Oracle® Communications ASAP

Cartridge Development Guide

Release 7.4.1
G13671-01
March 2025

Oracle Communications ASAP Cartridge Development Guide, Release 7.4.1

G13671-01

Contents

3 Configuring Network Element Connections

4 Mapping Network Element Commands to Actions, Entities, and Parameters

5 Creating and Configuring Atomic Actions

10 Configuring Dynamic and Static Event Templates for Return Parameters

11 Creating Java Connection Handlers

12 Creating Action Processors and Programs for Processing Requests and Responses

Preface

This guide provides guidance and best practices for creating Oracle Communications ASAP cartridges using Oracle Communications Service Catalog and Design - Design Studio for Activation.

Audience

This guide includes information for:

- Business analysts
- Cartridge service or network modelers
- Cartridge developers

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

This chapter provides an overview of the cartridge contents and the creation process.

About Cartridge Creation Options

Oracle Communications ASAP supports the following methods for creating cartridges:

- Design Studio for ASAP
- XML
- Stored Procedures

Design Studio for ASAP

The recommended development environment for creating a cartridge is Oracle Communications Service Catalog and Design - Design Studio. This guide describes the cartridge development process with Design Studio.

Design Studio simplifies the creation, assembly, and deployment of services across multiple domains. Design Studio functionality includes:

- Creating, deploying, and managing cartridges
- Extending cartridges into customer specific service configurations
- Managing and deploying complex multi-domain services to production, test, and development environments
- Modeling network element instances using predefined network element instance and connection attributes
- Creating and deploying patches

Design Studio has been optimized for developing Java-based ASAP cartridges.

Even though activation cartridges can be built outside Design Studio, this is not the recommended approach. Design Studio speeds up the development process and optimizes cartridge design and implementation by providing:

- Service model and Java stubs autogeneration feature
- Enforced naming conventions and consistency
- Cartridge documentation autogeneration
- Standard Eclipse development editors for business logic implementation (Java classes)
- Intuitive cartridge deployment/undeployment
- Testing harness
- Identifies problems and errors at development time
- GUI interface

ASAP cartridges created in Design Studio are validated again the ASAP schemas described in "About Cartridge XML Schemas" during the build process and when deploying the cartridge service activation archive (SAR) file to the ASAP environment.

For more information, see the Design Studio Help.

XML

You can create ASAP Java using XML that must conform to ASAP cartridge schemas. See "About Cartridge XML Schemas". This is an older way of creating ASAP cartridges described in "Configuring Services Using XML."

Stored Procedures

Stored procedures have been deprecated.

You can directly add cartridge-specific information to the ASAP databases using SQL*Plus stored procedures. All cartridge-related stored procedures are described in "Configuring Services Using Stored Procedures."

About ASAP Cartridges

ASAP cartridges are discrete software components developed for ASAP. An ASAP cartridge provides specific domain behavior on top of the core ASAP software. This domain behavior includes a part of, or all services on a network element (NE), element management system (EMS), or network management system (NMS). In this guide, all of these systems are collectively called NEs.

An ASAP cartridge is not a standalone component, but it operates in conjunction with the core ASAP software. Cartridges can be designed for a specific vendor, technology, and software load, and elements within each network cartridge can be reused in the creating of common or mixed service model cartridges. For more information, see "About Service Modeling."

An ASAP cartridge can be used to configure ASAP to provision the following:

- NEs from a specific vendor (for example, Nokia).
- Technologies, such as HLR and GSM.
- Services that are supported on an NE, such as Wireless, Optical for VoIP, IPTV, or high speed internet.

ASAP Cartridge Contents

An ASAP cartridge contains the following components:

- An interface from ASAP to the NE that includes the following:
  - NE, NE template, or dynamic NE template containing connection protocol details, connection parameters required by the connection handler, and other services.
  - One or more connection handlers with associated Java code to run connection details provided in the NE, NE template, or dynamic NE template elements.
- A mapping of NE generated user exit types that you defined in the action processor methods to one of eight ASAP base exit types. You can also optionally map your user exit types to a regular expression search pattern.
- A set of action processors that include the following:

  - The action processor type: A Java processor class.
  - The action processor class: This Java class can be manually or automatically generated using Design Studio.
  - The action processor method: If you selected the option to autogenerate your Java code, Design Studio creates an **execute** method contained in the action processor class where you must implement the man-machine language (MML) commands that ASAP sends to the NE using the attributes and parameters specified in the atomic actions. You must also specify logic for your user exit types in this method. If you did not select the option to autogenerate your Java code, Design Studio allows you to create and select your own method.
- A set of atomic actions in the form of Atomic Service Description Layer (ASDL) commands that include the following:
  - A list of user-defined attributes that include the kind of routing support required to route the atomic action to an NE, and any parameters required by the associated MML command that ASAP implements and sends to the NE.
  - Rollback, retry, and index related atomic action configuration attributes.
  - A list of associated action processors that implement the MML command that ASAP sends to the NE.
- A set of service action commands in the form of Common Service Description Layer (CSDL) commands that form meaningful service actions. Each service action can incorporate one or more atomic actions.
- Sample work orders.

Cartridge Creation Workflow

You should fully understand the functionality and attributes for each NE that your cartridge must manage before you start to develop an ASAP cartridge. With this understanding, you can develop a service model focused on capturing the re-usable behavior in each NE.

The following list outlines the workflow required to build a cartridge. For additional details, rules, and guidelines for each step in the cartridge creation process, refer to subsequent sections in this document.

1. Select your cartridge type: service or network cartridge.
2. Define your NE details (for example, connection protocol and the maximum number of connections the NE supports, and so on). You can also create NE templates and dynamic NE templates at this time.
3. Define the corresponding connection handler for your NE.
4. Implement one or more Java classes with methods to run the NE connection details. You must associate your Java classes to your connection handler.
5. Identify the MML commands or API calls and parameters that your NE requires.
6. Map user-defined exit types (UDETs) to a base exit type.
7. Create corresponding atomic actions for each MML command or API call.
8. Specify the parameters used for the MML command or API call in the corresponding atomic action.
9. Create action processors and associated Java **execute** methods that implement the associated MML command.

10. Configure the Java **execute** method to handle MML response messages from the NE. Associate each NE response message to a UDET.
11. Associate the action processors to a corresponding atomic action.
12. Create a service action and associate it to one or more atomic actions to create a meaningful service.
13. Create sample work orders.
14. Deploy the cartridge.
15. Test the cartridge.

About Cartridge XML Schemas

ASAP cartridges created using Design Studio or with XML must conform to the *ASAP_Home***/xml/xsd/ServiceModel.xsd** schema. The SAR file structure (created automatically when you use Design Studio, or manually when you use XML) must conform to the *ASAP_Home***/xml/xsd/SA_Archive.xsd** schema.

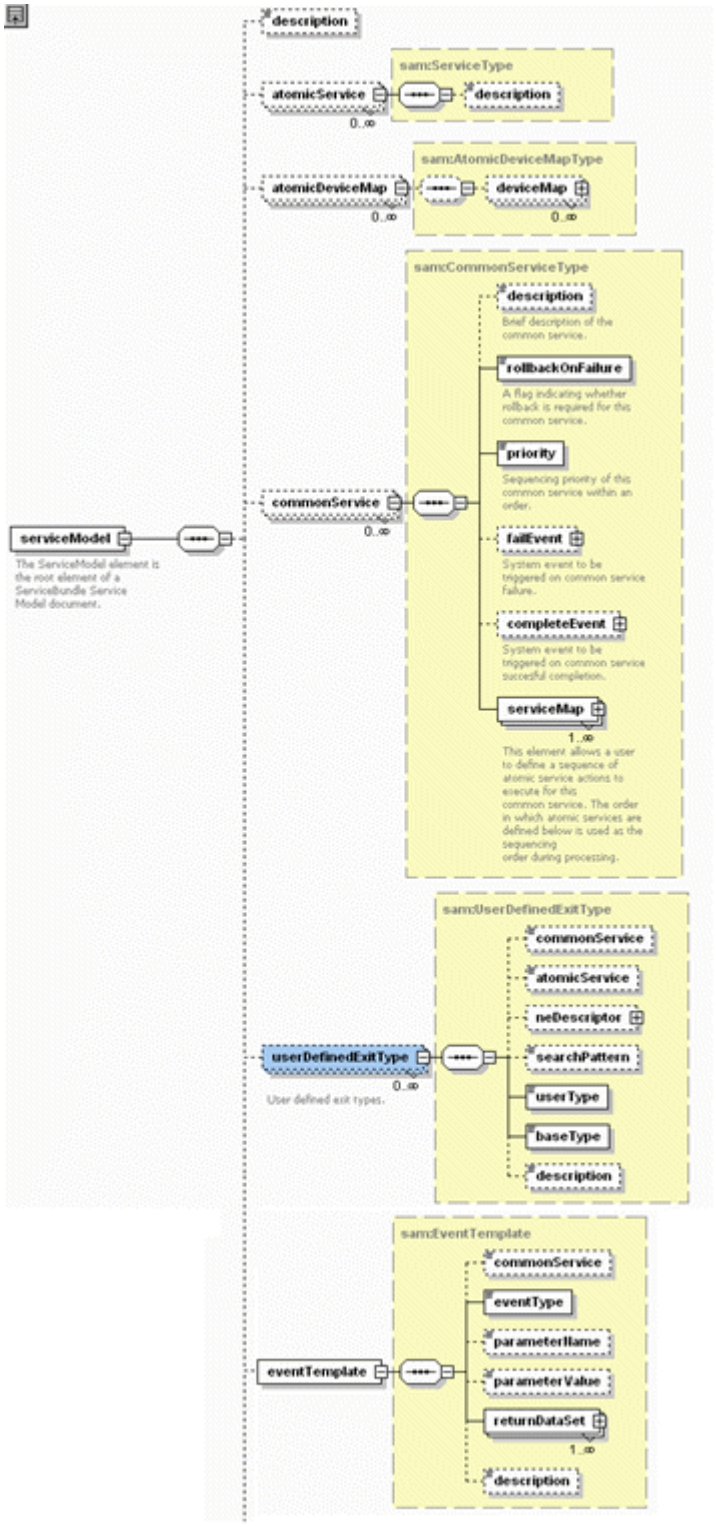
This section describes the uses and structure of **ServiceModel.xsd** and **SA_archive.xsd**.

ServiceModel.xsd

The **ServiceModel.xsd** file defines the content and structure of one or more **ServiceModel.xml** files. This file ensures that the element hierarchy and document structure of the **ServiceModel.xml** file are correct and ensures that element and attribute content adheres to the defined datatype.

Figure 1-1 shows the element and structure of the **ServiceModel.xsd** schema as described in the *Java Online Reference* available with the ASAP installation files.

Figure 1-1 ServiceModel.xsd Elements and Structure



The **ServiceModel.xsd** file contains the elements and structure to define:

- Atomic actions and their associated rollback conditions, timeout and retry settings, parameters and associated devices and software loads
- Device mappings (atomicDeviceMap), which provide a type definition to map atomic actions to NE types
- Service actions and their associated rollback conditions, priorities, provisioning events, and mappings to atomic actions
- Base-exit-type-to-user-exit-type mappings
- Event template mappings to return extended event information

> **Note:**
>
> Design Studio automatically conforms to this schema when you generate a cartridge, although the Design Studio GUI screens do not necessarily map to each schema elements.

Depending on the service modeling strategy, the service definition can be contained in one or more service model files. For example, all of your service definitions can be contained in a single service model file. Alternatively, larger organizations can distribute, add, modify, and delete service actions over three different service model files or create a service model file for each service.

The **ServiceModel.xsd** file is fully annotated, and the *ASAP_Home*/**samples/sadt** directory contains sample service models.

The XML files you create that contain the service models can have any name, provided the **<ServiceModel>** element in the activation model deployment descriptor (**activation-model.xml**) correctly references it.

SA_archive.xsd

The **SA_Archive.xsd** file is the schema upon which **activation-model.xml** is based. The **activation-model.xml** file identifies the components contained in the service activation archive to be deployed by the SADT or the **installCartridge** script.

> **Note:**
>
> Design Studio automatically conforms to this schema when you generate a cartridge SAR file.

These components include, at a minimum, one or more service models and the required JInterpreter provisioning classes. You can optionally include other components, such as:

- Customized SQL (the SQLDeploy type allows you to add customer data to both the SRP and NEP schemas).
- Documentation (including design guidelines, API documentation, and so forth).

The ComponentType attribute appears as follows in the **SA_Archive.xsd** file:

```
<xsd:complexType name="ComponentType">
<xsd:annotation> <xsd:documentation>A component type can one of either a serviceModel,
```

```
or a customized SQL.
         </xsd:documentation>
</xsd:annotation>
<xsd:choice>
<xsd:element name="serviceModel" type="am:XMLFileType"/>
<xsd:element name="javaProvisioningFile" type="am:ProvisioningClassFileType"/>
<xsd:element name="srpSQLFile" type="am:SQLDeployType"/>
<xsd:element name="nepSQLFile" type="am:SQLDeployType"/>
</xsd:choice>
```

The **activation-model.xml** file must reside in the **META-INF** directory.

The **SA_Archive.xsd** file is fully annotated, and the *ASAP_Home*/**samples/sadt** directory contains **activation-model.xml** files contained in the sample SAR files included with ASAP.

About Service Modeling

ASAP supports the following service models:

- Vendor, technology, and software load-specific service model

  This service model aligns common service actions and atomic actions with one vendor, technology, and software load. Design Studio for ASAP refers to this model as a network cartridge.

- Common service model

  This service model groups service actions and atomic actions for different vendor, technology, and software loads into one service cartridge. Each service action and atomic action combination supports only one vendor, technology, and software load. Design Studio for ASAP refers to this model as a service cartridge.

- Mixed service model

  This service model associates a service action to atomic actions created for different vendor, technology, and software loads. Design Studio for ASAP also refers to this model as a service cartridge.

For more information on these service models, see *Design Studio Modeling Activation*.

You must design your cartridges to enable the use of the service models you require, for example, by applying naming conventions and parameter standards across cartridges so that merging of cartridge-specific objects into a common or mixed service model can occur on a customer project. This guide provides guidelines and best practices for creating each element using such consistent conventions and standards to facilitate service modeling.

2

Creating a Cartridge Project

This chapter describes how to define an Oracle Communications ASAP cartridge project.

About Cartridge Types

ASAP provides two cartridge types that support the three service models (see "About Service Modeling"):

- Network cartridges
- Service cartridges

Network cartridges can be used to implement the vendor, technology, and software load specific service models. Service cartridges can be used to implement both common and mixed service models.

> **Note:**
>
> You can select the cartridge type and all components described in this chapter using the New Studio Activation Cartridge Project Wizard in Design Studio for ASAP. For more information about this wizard, see the discussion on setting up an activation cartridge in Design Studio Help.

Defining Network Cartridge Project Parameters

Network cartridges target a single vendor, technology, and software load. The development process starts with the network element (NE) interface documents, identifying the services and commands supported and then deciding which set of services to be implemented. Specific customer business logic has no impact, because the solution layer must be implemented as a service cartridge. The scope is to develop generic, reusable libraries of atomic actions, which can then be used for custom solutions projects.

Network cartridges typically support a one-to-one mapping between service action and atomic actions, simplifying service modeling; however, this pushed back the problem of creating meaningful services to the work order level. For more recommendations about scenarios where network cartridges are appropriate, see the Design Studio Help.

You can purchase network cartridges from Oracle, or you can create your own network cartridge.

Defining network cartridge project parameters includes the following tasks:

- Defining Network Cartridge Identification Tokens
- Defining the Scope of the Network Cartridge
- Creating a Design Studio Project

Defining Network Cartridge Identification Tokens

Name each network cartridge using elements that uniquely identify it. The following items are included in a cartridge name:

- Vendor
- Technology
- Software load

> **Note:**
>
> You define these three elements using the New Studio Activation Cartridge Project Wizard in Design Studio for ASAP. For more information about this wizard, see the discussion on setting up an activation cartridge in the Design Studio Help.

Selecting the Vendor Token

The Vendor token is a string that uniquely identifies the manufacturer of the NE: for example, **ALU** for Alcatel-Lucent, or **ERIC** for Ericsson. It is embedded in the service modeling object names and Java method names. Use Table 2-1 to select a vendor token, otherwise check the NASDAQ symbol for hints but do not use symbols that are cryptic, for example the SONUS NASDAQ symbol is SONSE which is not as meaningful as SONUS.

Table 2-1 Vendor Token Examples

Company	Vendor Token
Alcatel-Lucent	ALU
Ericsson	ERIC
Cisco	CSCO
Comverse	CMVT
Copper Mountain	CMTN
Logica	LGIA
Lucent	LUC
Nokia	NOK
Nortel	NT
Redback	RBAK
Siemens	SIEM
Sonus	SONUS
Vodafone	VF

Selecting the Technology Token

The technology token is a string that identifies the category of services or vendor classification of equipment to which the NE belongs: for example, HLR for Home Location Register or

DSLAM for Digital Subscriber Line Access Module. In some cases, a vendor specific term (such as DMS or STINGER) may be used in place of the technology token.

See Table 2-2 for technology token examples.

Table 2-2 Technology Token Examples

Vendor	Technology Token	Description
Generic	`HLR`	Home Location Register NE
Generic	`DSLAM`	Digital Subscriber Line Access Module
Generic	`VMS`	Voice Mail Server
Generic	`SMS`	Short Message Server
Nortel	`DMS`	Digital Multiplex System Voice NE
Alcatel-Lucent	`STINGER`	Digital Subscriber Line Access Module

Selecting the Software Load Token

The software load is an alphanumeric string representing the software load of the element management system (EMS) that manages the NE, or the software load running on the NE itself. The selection of the software load to be supported is based on the entity (EMS, Network Management System (NMS), or NE itself) that the cartridge is designed to interface with.

A software load containing a minor release number (for example, 1.2) has a corresponding software load token of 1-2. This is the same token used in the name and configuration of the sample NE and in the atomic action to Java method mapping (see the **sftwr_load** token of **tbl_nep_asdl_prog**). Do not use periods in the name of the software load token, use dashes instead.

A cartridge only supports one technology and software load. When the software load for a particular technology changes, build a new cartridge to support the new software load.

In general, create a new cartridge release when a major or minor change in the software load occurs, and specifically when the changes between these releases are significant. Some vendors make significant changes in their software between minor releases (for example 1.2 to 1.3); other vendors make the significant changes in their cartridges between major releases (for example 3.0 to 4.0).

Using an x in the second or third digit of the cartridge software load value indicates that the release does not have significant changes for any releases that change the digit marked as x. For example, a cartridge marked with software load 1-2-x, assumes that small changes occur in the third digit. Changes in the cartridge may be needed if additions are made to the NE software as part of such a release, but the cartridge software load remains intact and the cartridge remains backward compatible.

Defining the Scope of the Network Cartridge

Two approaches can be taken in determining the scope of the network cartridge:

- Comprehensive - aimed at supporting as much functionality as provided by the NE. You may develop more than one service package (see "Selecting Entity Tokens") for the various services supported on an NE (for example, Frame Relay, FRATM, and ATM services). For cartridges supporting many different types of services, the comprehensive approach can require significant development effort.

- Service-specific – often driven by a customer request or market demand for support for a particular set of services on an NE: for example, ATM PVCs. Because the scope of the cartridge is limited to a subset of functionality, this approach often requires less development effort. Additional services in the form of sub-cartridges can be supported on the NE in the future.

Factors influencing the approach that is taken include the time available to implement the cartridge, customer priorities and the number of services provided by the NE.

Creating a Design Studio Project

Design Studio for ASAP automatically creates your directory structure for you as you add new service actions, atomic actions, and action processors.

The project name should identify the name of the vendor, the technology, and the software load to differentiate your cartridge from other cartridge projects.

Defining Service Cartridge Project Parameters

Service cartridge can select components from any network cartridge to create customized service models that can simultaneously activate and configure diverse NEs from any vendor, technology, and software release.

For more information about the kinds service cartridges, see the Design Studio Help.

Importing and Extending Network Cartridges in Service Cartridges

Service cartridges extend and customize the services provided in network cartridges. To access the service action, atomic actions, action processors, network connections, user exit types, and event templates configured in network cartridges, you must import the network cartridges in Design Studio before creating a service cartridge project.

For more information about importing cartridge projects from SAR files, see Design Studio Help.

> **Note:**
>
> Importing cartridge projects from SAR files is deprecated functionality. Oracle recommends that you distribute and deploy Design Studio Projects rather than SAR files.

> **Note:**
>
> You can reuse service actions, atomic actions, action processors, network connections, user exit types, and event templates configured in network cartridges purchased from Oracle in a customized service cartridge. However, the source code for Java action processor classes, methods are not provided. If you require access to specific code in order to extend existing network cartridge, Java implementations, request access by raising a service request with Oracle Support.

3
Configuring Network Element Connections

This chapter describes how to configure Oracle Communications ASAP to connect to a network element (NE).

About Network Element Configuration Components

Every cartridge must create the following components to enable a connection to an NE:

- NEs and NE connections: see "About Network Elements and Network Element Connections"
- NE connection handler and associated Java code: see "About Configuring a Java Network Connection Handler"
- NE to network element processor (NEP) mapping: see "Mapping a Network Element to a Network Element Processor"

You can also create the following:

- NE templates

  NE templates provide reusable NE information that can be used to quickly create new NEs with similar attribute requirements. For more information, see the Design Studio Help.

- Dynamic NE templates

  Use the Dynamic NE Template editor to define a dynamic NE template entity. The entity routes orders based on network and communication data provided as order parameters, rather than using preconfigured static, locally maintained data. For more information, see "Configuring Dynamic Routing ."

About Network Elements and Network Element Connections

ASAP supports two types of NE connections:

- **Host**: Indicating a programmable NE directly connected to ASAP.
- **Remote**: Associated with the host NE and programmed through the designated host. ASAP routes service requests in the form of atomic actions through the host NE to the appropriate remote NE.

A host NE is an NE that has an interface through which remote NEs can be programmed. Several remote NEs covering a given area can be associated with a host NE, which increases the effective coverage of the NE group. Host NEs are not required to have remote NEs assigned to them.

ASAP can interface with many NE technologies and software loads over several logical and physical interfaces. The definitions for each host NE resides in **tbl_host_clli** in the service activation request manager (SARM) database. The records in this table define the different technologies (switch types) and software loads that are currently used by all service request processors (SRPs) and NEs in the system. The Java method interpreters reference this table to find the technology and software version for each SRP/NE in the system.

The definition for the remote NE resides in the user-configurable table (**tbl_clli_route**), which maps host NEs to remote NEs. Work orders sent to ASAP target the remote NE value populated within **tbl_clli_route**, so if the target NE is the host NE, then you must enter the name of the host NE in the remote NE field.

For more information about these tables, see *ASAP Developer's Guide*.

You can create and configure a host NE with Design Studio using the Network Element Wizard. You can designate one or more remote NEs or specify a host NE as the target for work orders after you have created a host NE from the Network Element editor **Target Network Element** tab.

To create and configure host NEs and remote NEs, see the following sections:

- Network Elements and Network Element Connections
- Adding Target Network Elements
- Setting Network Element Throughput Control

Network Elements and Network Element Connections

When you create an NE, you populate the **tbl_host_clli** table. This static table contains the host NE, the technology, and the software load of each NE in the ASAP system. It also contains records for each host NE to which the NEPs interface.

You can create an NE using Oracle Communications Service Catalog and Design - Design Studio with the Network Element Wizard.

tbl_resource_pool is a static table that defines the collection of command processors (devices) that the NEP uses to establish connections to NEs. Groups of command processors are called resource pools. Each NE configuration determines a primary resource pool that defines one or more devices the NEP uses to connect to that NE. These devices are not used to connect to other NEs. Each NEP has an auxiliary resource pool that contains devices used by the NEP to establish connections to any NE managed by the NEP. These primary and auxiliary resource pools are defined in this table. You must populate this table to add command processors.

The devices contained in resource pools are configured for a specific type of connection protocol.

The maximum connections setting for an NE must not exceed the number of devices in the primary resource pool.

If an NE allows for multiple simultaneous connections, the NE should have more than one device configured in its primary resource pool. Oracle recommends two or more connections in the resource pool.

For more information about connection pools, see the discussion about the NEP session manager in *ASAP Server Configuration Guide*.

Creating and Configuring Network Element and Network Element Connections

To create and configure NEs and NE connections:

1. In Design Studio, open an Activation project.
2. Select the **Studio** menu, then select **New**, then select **Activation**, and then select **Network Element**.

   The Network Element Wizard appears.

3. In the **Entity** field, enter an entity name.
4. Click **Finish**.

   The Network Element editor appears.
5. In the **General** tab, do the following:
   a. In the **Connection Pool Name** field, enter a connection pool name. Creates a connection pool of devices that the NEP uses to establish connections to NEs. For more information about connection pools, see *ASAP Server Configuration Guide*.
   b. In the **Protocol** field, enter a connection protocol. ASAP supports multiple communication protocols, and provides optional pre-configured parameters for these protocols. For more information about these protocols and parameters, see "About Communication Protocol Parameters."
   c. In the **Drop Timeout (minutes)** field, enter the drop timeout threshold in minutes. This field specifies the time threshold in which an NE receives no work orders from ASAP after which ASAP drops the connection. For more information about the Drop Timeout parameter, see *ASAP Server Configuration Guide*.
   d. In the **Spawn Threshold (AA)** field, enter the spawn threshold. This field specifies the number of pending atomic actions in an NE connection queue before the SARM spawn a new NE connection. For more information, see *ASAP Server Configuration Guide*.
   e. In the **Maximum Connections** field, enter the maximum number of connections. This field specifies the maximum number of connections that can be established to an NE. For more information, see *ASAP Server Configuration Guide*.
   f. In the **Kill Threshold (AA)** field, enter a kill threshold. This field specifies the termination of an NE connection when the number of atomic actions within an NE queue falls below this threshold. For more information, see *ASAP Server Configuration Guide*.
   g. In the **Retry Count** field, enter a the maximum number of retries. This field specifies the maximum number of retries, if an NE work order requests times out. If the number of retries exceeds retry count, then the order fails and rolls back. This attribute is configurable at the NE level, the atomic action level, the system level, and the work order level. For more information, see "About Retry Properties."
   h. In the **Retry Interval** field, enter a retry interval time. This field specifies the time period in seconds between NE retries. This attribute is configurable at the NE level, the atomic action level, the system level, and the work order level. For more information, see "About Retry Properties."
   i. In the **Throughput** field, enter the minimum number of transaction per NE instance. This field specifies the NE instance throughput control – the minimum number of transaction per NE instance. For more information, see "Setting Network Element Throughput Control."
   j. In the **Transaction Per** field, enter a time value for the **Throughput** parameter. For more information, see "Setting Network Element Throughput Control."
6. In the **Connection** tab, click **Add**.

   The Add Predefined Parameters dialog box appears.
7. Do one of the following:
   - To accept the auto generated parameters that ASAP preconfigures for the protocol, click **Yes**.
   - To create your own parameters click **No**.

Adding Target Network Elements

tbl_clli_route is a static table that contains the mapping between a remote NE and its host NE. You must populate this table if you want to specify a remote NE-to-host NE mapping. If you do not want to use a remote NE, you must specify the host NE as the target NE. Work orders are routed based on the **Target NE Name** field (called **mach_clli** in **tbl_clli_route**). In addition, you can associate individual atomic actions to specific remote NEs.

Any changes you make to the mapping relationships between host NEs to remote NE take effect at runtime. All other changes require that you restart the SARM.

To configure and create a network connection:

1. In Design Studio, open an Activation project.
2. Open an existing Network Element.
3. From the Network Element editor, on the **Target Network Elements** tab, click **Add**.
4. In the **Target NE Name** area, do one of the following:
   - If you want to route work orders to the host NE, enter the name of the host NE.
   - If you want to route work orders through the host NE to one ore more remote NEs, enter the name of the remote NE.
5. (Optional) If you want to associate an atomic action to your NE, click **Select** and add an atomic action from the list of available atomic actions. This option is available only if you have already created atomic actions.

Setting Network Element Throughput Control

Throughput control mechanism controls the number of transactions per unit of time. This mechanism ensures that networks elements are not overloaded.

To prevent certain types of NEs from becoming overloaded, it may be necessary to control the volume of transactions that are being sent from ASAP. A central throughput control mechanism enables you to configure a specific throughput per unit of time for NE instances, which ensures that no more than a specific number of transactions are sent to the NE per unit of time.

Consider the following scenario:

It has been discovered that the throughput limitations of a specific NE (that responds to ASAP asynchronously) require that no more than 20 transactions per second can be sent to the NE. Otherwise, some response messages are not generated and are therefore never received by ASAP. To prevent overloading and ensure the NE generates all required response messages, the service modeler configures throughput controls for this NE instance as described below.

To configure the throughput control for a NE instance:

1. In the NE Template editor, modify the throughput properties used to create new NE instances.

   When modifying the properties used to create new NE instances, you ensure that any future NE instances use the appropriate throughput properties. To do this, update the throughput values in the NE Template editor as follows:

   a. In the **Throughput** field, enter **20** as the number of transactions.

      Valid **Throughput** field values range from 1 - 9999.

   b. In the **Transactions Per** field, enter **Seconds** as the unit of time.

2. In the Network Element editor, modify the throughput properties for any existing NE instances of that type.

   Update the throughput values as follows:

   a. In the **Throughput** field, enter **20** as the number of transactions.

      Valid **Throughput** field values range from 1 - 9999.

   b. In the **Transactions Per** field, enter **Seconds** as the unit of time.

3. In the Dynamic NE Template editor, modify the throughput properties for any existing Dynamic NE Template used for NE instances of that type.

   Update the throughput values as follows:

   a. In the **Throughput** field, enter **20** as the number of transactions.

      Valid **Throughput** field values range from 1 - 9999.

   b. In the **Transactions Per** field, enter **Seconds** as the unit of time.

4. Save all modified NE templates, NEs, and dynamic NE templates.

   You can now deploy the configuration to an ASAP environment for testing.

About Configuring a Java Network Connection Handler

An NE connection handler associates an NE to Java code that implements the connection from the NEP to the NE.

In Design Studio, when you create an NE, you must choose a supported protocol for your NEP-to-NE connection. Then you can add one or more connections to the NE. ASAP provides you with optional base connection parameters. If you choose to accept these base parameters, ASAP will automatically generate supporting Java code (for more information about autogenerated protocol-specific communication parameters, see "About Communication Protocol Parameters").

ASAP sends these communication parameters to Java methods that you create to implement the connection. For more information about the Java code used to implement the connection, see "Creating Connection Methods and Helper Classes."

> **Note:**
>
> Communication parameters are not part of the data dictionary used for atomic actions.

Creating an Network Element Connection Handler

To create an NE connection handler:

1. In Design Studio, open an Activation project.
2. Select **Studio**, then select **New**, then select **Activation**, then select **NE Connection Handler.**

   The NE Connection Handler Wizard appears.

3. Do the following:

   - In the **Project** field, enter the name of the project.

- In the **Name** field, enter the name of the network connection handler element.
- In the **Folder** field, you can choose to create a new folder, or select an existing folder.

4. Click **Finish**.

   The NE Connection Handler editor appears.

5. In the Connection Handlers section, click **Add**.

   A new connection handler appears with the same vendor, technology, and software load of the project.

6. Click **New.**

   The Studio Activation Java Connection Handler Wizard appears.

> **Note:**
>
> Ensure that a dot does not precede the package name. If a dot precedes the package name, remove it.

7. In the **Name** field, enter a connection handler name.
8. From the **Connection Type** list, do one of the following:
   - To create a new telnet NE connection handler, select **Telnet**. Telnet NE Connection Handler automatically generates the code for telnet connections. This extends the telnet connection to support the interface. The NE Connection Handler editor indicates where additional code is required.
   - To create a custom NE connection handler, select **Custom**. Use this NE Connection Handler if the connections are not telnet. Custom Connection Handlers generate a skeleton to implement the IconnectionHandler and extends the base NE connection class. The NE Connection Handler editor indicates where additional code is required.
9. Click **Finish**.

> **Note:**
>
> The code is generated after but is not synchronized (that is, it does not automatically generate every time you change the NE Java code.) The developer must manage all the changes to automatically generated classes after they are created.

Mapping a Network Element to a Network Element Processor

You must map NEs to NEPs. NEPs perform the following tasks related to NE connectivity:

- Support a session manager that manages high level interaction with an NE. This includes routing to resource pools and determining which command processor (a thread that implements user-defined Java methods for connecting to an NE) to use within a resource pool.
- Provide interpreters (JInterpreter for Java methods) that run custom code that handles protocol and device-specific communication with NEs.

- Support for a connection handler method within the command processor that provides a transparent interface between the user-created Java methods and the protocol-specific communication details: for example, TCP/IP, serial, SSH, SSH FTP, and so on.
- Support for a Multi-Protocol Manager (MPM) within the command processor. ASAP maintains protocol-specific communication parameters in the SARM and loads them from the database by the NEP after you determine the communication protocol to use and prior to connecting to the NE.
- Manage connect, disconnect, login, connection spawning thresholds, connection destruction thresholds, maximum connections, and device throughput as defined in the information configured in ASAP cartridges.

For more information about NEP functionality, see *ASAP Server Configuration Guide*.

To map an NE to an NEP:

1. In Design Studio, select **Studio**, then **New**, then **Project**, then **Environment Project.**

   The New Studio Environment Project Wizard appears.
2. In the **Project name** field, enter a project name.
3. Click **Finish**.

   The Open Associated Perspective? dialogue box appears.
4. Click **No**.
5. From an Activation project, select **Studio**, then **New**, then **Environment**, then **Studio Environment.**

   The Studio Environment Wizard appears.
6. In the **Name** field, enter a name for the Studio Environment.
7. Click **Finish**.
8. From an Activation project, select **Studio**, then **New**, then **Environment**, then **NEP Map.**

   The NEP Map Wizard appears.
9. Do the following:
   a. From the **Project** list, select an environment project.
   b. From the **Studio Environment** list, select an environment.
   c. In the **Entity** field, enter a name for the NEP-to-NE mapping.
   d. Click **Finish**.

      The NEP Map editor appears.
10. In the **Network Element Processor Map** area, click **Add**.

    The Select a Network Element screen appears.
11. Select the NE you want to map your NEP to.
12. Click **OK**.
13. (Optional) If you have more than one NEP server, you can specify the name of the NEP server in the **NEP Server** field.

4

Mapping Network Element Commands to Actions, Entities, and Parameters

This chapter describes how to map network element (NE) commands to cartridge actions, entities, and parameters.

About Identifying Network Element Commands and Parameters

The bottom-up methodology begins with the identification of the NE commands, man-machine language (MML) commands or API calls, to be supported in the NE specification for the relevant service packages. You must develop an understanding of the services provided by the NE and the sequence in which the commands are provisioned to implement the services. This simplifies the effort of identifying action processors, atomic actions, and service action commands. You must also identify the parameters required to provision each action.

Defining Actions and Entities

Every service action, atomic action, and action processor consists of a combination of the following:

- Vendor, technology, and software load (see "Defining Network Cartridge Project Parameters" and "Defining Service Cartridge Project Parameters")
- Actions (see "Selecting the Action Tokens")
- Entities (see "Selecting Entity Tokens")

> **Note:**
>
> Using Design Studio for ASAP, you can define actions and entities using the Atomic Action Wizard, Service Action Wizard, Action Processor Wizard, or Cartridge Layout tool. For more information on these features, see Design Studio Help.

Selecting the Action Tokens

This set of tokens represents the actions that can be taken on the NE. Different NE vendors may use different tokens to represent identical actions (for example, ADD and SET). NEs from different vendors may use similar tokens to represent different actions (for example ADD and ACTIVATE). Oracle recommends that you select one of the mainstream actions (as shown in the list below) without distorting the meaning of the action taken. If this is not possible, select the action token reflected in the vendor documentation.

Actions can be any verb however the mainstream actions recommended by Oracle are as follows:

- ADD

- DEL
- CHG
- ACTIVATE
- DEACTIVATE
- QRY

The action token used in a service action is in most cases the same as the action token used in the corresponding atomic action, and action processor when there is a one-to-one mapping. Many-to-one mapping reflects the net result of the actions taken at the atomic action and action processor level in the action of the service action. Action processors in most cases should use the same action as defined in the atomic action.

Table 4-1 provides an example of a service action-to-atomic and action processor mapping.

Table 4-1 Service Action, Atomic Action, and Action Processor Action Mapping

Service Action Verb	Atomic Action Verbs	Action Processor Verbs
C_ALU-MOCA_R6_**ADD**_CAW	A_ALU-MOCA_R6_**ASSIGN**_CAW A_ALU-MOCA_R6_**ENABLE**_CAW	I_ALU-MOCA_R6_**ASSIGN**_CAW I_ALU-MOCA_R6_**ENABLE**_CAW

Selecting Entity Tokens

NEs can have various domain-specific entities that require further specification. These entities are the recipient of the action verbs.

In some cases, the volume of services supported on an NE requires that you create logical functional groups of services called service packages. For example, a cartridge for an NE that supports various types of data services might have the following service packs:

- ATM
- FRAME
- FRATM
- BGP

When multiple service packages organize a cartridge, a common service package can contain common components, if applicable, such as connection classes, helper classes, common actions taken across service packages, and so on.

You can also choose the service names that can be manipulated in the cartridge. Services could be subscribers, features such as call waiting, three way calling, or logical components such as cross connects. For example:

- X-CONN
- SUBS
- CALL-FORWARD
- THREE-WAY-CALLING
- PORT
- GSM-SUBS

Service package and service name should be used in the naming convention of service action, atomic action, and action processor commands. Separate compound service package and service sub-tokens with a dash rather than an underscore. For example:

Table 4-2 Service Action, Atomic Action, and Action Processors Entity Tokens

Service Action Entity	Atomic Action Entity	Action Processor Entity
C_NOK-HLR-R4_ADD_**BGP-SUBS**	A_NOK-HLR-R4_ADD_**BGP-SUBS** A_NOK-HLR-R4_ENABLE-**BGP-SUBS**	I_NOK-HLR-R4_ADD_**BGP-SUBS** I_NOK-HLR-R4_ENABLE_**BGP-SUBS**

Generating a Cartridge Layout

For activation network cartridges, service actions, atomic actions, and action processors are created and linked in a 1:1:1 relationship for all combinations of the actions and entities you specify.

For an activation service cartridge, only service actions are created, allowing a non-restricted association either with already existing network activation cartridges atomic actions or new atomic actions defined as part of the solution. For activation service cartridges, a decision must also be made about the type of service model you create (common, mixed, or vendor/technology/software load-specific), which affects the naming convention used for the atomic actions.

When using the cartridge generation feature, you specify the actions that will be performed by the cartridge (for example, ADD, MOD, DEL, QUERY and so on) and the entities targeted by these actions (for example, PORT, SUBSCRIBER, SUBSCRIPTION, LINE, and so on). After entering this information into the Project editor **Cartridge Layout** tab, you can generate a framework model by clicking the **Generate Cartridge** button.

Design Studio uses the action and entity tokens in the Java autogeneration feature (class and package names) therefore do not use special characters (like dashes) when naming these components. Use single tokens when defining actions and entities. Use standard names for actions whenever possible, like ADD, DELETE, QUERY and MODIFY across all the cartridges. Use short and descriptive names for entities. Whenever the cartridge template cannot be auto generated in a single pass, use the feature several times. For example:

- In the first pass, generate a template for ADD, DELETE, QUERY, MODIFY actions and SUBSCRIBER and FEATURE entities
- Remove actions and entities for which the template has been generated under the **Cartridge Layout** tab and add new actions and entities (like ENABLE, DISABLE, BLOCK, UNBLOCK actions and SERVICE entity).

The cartridge generation feature does not overwrite a framework that already exists. Rather, it adds to framework new and modified actions and entities. Additionally, Design Studio does not delete old actions or entities. You can, however, delete them manually.

About Parameter Types

Each atomic action has parameters that are sent to the NEP by the atomic action. The parameters determine whether the SARM transmits a particular atomic action.

tbl_asdl_parm is a Static Table that is used by the SARM to define the parameter labels and values associated with a given atomic action. It also provides the mapping between the service action parameter labels received from the service request processor (**csdl_lbl**) and the atomic action parameter labels (**asdl_lbl**) transmitted to the NEP for interpretation.

For each service action label (**csdl_lbl**), the SARM checks the current service action parameter name-value pairs for a matching label. If no matching label is found, it checks for a label in the work order global parameter name-value pairs. If no matching label is found in either of these parameter name value pairs and the parameter type (param_typ) which is mandatory, the default value (default_vlu), is used.

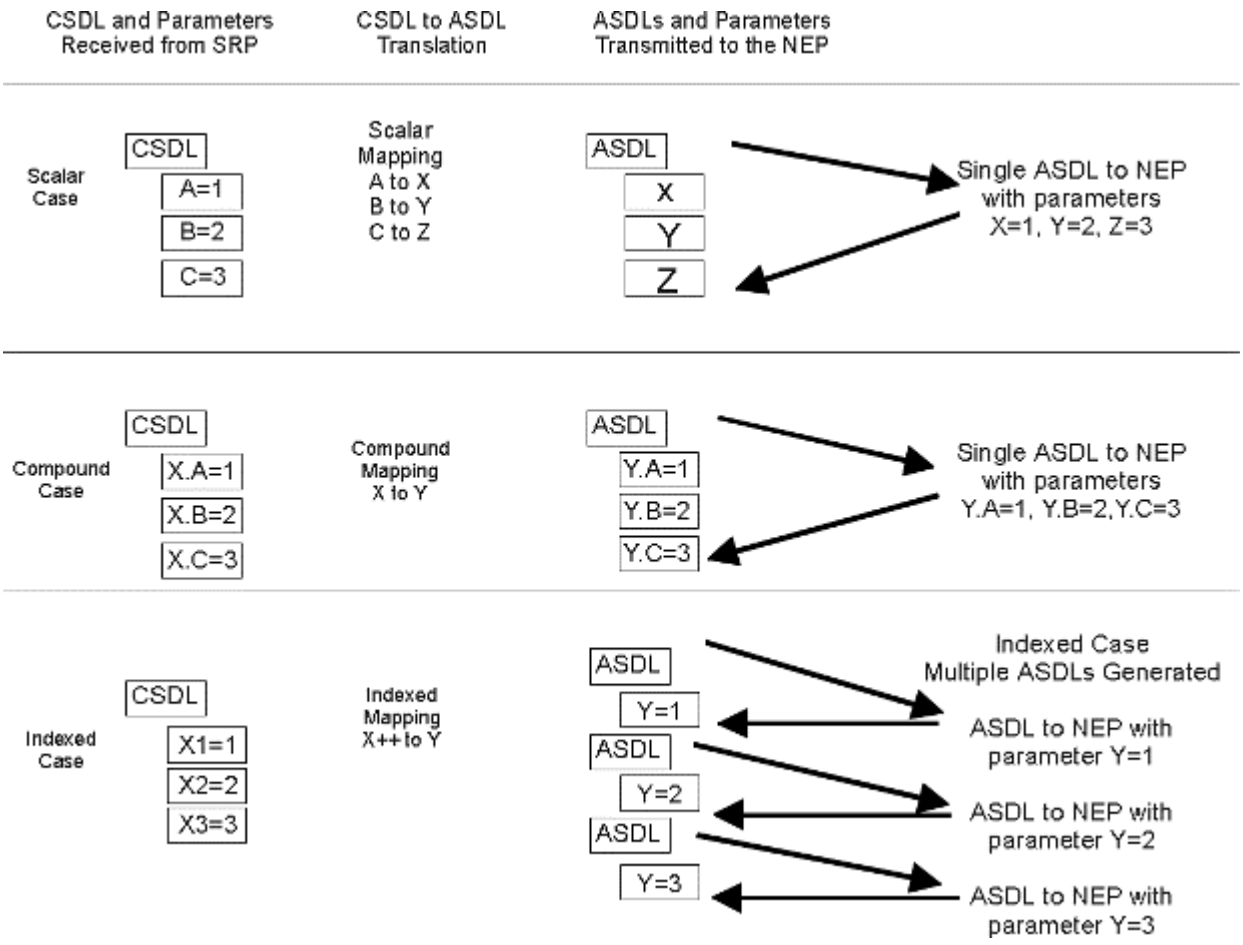
If no default value is set, the SARM registers an atomic action parameter mapping failure. If the parameter is indexed, the **csdl_lbl** must contain a **++** or the SARM will not start.

Atomic action parameters can be one of the following types:

- **R**: required scalar
- **O**: optional scalar
- **C**: required compound
- **N**: optional compound
- **M**: mandatory indexed
- **I**: optional indexed
- **X**: required XML
- **Y**: optional XML
- **P**: required XPATH
- **Q**: optional XPATH
- **+ –**: the current index value for this atomic action. Only applicable to indexed atomic actions.

> **Note:**
>
> You can create, modify, or delete new runtime parameters, specify the parameter type, and specify whether the parameter is optional or mandatory using the Design Studio Data Schema editor. See Design Studio Help references for the **Activation** tab and the **Details** tab for the ASAP Data Schema editor.

Figure 4-1 ASAP Parameter Types



Default Values Rules and Guidelines

Provide default values for parameters only when the NE documentation suggests that one value or setting is much more common to use than another.

About Creating a Data Dictionary

You may use a data dictionary with ASAP in the following two scenarios:

- When you have identified the NE commands and identified actions and entities (or generated a cartridge framework), you must create a data dictionary of all parameters required for the NE commands. In this scenario, you are creating the data dictionary based on the information you have gathered about the NE.
- You import an existing data dictionary into Design Studio, and you must associate the relevant data or structured elements to NE commands, actions, and entities. In this scenario, the data dictionary already exists, and you must map and configure these data elements for use with ASAP.

For more information about creating a data dictionary, see Design Studio Help.

After you have created the data dictionary, you must encapsulate the parameters within atomic actions. In most cases, you create one atomic action for each of the provisioning actions that

can be taken on the NE. For more information about creating atomic actions, see "Creating and Configuring Atomic Actions ."

Creating an ASAP Cartridge Project Data Dictionary Using Design Studio

Design Studio automatically creates a data dictionary for each cartridge project when you create a new Design Studio cartridge project. Design Studio also creates data dictionaries for cartridges imported into Design Studio.

Scalar Parameters

Scalar parameters are conventional name-value pair parameters.

```
Service Action      C-ADD_FEATURE
     PARM  NE_ID    NEWYORK
     PARM  LEN      2111112
     PARM  LATA     516
     PARM  LCC      555
```

Creating a Scalar Parameter using Design Studio

To create a mandatory, optional, or indexed scalar parameter using Design Studio:

1. Select **Studio**, then select **Show Design Perspective**.
2. Select the **Data Element** tab.
3. Right click in the Data Element dialog box.
4. Select **Add Simple Schema Element**.

   The Create Data Schema Element wizard appears.
5. Enter the following:
   a. In the **Entity** field, enter the name of the project to which you want to add a scalar parameter.
   b. In the **Name** field, enter an element name.
   c. In the **Display Name** field, enter a display name. The Data Schema editor supports multiple languages for this field. The field adjacent to **Display Name** displays your language. You can define a **Display Name** field value for any language you select from the list. For more information, see Design Studio Help.
   d. In the Multiplicity field, select one of the following:
      - **Required**: This attribute makes the parameter mandatory.
      - **Optional**: This attribute makes the parameter optional.
      - **Range**: Any ranged parameter with a **Minimum** value greater than 0 is considered a mandatory ASAP parameter. Any ranged parameter with a **Minimum** value of 0 is considered an optional ASAP parameter.
6. Click **Finish**.

   The new parameter appears in the Data Element dialog box.
7. Click the new parameter.

   The Data Schema editor appears.
8. In the Element section, click the **Activation** tab.

9. From the **Runtime type** list, select **SCALAR**.
10. (Optional) Select **Indexed** to index the parameter.

Indexed Parameters

These parameters contain a sequential numerical index value to tell the SARM that it should run the same operation (for example, an atomic action) for all occurrences of that index. Consequently, if there are several options on a particular service action command (OPT1, OPT2, OPT3, etc.), you can specify the OPT parameter as an indexed parameter. When you specify the OPT parameter as an indexed parameter, the SARM generates several occurrences of that same atomic action, and each command has a different value for the option being transmitted to the NEP.

If there are 100 such indexed parameters on the service action command, the SARM transmits the same atomic action 100 times. Each time the SARM transmits the atomic action, the parameter has a different option value.

If an indexed parameter is configured to be transmitted on a given atomic action, only one indexed parameter value is transmitted with each atomic action, and the same atomic action is run repeatedly.

For instance, if the work order contains:

- OPT1 = 3WC
- OPT2 = CWT

And the service action-to-atomic action mapping contains the following:

```
Service Action Parameter   Atomic Action parameter
------------------------   -----------------------
OPT[++]                    OPTION
```

> **Note:**
>
> By convention, the ++ notation appears at the end of the label within square brackets. This convention makes it easy to identify the index.

That particular atomic action is run twice. The first time, the atomic action has an OPTION parameter with the value 3WC. The second time, the atomic action has an OPTION parameter with the value of CWT.

Regardless of whether the service action references a Java provisioning class, the service action has access only to one parameter, which in this example is OPTION.

Compound Parameters

Compound parameters contain structures or arrays of information that are represented by a particular structure name or compound parameter name. Each compound parameter can contain a large number of elements. If you use compound parameters, you only require a single entry in the ASAP translation tables to call the compound parameter and all its associated parameter elements.

If you configure a compound parameter to be transmitted on an atomic action, ASAP transmits all elements for the compound parameter to the NEP at the same time.

> **Note:**
>
> In the case of compound parameters, the base name of the parameter on the work order must be exactly as specified in the **tbl_asdl_parm** and the base name must not have a period in it.

For example, if there is a compound parameter with the base name **CMPNDPARAM** specified in the **tbl_asdl_parm** as type **C**, you can define a work order with the following parameters:

```
CMPNDPARAM1=value1
CMPNDPARAM2=value2
CMPNDPARAM3=value3
CMPNDPARAMABC=value4
```

A compound parameter can be used by selecting parameter type **C** or **N**. A compound parameter (whether it is indexed or not) does not trigger the multiple execution of the same atomic action.

The following formats are supported for compound parameters:

- **Format 1** – Suffix cannot contain a period. For example:
  - BasenameSuffixA
  - BasenameSuffixB
  - BasenameSuffixC
- **Format 2** – a period comes directly after the basename. For example:
  - Basename.SuffixA
  - Basename.SuffixB
  - Basename.SuffixC

> **Note:**
>
> The basename must match the name defined in **tbl_asdl_parm.**

Creating a Compound Parameter using Design Studio

To create a compound parameter using Design Studio:

1. Select **Studio**, then select **Show Design Perspective**.
2. Select the **Data Element** tab.
3. Right click in the Data Element dialog box.
4. Select **Add Structured Schema Element**.

   The Create Data Schema Structure wizard appears.
5. Enter the following:
   a. In the **Entity** field, enter the name of the project to which you want to add a scalar parameter.
   b. In the **Name** field, enter an element name.

c. In the **Display Name** field, enter a display name. The Data Schema editor supports multiple languages for this field. The field adjacent to **Display Name** displays your language. You can define a **Display Name** field value for any language you select from the list. For more information, see the Design Studio Help.

d. In the **Multiplicity** field, select one of the following:

- **Required**: This attribute makes the parameter mandatory.
- **Optional**: This attribute makes the parameter optional.
- **Range**: Any ranged parameter with a **Minimum** value greater than 0 is considered a mandatory ASAP parameter. Any ranged parameter with a **Minimum** value of 0 is considered an optional ASAP parameter.

6. Click **Finish**.

   The new parameter appears in the Data Element dialog box.

7. Click the new parameter.

   The Data Schema editor appears.

8. In the Element section, click the **Activation** tab.

9. From the **Runtime type** list, select **COMPOUND**.

> **Note:**
>
> All child elements inherit the **Activation** tab attributes from the base compound element.

10. (Optional) Select **Indexed** to index the parameter.

11. From the **Data Element** area, right click the new parameter.

12. Select **Add Simple Child Schema Element**.

> **Note:**
>
> Compound parameters do not support structured child schema elements.

13. Enter the following:

a. In the **Name** field, enter an element name.

b. In the **Display Name** field, enter a display name. The Data Schema editor supports multiple languages for this field. The field adjacent to **Display Name** displays your language. You can define a **Display Name** field value for any language you select from the list. For more information, see the Design Studio Help.

c. In the **Multiplicity** field, select one of the following:

- **Required**: This attribute makes the parameter mandatory.
- **Optional**: This attribute makes the parameter optional.
- **Range**: Any ranged parameter with a **Minimum** value greater than 0 is considered a mandatory ASAP parameter. Any ranged parameter with a **Minimum** value of 0 is considered an optional ASAP parameter.

14. Click **Finish**.

15. Repeat steps 7 to 10 for any additional parameters to be included in the compound parameter.

Compound Indexed Parameters

The compound parameter can have an index. If using a compound indexed parameter, the parameter type must be **C**. The following format is supported for only indexed compound parameters.

- Basename[1].Suffix
- Basename[2].Suffix
- Basename[3].Suffix

You can define compound parameters and indexed parameters at the same time. This allows for the specification of multi-dimensional data elements.

> **Note:**
>
> For an example of a compound indexed parameter, see "Scenario 4 – One Service Action to Multiple Atomic Actions Routed to Multiple NEs."

Compound Parameters Rules and Guidelines

Avoid the use of compound parameters unless absolutely necessary. Using compounds makes the SARMs error checking capability far less effective and makes order entry through Order Control Application (OCA) more difficult. When multiple sets of parameters that have variable numbers of elements must be passed to the same implementation method for provisioning, a compound parameter with an associated index can be used (the index is purely for logical representation of the data and should not be confused with the atomic action indexing capability in ASAP). For example, a Java method that provisions multiple features in an optimized manner could be passed a compound structure containing variables as shown:

```
FEATURE[1].NAME = 3WC
FEATURE[2].NAME = CFD
FEATURE[2].NUM_RINGS = 5
BLOCKED_NUMBER[1].PATTERN[1]
BLOCKED_NUMBER[1].PATTERN[2]
```

Whenever an index is used within an atomic action parameter label, the index is encapsulated within brackets (regardless of the type of ASAP parameter):

```
SUD[1].CODE = A
SUD[1].VALUE = 1
SUD[2].CODE = C
SUD[2].VALUE = 7
```

Though rarely configured within a cartridge, support for dynamic routing should be considered in certain scenarios such as IP (routers) configuration. In such cases, the reserved COMM_PARAM label should be configured as an optional compound in the parameter list for each atomic action.

XML Parameters

Thee XML and XPATH parameter types are used in service modeling for network actions (atomic actions), similar to existing scalar, index, and compound parameter type. XML can be used as values for both information parameters and extended work order properties.

If the network action (atomic action) contains an XML parameter; **JProcessor** class within the Java enabled NEP loads the XML data from the SARM database and makes the raw XML available as the value of the XML parameter and as a Document Object Model (DOM) object.

XML parameters pass structured information into ASAP. The values of these XML parameters must be well formed XML that can be successfully processed by a standard XML parser.

Creating an XML Parameter using Design Studio

To create an XML parameter using Design Studio:

1. Select **Studio**, then select **Show Design Perspective**.
2. Select the **Data Element** tab.
3. Right click in the Data Element dialog box.
4. Select **Add Simple Schema Element**.

   The Create Data Schema Element wizard appears.
5. Enter the following:
   a. In the **Entity** field, enter the name of the project to which you want to add a scalar parameter.
   b. In the **Name** field, enter an element name.
   c. In the **Display Name** field, enter a display name. The Data Schema editor supports multiple languages for this field. The field adjacent to **Display Name** displays your language. You can define a **Display Name** field value for any language you select from the list. For more information, see the Design Studio Help.
   d. In the **Multiplicity** field, select one of the following:
      - **Required**: This attribute makes the parameter mandatory.
      - **Optional**: This attribute makes the parameter optional.
      - **Range**: Any ranged parameter with a **Minimum** value greater than 0 is considered a mandatory ASAP parameter. Any ranged parameter with a **Minimum** value of 0 is considered an optional ASAP parameter.
6. Click **Finish**.

   The new parameter appears in the Data Element dialog box.
7. Click the new parameter.

   The Data Schema editor appears.
8. In the Element section, click the **Activation** tab.
9. From the **Runtime type** list, select **XML**.

XPath Parameters

The XPath parameter type defines an XPath expression into XML data. From the runtime perspective, the JSRP, SARM and Java enabled NEP transfers XML data and XPath expressions to each other by saving the complex data into the SARM database, loading them from the database, and evaluating the XPath expression against the XML data.

When you provision a work order, the SARM loads the XML data from the SARM database and evaluates an XPath expression against the XML data in the following cases:

- An XPath parameter is used as part of service action spawning logic to determine whether an atomic action should be spawned or not
- An XPath parameter is used to spawn multiple instances of the same atomic actions depending on how many instances of XML elements are present in the work order

If there is an XPath parameter present in the atomic action, **JProcessor** evaluates the associated XPath expression when a user requests the value of the XPath parameter.

XPath parameters provide a mechanism to extract fragments from another XML parameter at runtime. In ASAP they are always used in association with an XML parameter, called in Design Studio, the **Dependent XML**. Design Studio enforces the association when defining XPath parameter in the context of an atomic action, but not in the context of data schema entity. If the association is defined in the context of a data schema entity, Design Studio makes an attempt to recreate it when the XPath data element is used in the context of an atomic action.

Creating an XPATH Parameter using Design Studio

To create an XML parameter using Design Studio:

1. Select **Studio**, then select **Show Design Perspective**.
2. Select the **Data Element** tab.
3. Right click in the Data Element dialog box.
4. Select **Add Simple Schema Element**.

   The Create Data Schema Element wizard appears.
5. Enter the following:
   a. In the **Entity** field, enter the name of the project to which you want to add a scalar parameter.
   b. In the **Name** field, enter an element name.
   c. In the **Display Name** field, enter a display name. The Data Schema editor supports multiple languages for this field. The field adjacent to **Display Name** displays your language. You can define a **Display Name** field value for any language you select from the list. For more information, see the Design Studio Help.
   d. In the **Multiplicity** field, select one of the following:
      - **Required**: This attribute makes the parameter mandatory.
      - **Optional**: This attribute makes the parameter optional.
      - **Range**: Any ranged parameter with a **Minimum** value greater than 0 is considered a mandatory ASAP parameter. Any ranged parameter with a **Minimum** value of 0 is considered an optional ASAP parameter.
6. Click **Finish**.

The new parameter appears in the Data Element dialog box.

7. Click the new parameter.

   The Data Schema editor appears.

8. In the Element section, click the **Activation** tab.
9. From the **Runtime type** list, select **XPATH**.
10. (Optional) Select **Indexed** to index the parameter.
11. In the Dependent XML field create or select a dependent XML. This attribute displays the path of the XML file that defines the parameter. This field is available only for the XPATH run-time type parameter.

Grouping Scalar Parameters using Design Studio Structured Elements

You can group ASAP scalar parameters in Design Studio by using the structured schema element feature. The structure element is a container that holds ASAP parameters. For example the following scalar groups can be defined using two levels of structure elements:

```
Structure element1
     Structure element2
               Scalar1
               Scalar2
Structure element3
     Structure element4
               Scalar3
               Scalar4
```

In a real world scenario, these structure could be as follows:

```
Person
     Name
         First_name
         Last_name
Place
     Address
         Number
         Street
```

The structure elements used in Design Studio are converted into individual ASAP scalar parameters by absorbing the structured element names into the scalar parameter name. The example used above describing a person and place would by default look as follows as ASAP parameters:

```
Person_Name_First_name
Person_Name_Last_name
Place_Address_Number
Place_Address_Street
```

The default character used to separate the elements in the ASAP parameter names is the underscore (_). It is possible to change this character. See Design Studio for more information.

To group scalar parameters using Design Studio:

1. Select **Studio**, then select **Show Design Perspective**.
2. Select the **Data Element** tab.
3. Right click in the Data Element dialog box.
4. Select **Add Structured Schema Element**.

The Create Data Schema Structure wizard appears.

5. Enter the following:
   a. In the **Entity** field, enter the name of the project to which you want to add a scalar parameter.
   b. In the **Name** field, enter an element name.
   c. In the **Display Name** field, enter a display name. The Data Schema editor supports multiple languages for this field. The field adjacent to **Display Name** displays your language. You can define a **Display Name** field value for any language you select from the list. For more information, see Design Studio Help.
   d. In the **Multiplicity** field, select one of the following:
      - **Required**: This attribute makes the parameter mandatory.
      - **Optional**: This attribute makes the parameter optional.
      - **Range**: Any ranged parameter with a **Minimum** value greater than 0 is considered a mandatory ASAP parameter. Any ranged parameter with a **Minimum** value of 0 is considered an optional ASAP parameter.
6. Click **Finish**.

   The new parameter appears in the Data Element dialog box.
7. Click the new parameter.

   The Data Schema editor appears.
8. In the Element section, click the **Activation** tab.
9. From the **Runtime type** list, select **SCALARS**.
10. From the Data Element area, right click the new parameter.
11. Select one of the following:
    - **Add Simple Child Schema Element**: Select this attribute if you want to immediately define xml or scalar parameters within the first structured element. If you select this option, go to step 12.
    - **Add Structured Child Schema Element**: Select this attribute if you want additional structured child schema elements below the first structured element. If you select this option, repeat steps 5 to 11.
12. Enter the following:
    a. In the **Name** field, enter an element name.
    b. In the **Display Name** field, enter a display name. The Data Schema editor supports multiple languages for this field. The field adjacent to **Display Name** displays your language. You can define a **Display Name** field value for any language you select from the list. For more information, see the Design Studio Help.
    c. In the Multiplicity field, select one of the following:
       - **Required**: This attribute makes the parameter mandatory.
       - **Optional**: This attribute makes the parameter optional.
       - **Range**: Any ranged parameter with a **Minimum** value greater than 0 is considered a mandatory ASAP parameter. Any ranged parameter with a **Minimum** value of 0 is considered an optional ASAP parameter.
13. Click **Finish**.

14. Repeat steps 12 to 13 for any additional parameters to be included in the scalar or xml parameter group.

5

Creating and Configuring Atomic Actions

This chapter describes how to create and configure Oracle Communications ASAP atomic actions.

About Creating and Configuring Atomic Actions

An Atomic Service Activation Layer (ASDL) or atomic action is an ASAP command that is associated with a particular Common Service Description Layer (CSDL) or service action command. A service action describes the service action to be performed, and can contain one or more atomic action. The atomic actions associated with the service action performs the operations on one or more Network Elements (NEs) in order to fulfil the services action.

The naming convention for a network cartridge atomic action is as follows:

A_*vendor*-*technology*_*softwareload*_*action*_*entity*

where:

- **A**: indicates an atomic action.
- *vendor*: vendor identifies the manufacturer of the NE. See "Selecting the Vendor Token."
- *technology*: technology identifies the category of services or vendor's equipment classification to which the NE belongs. See "Selecting the Technology Token."
- *softwareload*: softwareload represents the version of the EMS that manages the NE, or the version running on the NE. See "Selecting the Software Load Token."
- *action*: action is the action that can be taken on the NE. See "Selecting the Action Tokens."
- *entity*: entity is a domain-specific entity that is the recipient of the action. See "Selecting Entity Tokens."

The tokens in the name are separated by underscore characters. Compound tokens include a dash as a separator. If the software load token includes a dot (**.**), the system replaces it with a dash. All characters in the name must be in upper case.

If entities are used, *entity* must include the service package in its name. For example an atomic action belonging to the GSM service package would be named as follows:

```
A_CSCO-IOS_12-2-X_ADD_GSM-MAX-PREFIX
```

Service cartridge atomic actions do not have to follow the naming convention.

Design Studio for ASAP enforces this naming convention when you create an atomic action using the Atomic Action Wizard.

Creating and Configuring an Atomic Action

You can create an atomic action using Design Studio with the Atomic Action Wizard.

Each atomic action within the SARM has a configuration record that you can set up. This record contains the following attributes:

- Atomic action timeout and retry properties.
- Atomic action used for rollback: Determines which rollback atomic action the SARM must use if a rollback is required.
- Routing support: You can choose the routing method you want to use to send the atomic action to the Network Element Processor (NEP).

tbl_asdl_config is a user-populated static table that defines the atomic action configuration information required to handle routing and rollback at the atomic action level. It is used by the SARM to determine whether rollback is required for this atomic action, and if so, the rollback atomic action to use.

To configure an atomic action using Design Studio:

1. Select **Studio**, then **New**, then **Activation**, then **Atomic Action.**
2. From the Atomic Action Wizard, do the following:
   - Enter an action name that corresponds to a network element (NE) command.
   - Enter an entity name that corresponds to an NE service name or service package you want to configure.
3. Click **Finish**.

   The Atomic Action editor appears.
4. In the **Parameters** tab, add right click in the dialog box.
5. Do one of the following:
   - If you want to add a simple element:
     a. Click **Add Simple Data Element**.

        The Add Simple Element dialog box appears.
     b. Select one or more elements.
     c. Click **Finish**.
   - If you want to add a structured element:
     a. Click **Add Structured Data Element**.

        The Add Structured Element dialog box appears.
     b. Select one or more elements.
     c. Click **Finish**.
6. In the **Details** tab, select a routing method for the atomic action from the **Routing Support** list:
   - **None**: Indicates that no routing method has been selected.
   - **DN Routing**: Indicates that DN routing has been selected. For more information about DN routing, see "Configuring Atomic Action Routings by Using a Directory Number."
   - **NE Routing**: Indicates that NE routing has been selected. For more information about NE routing, see "Configuring Atomic Action Routings by Using a Network Element."
   - **ID Routing**: Indicates that ID routing has been selected. For more information about ID routing, see "Configuring Atomic Action Routings by Using ID_ROUTING."
   - **User Defined Routing**: Indicates that user-defined routing has been selected. For more information about user-defined routing, see "Configuring Atomic Action Routings by Using USER_ROUTING."

- **Dynamic Routing**: Indicates that dynamic routing has been selected. For more information about user-defined routing, see "Configuring Dynamic Routing ."

> **Note:**
>
> Selecting a routing method populates parameters specific to the routing method you selected in the Atomic Action editor **Parameters** tab.

7. In the **Details** tab, select atomic action routing configuration information from the **Details Information** section:
   - **Provide Parameter Count**: Select to indicate that the NEP should send the current index value for the atomic action.
   - **Index Count**: Specify the name of the parameter for obtaining the index value in Java provisioning classes.
   - **Timeout (Second)**: Specify the number of seconds before the ASAP server considers an atomic action in-progress as failed. The default value is 0, which means ASAP server will not consider the atomic action in-progress as failed. For more information, see "About Retry Properties."
   - **Rollback Atomic Service**: Specify an atomic action that rolls back the changes of the current atomic action in a failure scenario.

     For example, atomic action A is mapped to service action B. The rollback is configured on the service action. On the Atomic Action editor, in the **Details** tab, for the atomic action entity A, you select an atomic action Y. In case of a failure scenario, the service action B is rolled back and atomic action Y is called to rollback the action of atomic action A. For more information, see "About Configuring a Rollback Atomic Action."
   - **Retry**: Enables the **Retry Count** and the **Retry Interval** fields. For more information, see "About Retry Properties."
   - **Retry Count**: Specifies the number of times the atomic action can be tried at the NE. For more information, see "About Retry Properties."
   - **Retry Interval (Second)**: Specifies the time interval, in seconds, between each retry attempt by ASAP. For more information, see "About Retry Properties."
8. In the **Mappings** tab, click Add.

   The Action Processor Selection Dialog appears.
9. Do one of the following:
   - If you have already created an action processor, select the action processor and click **OK**.
   - If you have not created an action processor and you want to create one now, click **New**.

     The Action Processor Wizard appears. For more information about creating and configuring an action processor, see "Creating an Action Processor."

About Retry Properties

Retry properties instruct the SARM to retry an atomic action according to the **Retry Count** and **Retry Interval** parameter that you have configured. If the atomic action does not complete after the final retry, the SARM fails it.

Timeout and retry attributes are configurable at:

- The atomic action level using the **Timeout (second)**, **Retry Count,** and **Retry Interval (second)** attributes. These attributes are defined in the Design Studio Atomic Action editor **Details** tab.
- At the NE level using the **Drop Time Out (minutes)**, **Retry Count** and **Retry Interval** attributes. These attributes are defined in the Design Studio Network Element editor **General** tab.
- At the work-order level.
- At the system level.

If an atomic action needs to be retried, the atomic action timeout and retry attributes are applied first. If no atomic action timeout and retry attributes are configured, the attributes configured for the NE apply. If no timeout and retry attributes are configured for the NE, the work order attributes are applied. If no work order timeout and retry attributes are configured, system-wide attributes are used.

If ASDL_TIMEOUTS is disabled in the **ASAP.cfg** file, all atomic action timeouts are disabled, regardless of whether timeout and retry data is configured for the atomic action.

These properties are specified on the work order. Default retry properties are also specified in **ASAP.cfg**.

Table 5-1 Retry Properties

Property	Description
NUM_TIMES_RETRY	Specifies the number of atomic action retries to be applied to an atomic action if the atomic action fails with a “Fail But Retry" condition.
RETRY_TIME_INTERVAL	Defines the time interval between atomic action retries if an atomic action fails with a “Fail but Retry" condition.

When defining hard error thresholds, you must consider the following points:

- The host NE, atomic action, and atomic action user exit code must already be defined.
- The same host NE, atomic action, and atomic action user exit code combination can only be used once.

For more information about configuring user exit types, see "Configuring Base Exit and User Exit Types ."

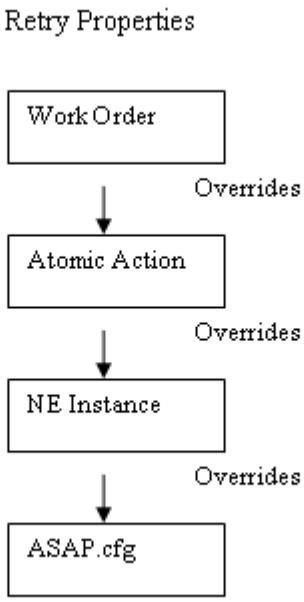
Because different NEs often have different retry requirements, it is necessary to provide a flexible retry mechanism that enables retry properties to be specified at the NE instance level and at the atomic action level (this is in addition to the ability to configure a single set of system-wide retry properties, which apply to all atomic actions and all NEs that trigger a retry).

Flexible retry configuration in ASAP enables specification of retry properties in the following locations:

- **ASAP.cfg:** This configuration file contains values for the Number of Retries and the Retry Interval, which will be used whenever a retry occurs, on any NE or atomic action, if no other values are configured elsewhere.
- **Work Order:** If the Number of Retries and Retry Time Interval are specified on a work order, these values will override those defined elsewhere in the system (including the **ASAP.cfg:** file, atomic action level, or NE instance level).

- **Atomic Action:** If you specify the Number of Retries and Retry Interval at the atomic action level, and a retry is encountered on any of the action processors mapped to that atomic action, the values you specify will be used. These values will override those defined at the NE instance level and at the **ASAP.cfg:** level.
- **Network Element Instance:** If you specify the Number of Retries and Retry Interval on the Network Element editor, any command triggering a retry against this NE instance will use the retry values you specify. These values will override those defined at the **ASAP.cfg:** level.
- **NE Template:** If you specify the Number of Retries and Retry Interval on the NE Template editor, any NE created from the template will inherit the retry values you specify.
- **Dynamic NE Template:** If you specify the Number of Retries and Retry Interval on the Dynamic NE Template editor, any NE instances dynamically created using the template will inherit the retry values you specify. These values will override those defined at the **ASAP.cfg:** level.

Figure 5-1 Retry Properties Locations



Example 1: Configuring Retry Properties at the Network Element Instance Level

A specific vendor's NE often responds with a FUNCTION BUSY message, meaning that it cannot presently process commands and that the command should be retried at a later time (there is not necessarily any problem with the command itself, but the load on the NE is too large at this particular moment). Best practices dictate that a command will eventually succeed if tried 3 times with a 10-second interval between tries. To ensure that the command is properly retried, the service modeler should configure the retry properties at the NE instance level. The work order will fail only if the configured Number of Retries is exceeded.

To configure retry properties at the NE instance level:

1. In the User Defined Exit Type editor, update the user-defined exit type configuration entry that corresponds to the FUNCTION BUSY response to specify an exit type of RETRY when this response message is encountered.
2. Modify the retry properties for any existing NE instances of that type.

   To do this, update the retry values in the NE editor for each NE instance as follows:

   - In the **Number of Retries** field, enter **3**.
   - In the **Retry Interval** field, enter **10**. (seconds)
3. Modify the retry properties for any existing Dynamic NE Template used for NE instances of that type.

   To do this, update the retry values in the Dynamic NE Template editor as follows:

   - In the **Number of Retries** field, enter **3**.
   - In the **Retry Interval** field, enter **10**. (seconds)
4. Ensure that all NE templates, NEs, and dynamic NE templates that were changed have been saved.

   After saving, you can deploy the configuration to an ASAP environment for testing.

Example 2: Configuring Retry Properties at the Atomic Action Level

When trying to change the LEN on a specific vendor's NE, the NE responds with an INVALID STATE error message if the customer line is in use. In this scenario, best practices dictate that ASAP retry the atomic action 10 times with an interval of 300 seconds between each attempt before a failure is be generated. The following example demonstrates how the service modeler configures the retry properties at the atomic action level to meet this criteria.

1. In the User Defined Exit Type editor, update the user-defined exit type configuration entry that corresponds to the INVALID STATE response to specify an exit type of RETRY when this response message is encountered.
2. When examining this NE's retry requirement, there are two options that would support the requirement:

   a. Modify the retry properties for the NE template (so that the configuration is carried over to any new NE instances that are created), for each NE instance of that type, and for each Dynamic NE template of that type.

   b. Modify the retry properties for the specific service action (change LEN). In this example, assuming the change LEN atomic action is specific to the vendor equipment in question (either a common atomic action mapping to only one vendor and technology, or a vendor and technology-specific atomic action mapping to a single action processor), and assuming the retry behavior specified for this requirement is unique to the atomic action (change LEN), then simply update the retry properties for the atomic action.

   > **Note:**
   >
   > Option a) requires multiple updates (to the NE Template, each NE instance, and each Dynamic NE Template). Option b) requires a single update.

3. Modify the retry properties for the change LEN atomic action.

   Update the retry values in the Atomic Action editor as follows:

a. In the **Number of Retries** field, enter **10**.

b. In the **Retry Interval** field, enter **300**. (seconds)

> **Note:**
>
> To update the retry value in an editor field, activate the field by selecting the corresponding check box. Retry values have no digit limit but must be positive integers. Retry values can be 0 if overriding the **ASAP.cfg** configured retry values is required.

4. Save changes to atomic actions.

   You can now deploy the configuration to an ASAP environment for testing.

About Delayed Failure Properties

Delayed Failure properties instruct the SARM to continue provisioning an order until the Order Delayed Failure Threshold is reached and the order is failed. These properties are work order properties.

Table 5-2 Delayed Failure Properties

Property	Description
Delayed Failure Property	Requests the SARM to treat all hard errors on atomic actions as Delayed Failures. The SARM skips any subsequent atomic action in the service action, continues provisioning at the next service action, and then fails the order. You can use the Delayed Failure property to override the ASDL_EXIT configuration in the Java method. This property should only be set when there are no dependencies on subsequent service actions on the work order. Upon hard failure of an atomic action, the associated service action is failed by ASAP, even if the Delayed Failure property is set.
Order Delayed Fail Threshold	Specifies the number of delayed failures that a particular order can have before the order is explicitly failed. This property is intended for batch orders.

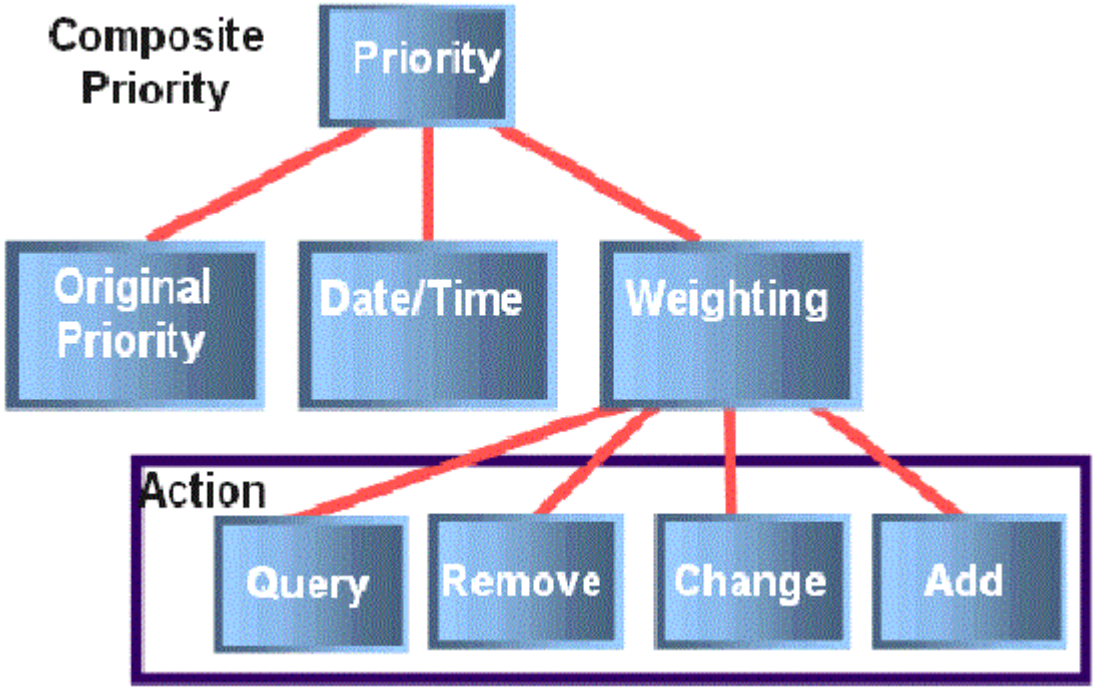
Rollback must be turned off for delayed failure to work.

About Composite Priorities

The composite priority mechanism ensures a balance between maximizing throughput and the need to provision higher priority atomic actions over those with lower priority. This mechanism does not guarantee the explicit sequential execution of work orders. Rather, it is designed to ensure that high priority orders are not impeded by lower priority orders that are in progress at the same time. ASAP will use any available processing power to activate orders, and does so by activating many orders in parallel across many network devices.

After orders are placed in the in-progress queue, each atomic action on the order inherits the work order properties including the due date and time, order priority and action. These attributes are used to determine where the atomic action should be placed in the pending queue but do not guarantee that it will be provisioned in advance of any other atomic action. The following diagram shows the importance of the attributes (from left to right) in the prioritization of the atomic action. Details of the algorithm are explained in the main flow of the use case.

Figure 5-2 Composite Priorities



ASAP maintains one pending queue for each NE. Many orders are processed at the same time but only a single atomic action is active for each order at any given time due to the serial nature of atomic action processing within an order. In other words, if there are 100 orders in progress, there are 100 active atomic actions. While ASAP is processing a high priority atomic action for one work order, atomic actions from lower priority orders will also be processed against different NEs. ASAP retrieves future-dated orders from the database based on their due date/time, and subjects these orders to composite prioritization at the atomic action level. When a work order is submitted to ASAP, it is subject to BATCH_SLEEP_INTERVAL, which is the time period between SARM database queries for orders that have become due.

Composite priorities operate as follows:

1. A work order is submitted into ASAP with the following attributes:
   a. Due date and time
   b. Priority
   c. Action (query, remove, change or add)
2. When the order arrives at its due date and time and BATCH_SLEEP_INTERVAL expires, its first atomic action, referred to as the active atomic action for the purposes of this example, is inserted into the pending queue according to the following algorithm:
   a. Search through the pending queue comparing the priority of the active atomic action (as inherited from the work order) to those already in the queue. If there are no atomic actions with identical priorities insert the atomic action into the queue according to its priority (in other words, an active atomic action with a priority of 4 is inserted behind an atomic action with a priority of 3 but ahead of an atomic action with a priority of 6 – the lower the number, the higher the priority and hence the closer to the front of the pending queue) and proceed to step 3.

b. For the subset of atomic actions in the pending queue whose priorities match the active atomic actions priority, ASAP examines the due dates and times of each and inserts the active atomic action into the queue according to its due date and time. In other words, the active atomic action is inserted behind atomic actions with older due dates and times but ahead of atomic actions with newer due dates and times. atomic actions with older due dates and times are closer to the front of the pending queue. Go to step 3.

c. For the subset of atomic actions in the pending queue whose priorities and due dates and times match the priority and due date and time of the active atomic action, insert the atomic action into the pending queue according to its action. An active atomic action with an action of “Query" is inserted ahead of atomic actions with other actions. The priority of the action from highest to lowest is Query, Remove, Change, Add.

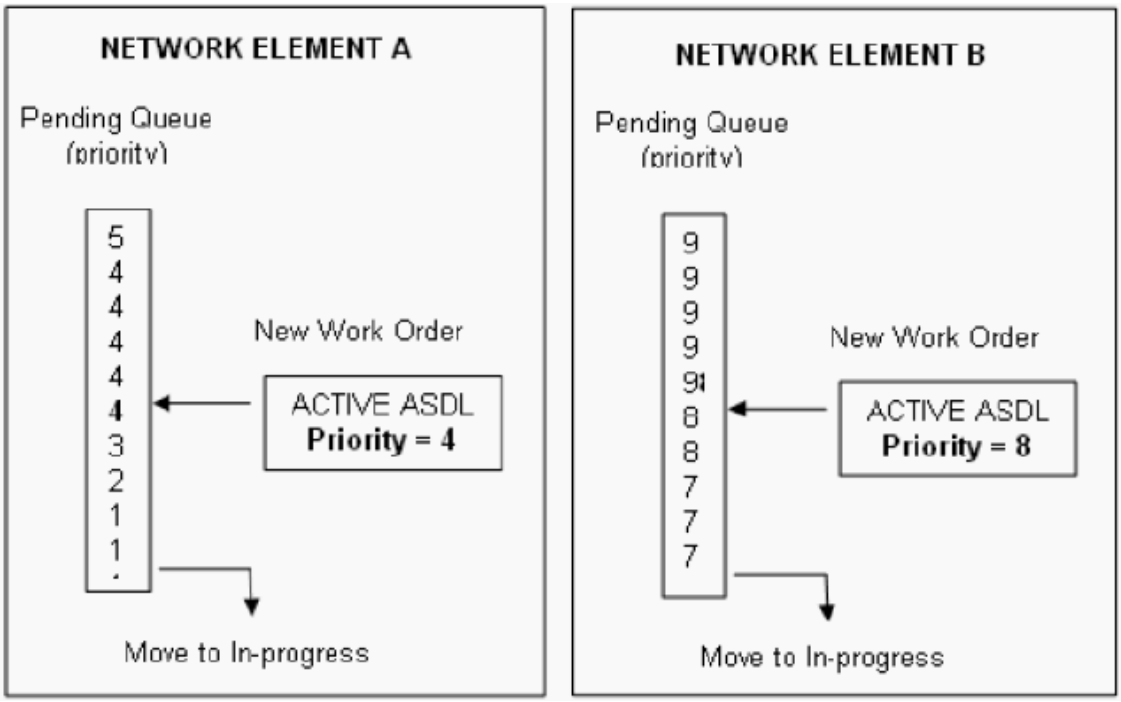
3. Eventually the atomic action is moved to the in-progress queue where it provisions and completes. While the SARM is being notified that the atomic action has completed an idle connection is detected and another atomic action may be scheduled.

   If the active atomic action is placed in the retry queue, the retry timer starts. During the time the active atomic action remains in the retry queue other atomic actions may be scheduled.

   When the retry time interval expires and the atomic action is placed back in the pending queue, step 2 is repeated.

Figure 5-3 shows multiple pending queues (one for NE A and one for NE B). NE A has many high priority atomic actions (for example: priority 1, 2) in its pending queue while NE B has many lower priority atomic actions (for example priority 7, 8, 9) in its pending queue. Because there only low priority atomic actions in NE Bs pending queue, these will be provisioned at the same time as the high priority atomic actions on NE As pending queue.

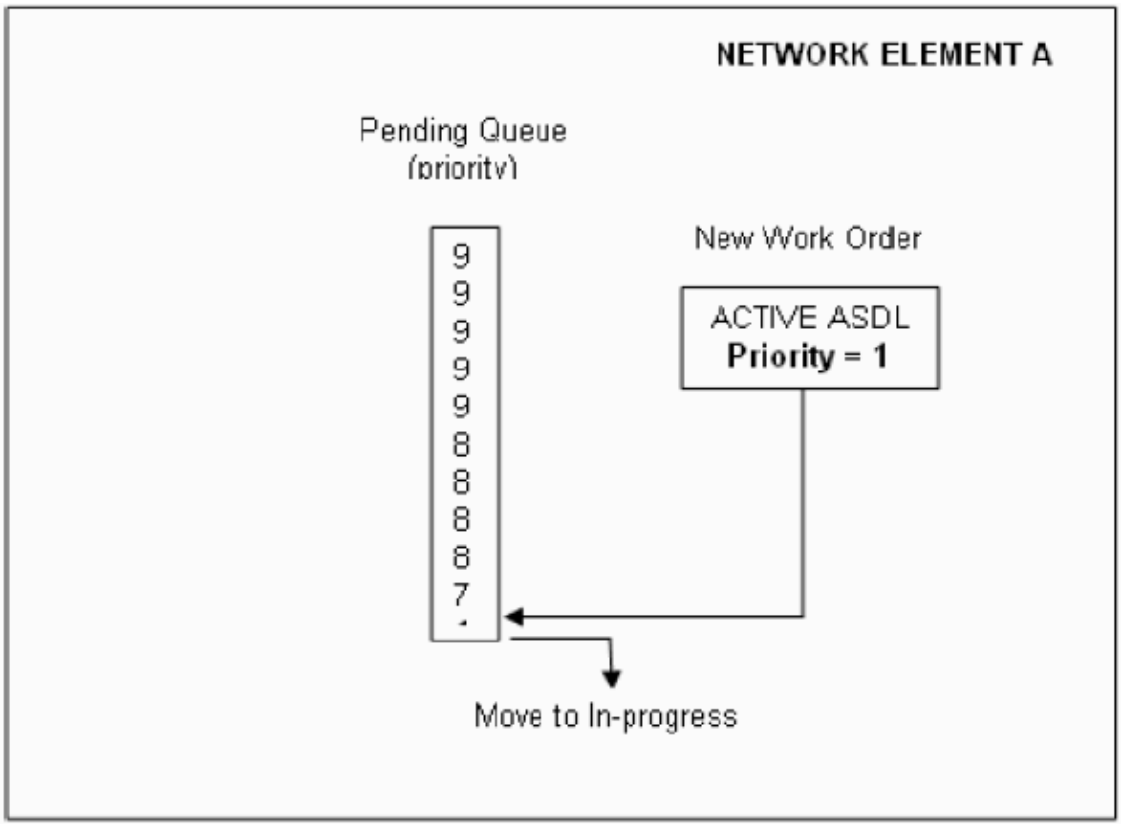
Figure 5-3 Pending Queues



The following diagram shows a single queue containing low priority atomic actions when a high priority atomic action arrives. The high priority atomic action is inserted ahead of all lower

priority atomic actions in the pending queue and as a result will be placed in-progress before any of the others. When the high priority atomic action has completed, the SARM must be notified and an idle connection will be detected. At this time another atomic action (possibly of greater, equal or lower priority) may be scheduled (for example: in this example the atomic action with priority 7).

Figure 5-4 Pending Queues



About Configuring a Rollback Atomic Action

You can configure atomic actions in the system to perform rollback on a failed provisioning activity by setting its rollback flag and specifying a rollback atomic action. For example, if you have an atomic action for creating a service, you can select a rollback atomic action from deleting a service. See "Creating and Configuring an Atomic Action" for instructions about enabling the rollback feature and assigning a rollback atomic action to a standard atomic action. You must also enable the rollback functionality at the service action level to enable atomic action rollback. To enable rollback at the service action level, see "Enabling the CSDL Rollback Functionality."

> **Note:**
>
> The SARM will only roll back atomic actions that you have configured with these settings.

Atomic actions can perform the following types of rollback:

Table 5-3 Atomic Action Rollback Types

Rollback	Description
Provisioning Rollback	Used when a work order fails while provisioning.
Cancellation Rollback	Used when a cancellation request is applied to an existing order in the SARM.
Correction Rollback	Used when a correction request is applied to an existing order in the SARM.

About Rollback Atomic Action Parameters

The parameters that are sent to the rollback atomic action are automatically pre-determined, consequently, you do not need to define or configure the atomic action parameters for a rollback atomic action in **tbl_asdl_parm**. Rollback parameters are created using the **SEND_PARAM** action function with an option of R (or ReturnRollbackParam) in the JInterpreter.

If a rollback parameter is created for an atomic action using **SEND_PARAM**, the value of this parameter remains the same for the rollback atomic action. For example, if a rollback parameter is created in another atomic action using the same name as the initial rollback atomic action parameter, the value of this new rollback parameter will not overwrite the value provided to the initial rollback atomic action. If you send a rollback atomic action parameter that has the same name as the forward atomic action, the rollback atomic action parameter takes precedence. When the rollback atomic action is run, it receives the value of the rollback atomic action parameter.

The rollback parameters created by a particular atomic action are provided exclusively to its rollback atomic action, and are not shared with other atomic actions. You cannot use rollback parameters to share information between rollback atomic actions.

About Atomic Action Rollback Functionality

The following sections describe additional considerations for rollback functionality.

Rollback Order

Atomic actions are rolled back in reverse order of completion. When the rollback process begins, the last completed atomic action is rolled back first, followed by the second-to-last completed atomic action, etc.

Rollback Failure

The rollback of an atomic action can either complete or fail. During rollback processing, the status of every rollback atomic action is recorded as either **Completed** or **Failed**.

If the configuration variable is set to **0**, the service action status will be set to "rollback successful" even if one or more rollback atomic actions fail to complete. The failure of a rollback atomic action is ignored and the rollback of previous atomic actions continues.

If the configuration variable is set to **1**, the service action status will be set to "rollback failed" if a rollback atomic action fails for any reason.

Order Timeout

The order timeout parameter is ignored on rollback.

Rollback Completion

Rollback processing ends when the final rollback atomic action has either completed or failed. If the rollback was initiated as a result of a cancellation, a work order Completion Notification is sent to the SRP. In all other cases, the SRP receives a work order Failure Notification.

Rollback Upon Failure

When a work order fails, the SARM performs the following rollback steps:

1. As the SARM loads a work order for provisioning, it scans all of the service actions in the work order to determine if one or more has been configured for rollback in the event of failure.
2. If none of the service actions have been configured for rollback in the event of failure, rollback is not performed if the work order fails.
3. If rollback has been configured on one or more service actions *and* the work order property specifies rollback, the SARM sets a global flag on the work order to indicate that rollback is required.
4. If the work order fails, the SARM notifies the SRP that rollback is to be performed and starts the procedure.
5. When rollback is complete, the SARM sends an Order Failure notification to the SRP.

> **Note:**
>
> During normal provisioning, when atomic action failure occurs, the SARM immediately fails the work order and rolls back all successfully completed atomic actions.

Rollback Upon Cancellation of an Order

When processing a work order cancellation, the SARM does not reference the service action rollback configuration, but invokes rollback at the atomic action level.

> **Note:**
>
> The ASAP work order cancellation functionality is intended to provide the ability to cancel a work order in the short period of time between the submission of an order to ASAP and the reception of an event indicating the order is in a final state (such as completed, failed). Oracle recommends that orders are not canceled outside this window as this can lead to additional un-needed performance overhead and fallout risk in ASAP. For example, terminating the service of a subscriber that has been successfully created means rolling back all of the original atomic actions rather than simply deleting the subscriber (a single atomic action). In addition, because data in ASAP should be maintained only for a limited period of time (see data purging and archival strategies section), use of cancellation functionality is subject to purging constraints.

The SARM performs the following rollback steps when a work order is cancelled:

1. When the SARM receives the cancellation request, it halts the work order when the current atomic action completes.
2. Before the rollback operation begins, the SARM notifies the SRP that the work order rollback is to be performed.
3. The SARM references the atomic action log to determine which atomic actions have been completed on the order.
4. The SARM rolls back completed atomic actions for which rollback is configured and rollback atomic actions are defined.
5. Upon completion, the SARM sends the SRP a Completion Notification.

Depending on the status of the work order when it is cancelled, a different rollback procedure is performed. The different work order status values and their corresponding rollback procedures are described in Table 5-4:

Table 5-4 Cancellation Order Status Rollback Procedures

Order Status	Rollback Procedure
Initial order	The order is cancelled and no provisioning is needed or occurs.
In Progress order	The SARM accepts the cancellation request and begins to roll back the order when the current atomic action on the work order completes. It reloads all completed atomic actions from the database, determines which ones require rollback by referencing their rollback flags, and then runs the rollback atomic actions. When the rollback procedure is complete, the SARM transmits a work order Completion Notification to the SRP. No reference is made to the work order rollback flag or to the rollback status of the service actions.
Completed order	The rollback procedure is identical to the procedure used for In-Progress orders, except there is no delay at the start, such as waiting for the last atomic action to complete before starting to roll back the order.
Failed order	The rollback procedure is identical to the procedure used for Completed orders.

Rollback Upon Revision to an Order

For the failed order to be rolled back explicitly, one or more service actions must be configured for rollback. The SARM automatically rolls back the work order before receiving and processing a new copy of it.

The rollback procedure for a revision or correction request depends on the state of the work order as described in Table 5-5:

Table 5-5 Revision Order Status Rollback Procedures

Order Status	Rollback Procedure
Initial order	The order is overwritten and no rollback occurs.
In Progress order	The SARM rejects the request for an order revision or correction from the SRP and no rollback occurs.
Completed order	The SARM rejects the order revision or correction request and no rollback occurs.
Failed order	If all service actions on the work order are terminated, then no explicit rollback is performed.

If explicit rollback is not performed upon receipt of an order failure and revision request, the activation of a new copy of the work order may cause a fallout at the NE because parts of the original work order may have been activated. After concluding that a provisioning request has failed, the queries switch and determines if the provisioning activity represented by this command has already been applied to the NE. If so, ASAP issues a soft error and continues processing the new order.

> **Note:**
>
> If rollback is not performed explicitly, the SRP can be designed to transmit a cancellation request on the original order, and then send a correction order that is dependent on the cancellation. In this way, the failed order is rolled back and the revision is only applied when the cancellation is complete.

Table 5-6 shows database variables and tables that you must configure to implement rollback of completed atomic actions:

Table 5-6 Rollback of Completed Atomic Actions Parameters

Variable	In Table
ignore_rollback	**tbl_asdl_config**
rollback_req	**tbl_csdl_config**

For information about these database tables, refer to *ASAP Developer's Guide*.

Configuring ignore_rollback

This configuration variable is located in the **tbl_asdl_config** table in the SARM database:

- If it is set to **Y**, rollback is ignored for the specified atomic action even if the rollback flag on the work order is set to **Y**.
- If it is set to **N**, rollback is required for the specified atomic action.

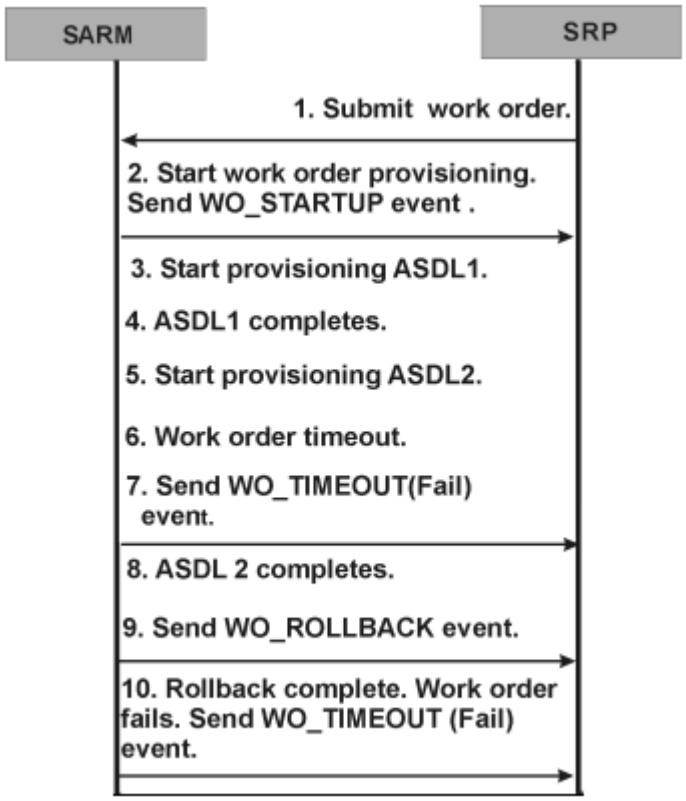
> **Note:**
>
> If employing the **delayed_failure** property (see "About Delayed Failure Properties"), rollback must be turned off. Service action-level rollback must be set to **N**, and ignore rollback must be set to Y.

The following example employs a configuration that requires:

- Setting the work order **wo_timeout** parameter to the required value.
- Configuring the rollback parameters in **tbl_asdl_config**.

In this example, the work order has one service action with three atomic actions. The expected result is that the work order fails after exceeding the time specified in the wo_timeout parameter on the work order and all completed atomic actions are rolled back.

Figure 5-5 Rollback Sequence of Operations



1. SRP submits work order to the SARM for provisioning.
2. The SARM starts provisioning the work order and sets the timer for work order timeout based on the timeout value. The SARM sends a **WO_STARTUP** event notification to the SRP.

3. The SARM starts provisioning the first atomic action in the work order.
4. The first atomic action is successfully provisioned.
5. The SARM starts provisioning the second atomic action in the work order.
6. While provisioning the second atomic action, a work order timeout occurs.
7. The SARM sends a WO_TIMEOUT (Fail) event notification to the SRP. The SARM resets the timer to zero and waits until the second atomic action completes.
8. When the second atomic action completes (with a Success or Fail status) all successfully completed atomic actions are rolled back.
9. The SARM sends a WO_ROLLBACK event notification to the SRP.
10. Rollback completes and the work order is failed. The SARM sends a WO_TIMEOUT (Fail) event notification to the SRP. The SARM may also send a WO_FAILURE event notification to the SRP.

6
Configuring Static Routing

This chapter describes how to configure static network element (NE) routing for Oracle Communications ASAP.

Configuring Static Network Element Routing

To increase the coverage of a host NE, several remote NEs covering a given area can be associated with a host NE. Service requests in the form of atomic actions are routed through the host NE to the appropriate remote NE.

ASAP determines the host NE using one of the following routing mechanisms:

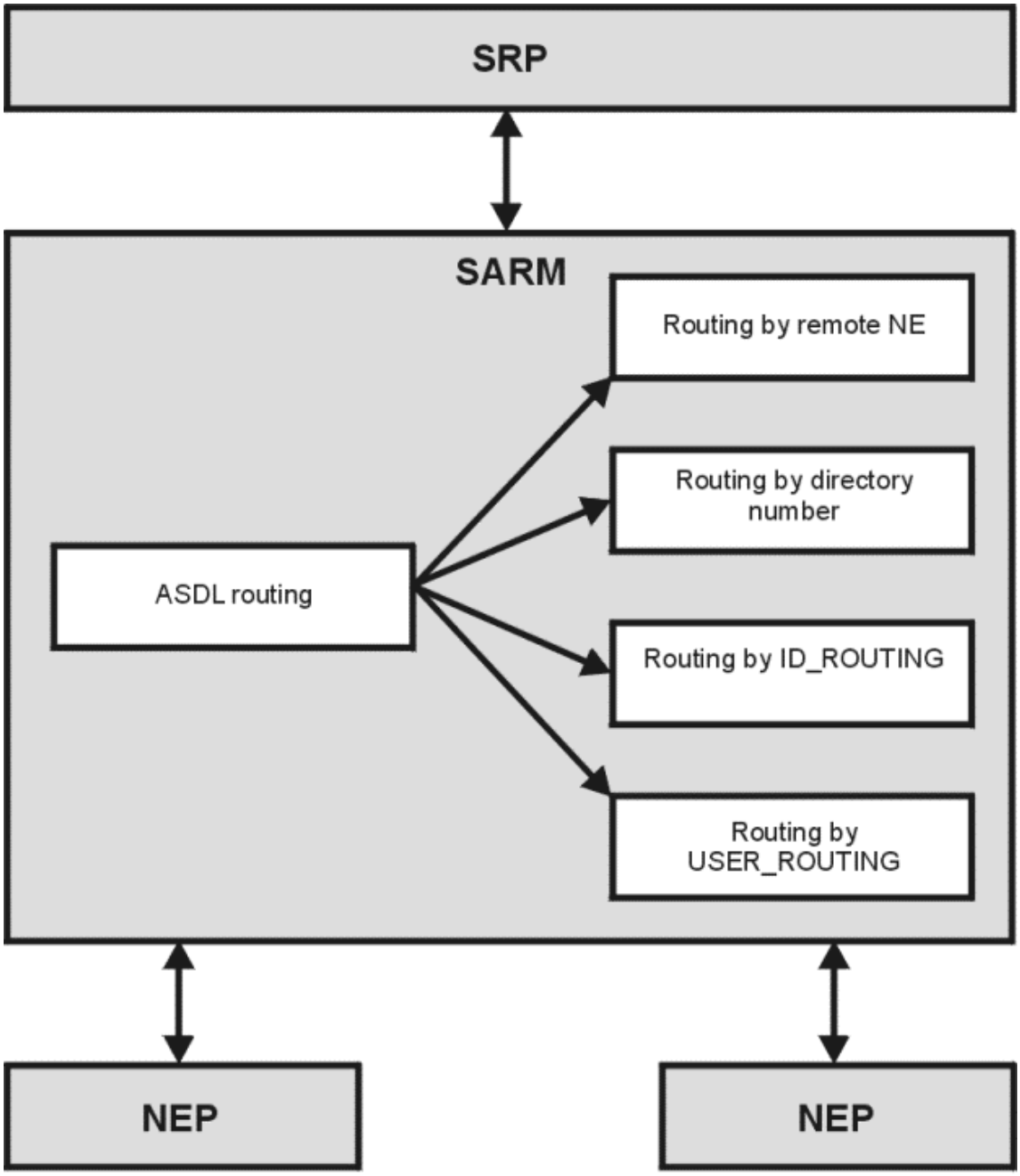
- **Dynamic Routing** – See "Dynamic Network Element Routing Scenarios."
- **Remote Network** – The NE identifier determines the communication parameters for the NE on which the service is to be provisioned. See "Configuring Atomic Action Routings by Using a Network Element."
- **Atomic Action Parameter** – ID_ROUTING information specified as an atomic action parameter. See "Configuring Atomic Action Routings by Using ID_ROUTING."
- **User Routing** – User-defined stored procedure that uses information from the USER_ROUTING atomic action parameter and/or the atomic action. See "Configuring Atomic Action Routings by Using USER_ROUTING."
- **Directory Number** – The host NE is not identified but is determined by the directory number specified on the work order. See "Configuring Atomic Action Routings by Using a Directory Number."

The routing logic has embedded priorities, which can affect the routing option you choose when multiple parameters are defined for work order information. Priorities between the routing logic are as follows:

- Routing by remote NE.
- Routing by ID_ROUTING.
- Routing by user-defined procedure.
- Routing by DN.

Figure 6-1 shows a system view of the atomic action routing logic in ASAP.

Figure 6-1 Atomic Action Routing Logic



Configuring Atomic Action Routings by Using a Network Element

The service action commands the Service Activation Request Manager (SARM) receives from the Service Request Processor (SRP) contain an NE identifier. This NE identifier is a reference to the communication parameters for the NE that ASAP should connect to. Based on these communication parameters, ASAP determines the host NE upon which the atomic action is to be provisioned.

The mandatory MCLI parameter that must be configured as an atomic action parameter when using static routing by NE ID must include the NE technology token as part its corresponding **asdl_lbl**. The convention is shown as follows:

`NE_ID_technology`

This is the same token that is used to populate the technology field when defining new NE instances to ASAP and in the naming convention for service action and atomic actions. Examples are shown in table 6-1:

Table 6-1 MCLI to NE_ID Technology Parameter Mapping

asdl_lbl (Atomic Action Label)	csdl_lbl (Service Action label)
MCLI	NE_ID_GWC
MCLI	NE_ID_HLR

An atomic action that queries an NE must be configured with a parameter (see **tbl_asdl_parm**) called **RET_PARM_TYPE** that has a default of IC indicating that both information parameters and service action parameters are to be returned from the implementation method. During the implementation of the associated method, these parameter combinations will be supported and the appropriate parameters and types shall be passed back to the SARM. Other possible values that the default may be changed to include:

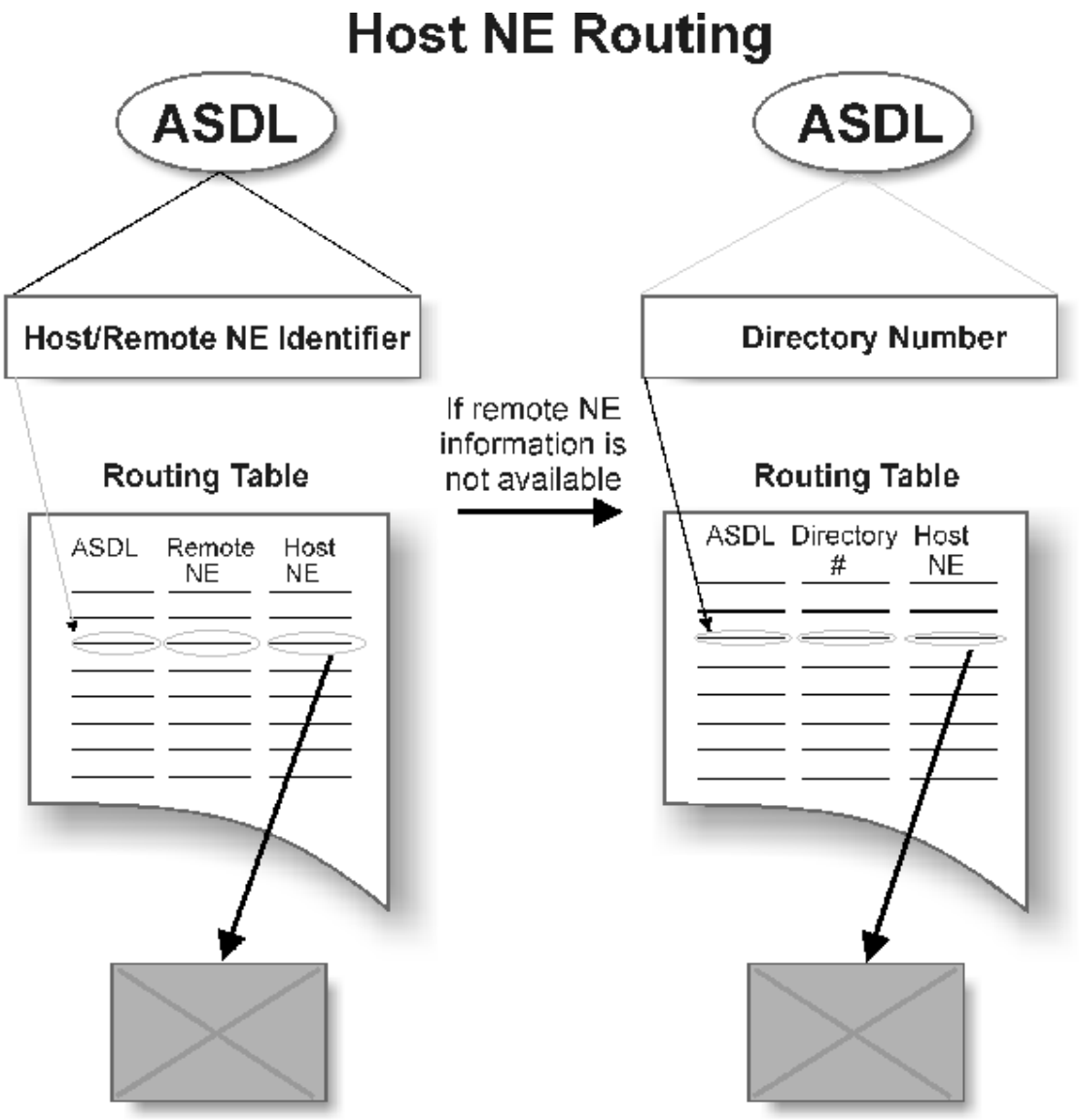
- **C**: service action parameters
- **W**: work order parameters
- **I**: information parameters
- **IC**: information parameters and service action parameters
- **IW**: information parameters and WO parameters

> **Note:**
>
> If a value is not provided for the RET_PARM_TYPE parameter or if it is left out of the atomic action parameter list, no parameters are returned from the query.

An atomic action may or may not be able to identify the host or remote NE to which it is to be routed. If the service action command received by the SARM contains a remote NE identifier, routing is achieved through a user-populated routing table in the SARM.

If the remote NE is not identified in the service action command, the host NE is determined by the directory number specified in the atomic action. The directory number consists of the NPA, NXX, and Line. Figure 6-2 illustrates this routing.

Figure 6-2 Routing by Host/Remote NE Identifier



Configuring Atomic Action Routings by Using ID_ROUTING

For flexible routing between atomic actions and NEs, you can use the ID_ROUTING atomic action parameter, and the **tbl_id_routing** database table. **tbl_id_routing** is a static database table that enables you to map between ID_ROUTING and the NE. Based on the information in the table, the ID_ROUTING is mapped to the host NE, which is loaded into memory when the SARM starts.

The **ID_ROUTING** parameter can be represented as any string of numbers and or characters to a maximum of 255 characters (or, in the case of an IP address, four sets of 255 characters – 255.255.255.255). You can define the parameter as part of a work order or a service action. **ID_ROUTING** can be a phone number, customer number, IP address, or any other identifier you choose.

If the work order provides **ID_ROUTING** information, such as phone number or customer number, you can get the host NE associated with the **ID_ROUTING** using the mapping table. The mapping table will provide the following matches:

- Exact matching
- Range matching

This allows precise matches or ranges within which the supplied parameters fall, so that one or multiple atomic actions can be routed to an NE at a time, based on configuration.

The **ID_ROUTING**/atomic action host common language location identifier code (CLLI) mapping table is binary-searched to get the associated host NE, associated with the atomic action to be provisioned and the **ID_ROUTING**. In the case of characters, the ASCII order is compared; in the case of numbers, the size of the number is compared.

The following mapping example displays how the characters can be compared.

```
"1" !="+1", "01"=="1","1"<"A", "A"<"a",  "A"<"AA"
```

> **Note:**
>
> Refer to *ASAP Developer's Guide* for information on using "= =" operators with IP addresses.

The stored procedures that you can use as external interfaces are the following:

- **SSP_list_id_routing** (RC1, **host_clli**) – Lists the host NE and **ID_ROUTING** mapping records in the SARM database.
- **SSP_new_id_routing** (**host_clli**, **asdl_cmd**, **id_routing_from**, **id_routing_to**) – Defines the host NE and **ID_ROUTING** mapping records in the SARM database.
- **SSP_del_id_routing** (**host_clli**, **asdl_cmd**, **id_routing_from**, **id_routing_to**) – Deletes the host NE and **ID_ROUTING** mapping records from the SARM database.

For more information on these stored procedures, refer to the *ASAP Developer's Guide*.

Routing by ID_ROUTING

The following steps must be followed when routing by **ID_ROUTING**:

- Populating the routing table (**tbl_id_routing**).
- Defining the atomic action parameter. A sample is located in *ASAP_Home***samples\ASDL_ROUTE\oraRoutingServices**.
- Defining the work order. A sample is located in *ASAP_Home***samples\ASDL_ROUTE\RoutingSrpInput**.
- Starting ASAP and submitting the work order.

The following example displays how to populate tbl_id_routing.

```
sqlplus -s $SARM_USER/$(GetPassword $SARM_USER 2)
<<HERE | grep -v "successfully completed"

set serveroutput on
var retval number

prompt Defining the ID_ROUTING Configurations
```

```
exec :retval := SSP_del_id_routing ;

exec :retval := SSP_new_id_routing ('BALTIMORE', '', 'BAL', 'CAL');
exec :retval := SSP_new_id_routing ('BALTIMORE', '', 'DEL', 'FAL);
exec :retval := SSP_new_id_routing ('BOSTON', '', '120000', '220000');

HERE
```

You can add new records to the database dynamically without downtime on the server by using the **Add new NE Configuration** command (**113**) of **asap_utils**. This command must be used after loading the ASAP database.

For more information about **asap_utils**, see *ASAP Server Configuration Guide*.

For more information about the **tbl_id_routing** table, see the *ASAP Developer's Guide*.

Configuring Atomic Action Routings by Using USER_ROUTING

You can perform atomic action routing by using a user-defined procedure. Routing by user-defined procedure provides the following:

- Allows for custom provided logic for atomic action routing
- Uses the atomic action parameter **USER_ROUTING**
- Uses the external interface **SSP_get_user_routing**
- Allows you to write your own routing logic using the predefined external user interface

The **USER_ROUTING** parameter can be represented as any string of characters to a maximum of 255 characters. You can define it as part of a work order, or as a service action parameter.

If the atomic action parameter **USER_ROUTING** information is provided in the work order, then the user-defined stored procedure is called. The user-defined procedure takes the asdl_cmd and the value of **USER_ROUTING** as input arguments, and returns the host NE to be routed.

You can use the following stored procedure as an external interface:

- **SSP_get_user_routing** (**user_routing**, **asdl_cmd**, **host_clli**, **ret_val**) – Returns a host NE (**host_clli**) that is used to route the atomic action. You must provide your own routing logic in the body of **SSP_get_user_routing** to find the host NE (CLLI) using the **USER_ROUTING** atomic action parameters, and the **asdl_cmd** if required.

For more information on the above stored procedure, refer to the *ASAP Developer's Guide.*

To use **USER_ROUTING**, perform the following steps:

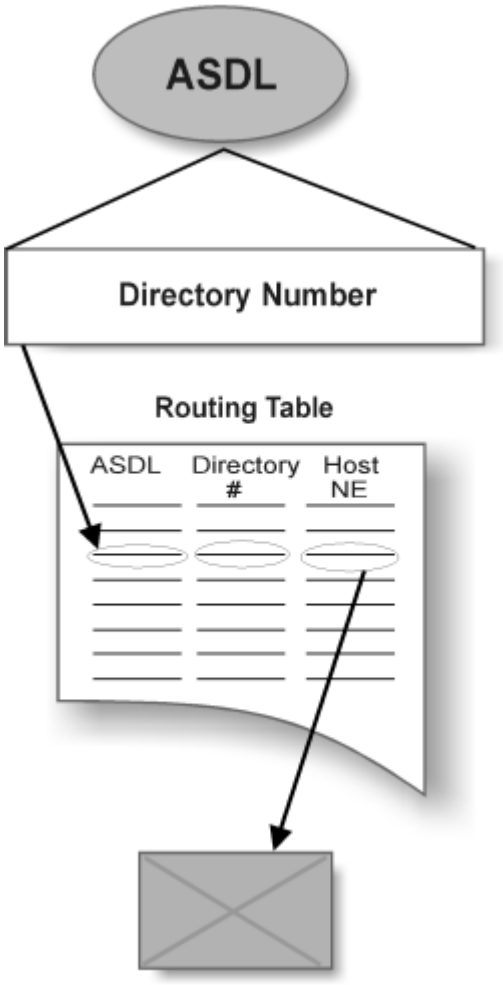
1. Write the stored procedure **SSP_USER_ROUTING**. A sample is located in *ASAP_Home***samples\ASDL_ROUTE\user_routing_proc.sp**.
2. Define and populate the routing table, if required. A sample is located in *ASAP_Home***samples\ASDL_ROUTE\user_routing_table.tbl** and *ASAP_Home***samples\ASDL_ROUTE\oraLoadRouting**.
3. Define the atomic action parameter. A sample is located in *ASAP_Home***samples\ASDL_ROUTE\oraRoutingServices**.
4. Define the work order. A sample is located in *ASAP_Home***samples\ASDL_ROUTE\RoutingSrpInput**.
5. Run ASAP and submit the work order.

When you choose a user-defined procedure with a database table, the database must be accessed every time the routing is requested. Consequently, there will be a slight performance degradation.

Configuring Atomic Action Routings by Using a Directory Number

Atomic actions are routed through a directory number (DN) identifier. The DN is identified on the work order and is passed to an atomic action as a parameter. Figure 6-3 illustrates atomic action routing by DN.

Figure 6-3 Routing by DN



Depending on your telecommunications situation, you may require routing based on different parts of the phone number/atomic action combination; for example, NXX and the first two digits of the line number.

An atomic action instance will be routed according to the DN/atomic action routing table if the atomic action parameter MCLI is set to SKIPCLLI on the work order. This value implies that the remote NE information is not known.

Before you can add a new routing, you must have already defined both the host NE and the atomic action in ASAP.

You can edit routing definitions provided the new routing definition does not already exist in ASAP.

- **SSP_new_dn_map** – This stored procedure defines atomic action routings by directory number.
- **SSP_list_dn_map** – This stored procedure lists directory mappings for atomic action, directory, exchange number, or for all of them.
- **SSP_del_dn_map** – This stored procedure deletes a directory number mapping from the SARM database.

7

Configuring Dynamic Routing

This chapter describes how to configure dynamic routing for Oracle Communications ASAP atomic actions.

Configuring Dynamic Network Element Routing

The Dynamic Network Element (NE) Routing feature allows ASAP to provision NEs based on network and communication data provided as order parameters rather than loaded from static Service Activation Request Manager (SARM) configuration tables. ASAP routes the translated Atomic Service Description Layer (atomic action) commands to the appropriate NEs based on specific routing information contained in the work order. This dynamically provided communication data is identical to the communication data normally defined in static tables and used by the devices [command processor threads in the Network Element Processor (NEP) to connect and log in to.

For information on configuring static routing, see " Configuring Static Network Element Routing."

Dynamic NE routing is most commonly used for IP-based provisioning, but is applicable to all downstream communication protocols. For example, it is possible to dynamically route provisioning tasks that use serial dialup connections in the downstream.

Dynamic routing functions as follows:

1. The SARM receives a work order.
2. The SARM uses the NE_ID (mapped to the atomic action label **MCLI**) parameter to determine if the order is to be routed statically or dynamically. The NE_ID, or any other Common Service Description Layer (service action) label defined for the parameter, identifies either an NE resource or a dynamic routing template resource configured in ASAP. Examples for mapping the atomic action label **MCLI** to different service action labels are provided later in this chapter.
3. If the **NE_ID** identifies a network template, the SARM uses this to dynamically set up a session manager, connection pool, and command processors.
4. The SARM uses the drop timeout (discussed later in this chapter) to terminate connections to the NE.
5. After the primary connection is dropped, the command processor, connection pool, and session manager are cleaned up.

Enabling Dynamic Routing

This section describes how to configure ASAP to enable dynamic routing.

Network Template Configuration

A network template describes an ASAP connection environment that consists of an NEP, primary connection pool and its attributes (spawn threshold, kill threshold, max connections,

and drop-timeout). ASAP uses the template to set up a connection pool and session manager for each dynamically identified NE.

In conventional static configurations, the work order identifies a real NE to be provisioned via the reserved atomic action parameter named **MCLI**. In dynamic routing, this parameter identifies a template for dynamic routing.

A static routing definition contains a parameter that references **MCLI** as follows:

```
<parameter name="MCLI" xsi:type="SimpleParameterType">
<required>true</required>
<parameterValueMap>NE_ID</parameterValueMap>
</parameter>
```

In this situation, a work order can identify the target NE by defining a parameter called **NE_ID** and assigning a value that references a statically configured NE resource in ASAP.

A dynamic routing configuration appears as follows:

```
<parameter name="MCLI" xsi:type="SimpleParameterType">
<required>true</required>
<parameterValueMap>TEMPLATE_ID</parameterValueMap>
</parameter>
```

In this situation, a work order can identify a network template by defining a parameter called **TEMPLATE_ID** and assigning a value that references a network template resource in ASAP.

Table 7-1 Comparing Static and Dynamic Configuration

#	Static	Dynamic
Atomic action reserved work order parameter **MCLI**	Identifies NE	Identifies network template
tbl_clli_route, **tbl_host_clli**, **tbl_nep**, **tbl_ne_config**, **tbl_resource_pool**, **tbl_comm_param**	Specifies connection environment for a specific NE	Specifies a dynamic routing template

You can use the Service Activation Configuration Tool (SACT) and **ActivationConfig.xsd** schema to create and deploy network templates. When deployed, the network template is identified in **tbl_ne_config**. For more information on the SACT, see *ASAP Server Configuration Guide*. You can alternatively use Design Studio to create and deploy network templates. For more information about using Design Studio to configure dynamic routing, see the Design Studio *Modeling Activation* Help.

If you are going to use SACT, you can write the XML configuration file from scratch, use an XML editor to generate the XML code, or use a sample from the *ASAP_home*/**samples/sadt/SampleCommonConfig** directory as the basis for the XML configuration file, where *ASAP_home* is the directory in which ASAP is installed.

A sample XML configuration file for dynamic routing appears below:

```
<dynamicRoutingTemplate name="DYN_DALLAS">
<nepServerName>NEP_S235</nepServerName>
<vendor>DYNAMIC_VENDOR</vendor>
<technology>DYNAMIC_TECH</technology>
<softwareLoad>DYNAMIC_SL</softwareLoad>
<maximumConnections>5</maximumConnections>
<dropTimeout>1</dropTimeout>
<spawnThreshold>3</spawnThreshold>
<killThreshold>0</killThreshold>
```

```
<read_timout>5</read_timeout>
<write_timeout>5</write_timeout>
<lineType>TELNET_CONNECTION</lineType>
<communicationParameter>
<label>DynLab1</label>
<value>
<value>DynVal1</value>
</value>
<description>DynDesc1</description>
</communicationParameter>
<label>LOGIN_PROMPT</label>
<value defaultValue="login:">
<value>login:</value>
<description>Login prompt.</description>
</communicationParameter>
<communicationParameter>
<label>READ_TIMEOUT</label>
<value defaultValue=5>
<value>5</value>
<description>Integer</description>
</communicationParameter>
<communicationParameter>
<label>WRITE_TIMEOUT</label>
<value defaultValue=5>
<value>5</value>
<description>Integer</description>
</communicationParameter>
</dynamicRoutingTemplate>
```

The template name (**DYN_DALLAS**) specified in the **dynamicRoutingTemplate** tag identifies the template that must be specified in the work order.

Parameters provided on the work order override statically defined values in the template.

For specific tag definitions, refer to the comments in **activationConfig.xsd**.

After you have written the dynamic routing XML configuration file, you can use a command-line tool to configure it into ASAP. For more information, see the *ASAP Server Configuration Guide*.

Dynamic Network Element Routing Scenarios

This section describes different routing scenarios and the configurations required to support them. Dynamic routing requires that communication parameters used in creating a connection must be passed down as order parameters.

Dynamic routing is supported by any protocol including TCP/IP. Consequently, ASAP cannot mandate keyword parameters to specify a target NE's communication parameters. For TCP/IP-based protocols, an IP address and port are usually sufficient parameters to specify a connection. Other protocols require different communication parameters: HTTP may include a URL, and Common Object Request Broker Architecture (CORBA) may use an Interoperable Object Reference (IOR) string. There is no limit to the set of communication parameters that can be used to uniquely identify a target NE.

Network Element Identification

NE identification is provided by means of the reserved compound parameter **COMM_PARM** and its reserved member **COMM_PARM.NE_ID**. Only work order parameters mapped to the key compound parameter of **COMM_PARM** are identified as dynamic communication parameters. The subset of communication parameters identified by **COMM_PARM.NE_ID** is used by ASAP to uniquely identify a specific NE.

For example, you could identify the communication parameters for an NE instance using TCP/IP connections with one or more of the following:

- **COMM_PARM.NE_ID.HOST_IPADDR**
- **COMM_PARM.NE_ID.PORT**
- **COMM_PARM.NE_ID.HOST_USERNAME**
- **COMM_PARM.NE_ID.HOST_PASSWORD**

For CORBA devices, the communication parameter may appear as follows:

- **COMM_PARM.NE_ID.IOR**
- **COMM_PARM.NE_ID.USERNAME**
- **COMM_PARM.NE_ID.PASSWORD**

Each parameter is used to create a key that uniquely identifies that NE. Based on the key, the SARM initializes a session manager. For instance, if you wanted only the IP address and security credentials to uniquely identify an NE, you would specify port as **COMM_PARM.PORT** (without the **NE_ID**) so that it does not come into play when identifying a target NE.

Another provisioning request with the same set of communication parameters but with a different user name and password identifies a different NE to ASAP. ASAP would create two sets of resources for each NE: connection pool, session manager, command processor, devices, and so on.

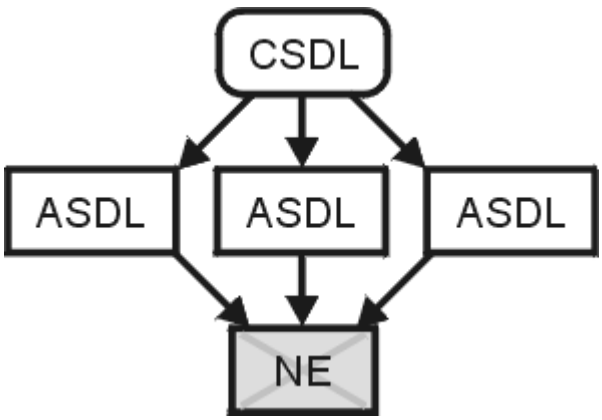
The following sections describe different routing scenarios.

Scenario 1 – One Service Action to Multiple Atomic Actions Routed to One NE

In this scenario, an upstream inventory system is used to maintain certain logical NE attributes including routing information. The set of NEs that will use dynamic routing have identical connection characteristics, consequently, they can share a single dynamic routing template. Work orders are submitted to ASAP with enough information to identify the template to be used for dynamic routing. All values configured in the template are then applied to establish the connection.

Figure 7-1 shows a single service action mapped to one or more atomic actions, all of which are routed to a single NE.

Figure 7-1 One Service Action to Multiple Atomic Actions Routed to One NE



The work order includes the following service action and communication parameters:

```
C-SERVICE
TEMPLATE_ID_A=TEMPLATE_1
COMM_PARM.NE_ID.URL=http://www.abc.com
COMM_PARM.NE_ID.HOST_USERNAME=jsmith
COMM_PARM.NE_ID.HOST_PASSWORD=<password1>
```

Table 7-2 shows the parameter mappings service model configuration.

Table 7-2 Scenario 1 Parameter Mappings

asdl_cmd	asdl_lbl	csdl_lbl	param_typ
A-SERVICE_1	COMM_PARM	COMM_PARM	Compound
A-SERVICE_1	MCLI	TEMPLATE_ID_A	Scalar
A-SERVICE_2	COMM_PARM	COMM_PARM	Compound
A-SERVICE_2	MCLI	TEMPLATE_ID_A	Scalar
A-SERVICE_3	COMM_PARM	COMM_PARM	Compound
A-SERVICE_3	MCLI	TEMPLATE_ID_A	Scalar

This configuration routes all three atomic actions to the same NE because they each receive the same set of communication parameters. Table 7-3 shows the downstream program (JInterpreter class) parameters:

Table 7-3 Scenario 1 Parameters

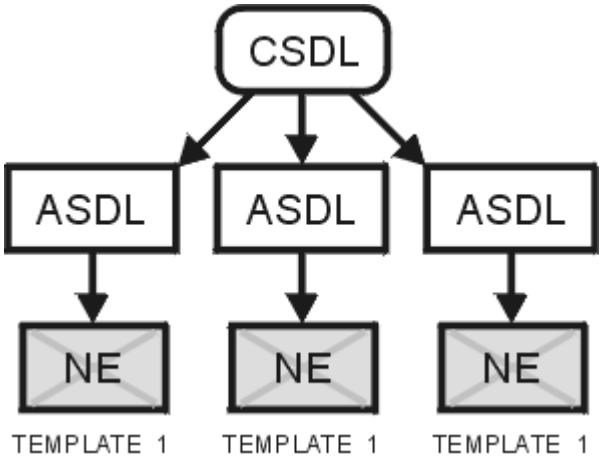
Label	Value
A-SERVICE_1	-
MCLI	TEMPLATE_1
URL	http://www.abc.com
username	jsmith
password	<password1>
A-SERVICE_2	-
MCLI	TEMPLATE_1
URL	http://www.abc.com
username	jsmith
password	<password1>
A-SERVICE_3	-
MCLI	TEMPLATE_1
URL	http://www.abc.com
username	jsmith
password	<password1>

The **COMM_PARM** and **COMM_PARM.NE_ID** are stripped from the work order parameter so that the downstream provisioning program receives the parameters in the name/value pair that is expected.

Scenario 2 – One Service Action to Multiple Atomic Actions Routed to Different NEs

Figure 7-2 presents a single service action mapped to two or more atomic actions, each of which is routed to a different NE but all using the same network template.

Figure 7-2 One Service Action to Multiple Atomic Actions Routed to Different NEs



The work order includes the following service action and communication parameters:

```
C-SERVICE
TEMPLATE_ID=TEMPLATE_1
SUBSCRPTION_A.NE_ID.URL=http://www.abc.com
SUBSCRIPTION_A.NE_ID.HOST_USERNAME=jsmith
SUBSCRIPTION_A.NE_ID.HOST_PASSWORD=<password1>
SUBSCRPTION_B.NE_ID.URL= http://www.def.com/
SUBSCRIPTION_B.NE_ID.HOST_USERNAME=dmiller
SUBSCRIPTION_B.NE_ID.HOST_PASSWORD=<password2>
SUBSCRPTION_C.NE_ID.URL=http://www.ghi.com
SUBSCRIPTION_C.NE_ID.HOST_USERNAME=djones
SUBSCRIPTION_C.NE_ID.HOST_PASSWORD=<password3>
```

Table 7-4 shows the service model configuration parameter mappings.

Table 7-4 Scenario 2 Parameter Mappings

asdl_cmd	asdl_lbl	csdl_lbl	param_typ
A-SERVICE_1	COMM_PARM	SUBSCRIPTION_A	Compound
A-SERVICE_1	MCLI	TEMPLATE_ID	Scalar
A-SERVICE_2	COMM_PARM	SUBSCRIPTION_B	Compound
A-SERVICE_2	MCLI	TEMPLATE_ID	Scalar
A-SERVICE_3	COMM_PARM	SUBSCRIPTION_C	Compound
A-SERVICE_3	MCLI	TEMPLATE_ID	Scalar

The atomic action parameter MCLI in this case identifies the network template. In static routing, the atomic action parameter MCLI identifies the NE.

This configuration routes each atomic action to a different NE. Table 7-5 shows the downstream program (JInterpreter class) parameters.

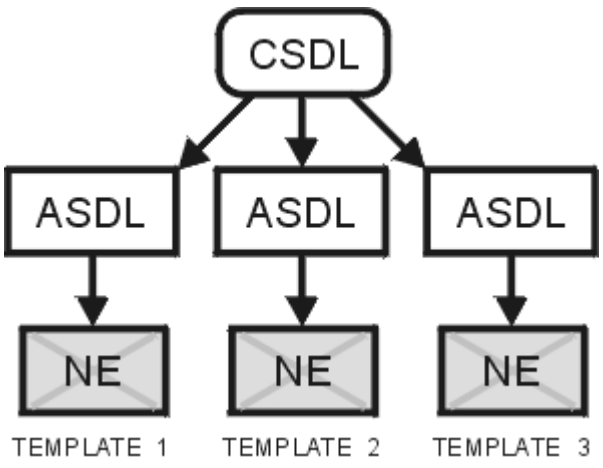
Table 7-5 Scenario 2 Parameters

Label	Value
A-SERVICE_1	-
MCLI	TEMPLATE_1
URL	http://www.abc.com
HOST_USERNAME	jsmith
HOST_PASSWORD	<password1>
A-SERVICE_2	-
MCLI	TEMPLATE_1
URL	http://www.def.com
HOST_USERNAME	jsmith
HOST_PASSWORD	<password2>
A-SERVICE_3	-
MCLI	TEMPLATE_1
URL	http://www.ghi.com
HOST_USERNAME	jsmith
HOST_PASSWORD	<password3>

Scenario 3 – One Service Action to Multiple Atomic Actions Routed to Different NEs

Figure 7-3 shows a single service action that is mapped to two or more atomic actions, each of which is routed to a different NE. Each NE is using a different network template.

Figure 7-3 One Service Action to Multiple Atomic Actions Routed to Different NEs



The work order includes the following service action and communication parameters:

```
C-SERVICE
TEMPLATE_ID_A=TEMPLATE_1
```

```
SUBSCRPTION_A.NE_ID.URL=http://www.abc.com
SUBSCRIPTION_A.NE_ID.HOST_USERNAME=jsmith
SUBSCRIPTION_A.NE_ID.HOST_PASSWORD=<password1>
TEMPLATE_ID_B=TEMPLATE_2
SUBSCRPTION_B.NE_ID.URL= http://www.def.com/
SUBSCRIPTION_B.NE_ID.HOST_USERNAME=dmiller
SUBSCRIPTION_B.NE_ID.HOST_PASSWORD=<password2>
TEMPLATE_ID_C=TEMPLATE_3
SUBSCRPTION_C.NE_ID.URL=http://www.ghi.com
SUBSCRIPTION_C.NE_ID.HOST_USERNAME=djones
SUBSCRIPTION_C.NE_ID.HOST_PASSWORD=<password3>
```

Table 7-6 shows the service model configuration parameter mappings.

Table 7-6 Scenario 3 Parameter Mappings

asdl_cmd	asdl_lbl	csdl_lbl	param_typ
A-SERVICE_1	COMM_PARM	SUBSCRIPTION_A	Compound
A-SERVICE_1	MCLI	TEMPLATE_ID_A	Compound
A-SERVICE_2	COMM_PARM	SUBSCRIPTION_B	Compound
A-SERVICE_2	MCLI	TEMPLATE_ID_B	Compound
A-SERVICE_3	COMM_PARM	SUBSCRIPTION_C	Compound
A-SERVICE_3	MCLI	TEMPLATE_ID_C	Compound

The atomic action parameter MCLI in this case identifies the network template. In static routing, the atomic action parameter MCLI identifies the NE.

This configuration routes each atomic action to a different NE. Table 7-7 shows the downstream program (JInterpreter class) parameters.

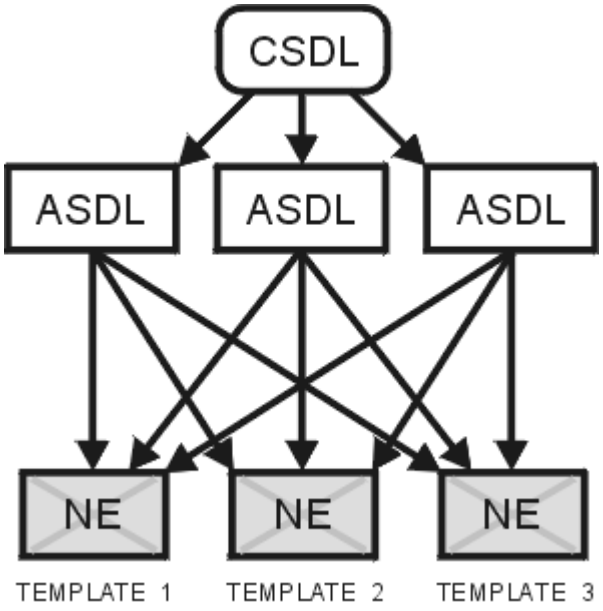
Table 7-7 Scenario 3 Parameters

Label	Value
A-SERVICE_1	-
MCLI	TEMPLATE_1
URL	http://www.abc.com
HOST_USERNAME	jsmith
HOST_PASSWORD	<password1>
A-SERVICE_2	-
MCLI	TEMPLATE_2
URL	http://www.def.com
HOST_USERNAME	dmiller
HOST_PASSWORD	<password2>
A-SERVICE_3	-
MCLI	TEMPLATE_3
URL	http://www.ghi.com
HOST_USERNAME	djones
HOST_PASSWORD	<password3>

Scenario 4 – One Service Action to Multiple Atomic Actions Routed to Multiple NEs

Figure 7-4 shows a case that differs from the previous scenario in that all of the atomic actions are sent to each NE.

Figure 7-4 One Service Action to Multiple Atomic Actions Routed to Multiple NEs



The work order includes the following service action and communication parameters:

```
C-SERVICE
TEMPLATE_ID[1]=<TEMPLATE_1>
SUBSCRIPTION[1].NE_ID.URL=http://www.abc.com
SUBSCRIPTION[1].NE_ID.HOST_USERNAME=jsmith
SUBSCRIPTION[1].NE_ID.HOST_PASSWORD=<password1>
TEMPLATE_ID[2]=<TEMPLATE_2>
SUBSCRIPTION[2].NE_ID.URL= http://www.def.com/
SUBSCRIPTION[2].NE_ID.HOST_USERNAME=dmiller
SUBSCRIPTION[2].NE_ID.HOST_PASSWORD=<password2>
TEMPLATE_ID[3]=<TEMPLATE_3>
SUBSCRIPTION[3].NE_ID.URL=http://www.ghi.com
SUBSCRIPTION[3].NE_ID.HOST_USERNAME=djones
SUBSCRIPTION[3].NE_ID.HOST_PASSWORD=<password3>
```

Table 7-8 shows the service model configuration parameter mappings.

Table 7-8 Scenario 4 Parameter Mappings

asdl_cmd	asdl_lbl	csdl_lbl	param_typ
A-SERVICE_1	COMM_PARM	SUBSCRIPTION[++]	Compound
A-SERVICE_1	MCLI	TEMPLATE_ID[++]	Compound
A-SERVICE_2	COMM_PARM	SUBSCRIPTION[++]	Compound
A-SERVICE_2	MCLI	TEMPLATE_ID[++]	Compound

Table 7-8 (Cont.) Scenario 4 Parameter Mappings

asdl_cmd	asdl_lbl	csdl_lbl	param_typ
A-SERVICE_3	COMM_PARM	SUBSCRIPTION[++]	Compound
A-SERVICE_3	MCLI	TEMPLATE_ID[++]	Compound

This configuration routes each atomic action to each NE. Table 7-9 shows the downstream program (JInterpreter class) parameters.

Table 7-9 Scenario 4 Parameters

Iteration	Label	Value
A-SERVICE_1		
-	MCLI	TEMPLATE_1
1	URL	http://www.abc.com
1	HOST_USERNAME	jsmith
1	HOST_PASSWORD	<password1>
-	MCLI	TEMPLATE_2
2	URL	http://www.def.com
2	HOST_USERNAME	dmiller
2	HOST_PASSWORD	<password2>
-	MCLI	TEMPLATE_3
3	URL	http://www.ghi.com
3	HOST_USERNAME	djones
3	HOST_PASSWORD	<password3>

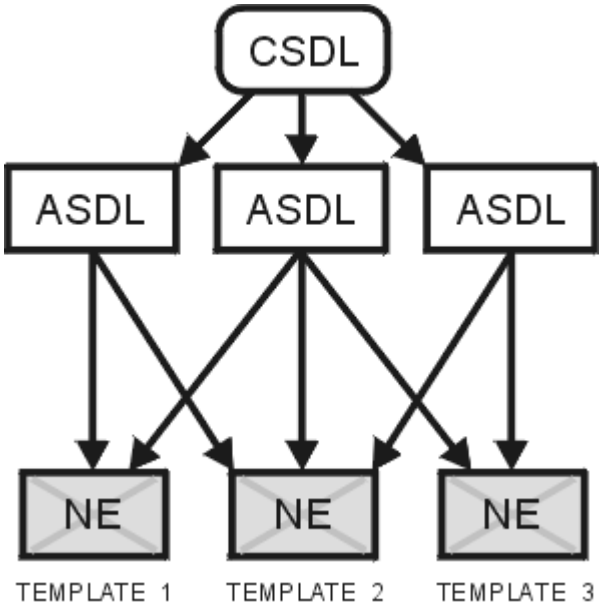
Each atomic action is called three times, each time with a different set of communication parameters.

Table 7-9 applies to A-SERVICE_2 and A-SERVICE_3 as well.

Scenario 5 – One Service Action to Multiple Atomic Actions Routed to Different NEs

Figure 7-5 shows atomic actions that are routed to one or more NEs, and others that are routed to another NE.

Figure 7-5 One Service Action to Multiple Atomic Actions Routed to Different NEs



The work order includes the following service action and communication parameters:

```
C-SERVICE
TEMPLATE_ID_A[1]=<template_1>
SUBSCRIPTION_A[1].NE_ID.URL=http://www.abc.com
SUBSCRIPTION_A[1].NE_ID.HOST_USERNAME=jsmith
SUBSCRIPTION_A[1].NE_ID.HOST_PASSWORD=<password1>
TEMPLATE_ID_A[2]=<template_2>
SUBSCRIPTION_A[2].NE_ID.URL=http://www.pqr.com
SUBSCRIPTION_A[2].NE_ID.HOST_USERNAME=dabrams
SUBSCRIPTION_A[2].NE_ID.HOST_PASSWORD=<password4>
TEMPLATE_ID_B[1]=<template_1>
SUBSCRIPTION_B[1].NE_ID.URL=http://www.abc.com
SUBSCRIPTION_B[1].NE_ID.HOST_USERNAME=jsmith
SUBSCRIPTION_B[1].NE_ID.HOST_PASSWORD=<password1>
TEMPLATE_ID_B[2]=<template_2>
SUBSCRIPTION_B[2].NE_ID.URL=http://www.pqr.com
SUBSCRIPTION_B[2].NE_ID.HOST_USERNAME=dabrams
SUBSCRIPTION_B[2].NE_ID.HOST_PASSWORD=<password4>
TEMPLATE_ID_B[3]=<template_3>
SUBSCRIPTION_B[3].NE_ID.URL= http://www.c.com/
SUBSCRIPTION_B[3].NE_ID.HOST_USERNAME=drichler
SUBSCRIPTION_B[3].NE_ID.HOST_PASSWORD=<password9>
TEMPLATE_ID_C[1]=<template_2>
SUBSCRIPTION_C[1].NE_ID.URL=http://www.pqr.com
SUBSCRIPTION_C[1].NE_ID.HOST_USERNAME=dabrams
SUBSCRIPTION_C[1].NE_ID.HOST_PASSWORD=<password4>
TEMPLATE_ID_C[2]=<template_3>
SUBSCRIPTION_C[2].NE_ID.URL= http://www.c.com/
SUBSCRIPTION_C[2].NE_ID.HOST_USERNAME=drichler
SUBSCRIPTION_C[2].NE_ID.HOST_PASSWORD=<password9>
```

Table 7-10 shows the service model configuration parameter mappings.

Table 7-10 Scenario 5 Parameter Mappings

asdl_cmd	asdl_lbl	csdl_lbl	param_typ
A-SERVICE_1	COMM_PARM	SUBSCRIPTION_A[++]	Compound
A-SERVICE_1	MCLI	TEMPLATE_ID_A[++]	Compound
A-SERVICE_2	COMM_PARM	SUBSCRIPTION_B[++]	Compound
A-SERVICE_2	MCLI	TEMPLATE_ID_B[++]	Compound
A-SERVICE_3	COMM_PARM	SUBSCRIPTION_C[++]	Compound
A-SERVICE_3	MCLI	TEMPLATE_ID_C[++]	Compound

This configuration routes atomic action

- A-SERVICE_1 to NE_1 and NE_2
- A-SERVICE_2 to NE_1, NE_2, and NE_3
- A-SERVICE_3 to NE_2 and NE_3

Table 7-11 shows the downstream program (JInterpreter class) parameters.

Table 7-11 Scenario 5 Parameters

Iteration	Label	Value
A-SERVICE_1	-	-
-	MCLI	<template_1>
1	URL	http://www.abc.com
1	HOST_USERNAME	jsmith
1	HOST_PASSWORD	<password1>
-	MCLI	<template_2>
2	URL	http://www.pqr.com
2	HOST_USERNAME	dabrams
2	HOST_PASSWORD	<password4>
A-SERVICE_2	-	-
-	MCLI	<template_1>
1	URL	http://www.abc.com
1	HOST_USERNAME	jsmith
1	HOST_PASSWORD	<password1>
-	MCLI	<template_2>
2	URL	http://www.pqr.com
2	HOST_USERNAME	dabrams
2	HOST_PASSWORD	<password4>
3	MCLI	<template_3>
3	URL	http://www.c.com
3	HOST_USERNAME	drichler
3	HOST_PASSWORD	<password9>

Table 7-11 (Cont.) Scenario 5 Parameters

Iteration	Label	Value
A-SERVICE_3	-	-
1	MCLI	<template_2>
1	URL	http://www.pqr.com
1	HOST_USERNAME	dabrams
1	HOST_PASSWORD	<password4>
-	MCLI	<template_3>
2	URL	http://www.c.com
2	HOST_USERNAME	drichler
2	HOST_PASSWORD	<password9>

Scenario 6 – Common URL

The following sample shows a common URL that is shared:

- Global Parameter

```
SUBSCRIPTION_A.NE_ID.URL=http://www.abc.com
```

- C-SERVICE

```
TEMPLATE_ID=<template_1>
SUBSCRIPTION_A[1].NE_ID.HOST_USERNAME=jsmith
SUBSCRIPTION_A[1].NE_ID.HOST_PASSWORD=<password1>
SUBSCRIPTION_A[2].NE_ID.HOST_USERNAME=dabrams
SUBSCRIPTION_A[2].NE_ID.HOST_PASSWORD=<password4>
```

Table 7-12 shows the service model configuration parameter mappings.

Table 7-12 Common URL Parameter Mappings

asdl_cmd	asdl_lbl	csdl_lbl	param_typ
A-SERVICE_1	COMM_PARM	SUBSCRIPTION_A[++]	Compound
A-SERVICE_1	MCLI	TEMPLATE_ID	Scalar

Table 7-13 shows the downstream program (JInterpreter class) parameters.

Table 7-13 Common URL Parameters

Iteration	Label	Value
A-SERVICE_1	-	-
-	MCLI	<template_1>
1	URL	http://www.abc.com
1	HOST_USERNAME	jsmith
1	HOST_PASSWORD	<password1>
-	MCLI	<template_2>

Table 7-13 (Cont.) Common URL Parameters

Iteration	Label	Value
2	URL	http://www.pqr.com
2	HOST_USERNAME	dabrams
2	HOST_PASSWORD	<password4>

Dynamic Routing Configuration Errors

If the maximum connections limit is reached, an exception is thrown indicating that the atomic action cannot be dispatched because all connections are in use. The atomic action is put in pending queue so that it can be processed when a connection is available.

A routing error (**ROUT_ERR**) event is logged and the work order fails in the following circumstances:

- The network template identifier is not defined on the order, or the identifier does not reference a valid template resource configured in ASAP.
- The combined total length of all communication labels and values exceeds 2048.

When dynamic communication parameters (as provided on an ASAP work order) are invalid (due to an incorrect IP address or port, for instance) the work order is not explicitly failed. Failing an order in this manner is generally reserved for incorrect activation parameters rather than invalid communication parameters. When incorrect communication parameters are detected (by the inability to establish a connection with the NE) the work order is placed in the retry queue. When the error in communication parameters is detected, use Order Control Application (OCA) to stop the order, change the invalid communication parameters and re-submit the order to ASAP.

OCA tracks the revision history of all orders.

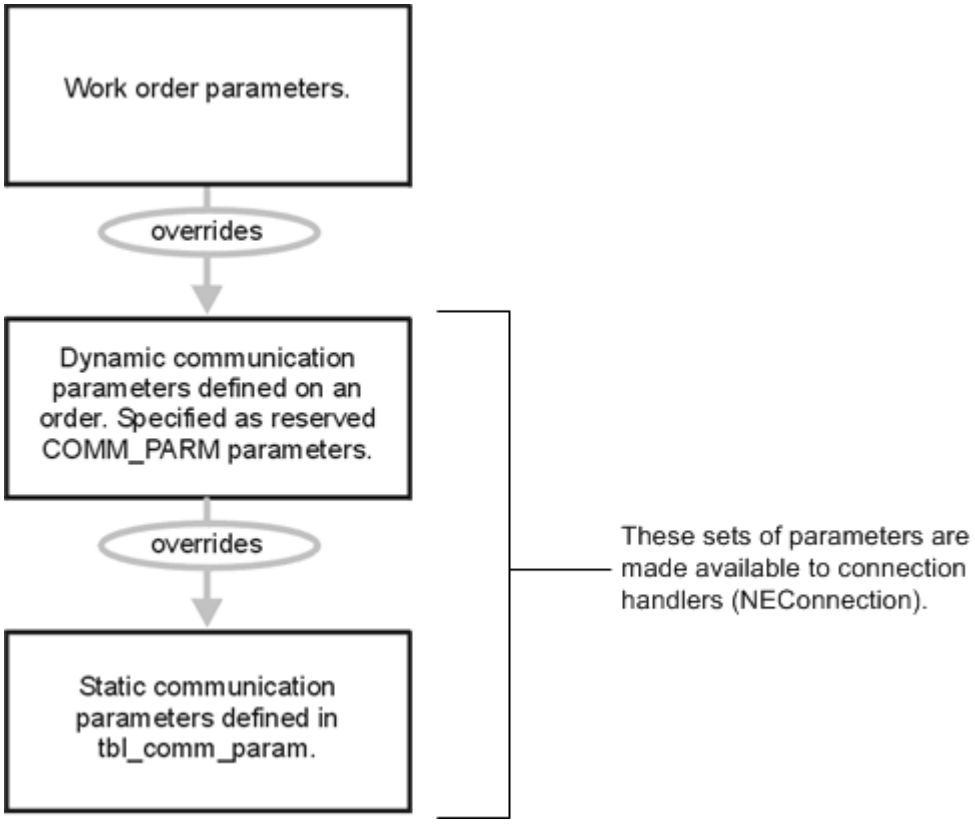
Refer to ASAP OCA User Guide for information about OCA.

Managing Communication and Order Parameters

Parameters defined with the same label as both communication and order parameters will conflict. In order of precedence, order communication parameters override static parameters if they have the same label. Oracle recommends that solutions developers not use conflicting labels for both communication and order parameters.

During provisioning, parameters contained in work orders override work order communication parameters (**COMM_PARM**), which override static communication parameters contained in **tbl_comm_param** (see Figure 7-6).

Figure 7-6 Order Parameter Precedence



Backward Support for MPM Protocols

Dynamic routing can be used in conjunction with Multi-Protocol Manager-supported protocols such as Telnet, FTP, and socket. These protocols require recognized keywords such as **HOST_IPADDR**, **HOST_NAME** and **PORT** to create a connection. These parameter names must be used to enable the MPM supported protocols.

An atomic action requires the following parameters to be routed using the MPM socket protocol:

- C**OMM_PARM.NE_ID.HOST_IPADDR** or **COMM_PARM.NE_ID.HOST_NAME**
- **COMM_PARM.NE_ID.PORT**

Software Load and Technology Type

Software load and technology type may be defined statically (tbl_host_clli) or provided by an upstream system as parameters on a work order.

Consequently, each atomic action requires the following reserved communication parameters:

- **COMM_PARM.NE_ID.SFTWR_LOAD** or **COMM_PARM.SFTWR_LOAD**
- **COMM_PARM.NE_ID.TECH_TYPE** or **COMM_PARM.TECH_TYPE**

These parameters can be defined dynamically on each order or statically in the network template.

The software load and technology type are established after when the NEP first establishes a connection to the NE. After the connection has been established, all subsequent values of **SFTWR_LOAD** and **TECH_TYPE** received from subsequent work orders destined to the same NE instance are ignored. The software load and technology type are reloaded the next time ASAP sets up a session manager, connection pool, and command processors for that NE.

NE Configuration Parameters

Some of NE configuration parameters (such as **max_connections**, **drop_timeout**, **spawn_threshold**, **kill_threshold**, and **line_type**) may be provided by an upstream system as Work Order communication parameters.

The following work order communication parameters can be specified to override the defaults:

- **COMM_PARM.NE_ID.MAX_CONNECTIONS** or **COMM_PARM.MAX_CONNECTIONS**
- **COMM_PARM.NE_ID.DROP_TIMEOUT** or **COMM_PARM.DROP_TIMEOUT**
- **COMM_PARM.NE_ID.SPAWN_THRESHOLD** or **COMM_PARM.SPAWN_THRESHOLD**
- **COMM_PARM.NE_ID.KILL_THRESHOLD** or **COMM_PARM.KILL_THRESHOLD**
- **COMM_PARM.NE_ID.LINE_TYPE** or **COMM_PARM.LINE_TYPE**

These parameters are used to initialize the session manager and command processors. After the session is established for the NE, parameters those coming from subsequent work orders to the same NE instance will be ignored until the session manager is removed from memory (when the primary connection to the NE is closed).

8
Creating Service Actions

This chapter describes how to create service actions for Oracle Communications ASAP.

About Creating and Configuring Service Actions

A Common Service Description Layer (CSDL) or service action command is an ASAP command that is associated with a particular work order. The service action command is associated with one or more operations on one or more network elements (NEs).

Service action command names are comprised of the string C_ (for Service Action) as well as attributes including the cartridge identification elements (tokens), actions, and services that have been selected for the cartridge. The tokens are separated by underscores, and compound tokens (if required) include a dash as a separator. If the software load token includes a "." it is replaced by a dash. All characters in the name must be in uppercase. The naming convention is as follows:

```
C_vendor-technology_softwareload_action_entity
```

Where

- **C_**: This prefix indicates a service action.
- *vendor*: see "Selecting the Vendor Token"
- *technology*: see "Selecting the Technology Token"
- *softwareload*: see "Selecting the Software Load Token"
- *action*: see "Selecting the Action Tokens"
- *entity*: see "Selecting Entity Tokens"

> **Note:**
>
> If service packages are used, the service token should include the service package in its name. For example a service action belonging to the BGP service package would be named as follows:
>
> ```
> C_CSCO-IOS_12-2-X_ADD_BGP-MAX-PREFIX
> ```

Identify and model meaningful services as service action commands. The first step is to create a one-to-one mapping between each service action and atomic action. For example, an atomic action that adds three-way calling to a subscriber line should have an associated service action that allows for this feature to be activated individually by an upstream system:

Table 8-1 Service-Action-to-Atomic-Action Mapping (One-to-One)

Service Action	Atomic Action
C_NOKIA_HLR_M11_ADD-3WC	A_NOKIA_HLR_M11_ADD-3WC

Where possible, also model other meaningful services. For example an atomic action to nail up a relay point on a Nortel Passport NE should also be modeled into a more meaningful service. The service action configuration should therefore include an individual service action that allows the relay point to be nailed up as well as a service action that implements a more meaningful service such as the activation of permanent virtual circuit (which makes use of the atomic action to nail up a relay point among other atomic actions that are used to construct the PVC):

Table 8-2 Meaningful Service-Action-to-Atomic-Action Mapping

Service Action	Atomic Action	Meaningful Service
C_NT-PP_12-4_SPECIFY_NRP	A_NT-PP_12-4_SPECIFY_NRP	No
C_NT-PP_12-4_ADD_ATM-PVC	A_NT-PP_12-4_ADD_VCC A_NT-PP_12-4_SPECIFY_NRP A_NT-PP_12-4_CREATE_X-CONN	Yes

Service cartridge service actions do not have to follow this naming convention.

Design Studio for ASAP automatically enforces this naming convention when you create a service action with the Service Action Wizard.

Creating Service Actions

To create a service action using Design Studio, use the following procedure:

1. Select **Studio**, then **New**, then **Activation**, and then **Service Action.**
2. From the Service Action Wizard, do the following:
   - In the **Action** field, enter an action name that corresponds to an NE command.
   - In the **Entity** field, enter an entity name that corresponds to an NE service name or service package you want to configure.
3. Click **Finish**.

   The Service Action editor appears.

Configuring Service Action Default Sequence

Service action level refers to the relative ordering of the service action within the work order. The SARM must have the service action level in case it receives service actions from a service request processor (SRP) or a Java SRP that is not in the work order in which it must be provisioned. Using the service action level, the SARM can re-order the service actions on an ASAP work order.

Assign service action levels based on the logical sequence in which the service action commands would need to occur if they were contained within a single work order. For example, on some NEs where a change line command is not available, a work order may contain service actions to

1. Query the line for line attribute information.
2. Delete the line.
3. Recreate the line with new attributes.
4. Re-assign the old attributes

Assigning the levels as shown in the following list ensures that service actions are run in the correct order if they were for some reason out of sequence on the original order:

- Query 100
- Delete 120
- Add 140
- Modify/change 160

To configure a service action sequence using Design Studio:

1. From the Service Action editor, click the **Properties** tab.
2. In the **Level** field, enter a service action sequence level.

Configuring Service Action Fail and Complete Events

You can optionally configure a service action to trigger a return event to the SRP or JSRP when it receives a defined event.

- **Service Action Completion Event** – The event that is triggered if this service action completes successfully. These events can either be system events or custom events. For information about system events and configuring system events see the *ASAP System Administrator's Guide*.
- **Service Action Failure Event** – The event that is triggered if this service action fails. These events can either be system events or custom events. For information about system events and configuring system events, see the *ASAP System Administrator's Guide*.

Each service action command must be defined in a static user-configured translation table that specifies the particular characteristics of the service action command. **tbl_csdl_config** is a user-created static table that contains all service actions. Each work order submitted to can have one or more associated service actions.

To configure a service action fail or complete event using Design Studio:

1. From the Service Action editor, in the **Properties** tab, select a service action completion event from the **Service Action Completion Event** list.
2. From the **Service Action Failure Event** list, select a service action failure event.

About Mapping a Service Action to Atomic Actions

After you have defined service action commands, atomic actions, and atomic action parameters, you can establish mapping relationships between the service action commands and atomic actions. You must define which atomic actions are transmitted to the NEP for a given service action command.

A service action command can have one or more atomic actions. Multiple atomic actions must be performed in the correct sequence, otherwise the service action can fail. This sequence is identified when creating the mapping relationship. In the example below, the atomic action **CREATE_LINE** must be performed before any options are added to that line.

Table 8-3 Service-Action-to-Atomic-Action Mappings

Service Action	Atomic Action	Parameters	Description
CREATE_RES_LINE	CLEAR_INTERCEPT	MCLI="NEWYORK", NPA="516", NNX="555", LINE="1212"	Clear the intercept for the directory number before adding line.
CREATE_RES_LINE	CREATE_LINE	MCLI="NEWYORK", LEN="2111112", NPA="516", NNX="555", LINE="1212", PARTY="I", PIC="333"	Create the line in the NE.
CREATE_RES_LINE	SET_OPTION_ON	MCLI="NEWYORK", LEN="2111112", NPA="516", NNX="555", LINE="1212", OPT="TTR"	Add the Touch Tone feature to the line.
ADD_FEATURE	SET_OPTION_ON	MCLI="NEWYORK", LEN="2111112", NPA="516", NNX="555", OPT="CAW"	Add the Call Waiting feature to the line.

Note:

Any changes or additions you make to mapping relationships only take effect after the SARM server is restarted.

About Limiting Independent Network Element Commands to Optimizing the Network Element Interface

The goal of NE interface optimization is to limit the number of independent NE commands (either MML or API calls) that ASAP sends to an NE. Collect related service activation requests and combine them into a single service activation request that ASAP sends to the NE. This avoids performance overhead associated with checking multiple NE responses and provides a higher degree of throughput to the NE.

In the non-optimized (standard) approach, a number of independent atomic actions are combined together to create a service action as shown in the following example:

```
C_NOKIA-HLR_M11_ADD_FEATURES
                    A_NOKIA-HLR_M11_ADD_CW
                    A_NOKIA-HLR_M11_ADD_3WC
```

```
A_NOKIA_HLR_M11_ADD_CF
…others feature atomic actions…
```

In this example, a service, **C_NOKIA-HLR_M11_ADD_FEATURES**, spawns multiple atomic actions that activate features on a subscriber line. Because each Java method has one feature-related NE command embedded in it, each atomic action that runs sends one NE command to the NE and each atomic action is responsible for checking the response from the NE to verify its success. Though this approach enables tight feature-specific parameter checking in the SARM, the error checking required after ASAP sends each command creates a significant amount of overhead. This approach is suitable when ASAP provisions a small volume of work orders and performance is not a major factor; however, when ASAP provisions a large volume of work orders, this approach may impair the performance of ASAP.

When NE interface optimization is used, a Java method is created that combines the multiple feature requests into one or more NE commands. Some NEs have a length limit to the command string and therefore some splitting of commands may be necessary. The Java method is used to examine the feature flags on the work order and then construct a larger NE command. A generic atomic action maps to the main Java method and all of the possible atomic action parameters for adding the supported features to the subscriber line are associated with the following atomic action:

```
C_NOKIA-HLR_M11_ADD_FEATURES
                    A_NOKIA-HLR_M11_ADD_FEATURES
```

When a Java method sends more than one command to an NE, it must be transaction oriented. For example, if a Java method sends multiple commands to an NE and the third command fails, it is necessary to roll back the previous two commands before failing the atomic action

An optimized design requires that all of the atomic action parameters provided to the Java method be configured as optional in the SARM. This reduces the error-checking ability of the SARM and results in a higher degree of fallout at the NE level. Additional coding, maintenance, and testing effort is also required within each Java method. The benefits of this design includes reducing the number of independent commands that are sent to the NE and reducing the number of atomic actions and responses that ASAP must manage. For more information about atomic action to Java method and MML command ratio, see "About the Ratio of Provisioning Commands to Atomic Actions."

In addition to providing a standard set of atomic actions that map to the individual NE commands on an NE, it may be possible to implement atomic actions that support NE interface optimization if coupling of commands is supported by the NE. This is most common in the voice networks where numerous features (such as creating a subscriber and adding a number of features) are needed to provide different levels of service to a customer. If the NE does support this functionality, it must be supported in the cartridge in addition to the standard service modeling approach.

Adding Atomic Actions to a Service Action

To add atomic actions to a service action using Design Studio:

1. From the Service Action editor, click the **Atomic Actions** tab, and then click **Add**.

   The Atomic Action Selection dialog box appears.

2. In the **Matching items** list, select an atomic action.

3. Click **OK**.

   The atomic action you selected is added to the atomic action list in the **Atomic Actions** tab.

> **Note:**
>
> Atomic actions are run in the order in which they are added to a service action from the top of the list to the bottom. You can change the position of atomic actions in **Atomic Actions** tab using the up and down arrows.

About Atomic Action Spawning Logic

When given a particular service action command and its parameters, the SARM refers to a static user-populated translation table to generate one or more atomic actions for this service action with certain conditions. **tbl_csdl_asdl** is a static table that is used by the SARM and contains these mappings between service action commands and atomic actions. For each atomic action associated with a service action, the SARM verifies whether the atomic action should be spawned for the specified service action. The final determination of whether the atomic action is spawned depends on the atomic action parameter translation process specified in the **tbl_asdl_parm** database table.

ASAP's translation logic makes it possible to determine whether or not to spawn an atomic action based on a range of values, or based on equal, not equal, greater than or less than conditions. It is also possible to combine conditions using an AND or OR operator. This logic permits detailed computations required to run an atomic action on the NE, to be performed in the service-action-to-atomic-action translation step, rather than in the atomic action to NE translation step, and streamlines bandwidth usage during NEP to NE communications. The expanded set of possibilities for service-action-to-atomic-action translation allows for a greater flexibility in the translation and mapping process and a more efficient processing effort.

The generation of each atomic action can also depend on the results of previous atomic actions that return parameters to the SARM upon successful completion.

This arrangement provides a mechanism for flexible translation, allowing the SARM to use one of the following methods to perform service-action-to-atomic-action translation.

- Unconditional translation: The SARM always generates the atomic action for this service action. ASAP supports the following unconditional translation option:
  - **Always** – The SARM always generates the atomic action for this service action.
- Conditional translation: The SARM uses conditional logic to decide whether to generate the atomic action. If the label and/or values associated with the conditional translation are not configured in the database, the atomic action fails. ASAP supports the following conditional translation options:
  - **Always** with **Include Expression** option: The user can define a logical expression using a number of criteria for a service action parameter. The range of options available allows an atomic action to be generated if the service action parameter value is within a set range of values or if the service action parameter is greater than, or less than, or equal to, a specified value. More than one condition can be combined in the expression, using an AND or OR operator. For more information about creating logical expressions, see "Components of Service-Action-to-Atomic-Action Translation Expressions" and "Defining Service Action-Atomic Action Translation Expressions."
  - **Defined** – The SARM only generates a particular atomic action if the stated service action parameter is defined on the service action.
  - **Not Defined** – The SARM only generates a particular atomic action if the stated service action parameter is not defined on the service action.

– **Equals** – The SARM only generates a particular atomic action if the stated service action parameter is defined on the service action and has a particular parameter value.

> **Note:**
>
> Always use optional parameters to spawn atomic actions using the **Equals**, **Defined**, and **Not Defined** conditional translation options. Using a mandatory parameter creates error messages if the mandatory parameter is not used.

Configuring Atomic Action Spawning Conditions

To configure atomic action spawning conditions using Design Studio:

1. From the Service Action editor, in the **Atomic Actions** tab, select one of the following options:
   - **Always**
   - **Defined**
   - **Not Defined**
   - **Equals**
2. (Optional) Select **Include Expression**. This option is only available with the **Always** condition.
3. (Optional) Enter a logical expression in the **Include Expression** text box. This option is only available with the **Always** condition.
4. (Optional) Enter an atomic action parameter label in the **Parameter Label** field. This option is available for the **Defined**, **Not Defined**, and **Equals** conditions.
5. (Optional) Enter an atomic action parameter label value in the **Parameter Value** field. This option is only available for the **Equals** condition.

Components of Service-Action-to-Atomic-Action Translation Expressions

The **eval_exp** column (also called **Include Expression** field in Design Studio) in the **tbl_csdl_asdl_eval** table contains an algebraic expression (as a string) that combines all the parameters to be checked. If the value of the **eval_exp** in the **tbl_csdl_asdl_eval** is not NULL, the string is parsed and the expression is evaluated to TRUE or FALSE. If the expression cannot be evaluated (for example, incorrect semantics or non-existent parameter), the translation fails.

The expression for the service-action-to-atomic-action translation contains **<parameter operator [value]>** groups. To specify the order operations, you must use brackets in the algebraic expression. A string with a length of 255 can accommodate a number of conditions. The average is 20 groups of **<parameter operator [value]>** groups. Each parameter or value should not exceed 30 bytes in length.

Supported Parameters for Translation Expressions

The parameters that may be included in the expression and to be checked are:

- Service action parameters and work order parameters
- Technology and software load

Table 8-4 shows the predefined parameters available to use with each translation expression.

Table 8-4 Predefined Parameters

Parameters	Description
HOST_NE	Remote NE.
TECH	Technology or NE type. This parameter is read from memory and loaded when SARM starts.
SFTWR	Software that runs on the NE. This parameter is read from memory and loaded when SARM starts.

The parameters in the algebraic expression must match one of the parameter names that come from SARM, such as service action or work order parameters. You can define all other parameters and you can check any number of parameters in the algebraic expression (limit of 255 bytes.)

Supported Operators for Translation Expressions

Table 8-5 shows the operators you can use in the service-action-to-atomic action translation expression.

Table 8-5 Service-Action-to-Atomic-Action Translation Expression Operators

Operators	Description
Boolean: AND, OR and NOT	These Boolean operators are applied against Boolean values returned by operations performed on parameters. For example: NUM3 < 9999 AND NUM2 <333 The expression to enter in tbl_csdl_asdl is (NUM3 < 9999) AND (NUM2 <333)
Operators to be used against parameters	Parameter value not required: ISDEF, NOTDEF. For example: NOTDEF VAR7 AND NUM2 > 333 The expression in tbl_csdl_asdl is (NOTDEF VAR7) AND (NUM2 > 333)
Parameter value is required	Integer operators: >, <, >=, =<, =, != String operators: LIKE, !LIKE For example: (VAR7=72) AND (CENTER !LIKE “YORK")

Supported Values for Translation Expressions

The operators >, <, >=, =<, != and = are used only for integer values. The values provided for these operators in the expression must be convertible to an integer or the translation will fail. These operators trigger the conversion when the expression is parsed.

For string values, you use the operators LIKE and !LIKE (that is, not LIKE) and the values require quotation marks, for example, CENTER **!LIKE** “YORK". These operators accept the wildcards % and ?. For example: (TECH **LIKE** “D?S") AND (SFTWR **!LIKE** “BCS%")

A value is not required for the ISDEF and NOTDEF operators. The groups evaluate to TRUE or FALSE and they can be aggregated together using the operands AND and OR.

All other parameters to be included in the expression can have any name, as long as they match a service action or work order parameter label. If a parameter label is not defined on the incoming work order, translation will fail.

Defining Service Action-Atomic Action Translation Expressions

The use of brackets in the service-action-to-atomic-action translation expression to specify the order of the operations simplifies the following aspects of the service-action-to-atomic-action translation process:

- Precedence of the operators is already determined
- Increased performance when evaluating each atomic action for service action translation
- Coding, maintenance, and enhancements of service actions and atomic actions

To avoid errors, you must use spaces between Literal operators and labels and Literal operators and values.

The format is:

```
(DIS LIKE "B747")
```

> **Note:**
>
> Literal operators are: LIKE, !LIKE, ISDEF, NOTDEF. The operators AND and OR always have their operands in brackets, therefore no blank space is mandatory on either side.

A space between other operators and the operands is recommended but not mandatory.

```
(AAA < 72) AND (NOTDEF BBB)
```

To specify the order of the operations, each operator and its operands must be enclosed in a set of brackets:

```
((A < 8) OR ((NOTDEF B) AND ((C != 3) OR  (NOT(D = 9)))))
```

Table 8-6 shows the possible values in the **eval_exp** column of **tbl_csdl_asdl_eval**.

Table 8-6 eval_exp column values

Value	Description
NULL	Used when you do not require the enhanced service-action-to-atomic- action translation. You can leave the eval_exp column of tbl_csdl_asdl_eval empty. This expression translates to TRUE, which means the translation relies completely on the cond_flag column. As a result, existing functionality is not affected and an AND is placed between the new and the existing functionality. These conditions are identified in columns cond_flag and eval_exp. An atomic action is valid only if both conditions are satisfied.
Valid algebraic expression	If you require the enhanced service-action-to-atomic-action translation, you must use a valid algebraic expression that evaluates to TRUE or FALSE in the eval_exp column of the tbl_csdl_asdl_eval table. The evaluation of an expression fails if it finds any syntax error in the expression, or if it cannot get a value for a parameter when the value is required.
TRUE	The string that is evaluated by the C code to the boolean value TRUE. This is similar to A (always) for existing functionality.

If the evaluation of the expression fails, a **SYS_ERR** diagnostic message is logged in the diagnostic file, and the atomic action is not included on the service action.

The expression evaluates to TRUE or FALSE, which would result in spawn or don't spawn the atomic action, respectively. You must ensure that the translation expression is correct. When the SARM starts or the configuration changes are dynamically re-loaded, the syntax of the translation expressions is checked.

Translation Function Conflicts

Ensure that there is no conflict between the conditions set by the different entities, if you use the following:

- Conditional service-action-to-atomic-action spawning logic
- Standard service-action-to-atomic-action spawning logic
- Atomic action to program or Java method mapping logic

You must also consider that the SARM service-action -to-atomic-action spawning logic creates an **AND** condition between these entities.

About Service Actions and Rollback

As part of the service action configuration, you must identify whether the service action rolls back in the event of failure. When the SARM begins provisioning a work order, it scans each service action on the order to determine if rollback has been configured. If rollback has been configured for one or more service actions, the SARM flags the work order as requiring rollback in the event of failure.

Table 8-7 shows the service action level rollback qualifications.

Table 8-7 Service Action-Level Rollback Qualifications

Qualification	Description
Atomic actions must also be configured for rollback separately	Atomic actions associated with the service actions are not affected by the rollback configuration for service actions.
Override default behavior using order property	The service action Level Rollback Configuration defines the default behavior for a service action. This setting is ignored if the work order Rollback Property specifies *no* rollback in the event of a work order failure.
Service action rollback configuration is a prerequisite for work order rollback	If the WO Rollback Property is turned on and no service actions are configured to require rollback, the work order will not roll back in the event of a failure.
The end state of rolled back service actions is unknown	During rollback processing, a list of all completed atomic actions is obtained. Regardless of whether an atomic action completes or fails, rollback of previous atomic actions continues. Consequently, the end state of the service action is unknown and must be tested.

Enabling the CSDL Rollback Functionality

The **rollback_req** configuration variable is located in **tbl_csdl_config** in the SARM database. If it is set to Y, rollback occurs. If it is set to N, no rollback occurs.

If a service action that requires rollback fails, the dynamic work order structure in the SARM memory is flagged and the entire work order is rolled back.

> **Note:**
>
> If employing the delayed_failure property (see "About Delayed Failure Properties"), rollback must be turned off, rollback_req must be set to N, and ignore rollback must be set to Y.

To enable CSDL rollback functionality using Design Studio:

1. From the Service Action editor, click the **Properties** tab, select **Rollback**.

Enabling Work Order Rollback Functionality for the Service Request Processor Emulator

The **wo_rback** configuration variable located in **tbl_wo_def** in the SRP database defines whether the work order rolls back in the event of failure.

> **Note:**
>
> **tbl_wo_def** and this variable are only applicable when using the SRP emulator.

- If set to **Y**, rollback for a work order occurs if the work order times out or fails.
- If set to **N**, no rollback for the work order occurs.
- If not specified, the SARM uses **D**, the default value. In this case, rollback depends on the service action parameter rollback_req in tbl_csdl_config. If rollback_req is set for service action, then the work order rolls back when it times out or fails.

About Configuring a Rollback Point (Point of No Return)

You can configure atomic actions so that when a rollback situation occurs, rollback will only be partially performed, stopping at the atomic action configured to be the point of no return. By configuring the **Rollback Point** (**pointOfNoReturn** in the XML) value in an atomic action, you can cause the following actions:

- No point of no return functionality - Rollback is performed normally.
- **State**: An atomic action can be configured as the rollback point (also called a point of no return) for partial rollback. If rollback occurs, and execution has continued beyond this point, execution is rolled back to this atomic action but no further.
- **Stop**: An atomic action can be configured as the rollback point for a rollback. After execution has continued past this atomic action, no rollback can occur.

An example scenario with atomic action1, atomic action2, atomic action3 and atomic action4, with atomic action2 configured for point of no return for partial rollback (for example, with a PNR=1), would work as follows. atomic action1, atomic action2 and atomic action3 run correctly. A work order timeout occurs on atomic action4. atomic action3 is rolled back but

because atomic action2 is considered to be the point of no return for partial rollback, neither atomic action2 nor atomic action1 is rolled back.

In the same scenario, if atomic action2 is configured for point of no return with no rollback (for example, PNR=2), then no rollback occurs at all.

Configuring a Rollback Point

To configure a point of no return using Design Studio:

1. From the Service Action editor, click the **atomic actions** tab, and then click the empty Rollback Point field for an atomic action.

   A list appears.

2. Select one of the following options:

   - Leave the field blank to maintain full rollback functionality.
   - **State**
   - **Stop**

9

Configuring Base Exit and User Exit Types

This chapter describes how to define an Oracle Communications ASAP user-defined exit types (UDETs) and map them to ASAP base exit types.

About User Errors and Thresholds

Network element errors can be associated with exit types. An exit type reflects the status of an atomic action at any point during the processing of that atomic action. Atomic action exit types that are associated with atomic action completion and failure scenarios are termed base exit types. The Service Activation Request Manager (SARM) database contains several tables that reference base exit types (in other words, contain a base_types column). You can define custom user-defined exit types (also known as user errors) that you can then map to ASAP events. User-defined errors are stored in the Service Request Manager (SARM) **tbl_user_err** table.

After you have defined user errors, you can define user error thresholds to elicit user-defined responses or events should the failure threshold for an atomic action be exceeded. For example, if a host network element (NE) returns a given error notification from a specific atomic action a given number of times (the defined threshold), the appropriate user-defined event is issued. User defined error thresholds are stored in **tbl_err_threshold**.

About Base Exit Types

In provisioning, when a user exit type is returned from a JNEP Java code, the corresponding base type is found. For the JNEP Java code, JNEP finds the base type and sends this base type to the NEP.

There are seven base types defined in SARM database **tbl_user_err**. If you try to define more base types in **tbl_user_err**, at startup and load time, the NEP server detects it as an error and terminates the server.

These base exit types include the following:

- **SUCCEED** – The atomic action ran successfully. The NEP successfully completes the atomic action and the SARM provisions the next atomic action on the work order. The completed atomic action and its associated parameters are saved by the SARM to facilitate rollback, if necessary.
- **FAIL** – Hard error; the atomic action failed, which results in the failure of the work order. The JInterpreter method sends this response if the provisioning activity has failed and work order provisioning must stop. The SARM then fails the entire work order and notifies the SRP of the work order failure event. If rollback is configured on the work order, the SARM rolls back any previously completed atomic actions.

> **Note:**
>
> **FAIL** is predefined in the Java code like other base types. You may define other user exit types based on it.
>
> To use this exit type from Java code, the Java method should call the **setASDLExitType** method. Oracle recommends adding error text to the failure to clarify diagnostic messages. For example:
>
> ```
> setASDLExitType("FAIL", "User error text for FAIL");
> ```

- **RETRY** – The atomic action will be retried later. The JInterpreter method sends this response when it is determined that the atomic action must be retried. While the atomic action is waiting to be retried, the connection to the NE can be used to provision other atomic actions destined to that NE. This is in contrast to the Maintenance Mode condition in which no other atomic actions are transmitted to the NE. If the atomic action does not complete after the final retry, the SARM fails it. The atomic action and associated parameters are logged only once by the SARM after the final try. When the retry threshold is reached, a system event indicates that the atomic action (and consequently the work order) has failed. Using the Control subsystem, you can map this system event to generate the appropriate alarm.

  The number and frequency of retry attempts are governed by work order properties, and in the event that these properties are not defined on the work order, by **ASAP.cfg** configuration parameters.

  For more information about RETRY behavior, see "Behaviors of RETRY and RETRY_DIS."

- **RETRY_DIS** – This is a base type which is similar to **RETRY**. This base type means the atomic action did not complete and needs to be retried. You can base new user exit types on **RETRY_DIS**. (DIS refers to *disconnect* rather than *disable*).

  When **RETRY_DIS** is returned from a Java provisioning class, the related port is disconnected. The atomic action will fail but will be retried by NEP server. The NEP server will indicate this to the SARM server. Just before the NEP server retries the atomic action, it has to establish a new connection because it was disconnected. The NEP will establish a connection again and will retry the same atomic action. This will continue as long as **RETRY_DIS** is returned from the Java provisioning class.

  If the parameter **IO_ASDL_RETRY** (which has a default value of 0) is defined as 1 in the **ASAP.cfg** file, the atomic action with **RETRY_DIS** exit code will be only retried as many times as is indicated by **NUM_TIMES_RETRY**. In this case, **RETRY_DIS** functions similar to **RETRY** with the difference of the additional disconnect and reconnect after each atomic action retry.

  The **NUM_TIMES_RETRY** parameter represents the number of times the SARM sends an atomic action to the NEP to be processed after the NEP returns it with a 'Fail but Retry' status. A work order is failed when the number of retries equals the value specified for **NUM_TIMES_RETRY**. The **RETRY_DIS** base exit type can occur if a device is found to be in an abnormal state and must be manually reset.

  For more information about **RETRY_DIS** behavior, see "Behaviors of RETRY and RETRY_DIS."

- **MAINTENANCE** – The atomic action failed because the NE is currently unavailable to receive provisioning requests. The JInterpreter method sends this atomic action response after being notified that the NE is currently unable to accept provisioning updates. The NEP automatically logs off and disconnects from the NE. On receiving this atomic action response, the SARM moves the atomic action from the In Progress atomic action queue to

the Pending queue, and then marks the status of the NE as Maintenance. The SARM waits for the NEP to transmit an NE Available notification after it has successfully re-established its primary connection to the NE. The atomic action and its associated parameters are not logged by the SARM because the atomic action itself did not actually fail.

- **SOFT_FAIL** – The atomic action has encountered an error has occurred that is not serious enough to halt the successful provisioning of the order. For example, this event can occur when the JInterpreter method logic detects that a user is attempting to add a feature to a line that already has that option. The SARM receives the relevant atomic action response and manages it in the same manner as an atomic action completion, with one exception. The SARM sets the Exceptions flag on the work order Completion notification returned to the SRP to indicate that some of the provisioning activity that was requested by the originating system was not performed. The failed atomic action and its associated parameters are saved by the SARM.
- **DELAYED_FAIL** – The atomic action had failed during provisioning. The SARM skips any subsequent atomic action in the service action, continues provisioning at the next service action, and then fails the order. The failed atomic action and its associated parameters are saved by the SARM. Rollback and delayed failure are incompatible because the intent of this base type is to continue provisioning subsequent Service Actions, while rollback would reverse successfully provisioned Service Actions. It it therefore recommended that you set the service action rollback parameter to N and the **ignore_rollback** parameter to **Y**. Delayed failure should only be used when there are no dependencies on subsequent Service Actions. If dependencies exist, the subsequent provisioning actions will fail.
- **STOP** – The atomic action has stopped processing. The JInterpreter method sends this response when it detects that the work order has been set in a stopped state. While the work order is in a stopped state, the Work Order Manager in the SARM only accepts requests to resume or cancel this work order. You can submit such requests through an Order Control Application (OCA) or Service Request Processor (SRP) API call by resubmitting the stopped work order.

Behaviors of RETRY and RETRY_DIS

When **IO_ASDL_RETRY** is a set to 0 (the default value):

1. The atomic action will fail but will be retried again (same behavior for both **RETRY** and **RETRY_DIS**).
2. The related port used will be kept intact for **RETRY** but disconnected for **RETRY_DIS**.
3. The port will be connected again (only for **RETRY_DIS**).

For RETRY, retry happens according to the **NUM_TIMES_RETRY** and **RETRY_TIME_INTERVAL** parameters, but not for **RETRY_DIS**.

For **RETRY_DIS**, steps a), b) and c) will be repeated as long as the Java provisioning class for that atomic action returns with **RETRY_DIS**.

When **IO_ASDL_RETRY** is set to 1:

1. Atomic action will fail but will be retried again (same behavior for **RETRY** and **RETRY_DIS**).
2. The related port used will be kept intact for **RETRY** but disconnected for **RETRY_DIS**.
3. The port will be connected again (only for **RETRY_DIS**).

For both **RETRY** and **RETRY_DIS**, retry happens according to the **NUM_TIMES_RETRY** and **RETRY_TIME_INTERVAL** parameters.

Steps a), b) and c) will be repeated **NUM _TIMES_RETRY** times only.

How to use **RETRY_DIS**:

JNEP

RETRY_DIS is predefined in the Java code like other base types. You may define other user exit types based on it.

To use this exit type from Java code, the Java method should call the **setASDLExitType** method. For example:

```
setASDLExitType("RETRY_DIS", "User error text for RETRY_DIS");
```

About User Exit Types

User exit types allow cartridge developers and systems administrators to map atomic action exit codes to one of the predefined base exit types. Base exit types determine the product behavior. Cartridges map return codes and status values from an NE to a user-defined exit type.

Regular expressions (regex) are used to perform pattern searches on responses from NEs. The pattern used is stored in **tbl_user_err** in the SARM database. The user exit type contains a regex pattern that is applied at runtime.

Regular expressions enable users to associate a series of responses to a specific base type. For example, a regular expression **6** can identify a pattern where any response with the character **6** followed by any number of characters will translate to base type of FAIL. Regular expressions can also allow very specific searches within a response from an NE.

Regular expressions are typically compiled before being run. Compilation produces a binary version of the expression and ensures that the syntax of the regular expression is correct. This compilation occurs using SACT and SADT when user exit types are deployed into ASAP as part of a cartridge. If the syntax is incorrect during compilation, SADT displays an error message and the deployment of the user exit type fails.

The supported regular expression version is consistent with Java 1.4.*x* regular expressions.

Using Regular Expression Search Patterns

The following provides additional regular expression search examples:

- ^.*\b(one|two|three)\b.*$ = matches a complete line of text that contains **one**, **two** or **three**.
- ^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).*$ matches a complete line of text that contains all of the words **one**, **two** and **three**.
- "[^"\r\n]*" matches a single-line string that does not allow the quote character to appear inside the string.
- \b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b matches any IP address.

For more information about search patterns, refer to the Java SE website:

http://docs.oracle.com/javase/8/

Using Search Patterns Against Long Switch Responses

There is a known issue with the Java.util.regex package from Java. Any match pattern with alteration on a string that is greater than 1400 bytes causes an exception and a stack overflow. This situation is not uncommon, particularly when implementing services and you want to

match the appearance of a word in a switch response. The following describes how to work around this issue.

In the following example, a command is sent to an NE and a multiline reply is received in which you want to match a keyword:

You may attempt to match the word **COMPLETED** in the reply as follows:

```
if (Pattern.matches("(.|\r|\n)*COMPLETED(.|\r|\n)*", replyString) ){
System.out.println("Matches \"COMPLETED\"");
} else {
System.out.println("No Match");
}
```

The problem will be encountered if the length of the replyString length exceeds 1400 bytes.

In the above sample, the “." signifies any character except a line terminator, that is, any of the following set of characters:

- \n (line feed, the UNIX line terminator)
- \r (carriage return)
- \u0085 (next line)
- \u2028 (line separator)
- \u2029 (paragraph separator)
- the sequence \r\n

Typically, to match . and \r and \n (or any combination of these), you would use (.|\r\n)* and this causes the problem.

However, Java Regexp enables you to match any characters including line terminator by means of an embedded flag expression. ((s).)* enables the flag to let "." match any character as well as line terminator. The problem is avoided by changing the search pattern to "((?s).)*COMPLD((?s).)*".

About User Exit Types for Unknown Errors

You must identify as many error codes or error messages from the NE as possible, and create user exit types for these errors. However; it is often difficult or impossible to map every possible error message. For these unknown error messages, create a catch-all user exit type, such as **NO_MATCH_FOUND** with a base exit type of **FAIL**.

The API call **setTypeByMatch** returns the error label (user-defined exit type field, as defined in Design Studio) for each match, but in case that no match is found (there is no modeled entry for this response pattern) it returns **NULL**. The code should associate all unknown errors with this type. **setTypeByMatch** can be overridden to handle this case. For example:

```
……..
logger.logDebug("NE REPLY: " + reply);
String exitValue = exitType.setTypeByMatch(reply);
logger.logDebug("Match returned for pattern <<" + reply + ">> is: "+ exitValue);
//If no match can be found among the defined exit types
if (exitValue == null) {
                exitValue = "NO_MATCH_FOUND";
                exitType.setTypeByMatch(exitValue);
}……..
```

About User Exit Types for Success Cases

Always identify the success case. This success case is the response pattern that means that the request successfully completed on the NE. Add it to the user-defined exit type mapping entries. This avoids failing the atomic action if no mapping is found for this case.

Mapping User Exit Types to Base Exit Types Based on Context

In some cases may require different atomic action exit types based on the context (like service) when the same response is received from the NE. For example, the same atomic action may be linked with various service actions (services) and should have a different exit status based on the service it is part of. The error code received is the same, but the outcome (fail, retry, warning) depends in this case on the incoming service action. For example, it is possible that the business requirements allow certain actions to be performed when creating an account but bar them when modifying the same account. In any of such cases, UDET granularity can be defined at service action or atomic action level. The UDET editor allows specifying the service action or atomic action for which the defined pattern and exit type will apply.

Creating New User Exit Types

Use the User Exit Type Wizard to create a user-defined exit type.

To create a new user-defined exit type:

1. Select **Studio**, then select **Show Design Perspective**.
2. Select **Studio**, select **New**, select **Activation**, then select **User Defined Exit Type**.
3. Select the project for this element and enter a name for the entity.
4. (Optional) Select a location for the entity.

   By default, Design Studio saves the entity to your default workspace location. You can enter a folder name in the **Folder** field or select a location different from the system provided default location. To select a different location:

   a. Click the **Folder** field **Browse** button.

   b. Navigate to the directory in which to save the entity.

   c. Click **OK**.
5. Click **Finish** to create the user-defined exit type.

Configuring User Exit Types

You can configure user-defined exit types using the User Defined Exit Type editor.

To configure a user exit type:

1. In the Cartridge view, double-click a User-Defined Exit Type entity to open the User Defined Exit Type editor.
2. In the User Defined Exit Types area, click **Add**.

   This enables the fields in the User Defined Exit Types Detail area of the editor and populates those fields with default values.
3. In the **Pattern** field, enter a value.

   For example, enter SUCCESS, DENIED, RESOURCE BUSY, and so on.

4. Select the corresponding base exit type.
5. Enter the User Defined Exit Type for this pattern.

   For example, you might enter **AA1_SUCCESS**.
6. Select **File**, then **Save**.

> **Note:**
>
> Use the **Service Action** and **Atomic Action** fields when creating Service Cartridges.

Examples: User Exit Types

Consider the following user exit type examples:

- Example: Unstable Network Element Connections
- Example: Configuration of Context Sensitive Exit Types
- Example: Exit Type Rationalization

Example: Unstable Network Element Connections

Problem: On an Ericsson network element during activation (after a successful connection and login to the network element) the login to the network element is randomly terminated. As an atomic action may be in progress against the network element at the time the connection was dropped it must be placed back in the queue for later activation and the connection must be re-established.

Solution: Configure a user exit type with the RETRY_DIS base type that triggers when the login prompt is detected during normal activation. This allows for the atomic action to retry at a later time after instructing ASAP to disable the current connection. If there is only one connection to the network element then ASAP eventually re-enables the connection and re-login.

Example: Configuration of Context Sensitive Exit Types

Problem: The customer has a network in which each HLR (referred to as a primary HLR) has a backup HLR (referred to as a secondary HLR). Services must be activated on both HLRs but if activations fail on primary HLRs the work order must be failed; if activations fail on secondary HLRs they must be soft failed.

Solution: Create different atomic actions that map to the same implementation. Configure two user-defined exit types that include the atomic action names in the configuration. Configure the base type for the primary atomic action with FAIL. Configure the base type for the secondary atomic action with SOFT_FAIL. The service model for this configuration is shown in the following diagram:

The user-defined exit type configuration is shown as follows:

User Type	Base Type	Pattern	Network Action
CRITICAL	FAIL	SUB&&EXISTS	A_HLR_ADD_SUB-PRIMARY
MINOR	SOFT_FAIL	SUB&&EXISTS	A_HLR_ADD_SUB-SECONDARY

In this example, whenever the response from the network element contains the strings `SUB` and `EXISTS` and the atomic action is A_HLR_ADD_SUB-PRIMARY, then a failure is triggered. Whenever the response from the network element contains the strings `SUB` and `EXISTS` and the atomic action is A_HLR_ADD_SUB-SECONDARY then a soft failure is triggered.

Example: Exit Type Rationalization

Problem: There are too many exit type entries with similar attributes present in the configuration, resulting in potentially high maintenance costs.

Solution: Where possible, collapse multiple exit type rows. For example, collapsing rows that have identical attributes other than the software load may be possible when the network element responses remain the same across software loads. A prime example of when exit type rationalization should occur is when multiple delivered cartridges are employed in the solution for the same network element. Because the user exit types in delivered cartridges always contain the vendor, technology, and software load attributes to ensure uniqueness, exit type rationalization is generally possible.

10

Configuring Dynamic and Static Event Templates for Return Parameters

This chapter describes how to create static and dynamic event templates for parameters returned from an network element (NE) as the result of an Oracle Communications ASAP work order.

About Static and Dynamic Event Templates for Return Parameters

Return parameters such as work order properties, information parameters, global work order parameters and service action return parameters can be returned on an ASAP Event. The details returned are controlled by template entries. These are configured using the **eventTemplate** object. The **serviceAction** and **eventType** attributes are used to identify the template. The **returnDataSet** object indicates which parameter names to retrieve. For more information refer to the descriptions of the **tbl_event_dataset** and **tbl_event_template** tables in the *ASAP Developer's Guide*.

Event templates can be configured statically or dynamically. Dynamic event templates are configured within work order properties sent to ASAP. Static event templates are configured within a cartridge. Dynamic event templates have precedence over static ones. Therefore, if there is any work order with a dynamic event template that matches an ASAP event related to that work order, no static event template will be checked.

For information about configuring dynamic event templates, see "Configuring a Dynamic Events Template." For information about creating a static event template, see Design Studio Help.

ASAP searches for any configured event template when any one of the following events occurs:

- Order Startup Event
- Order Complete Event
- Order Timeout Event
- Order Fail Event

> **Note:**
>
> If work order event is not an Order Fail Event, ignore the service action specified in the **Service Action** field.

For an Order Startup event:

1. ASAP searches for an event template with the event type **Order Startup Event** and that has the same parameter name and value as the work order.

2. If the search returns nothing, ASAP searches for an event template that has the event type **Order Startup Event**.
3. If the search returns nothing, no event template is configured.

For an Order Complete event:

1. ASAP searches for an event template with the event type **Order Complete Event** and that has the parameter name and value as the work order.
2. If the search returns nothing, ASAP searches for an event template that has the event type **Order Complete Event**.
3. If the search returns nothing, no event template is configured.

For an Order Timeout event:

1. ASAP searches for an event template with the event type **Order Timeout Event** and that has same parameter name and value as the work order.
2. If the search returns nothing, ASAP searches for an event template that has the event type **Order Timeout Event**.
3. If the search returns nothing, no event template is configured.

For an Order Fail event:

1. ASAP searches for an event template with the event type **Order Fail Event**, has the service action specified in the **Service Action** field, and that has the same parameter name and value as the work order.
2. If the search returns nothing, ASAP searches for an event template that has the event type **Order Fail Event**, has the service and has the same parameter name and value as the work order.
3. If the search returns nothing, ASAP searches for an event template that has the event type **Order Fail Event** and has the service action specified in the **Service Action** field.
4. If the search returns nothing, ASAP searches for an event template that has the event type **Order Fail Event**.
5. If the search returns nothing, no event template is configured.

Configuring a Dynamic Events Template

The extended work order property (or parameter) should be in the following format:

```
return_<event_template_name>%<event_type>%[service_action]
```

In this syntax, <...> means mandatory parameter, [...] means optional parameter; % is a separator.

Example:

```
return_ETEMP1%CompleteEvent%Service Action_1
```

Here,

- ETEMP1 is the event template name.
- CompleteEvent is the work order event type.
- Service Action_1 is the service action name.

An extended work order parameter in the format above is passed from the work order as an extended work order property. The value of that parameter, even if specified, is ignored.

```
return_dataset_<event_template_name>%<parameter_type>%[service_action]%<parameter_name>"
```

In this syntax, <...> means mandatory parameter, [...] means optional parameter; % is separator.

Example:

```
return_dataset_ETEMP1%infoParam%Service Action_1%MCLI
```

Here,

- ETEMP1 is the event template name.
- infoParam is the parameter type.
- Service Action_1 is the service action name (Service Action name)
- MCLI is the name of the parameter.

The parameter type has to be one of the following event parameter types:

- infoParam
- orderParameter
- serviceValue
- extendedWoProperty

Example (xml) 1:

```
<createOrderByValueRequest…..
……
….
…
<mslv-sa:extendedWoProperties>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_ETEMP1%orderStartupEvent</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP1%extendedWoProperty%apiClientId</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP1%extendedWoProperty%XYZ</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>XYZ</mslv-sa:name>
<mslv-sa:value>12349</mslv-sa:value>
</mslv-sa:extendedWoProperty>
</mslv-sa:extendedWoProperties>
</orderValue>
</createOrderByValueRequest>
```

Example (xml) 2:

```
<createOrderByValueRequest…..
……
….
…
<mslv-sa:extendedWoProperties>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_ETEMP1%orderStartupEvent</mslv-sa:name>
<mslv-sa:value/>
```

```
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP1%extendedWoProperty%apiClientId</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP1%extendedWoProperty%XYZ</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>XYZ</mslv-sa:name>
<mslv-sa:value>12349</mslv-sa:value>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_ETEMP2%orderCompleteEvent</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%infoParam%INFOP_N1</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%orderParameter%TESTP2</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%serviceValue%Service Action_TELNET%XML_csdl_P1</mslv-
sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
</mslv-sa:extendedWoProperties>
</orderValue>
</createOrderByValueRequest>
```

Example (xml) 3:

```
<createOrderByValueRequest…..
……
….
…
<mslv-sa:extendedWoProperties>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_ETEMP1%orderStartupEvent</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP1%extendedWoProperty%apiClientId</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP1%extendedWoProperty%XYZ</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>XYZ</mslv-sa:name>
<mslv-sa:value>12349</mslv-sa:value>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_ETEMP2%orderCompleteEvent</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
```

```
<mslv-sa:name>return_dataset_ETEMP2%infoParam%INFOP_N1</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%orderParameter%TESTP2</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%serviceValue%Service Action_TELNET%XML_csdl_P1</mslv-
sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%infoParam%INFOP_N2</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%infoParam%INFOP_A1_N1</mslv-sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>return_dataset_ETEMP2%infoParam%Service Action_TELNET_B%INFOP_B1_N1</mslv-
sa:name>
<mslv-sa:value/>
</mslv-sa:extendedWoProperty>
</mslv-sa:extendedWoProperties>
</orderValue>
</createOrderByValueRequest>
```

JSRP (OSS/J) Work Order Event Information

Additional information is returned for work order complete and failure events processed through JSRP servers. Network information is provided for failed services indicating what the last communicated network was when the service failed.

Extended work order complete and failure events contain the tags **failedServices** and **services**. This extension is configurable through a work order user property in order to provide backward compatibility. The failed services and services tags also contain the event template service parameters and info parameters, which may be used to pass upstream parameters that are relevant to services within an order.

For complete details of schema elements, refer to the ASAP Online Reference.

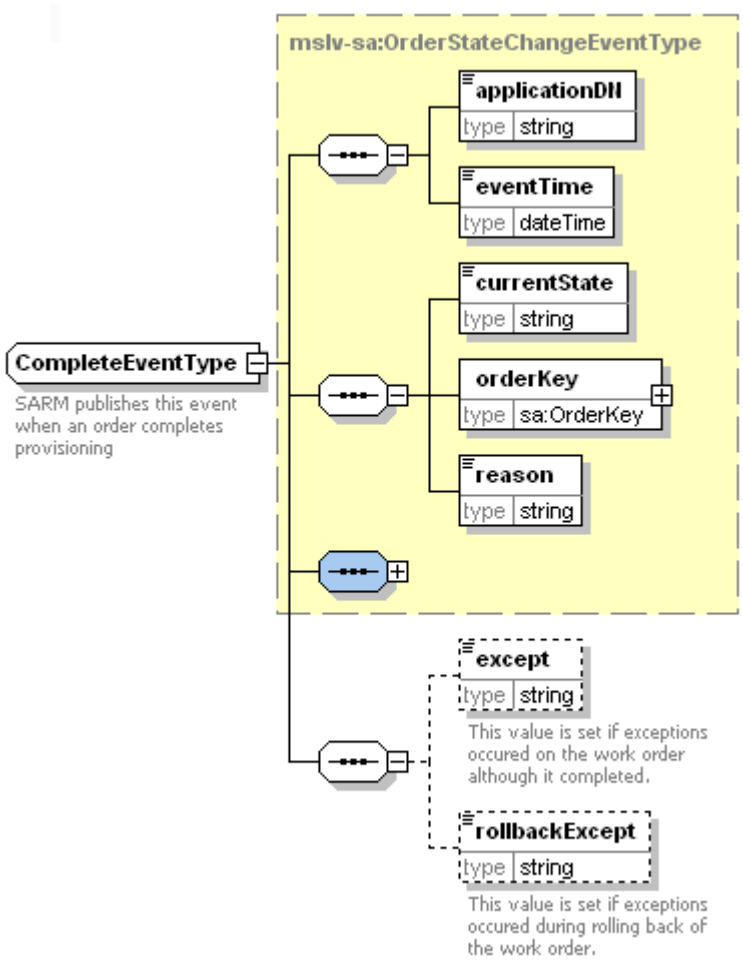
After ASAP is installed, you can access the schema files in the *ASAP_Home***/xml/xsd** directory.

Extended Work Order Complete and Failure Schemas

The work order complete event (**CompleteEventType**) schema type includes the extended tags - **failedServices**, and **services**.

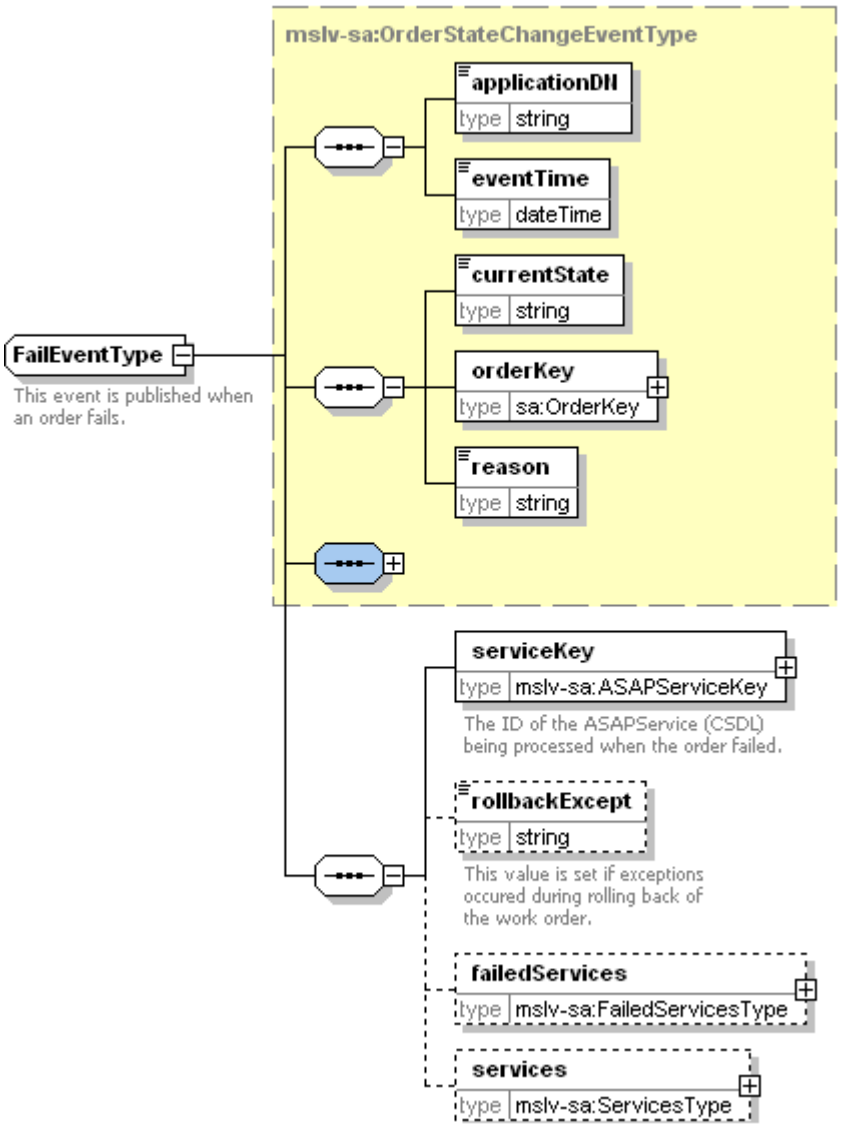
The failed event (**FailEventType**) schema type is extended as shown to include the two new tags **- failedServices**, and **services.**

Figure 10-1 Work Order Complete Event Schema



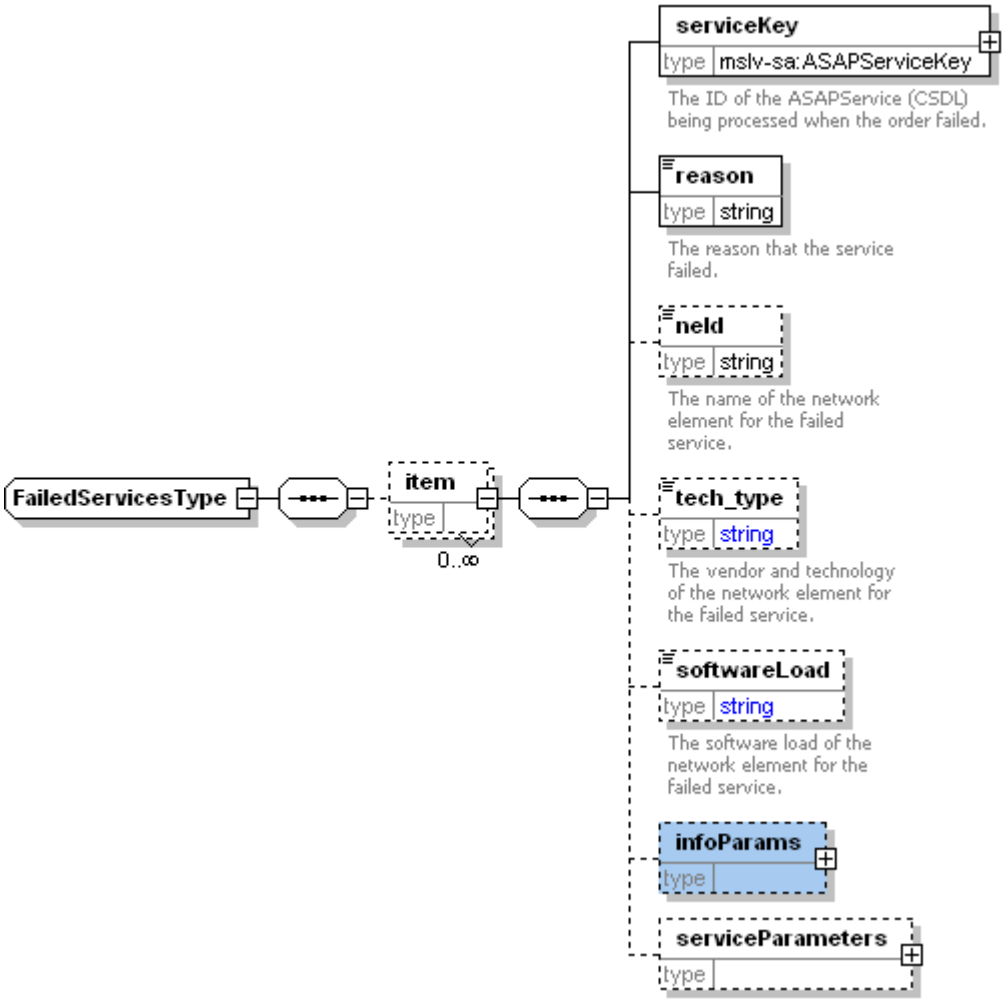
The failed event (**FailEventType**) schema type is extended as shown to include the two new tags - failedServices, and services.

Figure 10-2 Work Order Failed Event Schema



FailedServicesType Schema Type

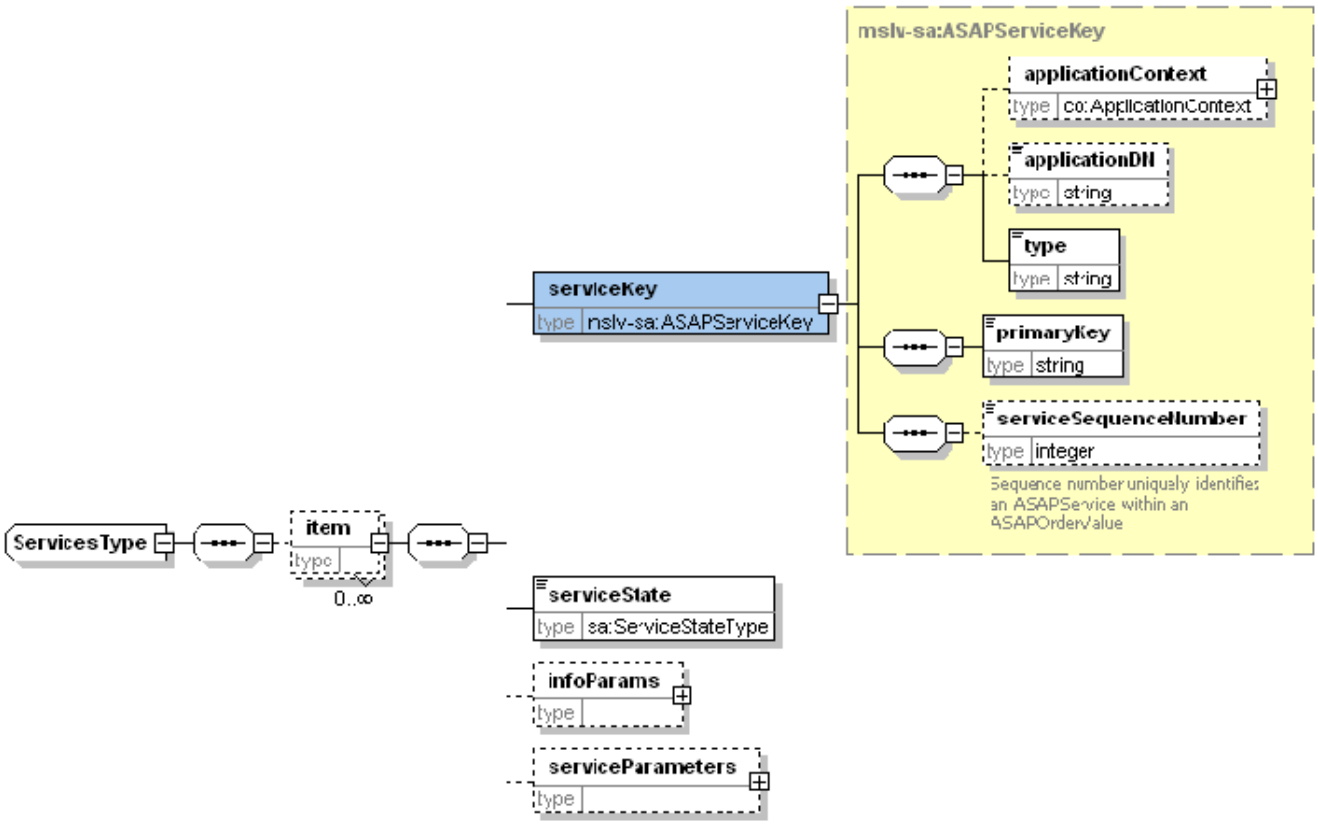
The **failedServicesType** tag contains information detailing a failed work order's services. A failed service has the new fields **reason**, **neId**, **tech_type** and **softwareLoad**. These fields give the reason the work order failed failure for the service specified by **serviceKey**, and identify the NE that a network action (for example, atomic action, atomic action) was executing when the failure occurred.



Services Schema Type

The **ServicesType** tag includes details on services for a work order, except those that failed. As shown, each service inside the **ServicesType** tag includes a tag **serviceState**, which contains the state of the service.

The information parameters (**infoParams**) and service parameters (**serviceParameters**) are shown within the related service (instead at the order level as with previous releases).



Controlling the Return of Enhanced Event Information with includeServiceActionDetail

The work order user property **includeServiceActionDetail** is used to control the inclusion of the work order complete (CompleteEvent) and failure (FailedEvent) types.

If includeServiceActionDetail is true, the **failedServices** and **services** information will be included in the work order complete and failure events. If includeServiceActionDetail is false or if includeServiceActionDetail does not exist (the default), then the extra information is not included in the work order complete and failure events.

For example:

```
<mslv-sa:extendedWoProperties>
<mslv-sa:extendedWoProperty>
<mslv-sa:name>includeServiceActionDetail</mslv-sa:name>
<mslv-sa:value>true</mslv-sa:value>
</mslv-sa:extendedWoProperty>
...
<\mslv-sa:extendedWoProperties>
```

JSRP Server Configuration Parameter INCLUDE_SERVICE_ACTION_DETAIL

The JSRP server configuration parameter **INCLUDE_SERVICE_ACTION_DETAIL** controls this feature in addition to the work order user property **includeServiceActionDetail**.

If the JSRP server configuration parameter is set to **true**, then the **failedServices** and **services** information will be included in every work order complete, failure, or timeout event. The work order user property will override the JSRP server configuration parameter. The JSRP server configuration parameter is defined in the deployment descriptor for JSRP in the deployed **ASAP$env_id.ear** file.

Additional Event Data

With augmented event data, the work order properties, infoparms, global work order parameters, and service action return parameters can be returned on an ASAP event.

This additional event data and the contents of the additional event data are controlled by template entries. The extra parameter information is sent from the SARM to the JSRP, eliminating the need for the JSRP to perform additional queries to the database. Additionally, the SRT is able to add XML event data to the JMS header properties.

Refer to *ASAP Developer's Guide* for schema and other information.

OSS/J Support by Schema Parameters

The ASAP JSRP supports the following:

- The co:type and sa:primaryKey tags of the sa:serviceKey tag in work orders are OSS/J compliant - the name of the service is provided by the tag co:type and the service instance number is provided by the tag sa:primaryKey.
- Soft failures (that is, exceptions) and rollback exceptions are provided based on a service (for example, Service Action), in addition to the work order level and rollback exceptions.
- You can specify the service sequence numbers for a work order. (Previous versions of ASAP number the services according to the order in which they are put inside a work order.)

> **Note:**
>
> Service and failed service information is only provided only for work order complete, failure, and timeout events.
>
> The enhancements to the events apply only to events processed through JSRP servers.
>
> The network information is provided only for failed services that indicate the last network communicated with when the service failed.

Work Order Property includeServiceActionDetail

The work order user property **includeServiceActionDetail** controls the extension of the work order complete, failure, and timeout event types with the two extended tags.

If the value is **true**, the **failedServices** and **services** information are included in the work order complete, failure, and timeout events. If the value is **false** or such a property does not exist (the default), then this extra information is not included in the work order complete, failure, and timeout events.

JSRP Server Configuration Parameter USE_ORIGINAL_INSTANCE_NUMBER

When the value of USE_ORIGINAL_INSTANCE_NUMBER is set to true, the **<co:type>** tag should be populated with service detail. The USE_ORIGINAL_INSTANCE_NUMBER parameter can be found in the **ejb-jar.xml** file in **ASAP$env_id.ear:srp** and in the **ejb-jar.xml** file in **SRT.ear**. Ensure that the values in both files match. For more information, see *ASAP Server Configuration Guide*.

11

Creating Java Connection Handlers

This chapter describes how to create Java implementations for network element (NE) connections and atomic action scripts that implement MML commands for Oracle Communications ASAP.

The following sections provide information about the Java connection handler:

The NE Connection Handlers with Java implementation manage the connections with network elements based on the communication parameters in an NE Template.

About Java Network Element Connection Handlers

The Java implementation NE Connection Handler needs to implement the `IConnectionHandler` interface, which provides a common interface for interacting with connections and requires few methods to be written.

Different types of NE Connection Handlers can be created:

- Telnet: When you create a new telnet NE Connection Handler, it generates code for telnet connections. This extends the telnet connection to support the interface. The NE Connection Handler editor indicates where additional code is required.
- Custom: Create this NE Connection Handler if the connections are not telnet. Custom Connection Handlers generate a skeleton to implement the `IconnectionHandler` and extends the base NE connection class. The NE Connection Handler editor indicates where additional code is required.

Creating New Network Element Connection Handlers

You use the NE Connection Handler Wizard to create new NE Connection Handler entities.

To create a new NE Connection Handler entity:

1. Select **Studio**, select **New**, select **Activation**, then select **NE Connection Handler**.

The NE Connection Handler Wizard appears.

2. Select the project for this element and enter a name for the entity.
3. (Optional) Select a location for the entity.

   By default, Design Studio saves the entity to your default workspace location. You can enter a folder name in the **Folder** field or select a location different from the system-provided default. To select a different location:

   a. Click the **Folder** field **Browse** button.
   b. Navigate to the directory in which to save the entity.
   c. Click **OK**.
4. Click **Finish** to create the NE Connection Handler.

Generating a Telnet Network Element Connection Handler Implementation

You need to generate a Telnet Network Element Connection Handler implementation if you want to extend a telnet connection to support the interface.

To generate a Telnet NE Connection Handler Implementation:

1. Create an NE Connection Handler with the NE Connection Handler Wizard.

   See "Creating New Network Element Connection Handlers" for more information.
2. In the Cartridge view, double-click the entity to open the NE Connection Handler editor.
3. In the editor, enter a description and select **Java Connection Handler** as the NE Connection Handler type.
4. Click **Add**.

   The **Vendor, Technology**, and **Software Load** fields are populated.
5. Click **New**.

   The Studio Activation Java Connection Handler Wizard appears.
6. Ensure that **Telnet** appears in the **Connection Type** field.

> **Note:**
>
> Ensure that a dot does not precede the package name. If a dot precedes the package name, remove it.

7. Click **Finish**.

   The code is generated ready for implementation.

> **Note:**
>
> The code is generated once but not synchronized (that is, it will not be rewritten and the developer owns the generated class).

Generating a Custom NE Connection Handler Implementation

Generate a custom NE Connection Handler implementation if you want to extend the base NE connection class of a connection other than telnet.

To generate a custom NE Connection Handler implementation:

1. Create an NE Connection Handler with the NE Connection Handler Wizard.

   See "Creating New Network Element Connection Handlers" for more information.
2. In the Cartridge view, double-click the entity to open the NE Connection Handler editor.
3. In the editor, enter a description and select **Java Connection Handler** as the NE Connection Handler type.
4. Click **Add**.

   The vendor, technology, and software Load fields are populated.
5. Click **New**.

   The Studio Activation Java Connection Handler Wizard appears.
6. In the **Connection Type** field, select **Custom**.

> **Note:**
>
> Ensure that a dot does not precede the package name. If a dot precedes the package name, remove it.

7. Click **Finish**.

   The code is generated ready for implementation.

> **Note:**
>
> The code is generated once but not synchronized (that is, it will not be rewritten and the developer owns the generated class).

About Communication Protocol Parameters

Communication parameters enable you to configure the information required to communicate through one of the ASAP-supported device interfaces. When the NEP command processor connects to an NE, these parameters are loaded into memory and used in the connection process. When the NEs are connected, they are loaded as NE program (Java program) variables prior to the execution of each program. This method ensures that the program has access to any user-defined information through the communication parameters.

Communication parameters can also be defined on the work order. These parameters defined on the work order override the statically pre-configured values contained in an ASAP cartridge. This feature is used for dynamic NE routing (see "Configuring Dynamic Routing ").

The NEP supports the following interfaces to the downstream network:

- Dedicated and dialup serial

- TCP/IP socket (standalone and in conjunction with the JInterpreter)
- Telnet
- Hostpad device interfaces
- LDAP, as the standard for uniform access to directory services.
- CORBA over IIOP (in conjunction with the JInterpreter)
- CAPI
- X.25, X.29, and TL1
- SFTP

Refer to the *ASAP Developer's Guide* for information on the action functions that support for these interfaces.

Interfaces using such technologies can be developed rapidly due to their script-driven nature, requiring little or no additional software development. These supported network interfaces allow telecommunications carriers to interface with external systems using simple scripts, thereby isolating end users from specific communication details.

Specifying Global or Local Communication Parameters

Using Design Studio, you can specify global communication parameters that apply to all connections to a particular NE. You can also customize a connection with local parameters that apply only to that connection.

Typically, a parameter is defined for a specific host NE and connection. If the various host NEs and connections share the same parameter values, however, the number of communication parameter entries can be reduced.

For example, you must define 33 mandatory X29 Pad interface-specific communication parameters for the X29 Pad interface. Most of these parameters have the same value. Defining each parameter separately for every X29 connection to each host NE results in the following:

- Considerable effort to configure these parameters one at a time.
- Additional memory resources required by the NEP to maintain these parameters in memory.

You can resolve these issues by defining one common set of parameters for all X29 connections to avoid repetition. Specifically, for a particular device interface, you can define parameters with the following groupings:

- All host NEs and all connections (common host, common device).
- All host NEs and a specific connection (common host, specific device).
- A specific NE and all connections to that NE (specific host, common device).
- A specific NE and a specific connection (specific host, specific device).

These parameters are processed in the order they are listed. They override any previous entries defined for the host NE and the device of a particular command processor in the NEP. Communication parameters defined on the work order override preconfigured values if the NEP is configured for dynamic NE routing.

User-defined Parameters

Communication parameters are available to every Java method that is running. You can specify various parameters that can be host NE or device-specific on the Java provisioning

method, and then the Java provisioning methods can employ host NE or device-specific processing.

Device-specific Interface Parameters

Typically, a parameter is defined for a specific host NE and connection. If the various host NEs and connections share the same parameter values, those values can be defined once to avoid repetition.

To communicate with the NE, ASAP opens a connection through the device interface, writes data to the device, and reads data from the device using I/O-related communication parameters.

The following communication parameters apply only to serial, telnet, and other terminal-based interfaces:

- Terminal based interface communication parameters.
- Serial interface communication parameters.
- Telnet interface communication parameters.
- Socket interface communication parameters.
- Generic interface communication parameters.

The following sections describe device interface types which are associated with mandatory parameters.

Table 11-1 illustrates the device types associated with each interface type:

Table 11-1 Interface – Device Type Matrix

Interface	Device Types	Applies to
Terminal-based communication devices	G – Generic Port Terminal-Based	Java
Terminal-based communication devices	T – Telnet Port	Java
Message-based communication devices	S – Socket Port	Java
Message-based communication devices	F - FTP Port for SFTP	Java
Message-based communication devices	W – LDAP Port	Java
Message-based communication devices	C – CORBA	Java

Table 11-2 describes the communication parameters that apply for terminal based interfaces.

Table 11-2 Common Terminal-based Communication Parameters

Parameter	Default	Description
VS_WIDTH	-	Virtual Screen width.
VS_LENGTH	-	Virtual Screen length.
VS_CRLF_MAP	-	A boolean flag that you can set to map LF to CR_LF automatically. The default is set not to map.

Table 11-2 (Cont.) Common Terminal-based Communication Parameters

Parameter	Default	Description
GR_WAIT_TIMEOUT	-	The wait timeout period, in seconds, that the thread reading from the Virtual Screen waits for the thread writing to the Virtual Screen to notify it of any new data. Increase this value if the processing fails before data arrives from the NE.

Generic Port Terminal Based and Generic Port Message-Based are specific to EDDs. For information on the communication parameters for these device types, see the discussion about generic EDD API parameters in the *ASAP Server Configuration Guide*.

tbl_comm_param contains communication parameters required for the NEP to communicate with various external systems. You must populate this table to configure communication parameters.

For more information about these parameters, see the discussion about NE API parameters in the *ASAP Server Configuration Guide*.

To set up a control connection to the appropriate server, you must also set up the communication parameters in the SARM database table **tbl_comm_param**.

CORBA support is offered for Java provisioning only. There is no **CORBA_IF_SUPPORTED** variable in **ASAP.cfg**.

The following sections describe the required parameters for each interface.

CORBA Interface Communication Parameters

The **CORBAConnection** class provides basic functionality to connect to NEs with CORBA interfaces. The custom classes can inherit from this class and implement the **connect()** and **disconnect()** methods.

The **CORBAConnection** class provides a wrapper around an ORB. The initialization of the ORB can be customized by extending **CORBAConnection** and overriding the functions **getInitialArguments** and **getInitialProperties**. Both of these values are used in the ORB.init call implemented in the connect method. The default is to use null for both arguments and properties, which loads the ORB provided by the JRE.

The “C" device type is used in **tbl_comm_param** and **tbl_resource_pool** to relate communication parameters to a CORBA device type.

In situations where the NEP is configured for LOOPBACK, all of the operations on **CORBAconnection** return success.

For more information on Class **CORBAConnection** and com.mslv.activation.jinterpreter, refer to the *ASAP Java Online Reference*.

Serial Port Hardwired Communication Parameters

The NEP server provides built-in support for serial port hardwired communications interfaces.

Table 11-3 lists and describes the serial port hardwired communication parameters that are in addition to the common terminal-based parameters described in Table 11-2.

Table 11-3 Serial Port Hardwired Communication Parameters

Parameter	Default	Description
TTY	Not Applicable	UNIX or Linux port. For a hardwired interface, this value is specific to each host. For a dialup interface, this value remains the same for all host NEs.
DIALUP_NUM	Not Applicable	Dialup number. This parameter is required only for a dialup interface and is different for every NE.
OPEN_TIMEOUT	Not Applicable	The wait timeout period, in seconds, that ASAP waits to open the device. The wait timeout parameter is only applicable to the serial interface.
WRITE_TIMEOUT	5	The wait timeout period, in seconds, that ASAP waits to write to the device.
READ_TIMEOUT	Not Applicable	The wait timeout period, in seconds, that ASAP waits to read from the device. Currently, this is only applicable to the socket interface.
DISABLE_PORT_ON_LOGIN	Not Applicable	Determines whether the port should be disabled if login to the NE fails. If the parameter is equal to zero, then the port is not disabled.
BAUD	Not Applicable	Baud rate for transmission. The valid values are ‘300', ‘600', ‘1200', ‘2400', ‘4800', ‘9600', and ‘19200'.
PARITY	Not Applicable	The parity, which can be either odd, even, or no parity. Enter ‘O' for odd, ‘E' for even, and ‘N' for no parity.
STOP	Not Applicable	Number of stop bits per character. The valid values are ‘1' and ‘2'.
SIZE	Not Applicable	Number of bits per character. The valid values are ‘5', ‘6', ‘7', ‘8'.

Serial Port Dialup Communication Parameters

The NEP server provides built-in support for serial port dialup communications interfaces.

Table 11-4 lists and describes serial port dialup communication parameters, in addition to the common terminal-based parameters described in Table 11-2.

Table 11-4 Serial Port Dialup Communication Parameters

Parameter	Default	Description
TTY	Not Applicable	UNIX or Linux port. For a hardwired interface, this value is specific to each host. For a dialup interface, this value remains the same for all host NEs.
DIALUP_NUM	Not Applicable	Dialup number. This parameter is required only for a dialup interface and is different for every NE.
OPEN_TIMEOUT	Not Applicable	The wait timeout period, in seconds, that ASAP waits to open the device. The wait timeout parameter is only applicable to the serial interface.
WRITE_TIMEOUT	5	The wait timeout period, in seconds, that ASAP waits to write to the device.

Table 11-4 (Cont.) Serial Port Dialup Communication Parameters

Parameter	Default	Description
READ_TIMEOUT	Not Applicable	The wait timeout period, in seconds, that ASAP waits to read from the device. Currently, this is only applicable to the socket interface.
DISABLE_PORT_ON_LOGIN	Not Applicable	Determines whether the port should be disabled if login to the NE fails. If the parameter is equal to zero, then the port is not disabled.
BAUD	Not Applicable	Baud rate for transmission. The valid values are ‘300', ‘600', ‘1200', ‘2400', ‘4800', ‘9600', and ‘19200'.
PARITY	Not Applicable	The parity, which can be either odd, even, or no parity. Enter ‘O' for odd, ‘E' for even, and ‘N' for no parity.
STOP	Not Applicable	Number of stop bits per character. The valid values are ‘1' and ‘2'.
SIZE	Not Applicable	Number of bits per character. The valid values are ‘5', ‘6', ‘7', ‘8'.

Telnet Port Communication Parameters

The NEP server provides built-in support for a TCP/IP Telnet communications interface. You can enable and configure the NEP Telnet driver to communicate with NEs using the standard Telnet terminal emulation.

ASAP also contains a Java telnet library. A virtual screen implementation is provided to simplify data manipulation.

NEConnection is an abstract class defined in package jinterpreter. All JInterpreter connection classes must extend this class in order to be evocable by the NEP. Oracle Communications provides the **TelnetConnection** class, which integrates the telnet and virtual screen implementations provided by the telnet library.

The **TelnetConnection** class also supports a piped stream interface, similar to the raw input stream available from the underlying TCP/IP connection. The read, write and **waitfor** operations defined on **TelnetConnection** act on the stream to retrieve and send data. This interface leaves the incoming data in a stream format for simple parsing scenarios. In simple parsing situations, a provisioning activity may only need to pick off a simple response string from the NE. In these situations, it can be simpler to use a **waitfor** call to track the response from the NE rather than use the structured format of the virtual screen.

By default, both the virtual screen and piped stream are enabled by the **TelnetConnection** class. The method **TelnetConnection.setStreamEnabled** (boolean enabled) can be used to enable or disable the stream.

In situations where the NEP is configured for **LOOPBACK**, the **InputStream** and **OutputStream** returned by the **StreamConnection** always return success for every read and write call. The **InputStream.read** methods return a size read integer of 1 with the value set to an empty character ' '. All of the **TelnetConnection** send and VirtualScreen get/read calls always return success.

For information on the JInterpreter API for the Telnet connection class, refer to the *ASAP Java Online Reference*.

Table 11-5 lists and describes the Java telnet port communication parameters for the JInterpreter, in addition to the common terminal-based parameters described in Table 11-2.

Table 11-5 Telnet Port Communication Parameters for the JInterpreter

Parameter	Default	Description
HOST_USERID	Not Applicable	User name.
HOST_PASSWORD	Not Applicable	Password.
OPEN_TIMEOUT	5 seconds	Connection establishment timeout (in seconds).
READ_TIMEOUT	1 second	Timeout for the telnet read functions (in seconds).
HOST_NAME	Not Applicable	Machine name for the host NE.
HOST_IPADDR	Not Applicable	Network IP address for the host NE.
PORT	23	Telnet service port. If SSH_SUPPORT is set to NO, then the PORT value applies. If SSH_SUPPORT is set to YES, the SSH_PORT value applies.
LOGIN_PROMPT	login:	Reserved. The login prompt expected in the telnet session.
PASSWORD_PROMPT	Password >	Reserved. The password prompt expected in the telnet session.

Note:

The default **TelnetConnection** class uses parameters defined in **tbl_comm_param** (**VS_LENGTH**, **VS_WIDTH**, **HOST_USERID**, **HOST_PASSWORD**, **HOST_NAME**, **HOST_IPADDR**, **PORT**) to initialize the telnet session. A solutions developer may use the provided **TelnetConnection** class as a connection handler as configured in **tbl_nep_asdl_prog**. It is also possible to extend the **TelnetConnection** class to override the provided functionality. For instance, a solutions developer may wish to override the connect, login or disconnect methods to implement custom functionality.

Refer to the Common Terminal-based Communication Parameters table for more information on virtual screen-related parameters.

SSH Telnet Communication Parameters

Table 11-6 contains the communication parameters for the SSH protocol in addition to the common terminal-based parameters described in Table 11-2.

Table 11-6 SSH Communication Parameters

Parameter	Default	Description
HOST_NAME	Not Applicable	The machine name of the NE.
HOST_IPADDR	Not Applicable	The IP address of the NE.
SSH_SUPPORT	NO	Indicates if SSH is supported. The valid values are YES or NO. If SSH_SUPPORT is set to NO, then the PORT value applies. If SSH_SUPPORT is set to YES, the SSH_PORT value applies.
SSH_PORT	22	The SSH port number.
SSH_VERSION	SSH2	The SSH version; either SSH1 or SSH2.

Table 11-6 (Cont.) SSH Communication Parameters

Parameter	Default	Description
SSH_AUTH_METHOD	PASSWORD	The authentication method; either PASSWORD or PUBLIC_KEY.
SSH_PREF_PUBLIC_KEY	PUBLIC_KEY_SSHRSA	The preferred public key; either PUBLIC_KEY_SSHRSA or PUBLIC_KEY_SSHDSS.
SSH_PREF_CIPHER_CS	CIPHER_BLOWFISH_CBC	The preferred CS cipher; either CIPHER_BLOWFISH_CBC, CIPHER_TRIPLEDES_CBC, TWOFISH128_CBC, TWOFISH192_CBC, TWOFISH256_CBC, TWOFISH_CBC, CAST128_CBC, AES128_CBC, AES192_CBC, AES256_CBC
SSH_PREF_CIPHER_SC	CIPHER_BLOWFISH_CBC	The preferred SC cipher: either CIPHER_BLOWFISH_CBC, CIPHER_TRIPLEDES_CBC, TWOFISH128_CBC, TWOFISH192_CBC, TWOFISH256_CBC, TWOFISH_CBC, CAST128_CBC, AES128_CBC, AES192_CBC, AES256_CBC
SSH_PREF_MAC_CS	HMAC_MD5	The preferred CS message authentication; either HMAC_MD5 or HMAC_SHA1.
SSH_PREF_MAC_SC	HMAC_MD5	The preferred SC message authentication HMAC_MD5 or HMAC_SHA1.
SSH_PREF_COMP_CS	COMPRESSION_NONE	The preferred CS compression COMPRESSION_NONE or COMPRESSION_ZLIB.
SSH_PREF_COMP_SC	COMPRESSION_NONE	The preferred SC compression COMPRESSION_NONE or COMPRESSION_ZLIB.
VS_TYPE	vt100	Virtual screen type.
VS_WIDTH	80	Virtual screen width.
VS_LENGTH	24	Virtual screen length.
SSH_TRANSPORT	SOCKET	Specifies the SSH Transport to be used. Set the value to **SOCKET** for direct SSH connections. Set the value to **HTTP_PROXY** if SSH connections need to be established via an HTTP proxy. The value **SOCKET** indicates SocketTransport and the value **HTTP_PROXY** indicates HttpProxyTransport.
HTTP_PROXY_HOST	NA	The host name of the HTTP proxy server. **Note**: Configure this parameter if **SSH_TRANSPORT** is set to **HTTP_PROXY**.
HTTP_PROXY_PORT	NA	The HTTP proxy port number. **Note**: Configure this parameter if **SSH_TRANSPORT** is set to **HTTP_PROXY**.

Login information for the NE/device needs to be populated in the Control database (**TBL_CLASSB_SECU**) in order for SSH to work. Table 11-7 lists and describes the SSH security parameters for the SSH protocol that should be stored in the Control database. For more information about securely storing NE login information, see "Implementing Secure Login Functionality."

Table 11-7 SSH Security Parameters

Parameter	Default	Description
HOST_USERID	NA	User name to log in to the NE.
HOST_PASSWORD	NA	For telnet: the password authentication: user password to login the NE. For SSH: Public key authentication: The passphrase used for the private key.
PRIV_KEY_FILE	NA	The machine name of the NE
HTTP_PROXY_USER	NA	The user name of the HTTP proxy server. **Note**: Configure this parameter if **SSH_TRANSPORT** is set to **HTTP_PROXY** and HTTP proxy requires authentication.
HTTP_PROXY_PASSWORD	NA	The HTTP proxy password. **Note**: Configure this parameter if **SSH_TRANSPORT** is set to **HTTP_PROXY** and HTTP proxy requires authentication.

Socket Port Communication Parameters

The NEP server provides built-in support for a TCP/IP socket-based communications interface. You can enable and configure the NEP socket driver to communicate with NEs using message-based communication.

A **SocketConnection** class is provided which wrappers a **java.net.Socket** instance. The **SocketConnection** class can be extended to provide custom functionality on top of the conventional socket interface. A **SocketConnection** implements the **StreamConnection** interface. The **StreamConnection** interface defines methods common to stream-based protocols.

By default, the connect method uses the communication parameters defined by **HOST_NAME**, **HOST_IPADDR** and **PORT**.

In situations where the NEP is configured for **LOOPBACK**, the **InputStream** and **OutputStream** returned by the **StreamConnection** always return success for every read and write call. The **InputStream.read** methods return a size read integer of 1 with the value set to an empty character ' '.

For information on the JInterpreter API for the Socket connection class, refer to the *ASAP Java Online Reference.*

Table 11-8 Socket Port Communication Parameters

Parameter	Default	Description
PORT	Not Applicable	Port of the remote socket listener.
OPEN_TIMEOUT	5	The wait timeout period, in seconds, that ASAP waits to open the device. The wait timeout parameter is only applicable to the serial interface.
WRITE_TIMEOUT	5	The wait timeout period, in seconds, that ASAP waits to write to the device.
READ_TIMEOUT	1	The wait timeout period, in seconds, that ASAP waits to read from the device. Currently, this is only applicable to the socket interface.

Table 11-8 (Cont.) Socket Port Communication Parameters

Parameter	Default	Description
DISABLE_PORT_ON_LOGIN	0	Determines whether the port should be disabled if login to the NE fails. If the parameter is equal to zero, then the port is not disabled.
SOCKET_CLIENT	Not Applicable	Socket server or client. The only valid value is 'C' because the communication is a Socket client.
HOST_NAME	HOST_CLLI	Machine name for the host NE.
HOST_IPADDR	Not Applicable	Network IP address for the host NE.
SOCKET_FAMILY	2	The only valid value is '2' because only the Internet address family is supported.

SFTP Port Communication Parameters

The SftpConnection class provides basic functionality to connect to NEs with SFTP interfaces. It can be used to perform standard SFTP commands like 'cd', 'get', and 'put'. For more information on Class **SftpConnection** and **com.mslv.activation.jinterpreter**, refer to the ASAP *Java Online Reference*.

Table 11-9 lists the port parameters for SFTP communication.

Table 11-9 SFTP Port Communication Parameters

Parameter	Default	Description
HOST_USERID	None	User name.
HOST_PASSWORD	None	Password associated with HOST_USERID
HOST_NAME	HOST_CLLI	Machine name for the host NE.
HOST_IPADDR	Not Applicable	Network IP address for the host NE.
PORT	21	Telnet service port.

LDAP Port Communication Parameters

The NEP server enables ASAP to communicate with LDAP (Lightweight Directory Access Protocol) Directory Servers through the LDAP protocol using TCP/IP. Connectivity to LDAP Directory Servers (NEs) is provided by the Multi-Protocol Manager. The LDAP interface allows inquiries, additions, modifications, and deletions of records stored in LDAP-enabled directories. The LDAP interface is implemented using Version 3 of the LDAP protocol.

The **LdapConnection** class provides a wrapper around a **netscape.ldap.LDAPConnection** class. As with **SocketConnection**, it provides a simple interface for returning the underlying **netscape.ldap.LDAPConnection** class for manipulation. A solutions developer is free to extend the default **LdapConnection** class to implement custom functionality.

For more information on Class **LdapConnection** and **com.mslv.activation.jinterpreter**, refer to the *ASAP Java Online Reference*.

In situations where the NEP is configured for **LOOPBACK**, all operations on **LdapConnection** return success.

HOST_NAME or **HOST_IPADDR** represent a **hostname** to which to connect or a dotted string representing the IP address of this host.

Table 11-10 LDAP Port Communication Parameters

Parameter	Default	Description
HOST_USERID	None	User name.
OPEN_TIMEOUT	5	The wait timeout period, in seconds, that ASAP waits to open the device. The wait timeout parameter is only applicable to the serial interface.
WRITE_TIMEOUT	5	The wait timeout period, in seconds, that ASAP waits to write to the device.
READ_TIMEOUT	1.0	The wait timeout period, in seconds, that ASAP waits to read from the device. Currently, this is only applicable to the socket interface. 0 = no timeout.
DISABLE_PORT_ON_LOGIN	Not Applicable	Determines whether the port should be disabled if login to the NE fails. If the parameter is equal to zero, then the port is not disabled.
HOST_NAME	HOST_CLLI	Machine name for the host NE.
HOST_IPADDR	Not Applicable	Network IP address for the host NE.
PORT	389	the TCP or UDP port number to which to connect or contact.
LDAP_VERSION	2	LDAP version to use. If VERSION2, use 2. If VERSION3, use 3.
SIZELIMIT	2	The size of the search results set asked from the directory server. Minimum 1, Maximum 500.

TL1 Port Communication Parameters

TL1 is a communication standard for specifying information exchanges between Operations Support Systems (OSSs) and NEs. Several NEs and/or Element Management Systems use TL1 for communication with external systems. TL1 can be used in conjunction with Telnet or X.25 protocols.

StreamConnection Interface

The **StreamConnection** interface allows a solutions developer to write JProcessor implementations independent of specific protocol APIs. This means that a single JProcessor implementation can reference only **StreamConnection** interface methods, and be able to switch underlying connection handler classes such as Socket and Telnet without having to modify the provisioning code.

All stream-based protocols such as socket and telnet implement the **StreamConnection** interface. See "Telnet Port Communication Parameters" and "Socket Port Communication Parameters" for more details.

For more information on the **StreamConnection** interface, refer to the *ASAP Java Online Reference*.

Creating Connection Methods and Helper Classes

Connection methods are used by ASAP core to establish a connection (also referred to as a device) and/or to login to an NE.

To implement a connection method and connection handler, you need to know:

- The NE activation interface and protocol
- The logic and parameters required for connecting and disconnecting to and from the NE
- Knowledge for implementing client protocols such as HTTP, web service, CORBA, TCP/IP
- Knowledge about third party libraries or frameworks required to implement a non TCP/IP protocol. For example, Apache AXIS for WSDL defined web service, Apache Common HttpClient for HTTP, and so on.
- If any secure data must be stored in the ASAP database

In Design Studio, a Java class maps to a connection handler (see "Creating an Network Element Connection Handler"). This Java class must implement the **IConnectionHandler** interface that provides a common interface for the various protocols. It contains some standard methods such as **connect ()** and **disconnect ().** The business logic depends on the NE interface protocol and connection/disconnection handshake sequence.

For non-telnet based cartridges the connection class should typically contain a **connect()** method and a **disconnect()** method. Helper methods are implemented to get at variables or objects that are stored by the connection class. For example the prompt is picked up from **tbl_comm_param** and when a provisioning method needs to get at it, it will invoke a method of the connection class to get the provisioning prompt. There may be helper methods for getting username, password, and so on. Sometimes a **send()** method may be implemented and called from the provisioning method after getting a reference to the connection object.

In case of a Telnet interface to the NE, Design Studio auto generates the skeleton code for the connection handler. The **TelnetConnection** class from the core framework is extended by the cartridge. The **login()** and **disconnect()** methods are implemented (the **connect()** method as supported in the core can be re-used as is). For telnet based cartridges there is also no need to implement **send()** and **waitfor()** methods because they are available in the core framework.

Other protocols, such as SOAP XML, TCP/IP, web services, CORBA, and so on) depend much more on the server side implementation; therefore more code has to be written to handle connections to these NEs. Write a dummy client outside the ASAP environment to test connectivity. Many of the provisioning guides give sample code illustrating how to connect and provision a service. After that is tested in a standalone mode, it can be ported into an ASAP cartridge because Java is platform independent.

Other cartridges that use generic protocols need to implement their own **send()** and **waitfor()** methods. Sometimes (for example with Soap/XML protocols) establishing a connection on the URL to the remote server does not guarantee that the connection is usable. In this case an actual query message for a non-existing subscriber is made within the connection class itself to ensure that the connection is valid. If this query fails, then a **ConnectionException()** is thrown back so that the connection can be retried instead of all the provisioning orders failing.

Creating a Provisioning Prompt

Where possible (for example for certain TCP/IP based protocols such as telnet) checking that the correct prompt and level are present should be performed before each command is sent to the NE. This should be implemented as a separate callable method. In addition a method

should be provided to be able to obtain the correct prompt and/or level in case an error has occurred.

Enabling Loopback Mode

When using the standard core ASAP send() and get() Java methods no additional loopback code should be required to be implemented in the cartridge because the standard loopback mechanism takes care of providing the exact responses requested.

Implementing Secure Login Functionality

In the current ASAP implementation login information for NEs is stored in **tbl_comm_param** in "clear" format. This makes it possible for sensitive data to be easily accessible by un-authorized persons (ASAP also automatically displays communication parameters in diagnostic files). It is very important to be able to store this type of data in a non-readable (encrypted) format.

There are two aspects to security: secure data storage and secure data encryption. The cartridge must be able to accommodate both:

1. Secure Data Storage - There are two types of data: ASAP secure data and custom secure data, which are identified by two class types (0- ASAP data, 1- custom data). ASAP secure data is stored in credential store factory (CSF) wallet located in *ASAP_Home*/**install/cwallet.sso** and custom secure data in **tbl_classB_secu** table in the control database. **tbl_classB_secu** allow entries in a name/value format with other fields for class type, security level, caching of data etc. The layout of this table is as follows:

```
SQL> desc tbl_classB_secu;
Name                                      Null?    Type
----------------------------------------- -------- ----------------------------
NAME                                      NOT NULL VARCHAR2(80)
VALUE                                     NOT NULL VARCHAR2(255)
CLASS                                     NOT NULL NUMBER(38)
S_CACHE                                   NOT NULL NUMBER(38)
C_DATE                                    NOT NULL DATE
DESC1                                              VARCHAR2(255)
```

2. Secure Data Encryption – Custom secure data can be stored either in "clear" or in "encrypted" format. ASAP secure data is always encrypted.

To load the class B data, which contains NE access information, use the following steps:

1. Create an input file that contains the data to be stored in **tbl_classB_secu**. For example:

```
################################################
#
# This info is added to 'tbl_classB_secu' entries -
# to be added using asap_security_tool
#
# Entries format: NAME:VALUE:CLASS:S_CACHE:DESCRIPTION
#
# Example: DMS_USER:user123:1:0:User name for DMS100 access
#
################################################
#
USER_NOKIA:username:1:0:Login name for Nokia HLR
PASS_NOKIA:password:1:0:Password for Nokia HLR
#
```

2. Load the content of this file into control database using **asap_security_tool** utility:

```
asap_security_tool -r <secure data input file>
```

To retrieve the secure data from the tables within the cartridge Java code use the methods provided in the **Security** Java class (see **Java Online Reference**). The following sample code shows how the encrypted user ID and password are retrieved from the secure tables:

```
logger.logDebug("Getting access secure data");
Security sec = ASCAppl.getSecurity();
try {
     String secUsername = sec.getSecureData("USER_NOKIA", 1);
     logger.logDebug("Retrieved secure user name");
     String secPassword = sec .getSecureData("PASS_NOKIA", 1);
     logger.logDebug("Retrieved secure password");
} catch (SQLException e) {
     logger.logDebug("Exception caught while retrieving secure data: " + e);
}
```

After being retrieved, this data is automatically decrypted and ready to be sent to the NE. Make sure that this data is not written into the cartridge diagnostics. Display 10 asterisks instead (the number of asterisks should not match the actual length of the password).

Connection Management Issues

Never fail a work order due to connection failure in the case where connection management (for example corba connections) is supported within the cartridge code. Orders are only failed when the NE returns an error message indicating that data on the order is invalid.

Where possible avoid explicitly disabling connections from within the cartridge code. ASAP core handles the disabling of connections when the connection class exits with failure.

When communication parameters necessary to establish a connection are missing (as determined in the connection class for the cartridge) the cartridge must log a meaningful error message to the diagnostic files to indicate which parameter is missing and what the expected parameter is used for.

A new and improved Java SEND method has been implemented in ASAP core which will not force the calling cartridge code to handle exceptions (for example IOException or TelnetException), but will manage these exceptions internally. When connections go down atomic actions should be put back in the appropriate queue and rescheduled by ASAP automatically. The core should manage disabling and re-enabling the device accordingly.

In some cases (with certain TCP/IP-based cartridges and possibly others) certain delays are incurred when connecting and/or logging into NEs (reference the Ericsson MSS-C cartridge). In such cases implement a communication parameter (tbl_comm_param) that allows for a delay (thread.sleep()) interval to be configured in the field. It should be possible to set this to 0 so that no delay is incurred.

Creating a Java Telnet Connection Class

This section describes the steps to create a Java Telnet connection class. Use these steps as a guideline for constructing your own cartridge.

To create a Java Telnet connection class:

1. Create an NE as described in "Creating and Configuring Network Element and Network Element Connections" with the following exceptions:
   a. In the **Protocol** field, select the **Telnet/SSH**.
   b. When adding a connection, accept the autogenerated parameters.

c. Select the **All Communications Parameters** tab.

d. Modify the communication parameter values that your connection requires.

e. Click **Add** Global to add any additional parameters that your connection requires.

f. Edit the **Label**, **Value** and **Description** columns to specify the new parameter. For example:

- In the **Label** field, enter **PROMPT**.
- In the **Value** field, enter #.
- In the **Description** field, enter **This value defines the initial prompt symbol**.

2. Create a Network Handler as described in "Creating an Network Element Connection Handler" with the following exceptions:

a. From the **Connection Type** list, select the **Telnet**.

b. In the **Class** field, click **New**.

The Studio Activation Java Connection Handler wizard appears.

c. In the **Name** field, enter a name for the connection handler.

d. In the **Connection Type** list, select the **Telnet**.

e. Click **Finish**.

The *connection_handler_name***.java** file opens in Design Studio (where connection_handler_name is the name of the connection handler).

3. In the *connection_handler_name***.java** file, get the connection parameters for any parameters you added in addition to the autogenerated parameters. For example:

```
try {
            String n_prompt = getCommParam("PROMPT");
            setPrompt(n_prompt);
```

This sample uses the **getCommParam** method to retrieve data from the **PROMPT** parameter defined in addition to the autogenerated parameters.

> **Note:**
>
> The following method gets all autogenerated parameters:
>
> ```
> super.login();
> ```

4. Add code to wait for the login prompt. For example:

```
-- Wait for the login prompt from the network element.
 if (login_prompt != null)
     waitfor(login_prompt);
 else
     waitfor("login:");
```

5. Add code to send the username. For example:

```
 this.sendln(userid);
```

6. Add code to wait for the password prompt. For example:

```
 -- Wait for the password prompt. --
 if (password_prompt != null)
     waitfor(password_prompt);
```

```
else
    waitfor("Password:");
```

7. Add code to send the password. For example:

```
this.sendln(password);
```

8. Add code to wait for the unix prompt. For example:

```
 -- Wait for the normal network element prompt. --
if (n_prompt != null)
    waitfor(n_prompt);
else
    waitfor(">");
```

In this example, the prompt was defined using the **PROMPT** parameter with the **=** value. Had this value not been defined, it would have used the default **>** value.

9. Add code to specify an error diagnostic message. For example:

```
} catch (Exception e) {
    Diagnostic.diag(Diagnostic.SANE, this, "Login failed: " +
    e.getMessage());
    throw new TelnetException("Login failed: " + e.getMessage());
}
```

10. Add code to specify a success diagnostic message. For example:

```
        Diagnostic.diag(Diagnostic.SANE, this, "Successfully logged in to the " +
"network element.");
}
```

12

Creating Action Processors and Programs for Processing Requests and Responses

This chapter describes how to create action processors and Java programs for atomic actions that implement man-machine language (MML) commands for Oracle Communications ASAP.

About Action Processors and Programs

The Network Element Processor (NEP) is the ASAP server component that manages interactions with network elements (NEs) and element management systems (EMSs). The NEP receives atomic actions from the service activation request manager (SARM) and uses programs to interact with the NE. Based on the programs, the NEP sends commands to the NE and returns responses from the NE to the SARM.

The NEP must choose the correct program to fulfill the atomic action. To determine which program to use, the NEP uses action processors. Action processors map atomic actions to programs. When you create ASAP cartridges, you can write Java programs from scratch, or configure the action processor to auto-generate Java programs.

This chapter describes the following:

- How to create and configure action processors.
- How to auto-generate Java command line interface (CLI) code and the situations where you need to write custom business logic. Auto-generating CLI code is available for cartridges that use CLI commands, such as TL1 over TCP/IP and Telnet over TCP/IP.
- How to auto-generate Java code stubs and the places where you need to write custom business logic. Auto-generating a Java stub is available for any cartridge type.
- Recommendations for writing Java programs from scratch.
- How to auto-generate unit test cases and the places where you need to write custom business logic.

> **Note:**
>
> While you can write a Java implementation, Oracle recommends that you auto-generate Java stubs. This method provides code that you would normally have to write yourself.

When the NEP receives an atomic action and its parameters from the SARM, the NEP determines what program to run based on the SARM **tbl_nep_asdl_prog** table. This table contains the mappings between atomic actions and programs that is defined in the action processor. The table defines the following columns:

- **asdl_cmd**: The atomic action passed to the NEP interpreter or jinterpreter.

- **tech**: The technology type of the NE that the NEP interpreter or jintepreter interacts with. With Java programs, the values in this column are a combination of vendor and technology separated by a dash (for example, ALU-FTTU).
- **sftwr_load**: The software version of the software currently running on the NE.
- **program**: The Java program that the jinterpreter must run to fulfill the atomic action.
- **interpreter_type**: A value of **J** indicates a Java program.

Table 12-1 shows how the same atomic action can map to various vendors, technologies, and software versions.

Table 12-1 Atomic-Action-to-Program Mappings

asdl_cmd	tech	sftwr_load	program	interpreter_type
CLEAR_INTERCEPT	ALU-DMS	BCS35	com.alu.dms.bcs35.ClearInterceptProxy.execute	J
CREATE_LINE	ALU-DMS	BCS35	com.alu.dms.bcs35.CreateLineProxy.execute	J
SET_OPTION_ON	ALU-DMS	BCS35	com.alu.dms.bcs35.SetOptionOnProxy.execute	J
CLEAR_INTERCEPT	CSCO-INV	1.0	com.csco.inv.1_0.ClearInterceptProxy.execute	J
CREATE_LINE	CSCO-INV	1.0	com.csco.inv.1_0.CreateLineProxy.execute	J
SET_OPTION_ON	CSCO-INV	1.0	com.csco.inv.1_0.SetOptionOnProxy.execute	J

For more information about the NEP and the JInterpreter, see *ASAP Server Configuration Guide*. For more information about **tbl_nep_asdl_prog**, see *ASAP Developer's Guide*.

About the Ratio of Provisioning Commands to Atomic Actions

Whenever possible, map each atomic action to a program containing only one provisioning command. As a general rule, the fewer commands associated with the atomic action, the easier it is to use the atomic action as a building block in the implementation of higher level services. However, in some less common scenarios, several actions must be run on the NE in sequence. In such cases, you map a single atomic action to more than one action.

For example, NEs that require certain modes to be set before a provisioning command can be sent to the NE may need to encapsulate the commands to set the modes along with the provisioning command.

Review the following considerations before deciding whether to encapsulate several commands:

- Determine whether encapsulating the mode commands substantially increases ASAP and router performance.
- Determine whether the service model becomes less complicated when commands are encapsulated. Reducing service model complexity allows the service modeler to focus on implementing service offerings rather than on understanding and modeling mode setting dependencies for every service.
- Determine whether encapsulating mode commands removes the need to have complex mutex logic within the cartridge. For example, multiple ASAP work orders destined to the same NE may result in interleaved atomic actions. In some devices, without implementing mutex logic, atomic actions fail because the router is in an indeterminate mode for any given atomic action.

- Determine whether encapsulating mode commands removes the for implement additional logic for connection handlers. For example, if each atomic action sets its own mode, when a connection to a router is lost, at any point the connection handler would not have to determine whether any mode setting commands must be re-run.

For additional considerations at the service action level, see "About Limiting Independent Network Element Commands to Optimizing the Network Element Interface."

About Creating and Configuring Action Processors

An action processor maps an atomic action to a Java program. For every action processor, you need to define a program as the implementation that performs the work.

The naming convention for action processors is the same as the naming convention for atomic actions, with the exception of the prefix: Action processors use the prefix **I** whereas atomic actions use the prefix **A**. See "About Creating and Configuring Atomic Actions" for information about the naming convention.

Design Studio for ASAP automatically enforces this naming convention when you create an action processor with the Action Processor Wizard.

Creating an Action Processor

To create an action processor:

1. From an Activation project, select **Studio**, then select **New**, then **Activation**, and then **Action Processor.**
2. From the Action Processor Wizard, enter the following:
   - From the **Project** list, select a cartridge project in which to create the action processor.
   - In the **Action** field, the name of the action that the action processor performs (see "Selecting the Action Tokens").
   - In the **Entity** field, enter an entity that is the object of the action (see "Selecting Entity Tokens").
3. Click **Finish**.

   The Action Processor editor appears. From the action processor, you can either auto-generate code or associate the action processor with code that you have written yourself. For more information about these options, see:

   - Understanding the Auto-Generated Java CLI Code
   - Understanding the Auto-Generated Java Code Stubs
   - About Writing Java Programs from Scratch and Naming Conventions
   - Understanding Unit Testing

Understanding the Auto-Generated Java CLI Code

After you create the action processor, you can auto-generate the CLI Java code. You auto-generate the code when you want ASAP to interact with NEs that use CLI-based commands such as TL1 over TCP/IP or Telnet over TCP/IP. The code that is generated sends requests. You can include additional post processing logic if required. You must add business logic to the generated code for receiving responses.

> **Note:**
>
> The CLI code generation option is available for cartridges created or upgraded to ASAP 7.3.2 or later releases. If you are designing a cartridge that is not CLI-based, see "Understanding the Auto-Generated Java Code Stubs."

Before you can auto-generate CLI Java code, you must have created and configured the atomic action associated to the action processor. See "Creating and Configuring Atomic Actions " for instructions.

To auto-generate CLI Java code, you perform the following tasks:

- Configure the default command structure of the CLI commands that ASAP sends to the NE. For more information, see "About the CLI Command Structure Elements."
- The action processor editor **Request** tab. This tab defines the CLI commands that ASAP builds and sends to the NE. You can parse sample CLI commands to generate the elements names that are part of the outgoing command or manually add the command parameters. You can map these command parameters to atomic action parameters or define them as static parameters. For more information, see "Configuring CLI Command Requests."
- The action processor editor **Response** tab. This tab defines a response pattern and the position of the value within the response pattern that matches with an ASAP user exit type. You must also add response handling logic to the generated Java code. For more information, see "About Configuring CLI Command Responses."

Consider the following restrictions when choosing to auto-generate CLI Java code:

- The auto-generated code applies to network cartridges only; not to service cartridges.
- The auto-generated code supports only a one-to-one mapping ratio between the action processor and CLI command.
- If you want to use the CLI code generation function in cartridges developed before Design Studio 7.3, you must delete the old action processor and generate a new action processor. You must manually copy over any descriptive information contained in the old action processor.

About Configuring the CLI Command Structure

After you create an activation project, you can define the Java CLI command structure. The command structure defines the delimiters to use in CLI commands. You define a general command structure for request commands. When you configure the request commands, you can overwrite the general structure if a command you're configuring requires a different structure.

You must enable auto-parsing of CLI commands if you want Design Studio to automatically parse and map CLI command parameters to atomic action parameters when you configure your Java CLI command requests.

When automatically parsed, the command parameters are mapped to the atomic action parameters that have the same parameter names. If a command parameter name does not match any atomic action parameter name, you must manually map that command parameter to an action parameter.

About the CLI Command Structure Elements

After you create an activation project, you can define the CLI command structure. The command structure can have one or more of the following elements:

- **Header Body Separator**: Defines the separator between the header and the rest of the CLI command.
- **Parameter Separator**: Defines the separator between parameters within the CLI command.
- **Parameter Name-Value Separator**: Defines the separator between a parameter and its value.
- **Compound Parameter Encloser**: Defines the separator that encloses compound parameters.
- **Compound Parameter Index Separator**: Defines the separator between members of a compound parameter.
- **Command Tail**: Defines the character at the end of the CLI command.
- **End of Command Control Character**: Defines the command control character or string at the end of the CLI command. For example, a carriage return or a line-feed character.

Each command structure elements can take one of the following values:

- **NONE**: Select this value if the CLI command does not use the element.
- **: COLON**: Select this value if the element should be a colon.
- **, COMMA**: Select this value if the element should be a comma.
- **. DOT**: Select this value if the element should be a period.
- **= EQUAL**: Select this value if the element should be the equals sign.
- **; SEMI_COLON**: Select this value if the element should be a semicolon.
- **SPACE**: Select this value if the element should be a space.
- **CARRIAGE**: Select this value if the element should be a carriage return.
- **NEW LINE**: Select this value if the element should be a new line.
- **Ctrl+C**: Select this value if the element should be the Ctrl+C key combination.
- **OTHER**: Select this value if the element requires one or more characters not specified in this list. When you select OTHER, a field appears next to the list in which you enter one or more special characters. For example, if the **End of Command Control Character** is the word COMMIT, you could specify this using the OTHER option.

Configuring the CLI Command Structure

To configure the Java CLI command structure:

1. Open the activation project editor.
2. Click the **Command Structure** tab.
3. For each command structure element, do the following:
   a. From the **Header Body Separator** list, select the command structure element.
   b. From the **Parameter Separator** list, select the command structure element.

c. From the **Parameter Name-Value Separator** list, select the command structure element.

d. From the **Compound Parameter Encloser** list, select the command structure element.

e. From the **Compound Parameter Index Separator** list, select the command structure element.

f. From the **Command Tail** list, select the command structure element.

g. From the **End of Command Control Character** list, select the command structure element.

h. Click **OK**.

4. Enable the **Command Auto-Parsing** check box if you want to enable the **Command Auto-Parse Override** check box for all new action processors in the action processor editor **Request** tab.

5. Save the changes.

About Parsing and Configuring CLI Command Requests

Parsing Java CLI command parameters extracts the parameters from a sample CLI command and maps the parameters to corresponding atomic action parameters. You parse command parameters when you configure your CLI command requests. When you parse and configure CLI command requests, you can do the following:

- Manually parse the parameters or specify to automatically parse the parameters.
- Add logic that calls a helper method that manipulates atomic action parameters for name-value parameters or value-only parameters. Helper methods are not used for static string parameters because those parameters are not mapped to atomic action parameters.

  You can use the helper methods defined in these files:

  – **Utils.java**: This file contains predefined methods. See "Provided Methods for Manipulating Parameters" for more information.

  – **ReusableMethods.java**: You can define your own custom helper methods in this file. See "Defining Custom Methods for Manipulating Parameters" for more information.

Provided Methods for Manipulating Parameters

The **Utils.java** file contains helper methods that you can use to manipulate atomic action parameters and values. The methods include tasks such as appending characters to a parameter, concatenating two or more parameters, and so on. Design Studio generates the **Ultils.java** file when you configure your request CLI commands. You can access this file from the Design Studio **Package Explorer** view in the **src** directory.

All methods defined in **Utils.java** throw exceptions defined in the **ProvCartridgeException.java** file, which is auto-generated when you add a sample command.

Table 12-2 describes the methods contained in **Utils.java** and provides examples. For more information about using the methods, open the **Utils.java** file in the **Package Explorer** view. You can access this file after entering a CLI command to parse when configuring your CLI command requests.

Table 12-2 Utils.java Methods

Method	Description	Example
append	This method appends a value to a parameter.	The following appends the hash character to the MSISDN parameter: `append(MSISDN,"#")`
concat	This method concatenates multiple parameters together with a specified delimiter. The first value specifies the delimiter. You can specify multiple parameters after the delimiter using the Java variable arguments feature. The method returns a string after concatenating all the parameters passed. For example, you can use this method when you have to join two atomic action parameters into one parameter in the CLI command.	The following concatenates the MSISDN and LCC_CODE parameters using a dash delimiter: `concat("-","MSISDN","LCC_CODE")`
encloseWith	This method encloses a parameter between two strings.	The following encloses the MSISDN parameter between two hash characters: `encloseWith(MSISDN,"#","#")`
fixedLength	This method ensures that a parameter value is of a fixed number of characters. The method takes the required length of the parameter, the character to be used as padding, and the mapped parameter. If appender is true, the padding characters are prefixed otherwise it is suffixed. The method returns a string with a length equal to the length passed by prefixing or suffixing the padded character to the value passed.	The following appends the * padding character to the MSISDN parameter value if its value length is less than 5. Because the boolean flag is set to true, the padding character is added to the beginning of the value: `fixedLength("MSISDN",MSISDN,5, '*', true)`
prepend	This method adds a value to the beginning of a parameter.	The following adds the hash character to the beginning of the MSISDN parameter: `prepend(MSISDN,"#")`
replaceWith	This method replaces all occurrences of a string in the parameter value with a new string.	The following replaces dashes with hash characters within the MSISDN parameter value: `replaceWith(MSISDN,"--", "##")`
substring	This method extracts a substring from a parameter value. The substring is identified by its start and end character positions in the value. For example, if the value of MSDN were ABCDE and the start position is 3 and end position is 5, then the return value is CDE.	The following extracts the 3rd, 4th, and 5th characters from the value of the MSISDN parameter: `substring(MSISDN,3,5)`

Table 12-2 (Cont.) Utils.java Methods

Method	Description	Example
translate	This method translates a parameter value to a specified value. You specify the translation using value pairs separated by an equals sign (=). Delimit each pair with a comma (,). For example if an upstream system sends "Y" or "N", and the NE expects "Yes" or "No", you can translate the incoming values using "Y=Yes, N=No". This method returns the new value. If the method syntax is incorrect, the method returns null.	The following takes the ACTIVATE parameter value from the atomic action and translates the value from Y to Yes or from N to No.: `translate(ACTIVATE,"Y=Yes,N=No")`

Defining Custom Methods for Manipulating Parameters

When you configure you Java CLI command requests, you can add logic that calls a helper method to manipulate atomic action name-value parameters or value-only parameters. If you need helper methods other than those defined in the **Utils.java** file, you can define your own in the **ReusableMethods.java** file. (For information about the methods in the Utils.java file, see "Provided Methods for Manipulating Parameters.") Design Studio generates the **ReusableMethods.java** file when you configure your request CLI commands. You can access this file from the Design Studio **Package Explorer** view in the **src** directory.

The methods you define in **ReusableMethods.java** must throw exceptions defined in the **ProvCartridgeException.java** file.

Configuring CLI Command Requests

To configure a CLI command request:

1. Open the action processor editor and click the **Editor** tab.
2. From the **Type** list, select **CLI Code Generation**.

   The **Request** and **Response** tabs appears.
3. Click the **Request** tab.
4. If the CLI command you want to configure does not conform to the command structure you previously configured (in "About the CLI Command Structure Elements"), select **Overwrite** in Separators area and configure the command structure you need. Do the following:
   a. From the **Header Body Separator** list, select the command structure element.
   b. From the **Parameter Separator** list, select the command structure element.
   c. From the **Parameter Name-Value Separator** list, select the command structure element.
   d. From the **Compound Parameter Encloser** list, select the command structure element.
   e. From the **Compound Parameter Index Separator** list, select the command structure element.

f. From the **Command Tail** list, select the command structure element.

g. From the **End of Command Control Character** list, select the command structure element.

See "About Configuring the CLI Command Structure" for information about the structure elements and their values.

5. In the **Input Command** field, enter a sample CLI command to use as a template for defining the action processor parameters. The sample command should be a typical CLI command that you want ASAP to send to NEs to fulfill the atomic action that the action processor is associated with.

6. Parse the command and map the command parameters to action parameters by doing one of the following:

> **Tip:**
>
> Preview the structure of the command in the Preview area as you map CLI command elements to atomic action parameters.
>
> For example, the following command shows a header called HSDPA, a value-only parameter, two value-pair parameters, two static parameters, and ends with the COMMIT control character:
>
> ```
> HSDPA:<mcliVal>,LCC_CODE=<user_routingVal>,LINE=<lineVal>,static,
> program,;COMMIT
> ```

To automatically parse the CLI command:

a. Select the **Command Auto-Parsing** check box.

The **Parse Input Command** button becomes selectable.

b. Click the **Parse Input Command** button.

The command parameters are parsed based on the parameter structure separators you configured. The parameters appear in the Parameters area in the **Element Name** list.

If the parameter name does not match any atomic action parameter, then the parameter is still added to the **Element Name** list, but does not map to any atomic action parameter in the **Maps To** field.

c. If all parameter names are mapped to atomic action parameters, go to step 7.

d. If some parameters are not mapped to atomic action parameters, go to the instructions for manually parsing the CLI command below.

To manually parse the sample CLI command:

a. Using your mouse, highlight and right-click on a parameter in the command.

b. If you want to designate the parameter as the command header, select **Command Header**.

The parameter appears in the Command Header field.

c. If you want to designate the parameter as a command parameter, select **Command Parameter**, then select the value type to use in the command:

- **Name Value Pair**: When you select this option, the parameter name and value are used. You must select a parameter from the atomic action from which the value is populated.
- **ValueOnly**: When you select this option, only the parameter value is used. You must select a parameter from the atomic action from which the value is populated.
- **StaticString**: When you select this option, only the parameter name is used. Static strings do not contain values and cannot be associated with atomic action parameters.

After you make your selection, the highlighted parameter appears in the **Element Name** list. When you select one of the auto-generated parameters from the **Element Name** list, the associated atomic action parameters appear in the **Maps To** field (except for **StaticString** parameters).

7. (Optional) For **Name Value Pair** and **ValueOnly** parameters, add a line of logic in the **Parameter Logic** field that calls a helper method that manipulates the atomic action parameter and click **OK**.

   You can use the helper methods defined in either the **Utils.java** file (see "Provided Methods for Manipulating Parameters" or the **ReusableMethods.java** file (see "Defining Custom Methods for Manipulating Parameters."

   You can nest two or more commands in the **Parameter Logic** field. For example, the following line nests the **encloseWith** method within the **concat** method:

   ```
   concat("*",encloseWith(MCLI,"#","#"),MY_TEST)
   ```

8. If you need to add more complex logic for a particular parameter that can be enabled by a one line method as in the **Parameter Logic** field, click **Edit Parameter Logic**. The first time you click this button Design Studio generates additional java files where you can make these modifications. If you make changes to the command parameter to atomic action parameter mappings after you have clicked the **Edit Parameter Logic** button for the first time, new files are not generated if you click the **Edit Parameter Logic** button again. For more information about the files generated when you click the **Edit Parameter Logic** button, see "About Auto-Generated and Synchronized CLI Java Files."

About Configuring CLI Command Responses

You must write additional code for handling CLI command responses. Design Studio generates the **ResponseHandlerImplementation.java** file when you configure your CLI commands. You can access this file from the Design Studio **Package Explorer** view in the **src** directory.

You can also configure the action processor to search for a specific response snippet in response messages before sending the response to the user defined exit type code you have written.

Configuring CLI Command Responses

To configure a CLI command response:

1. Open the action processor editor and click the **Editor** tab.
2. From the **Type** list, select **CLI Code Generation**.

   The **Request** and **Response** tabs appears.
3. Click the **Response** tab.
4. In the **Response Section**, **Response** area, **Response** field, enter a description of the response.

5. If you want the action processor to search for responses with specific headers or exit type patterns before sending the response to the user exit type code, do the following:
   a. In the **Response Section**, **Exit Type**, **Response Header** field, enter a response header.
   b. In the **Response Section**, **Exit Type**, **Exit Type Pattern** field, enter a response pattern.
   c. Click **Mark Positions** button.

      The Mark Positions Specifier dialog appears.
   d. In the **Start Position** field, enter a starting position from 1 to 10. The start position must be less than the end position.
   e. In the **End Position** field, enter an end position from 1 to 10. The end position must be greater than the start position.
6. Click **Edit Response Logic**. The **ResponseHanderIImplementation.java** file opens in the **Package Explorer** view.
7. Add response handling code to the file. For more information about the **ResponseHanderIImplementation.java** file, see "Auto-Generating the Java CLI Files."

Auto-Generating the Java CLI Files

To auto-generate the Java CLI code files:

1. Open the action processor editor and select the **Editor** tab.
2. From the **Type** list, select **CLI Code Generation**.
3. Click **New**.

   Design Studio automatically generates Java code.

   The **Class** field points to the auto-generated proxy java file (for example, alu.fttu.x74.ont.add.generated.AddOntProxy). This file contains a proxy class that is situated between the NEP and action processor and manages the interaction between them.

   The **Method** field points to the execute method within the processor java file (for example alu.fttu.x74.action.ont.add.AddOntProcessor.java).

   For more information about the auto-generated Java files and code and the areas where you must include additional business logic, see "About Auto-Generated and Synchronized CLI Java Files."
4. Click **Finish**.

About Auto-Generated and Synchronized CLI Java Files

After the Java files are generated, Design Studio automatically updates those files that are synchronized whenever you build the cartridge. Never make changes to synchronized files because Design Studio overwrites these files when you build the cartridge. You can, however, modify files that are not synchronized.

Oracle recommends that you backup the *cli_project***/src** directory (where *cli_project* is the activation project that contains the CLI code) to a source control system. This directory contains all the modifiable CLI Java files. For more information about backing up this folder, see *Developer's Guide*.

Table 12-3 describes the Java files that are created when you auto-generated them, and whether the generated files are synchronized and modifiable.

> **Note:**
>
> In the files specified in Table 12-3, *action* and *entity* represent the action and entity values you selected when you created the action processor (see "Creating an Action Processor").

Table 12-3 CLI Code Generation Java Files

Java File	Description	Sync	Mod
*ActionEntity***Processor.java**	This file contains the *ActionEntity***Processor** class that implements the generated processor interface. The *ActionEntity***Processor** class includes the **execute** class and the following methods: • **ILogger** is an interface for debug logs. When the processor is running on the ASAP system, it logs to the Diagnosis log. If you are running the processor in JUnit, you can use other implementations of logger to log to the console instead. • **IExitType** enables you to set the exit type explicitly or by matching a response string against the user-defined exit types. • *ActionEntity***Input** For more information, see *ActionEntity***Input.java** in this table. • *ActionEntity***Output** For more information *ActionEntity***Output.java** in this table. • IConnectionHander For more information, see "Creating Java Connection Handlers." The logic in the **execute** class is fully functional when you auto-generate the CLI Java files. The code is generated based on the command element to atomic action mappings that you configured using the action processor editor **Request** tab. The code is also based on the **Exit Type Pattern** field in the action processor editor, **Response** tab. You specify a response snippet in this field that the code searches for in response messages. You use the **Mark Positions** button to specify what part of the response snippet to verify before sending the response to the user defined exit type code you have written. For more information about exit types, see "Creating Java User Exit Types." The *ActionEntity***Processor.java** file is auto-generated when you click the **New** button as described in "Auto-Generating the Java CLI Files." However, if the file already exists when you click the **New** button, then the file will not be auto-generated and will not reflect any changes that you made to the action processor since you first auto-generated the file. If you want to Design Studio to auto-generate the file, you must delete the old file first.	No	Yes
ModifyGeneratedMML.java	This file contains methods you can use to augment the generated MML command if you require any additional post processing. For example, you may need to encrypt the CLI command before sending it.	No	Yes
ResponseHandlerImplementation.java	This file contains the ResponseHandlerImplementation class that implements the ResponseHandlerInterface. You must do the following in this file: • Declare return parameter variables. • Write response parsing code. • Return the parameter value in the response.	No	Yes

Table 12-3 (Cont.) CLI Code Generation Java Files

Java File	Description	Sync	Mod
*ActionEntity***Input.java**	This file contains the InputBean. The InputBean has set and get methods for all parameters of the atomic action and provides setters and getters for manipulating parameters. If the parameter is a scaler (simple type), it is received as a string and can be used immediately. For information about compound parameters, see "Understanding Generated Code for Compound Parameters" If you need to add logic to the input parameters, for example, if you are mapping a CLI command to more than one compound parameter, you can click the **Edit Parameter Logic** button in the action processor editor Request tab to generate files where you can add this logic. For more information about these generated files, see Table 12-4.	Yes	No
*ActionEntity***Output.java**	This file contains the Output class that enables you to populate output parameters. There are convenience methods for populating parameters to varying scope within a work order. Examples of parameters are as follows: • Action parameters, which are available to the service action • Input parameters • Global parameters, which are available to everything • Rollback parameters, which enable you to populate for the rollback action if it is defined in the atomic action The output parameters are not explicitly defined in the model, so there are no convenience methods. To set a parameter, you need to know its string name and include it.	Yes	No
*ActionEntity***ProcessorInterface.java**	This file contains the processor interface that is implemented by the *ActionEntity***Processor** class. This interface is synchronized whenever the cartridge model changes so the *ActionEntity***Processor** class always has the correct cartridge data available.	Yes	No
*ActionEntity***Proxy.java**	This file contains the **Proxy** that is situated between the NEP and Processor class and manages the interaction between them. **Proxy** sets up all classes used by the processor and initiates and calls the processor. Most importantly, the proxy simplifies the work required by the Processor by: • Creating all instances of the InputBean and initializing CompoundBeans so they are available and populated through the processor. • Performing much of the standard logging, including the entry and exit of the processor and the contents of the parameters passed in for debugging. • Extending the JProcessor. This isolates the portion of the Java processor code that needs to relate directly to the version of the activation, and allows the processor, its interface, and all its related classes and interfaces to run outside of the ASAP system and, therefore, to be unit tested When creating a Java processor from the action processor editor, the resulting class name is "Proxy" because the proxy gets initiated by the NEP (the Proxy is registered to be called in the activation).	Yes	No

Table 12-3 (Cont.) CLI Code Generation Java Files

Java File	Description	Sync	Mod
ConnectionHandler.java	This file contains the ConnectionHandler class that extends a protocol class and implements the IConnectionHandler. IConnectionHandler is an instance of the connection handler that is associated with the vendor, technology, and software version of the action processor. For the Telnet connection handler, the basic methods on the interface can be used to send requests (because it is string-based). For technologies (for example, SOAP or XML) that provide multiple convenience methods, the processor may test the type of connection handler and pass the request to a more specific connection handler to obtain access to the convenience methods. If you want to expose more explicit methods when writing a connection handler, you can define an interface that extends **IConnectionHandler** and ensure that those methods are available through that interface. The processor should always use an interface when interacting with the ConnectionHandler, to achieve the implementation in more than one way and allow for unit testing. For more information about unit testing, see "Understanding Unit Testing."	No	Yes
Base*ActionEntity***TestCase.java**	This file contains the unit test case for the action processor. For more information about this generated file, see "Understanding Unit Testing."	Yes	No
MMLConstructor.java	This file contains the code that builds the CLI command based on the command structure. For information about the command structure, see "About the CLI Command Structure Elements." The code does the following: • Adds the command header to the CLI command • Adds the header body separator after the header • Adds parameters to the CLI command after the header • Adds the command tail to the CLI command • Adds the command end-of-message character to the CLI command	Yes	No
MMLConstructor.Interface.java	This file provides the methods that the MML Constructor class implements.	Yes	No
ResponseHandlerInterface.java	This file contains the interface code that ResponseHandlerImplementation.java implements.	Yes	Yes
Separators.java	This file contains the command separators you specified when configuring the command structure. For more information about configuring the command structure, see "About the CLI Command Structure Elements."	Yes	No
ISystemParameters.java	This file contains the interface that extends the IBaseSystemParameters interface. The methods in IBaseSystemParameters class are implemented by the SystemParameter class.	Yes	No
SystemParameters.java	This file contains the SystemParameter class that implements the methods in the IBaseSystemParameters interface, such as getWorkOrderId(), getActionName(), and so on. The SystemParameter class extends the BaseSystemParameters class.	Yes	No

Table 12-4 describes the Java files that Design Studio creates when perform the tasks required to auto-generate Java CLI code.

> **Note:**
>
> The files generated when using the **Parameter Logic** field or the **Edit Parameter Logic** field specified in Table 12-4 use the *ElementName* variable. ElementName is the parameter name in the **Element Name** list when the parameter is a **Name Value Pair** (for example ID_ROUTING.java). *ElementName* is a combination of the atomic action parameter in the **Maps To** field followed by an underscore and the parameter name in the **Element Name** list when the parameter is **ValueOnly** (for example, MCLIValue_LINE.java).

> **Note:**
>
> The files generated when using the **Parameter Logic** field or the **Edit Parameter Logic** field specified in Table 12-4 that are not synchronized will not reflect any changes made to the mappings between the command parameters and the atomic action parameters after you generate these files for the first time. You must delete these files manually if you want to auto-generate them again.

Table 12-4 Parameter Logic, Edit Parameter Logic, and Input Command Java Files

Java Files	Description	Sync	Mod
ProvCartridgeException.java	This file contains the exception class used by the methods defined in the **Utils.java** file.	Yes	No
ReusableMethods.Java	This file contains the ReusableMethods class. You can use this file to define one-line methods that you can use when configuring your CLI command requests (see "Configuring CLI Command Requests"). This class should use the exception logic defined in **ProvCartridgeException.java**.	No	Yes
Utils.java	This file contains predefined one-line methods that you can use in the action processor editor **Request** tab **Parameter Logic** field. For more information about these methods, see "Provided Methods for Manipulating Parameters."	Yes	No
ElementName.**java**	This file is generated when you add a one-line method to the parameter logic when configuring your CLI command requests (see "Configuring CLI Command Requests").	Yes	No
*ElementName*_**Implementation.java**	This file is generated when you modify the parameter logic that you specified when configuring your CLI command requests. You can add custom logic to this file if you need to include more complicated processing instructions than those in **Utils.java** or in **ResuableMethods.java**.	No	Yes
*ElementName*_**Interface.java**	This file defines an interface for the *ElementName*_**Implementation** class in the *ElementName*_**Implementation.java** file. This file is generated when you modify the parameter logic that you specified when configuring your CLI command requests.	Yes	No

Backing Up Files

You should implement source control for the *cli_project*/**src** directory (where *cli_project* is the Design Studio Activation project root folder)

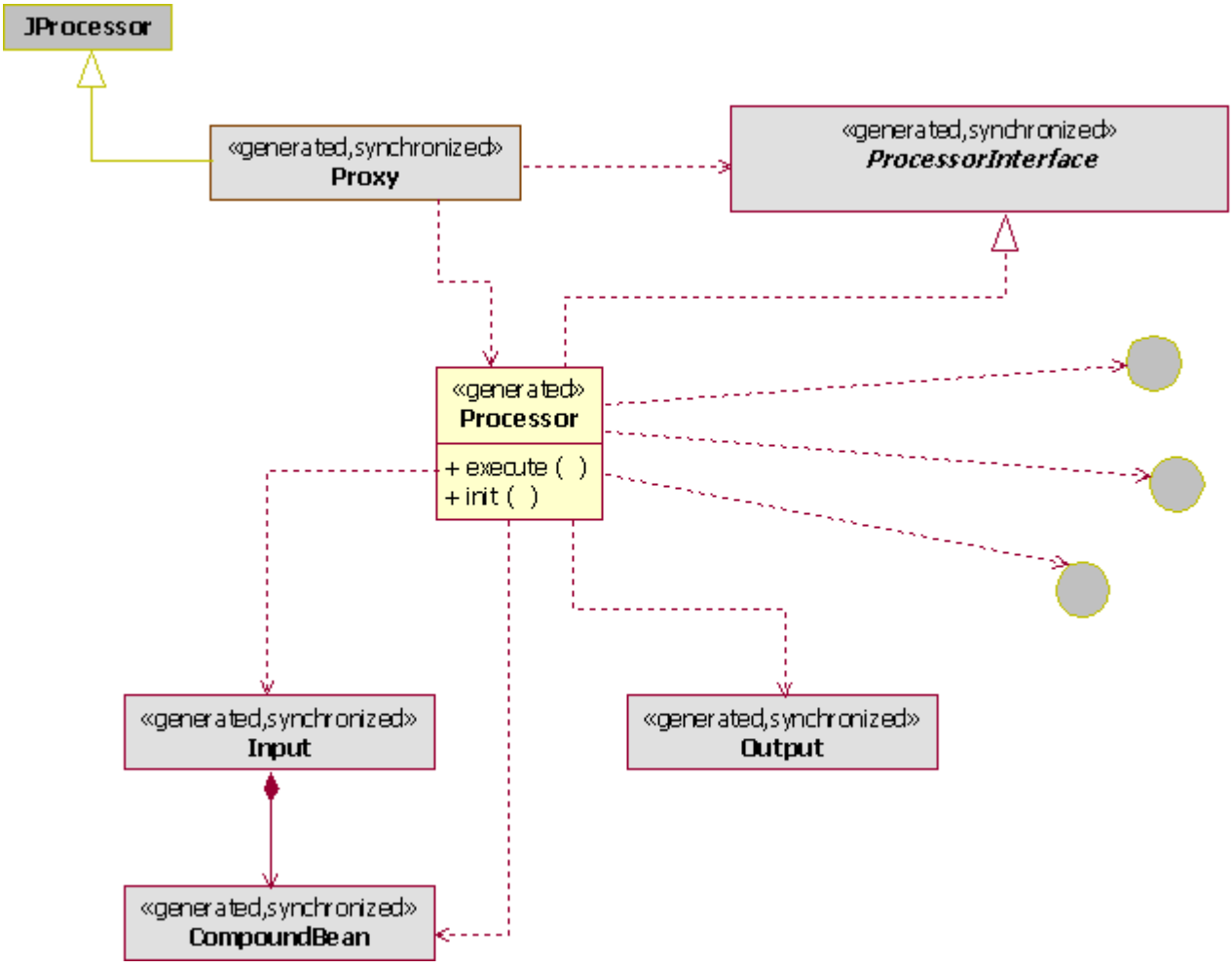
Understanding the Auto-Generated Java Code Stubs

You can use the action processor to auto-generate Java files with classes and methods configured for the protocol and attributes you selected. The auto-generated code are code stubs that provides a logical framework where you must include your business logic for sending and receiving responses.

The Java with code generation implementation for an action processor creates a Java processor that composes messages to be sent to a device, evaluates the response for errors, extracts output information from the response, and populates the information into output parameters.

Figure 12-1 shows some of the auto-generated files, whether they are synchronized, and how they relate to each other.

Figure 12-1 Generated and Synchronized Java Files



When you configure the action processor to auto-generate Java code stubs, the central class is the Processor. You must add business logic to this class. The Processor is created only once. The processor includes sample code based on the associated atomic action parameters at creation time. You should delay creating the processor until the action processor is associated with an atomic action that has fully defined parameters. If parameters are not yet

defined or the action processor is not yet associated with the atomic action, then the generated sample code will be incomplete and will require additional coding.

> **Note:**
>
> Synchronized classes or interfaces are rebuilt every time you save changes to atomic action parameters (for example, classes and interfaces are synchronized with the model and reflect the model). Therefore, you should never make code changes to any synchronized class or interface. Design Studio overwrites the code when you run the next build (with changes in the model). You should write code only for the Processor class.

There are 2 methods in the Processor:

- `execute`
- `init`

The main method is `execute`. When called, it is provided with the following:

- A number of classes to perform operations.
- An input class that contains all input parameters.
- An output class to populate the output parameters.
- Access to a logger.
- An implementation of the exit type to match responses against user-defined exit types and to set the exit type for the processor.
- Access to the Connection Handler to send requests and get responses from a connected device.

Auto-Generating the Java Stubs

To auto-generate Java stubs:

1. Open the action processor editor and select the **Editor** tab.
2. From the **Type** list, select **Java Action Processor (with Code Generation)**.
3. If you have not already done so, create and fully configure an atomic action that includes the required parameters. You must associate the atomic action to this action processor. For more information about creating and configuring an atomic action, see "Creating and Configuring an Atomic Action."

   > **Note:**
   >
   > If you do not create and configure the atomic action the auto-generated code will not function properly.

4. Click **New**.

   The Studio Activation Java Implementation Wizard appears.

5. In the **Package** field, enter a valid package name. You can use the default package name or enter a name of your choice. The default Package name uses the vendor, technology, software version, entity, and action that you selected when creating the action processor (see step 2).

   For example:

   ```
   alu.fttu.x74.ont.add
   ```

6. In the **Name** field, enter a name that appears in many of the auto-generated Java files and the classes they contain. You can use the default name or enter a name of your choice. The default name is a combination of the action and the entity.

   For example:

   ```
   AddOnt
   ```

7. Click **Finish**.

   Design Studio automatically generates Java code.

   The **Method** field points to the auto-generated proxy java file (for example, alu.fttu.x74.ont.add.generated.AddOntProxy). This file contains a proxy class that is situated between the NEP and Processor class and manages the interaction between them.

   The **Class** field points to the execute method within the processor java file (for example alu.fttu.x74.action.ont.add.AddOntProcessor.java). You must add business logic to this class.

   For more information about the auto-generated Java code and where you must include business logic, see "About Auto-Generated Java Files ."

About Auto-Generated Java Files

Table 12-5 shows the Java files containing the classes and interfaces used by the Processor.

Table 12-5 Auto-Generated Java Files

Java Files	Description	Sync	Mod
*ActionEntity***Processor.java**	This file contains the *ActionEntity***Processor** class that implements the generated and synchronized processor interface. The *ActionEntity***Processor** class includes the **execute** class with following methods: • `ILogger` is an interface for debug logs. When the processor is running on the Oracle Communications ASAP system, it logs to the Diagnosis log. If you are running the processor in JUnit, you can use other implementations of logger to log to the console instead. • `IExitType` enables you to set the exit type explicitly or by matching a response string against the user-defined exit types. • *ActionEntity***Input** For more information, see *ActionEntity***Input.java** in this table. • *ActionEntity***Output** For more information *ActionEntity***Output.java** in this table. • IConnectionHander For more information, see "Creating Java Connection Handlers." You need to write the logic in the `execute` class for each atomic action to achieve the desired action in a NE. Use the Java editor in the Package Explorer view of the Java perspective to write the code. When implementing the action processor, Design Studio provides you with support such as auto-generation of code and sample data. In Design Studio, this is currently set up specifically for the Telnet protocol (Soap, CORBA, and other protocols require more coding; for example, you must write your own logic for send methods, requests, to extend the connection class, and so on.). Code for the processor is auto-generated by the proxy (getter and setter methods for each parameter) which provides you with an API to manipulate the data. For example, for an incoming object, methods such as `getBilling` are auto-generated (the type of methods depend on the parameters specified in the service model and how they are mapped). You can use these auto-generated methods in the processor class to get the value for the parameters. To obtain the required method to get a value for a parameter, type the name of the parameter followed by a dot. This displays all available methods for the parameter.	No	No
*ActionEntity***Input.java**	This file contains the InputBean. The InputBean is tied to the parameters of the atomic action (has set and get methods for all parameters of the atomic action), and provides setters and getters for manipulating parameters. If the parameter is a scaler (simple type), it is received as a string and can be used immediately If the parameter is a compound, see "Understanding Generated Code for Compound Parameters."	Yes	No
*ActionEntity***Output.java**	The Output class enables you to populate output parameters. There are convenience methods for populating parameters to varying scope within a work order. Examples of parameters are as follows: • Action parameters are available to the service action. • Input parameters. • Global parameters are available to everything. • Rollback parameters enable you to populate for the rollback action if it is defined within the atomic action. The output parameters are not explicitly defined in the model, so there are no convenience methods. To set a parameter you need to know its string name and include it.	Yes	No
*ActionEntity***ProcessorInterface.java**	The processor interface is implemented by the *ActionEntity***Processor** class. This interface is synchronized whenever the cartridge model changes.	Yes	No

Table 12-5 (Cont.) Auto-Generated Java Files

Java Files	Description	Sync	Mod
*ActionEntity***Proxy.java**	`Proxy` is situated between the NEP and Processor class and manages the interaction between them. `Proxy` sets up all classes used by the Processor and initiates and calls the Processor. Most importantly, the proxy simplifies the work required by the Processor by: • Creating all instances of the InputBean and initializes CompoundBeans so they are available and populated through the processor. • Performing much of the standard logging, including the entry and exit of the processor and the contents of the parameters passed in for debugging. • Extending the JProcessor. This isolates the portion of the Java processor code that needs to relate directly to the version of the activation, and allows the processor, its interface, and all its related classes and interfaces to run outside of the ASAP system and, therefore, to be unit tested When creating a Java processor from the action processor editor, the resulting class name is "Proxy" because the proxy gets initiated by the NEP (Proxy is registered to be called in the activation). When you open that implementation it opens to the `Processor` class, where you write your code for editing.	Yes	Yes
ConnectionHandler.java	This file contains the ConnectionHandler class that extends a protocol class and implements the IConnectionHandler. IConnectionHandler is an instance of the Connection Handler that is associated with the vendor, technology, and software load of the action processor. For the Telnet Connection Handler, the basic methods on the interface can be used to send requests (because it is string-based). For technologies (for example, SOAP or XML) that provide multiple convenience methods, the Processor may want to test the type of Connection Handler and pass it to a more specific Connection Handler to obtain access to the convenience methods. If you want to expose more explicit methods when writing a Connection Handler, you can define an interface that extends the `IConnectionHandler` and ensure that those methods are available through that interface. The Processor should always use an interface when interacting with the ConnectionHandler, to achieve the implementation in more than one way and allow for unit testing. For more information about unit testing, see "Understanding Unit Testing."	No	Yes
Base*ActionEntity***TestCase.java**	This file contains the unit test case for the action processor. For more information about this generated file, see "Understanding Unit Testing."	Yes	No
ISystemParameters.java	This file contains the interface which extends the IBaseSystemParameters interface. Those functions within IBaseSystemParameters class are implemented by SystemParameter class.	Yes	No
SystemParameters.java	This file contains the the SystemParameter class that implements the respective functions of IBaseSystemParameters interface such as getWorkOrderId(), getActionName(), and so on, by extending BaseSystemParameters class.	Yes	No

Understanding Generated Code for Compound Parameters

The InputBean returns another bean that represents the compound if the parameter is a compound parameter with named members,. The returned bean has convenience methods to get the members within the compound. A compound bean for every defined type of compound parameter is created. You also get a set of instances of these beans based on the work order (you get a list of these). If the compound parameter does not have named members, it provides a vector of the members.

> **Note:**
>
> Specify the compound members whenever possible. Indicating the members will simplify the coding required and eliminate possible code to mode synchronization issues.

Multi-instance Compound parameters start at index one (e.g. CMPD[1]).

Bracket type (Index Parameter Identification Tokens) and delimiter (Indexed Parameter Delimiter) settings are configured on the Project editor **Cartridge Locations** tab in the Code Generation area. Design Studio applies these settings to all generated code within the cartridge. The following examples assume the defaults (square brackets with a period delimiter).

The following example shows a scalar parameter.

```
Service Action Parameter Name: SCALAR
Atomic Action Parameter Name: SCALAR
Order Format:
     SCALAR
Usage:
     String myscalar = parms.getMyScalar();
```

The following example shows a compound parameter with no members specified.

```
Service Action Parameter Name: CMPD
Atomic Action Parameter Name: CMPD
Order Format:
Entries will have the compound name as a prefix. There may be multiple entries with that
prefix. For example, a compound named "CMPD" may have the following entries on an order.
     CMPD
     CMPD.X
     CMPD.Y
     CMPD.Z
Usage:
     String mycmpd = parms.getMyCmpd
       String x = parms.getMyCmpd ("X");
       String y = parms.getMyCmpd ("Y");
       String z = parms.getMyCmpd ("Z");
```

The following example shows a compound parameter with members.

```
Service Action Parameter Name: CMPDMBR
Atomic Action Parameter Name: CMPDMBR
Order Format:
     CMPDMBR.A
     CMPDMBR.B
     CMPDMBR.C
Usage:
     MyCmdMbrBean mycmpdmbr = parms.getMyCmpdMbr();
       mycmpdmbr.getA();
       mycmpdmbr.getB();
       mycmpdmbr.getC();
```

The following example shows a multi-instance compound parameter with no members specified.

```
Service Action Parameter Name: CMPDMULTI
Atomic Action Parameter Name: CMPDMULTI
Order Format:
Entries will have the compound name as a prefix. There may be multiple entries with that
prefix. For example, a compound named "CMPDMULTI" may have the following entries on an
order.
     CMPDMULTI[1]
     CMPDMULTI[1].X
     CMPDMULTI[1].Y
     CMPDMULTI[1].Z
     CMPDMULTI[2].X
     CMPDMULTI[2].Y
     CMPDMULTI[2].Z
Usage:
     String mycmpdmulti = parms.getMyCmpdMulti ();
       String x1 = parms.getMyCmpdMulti (1, "X");
       String y1 = parms.getMyCmpdMulti (1, "Y");
       String z1 = parms.getMyCmpdMulti (1, "Z");
       String x2 = parms.getMyCmpdMulti (2, "X");
       String y2 parms.getMyCmpdMulti (2, "Y");
       String z2 = parms.getMyCmpdMulti (2,"Z");
```

The following example shows a multi-instance compound parameter with members.

```
Service Action Parameter Name: CMPDMULTIMBR
Atomic Action Parameter Name: CMPDMULTIMBR
Order Format:
     CMPDMULTIMBR[1].A
     CMPDMULTIMBR[1].B
     CMPDMULTIMBR[1].C
     CMPDMULTIMBR[2].A
     CMPDMULTIMBR[2].B
     CMPDMULTIMBR[2].C
Usage:
     MyCmpdMultiMbrBean[] mycmpdmultimbr = parms.getMyCmpdMultiMbr();
     for (int i = 0; i <mycmpdmultimbr.length; i++)
     {
       MyCmpdMultiMbrBean bean = mycmpdmultimbr[i];
       bean.getA();
       bean.getB();
       bean.getC();
     }
```

The following example shows an indexed compound parameter with no members.

```
Service Action Parameter Name: CMPDIDX[++]
Atomic Action Parameter Name: CMPDIDX
Order Format:
Entries will have the compound name as a prefix. There may be multiple entries with that
prefix. For example, a compound named "CMPDIDX" may have the following entries on an
order.
     CMPDIDX[0]
     CMPDIDX[0].X
     CMPDIDX[0].Y
     CMPDIDX[0].Z
     CMPDIDX[1].X
     CMPDIDX[1].Y
     CMPDIDX[1].Z
Usage:
     String mycmpdidx = parms.getMyCmpdIdx();
       String x = parms.getMyCmpdIdx ("X");
       String y = parms.getMyCmpdIdx ("Y");
       String z = parms.getMyCmpdIdx ("Z");
```

> **Note:**
>
> The implementation will be called multiple times, providing one instance of the compound during each call.

The following example shows a compound parameter with members.

```
Service Action Parameter Name: CMPDIDXMBR[++]
Atomic Action Parameter Name: CMPDIDXMBR
Order Format:
     CMPDIDXMBR[0].A
     CMPDIDXMBR[0].B
     CMPDIDXMBR[0].C
     CMPDIDXMBR[1].A
     CMPDIDXMBR[1].B
     CMPDIDXMBR[1].C
Usage:
     MyCmpdIdxMbrBean mycmpdidxmbr = parms.getMyCmpdIdxMbr();
       mycmpdidxmbr.getA();
       mycmpdidxmbr.getB();
       mycmpdidxmbr.getC();
```

> **Note:**
>
> - The implementation will be called multiple times providing one instance of the compound during each call.
> - For multi-instance compounds, member parameters cannot be set as required because the system cannot determine whether a member is present or if there are additional entries.

Example: Typical Processor Call Sequence

The proxy:

1. The proxy creates the input, the output, and the exit type classes.
2. The proxy populates the exit type classes and initializes them.
3. The proxy creates the processor that will be called and initializes it.
4. If the logger needs to be used by the processor, the proxy provides this during the `init` method call.
5. The proxy invokes the processor by calling the `execute` method with the input, output, connection, and exit type.
6. The processor obtains parameters from the InputBean to compose a message or command to be sent to a device.
7. The processor calls the `send request` to send that message to the device.
8. The processor sets the exit type based on the response.
9. The processor sets output parameters based on the response.

The processor may parse the response to obtain additional values for populating the output parameters.

10. The proxy cleans up the processor.
11. The proxy looks at the exit type that was set and populates it for return to the NEP, and cleans up the exit type.
12. The proxy extracts all output parameters for return to the NEP and to populate the work order.

    The proxy then cleans up this class and (16) the remaining classes.

> **Note:**
>
> You are only responsible for the items related to the processor (steps 6 through 9); the proxy handles all other items.

Writing Java Processor Execute Method Logic

The basic development steps to write the logic for the execute method of a Java Processor class are as follows:

To write Java Processor execute method logic:

1. Extract parameters from InputBean (retrieve information).
2. Use these parameters to build a command.
3. Send a message or command to the switch using the send request in Telnet.
4. Handle the response by setting the user-defined exit type.

   See "Configuring Base Exit and User Exit Types " for more information about setting the user-defined exit type.
5. Using the OutputBean, you have the option to return some parameters upstream to log, infoparm, and so on.

Occasionally, for Telnet, you may need to build some helper classes, perform data derivation, and create parsers.

Example: Telnet Provisioning Class Flow

The following list describes the flow for Telnet provisioning classes.

1. Initialize generic data
2. Get the connection reference
3. Get the NE ID
4. Enable the response log
5. Get the work order parameters and build the AsapParameter objects passing the parameter label and value
6. Build the provisioning command with a specific action type and append parameter objects to the command
7. Convert the command to a string

8. send the command to the switch
9. Obtain the NE response
10. Exit with the appropriate user-defined -> base exit type.

```
//***** initialize generic data: get connection reference, get ne ID, enable response log
initialize();
//****** get work order parameters and build AsapParameter objects passing the param
label and value
String imsi = getParam(IMSI);
String bserv = getParam(BSERV);
String msisdn = getParam(MSISDN);
String nbr = getParam(NBR);
AsapParameter imsiParm = new AsapParameter(IMSI, imsi);
AsapParameter bservParm = new AsapParameter(BSERV, bserv);
AsapParameter msisdnParm = new AsapParameter(MSISDN, imsi);
AsapParameter nbrParm = new AsapParameter(NBR, bserv);
//***** Build the provisioning command with a specific action type and append parameter
objects to the command
ProvisioningCommand cmd = new ProvisioningCommand(SAConstants.CREATE_BASIC_SRV); //
***create command for adding a basic service
cmd.append(imsiParm).append(bservParm).append(msisdnParm).append(nbrParm);
//***** command ready! convert it to string mml and send it to the switch
String strCmd = cmd.toString();
String reply = sendNeRequest(strCmd);
//*****handle response, set user exit type etc
handler.checkResponse(reply); //optional
UserExitType uet = handler.getUserExitType(reply);
setASDLExitType(uet.getUserExitType(), uet.getUserErrorText());
```

About Writing Java Programs from Scratch and Naming Conventions

You must extend the **com.mslv.activation.jinterpreter.JProcessor** class when writing Java code from scratch. Oracle recommends using the auto-generated code options. For more information about **JProcessor**, see *ASAP Java Online Reference*.

This section provides naming convention recommendations when writing Java programs from scratch.

Associating an Action Processors to the Java Code

To associate and action processor to the Java code you created:

1. Open the action processor editor and select the **Editor** tab.
2. From the **Type** list, select **Java Action Processor**.
3. In the **Class** field, enter a class name.
4. In the **Method** field, enter a method name.
5. Manually create the Java classes and methods to implement your network connection.

Java Package Naming Convention

The Java package naming convention consists of the constant prefix **com.oracle.cartridge.oss** in lowercase, with each of the tokens separated by a period (.)

character. Each of the tokens must be separated by an underscore (_) character. The format of a Java package is as follows:

```
com.oracle.cartridge.oss.vendor_technology_softwareload_entity_action_
```

Where:

- *vendor*: specifies the vendor name (see "Selecting the Vendor Token").
- *technology*: specifies the technology (see "Selecting the Technology Token").
- *softwareload*: specifies the software load (see "Selecting the Software Load Token"). Java class naming conventions exclude all period (.) characters from the software load token. For example, software load version 5.1 must appear as 51.
- *entity*: specifies the entity (see "Selecting Entity Tokens").
- *action*: specifies the action (see "Selecting the Action Tokens").

The following example illustrates the structure of a Java package used for the Alcatel-Lucent fiber to the user (FTTU) node running software load 7, providing the pay-per-view (PPV) service, with a buy action:

```
com.mslv.activation.cartridge.alu.fttu.7.ppv.buy
```

The convention used in most cartridges is based on the Metasolv name. For example com.mslv.cartridge.activation.cartridge.

Java Class Naming Convention

A Java class is a single entity that is contained within a Java package. The following list contains each of the types of cartridge Java classes and their corresponding naming conventions:

```
Connection class—Connection.java (for example HLRConnection.java)
Provisioning class—<*>Provisioning.java (for example HLRProvisioning.java)
Library class—<*>Lib.java
```

Class names should be nouns, in mixed case, with the first letter capitalized and with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. The wildcard token (*) used in the naming convention for provisioning and library classes is an optional string that can be used to either divide a provisioning or library class that is too large in size or identify a group of related features that are contained in the provisioning or library classes.

Java Helper and Utility Class Naming Convention

Java helper and utility class file names consists of a series of tokens that are separated by the underscore (_) character. Each token must begin with a lowercase letter. The ".jar" constant always appears at the end of the Java library file name to identify the file as a Java library file. The maximum allowable length for a Java library file name is dictated by the operating system.

The format of Java helper and utility classes is as follows:

```
vendor_technology_softwareload_entity.jar
```

Where:

- *vendor*: specifies the vendor name (see "Selecting the Vendor Token").
- *technology*: specifies the technology (see "Selecting the Technology Token").

- *softwareload*: specifies the software load (see "Selecting the Software Load Token"). Java class naming conventions exclude all period (.) characters from the software load token. For example, software load version 5.1 must appear as 51.
- *entity*: specifies the entity (see "Selecting Entity Tokens").

The following example illustrates the structure of a Java library file that contains the byte code to support VDSL service activation on a Alcatel-Lucent FTTU NE running software load 7.2:

```
alu_fttu_72_vdsl.jar
```

Java Method Naming Convention

Methods should be verbs. Tokens contained in the Java methods names are concatenated or separated using a combination of the period (.) and underscore (_) characters. The Java method naming convention consists of two tokens that are concatenated. The format of a Java method is as follows:

```
actionentity
```

Where:

- *action*: specifies the action (see "Selecting the Action Tokens").
- *entity*: specifies the entity (see "Selecting Entity Tokens").

The first letter of the action must appear in lowercase and the first letter of all subsequent tokens must appear in uppercase. The following example illustrates the structure of a Java method used for the Alcatel-Lucent FTTU NE supporting a pay per view (PPV) service:

```
addPpv
```

Java Variables Naming Convention

Variable names should be short yet meaningful. The choice of a variable name should be a mnemonic, and designed to indicate the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

Java Constants Naming Convention

The names of variables declared class constants should be all uppercase with words separated by underscores (_).

Understanding Unit Testing

Unit testing in Design Studio does not need to be implemented to complete a cartridge, although it is highly recommended for these reasons:

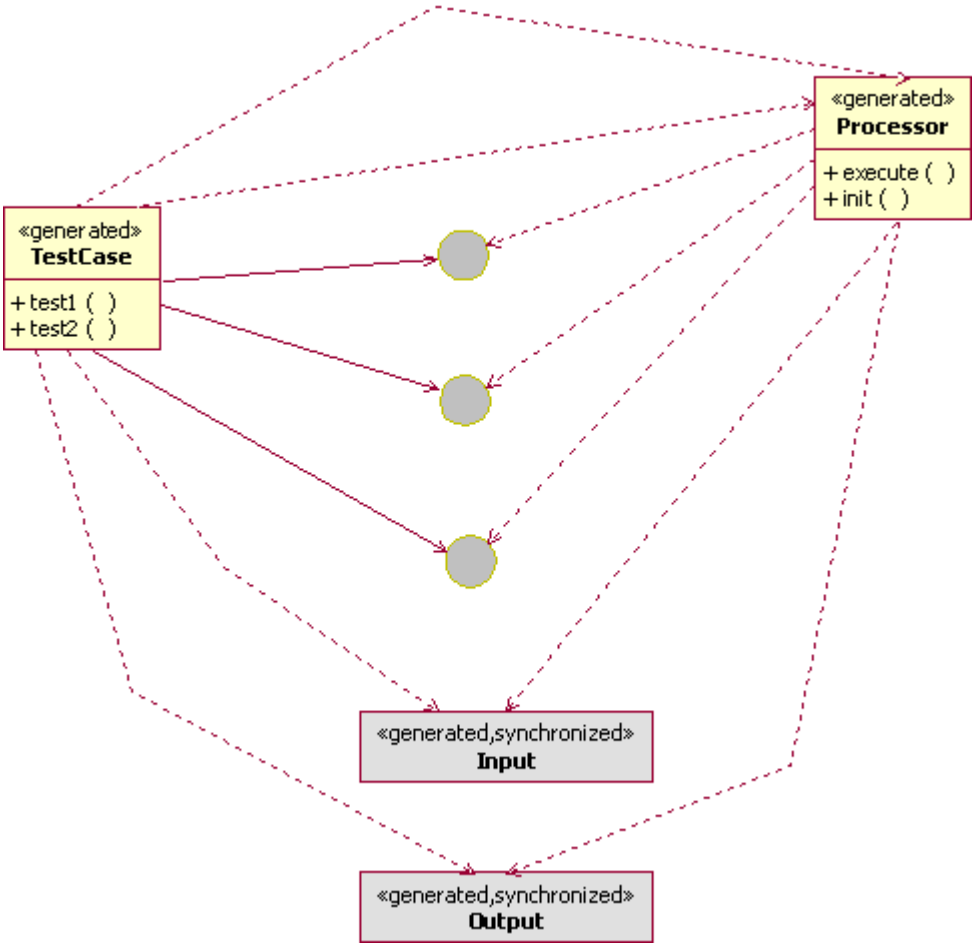
- Unit testing contributes to building quality code.
- Unit testing provides repeatable tests for regression.

You can test the processor outside of the ASAP system because the interfaces and generative classes of the Java processor are all independent of the ASAP system and its classes (the generated InputBeans and output are not tied to ASAP). To run the processor, a `TestCase` is generated once (with a sample test based on information at the time of creation), after which the developer owns it and can extend it.

The unit test framework initiates all tests in test subfolder. Unit testing is implemented as a JUnit test. JUnit tests can optionally be run with the JDT Debugger.

Figure 12-2 shows the generated test case and how it relates to the processor, input, and output files.

Figure 12-2 Generated Test Cases



The `TestCase` simulates the proxy for each individual test, and:

- Creates an implementation of the interfaces, either the real implementation or a stubbed test implementation.
- Generates input and output beans.
- Invokes the processor.

The `TestCase` is a JUnit`TestCase`. Each `TestCase` can contain many tests, and each test is defined by a no-parameter method beginning with "test".

The generated `TestCase` has a framework that provides a test. The test runs based on input files, which find the data and test criteria for a particular test. This framework enables developers to create simple files to define new tests. This works for any standard type of test where you pass in data and check the request to ensure it was sent as expected, and that the returned exit type is the one you expected. Also, this allows for a simple, standard response to be used inside the test.

Sample test classes are provided for simulating `IExit` and `ILogger`. A base output class provides the methods required for output classes.

Running Unit Test Cases

Run the `TestCase` class as a JUnit test, or as a Java application. Running as a JUnit test provides a richer user experience by providing results in the JUnit view. Running as a Java application allows the `TestCase` to be run as part of an automated test framework. Java application test case results appear in the Eclipse IDE in the Console view.

To run unit test cases:

1. Right click the TestCase class and select **Run As**.
2. Select **JUnit Test** or **Java Application**.

   Design Studio displays the results in the JUnit view or Console view, depending on your selection in step 2. Logging information is sent to the Console View.

Running Unit Tests with the JDT Debugger

To run unit test cases with the JDT debugger:

1. Set breakpoints in your Processor class as desired.
2. Right click the TestCase class and select **Run As**.
3. Select **JUnit Test** or **Java Application**.

   The unit test is run and the debugger will break as appropriate, allowing for full debugger functionality, including variable inspection and code stepping.

Understanding Unit Test Property Files

You use a set of property files to set up a unit test (both are property file and follow the Java property file format):

- **testdata** file (for example, **TestExample.testdata**)
- **testinfo** file (for example, **TestExample.testinfo**).

> **Note:**
>
> The **testinfo** file is optional. Design Studio uses defaults if it is not present.

Testdata file

The **testdata** format for naming the input parameters is similar to that within a work order. However, you must populate the test data with atomic action labels (and not service action labels). Run the unit test as if the parameters have been previously defaulted.

Apply the defaults that are normally set by the SARM (based on what is configured in the atomic action) as if they had been applied in the test data (the processor runs after those defaults have been set). The unit test data should be based on data that has already been defaulted and based on names relating to the atomic action label (and not the service action label).

When you fill in the test data for compounds or incoming repeating elements, use square brackets to indicate the index for a compound as in the following example.

```
# Example Action Processor input property file
NETID=ERIC-SDP_3-6-2-HOST
MSISDN=0701234567
FAF_LIST[1].FAF_N=0701237777
FAF_LIST[1].TSC=O
FAF_LIST[1].RCO=1
FAF_LIST[1].K=400
FAF_LIST[2].FAF_N=07052
FAF_LIST[2].TSC=4
FAF_LIST[2].K=100
FAF_LIST[3].FAF_N=071
FAF_LIST[3].K=500
```

Testinfo file

You can use this optional file to define the properties for which you are testing. You can also define what expected request the processor should create, the expected canned response returned to the processor, the expected exit type and whether it should be tested.

> **Note:**
>
> If you do not define a **testinfo** file, then by default the test case only tests whether the exit type is *succeed* (that is, to confirm that the test data has gone through).

```
# Example Action Processor test info property file
request.check=true
request.value=Test Message
response.value=Test Response
# Exit Type values:
# SUCCEED
# FAIL
# RETRY
# MAINTENANCE
# SOFT_FAIL
# DELAYED_FAIL
# STOP
exittype.check=true
exittype.value=SUCCEED
```

If you wish to have multiple request and response values in your test, you can specify multiple values in the **testinfo** file. Add a dot separated numeric suffix to the value (starting at 1).

If your request or response has multiple lines or special character, follow the standard Java property guidelines.

```
# Example Action Processor test info property file
request.check=true
request.value.1=Test Message 1
request.value.2=Test Message 2
response.value.1=Test Response 1
response.value.2=Test Response 2
# Exit Type values:
# SUCCEED
# FAIL
# RETRY
# MAINTENANCE
```

```
# SOFT_FAIL
# DELAYED_FAIL
# STOP
exittype.check=true
exittype.value=SUCCEED
```

Configuring a Unit Test

To configure a unit test:

1. Select **File**, select **New**, then select **File**.
2. Create a file ***name*.testdata**.

   For example, you might create a file called **TestExample.testdata**.

   > **Note:**
   >
   > Place this file in a subfolder of the action processor implementation package named **test**.

3. Enter the text for the file.

   The file format is a Java property file, so each entry specifies the parameter and its value.
4. Repeat steps 1 and 2 as necessary to create a second file ***name*.testinfo**.

   For example, you might create a file called **TestExample.testinfo**.

Understanding Java Libraries in Design Studio

There are several types of Java libraries available in Design Studio.

Referenced Libraries

Activation libraries are utilized by many cartridges and include the following:

- **studio_2_6_0.jar**: This library contains the base implementation files extending JProcessor for auto-generated Java stubs and auto-generated CLI code. For more information about JProcessor, see the *ASAP Java Online Reference*.
- **asaplibcommon.jar**: This library contains the core ASAP packages. For more information, see the *ASAP Java Online Reference*.
- **JInterp.jar**: This library contains the jinterpreter packages, classes, and methods that you can use to develop Java programs that the JNEP uses to communicate with NEs. For more information about the jinterpreter packages, see the *ASAP Java Online Reference*.

Activation libraries are automatically added to the project when you create an action processor. They are added to the project classpath to enable the Java development toolkit access.

> **Note:**
>
> - The **studio_2_6_0.jar** file is not installed by the ASAP installation. The **studio_2_6_0.jar** must be added to the ASAP installation prior to deployment of a Design Studio-created cartridge. Configure the **studio_2_6_0.jar**, **asaplibcommon.jar**, and **JInterp.jar** files on the server. See the discussion of installing a cartridge using Design Studio in *ASAP Installation Guide*.
> - When you are packaging a cartridge, exclude the **studio_2_6_0.jar**, **asaplibcommon.jar**, and **JInterp.jar** files. These JAR files are installed on the Activation server and are shared by all cartridges If you include these JAR files, Design Studio generates an error.

Other Libraries

Add other libraries to the **lib** folder under the project. Update the Java project properties to set the Java buildpath to make use of those libraries. See Eclipse help for adding folder or packages to the Java buildpath.

In the Project editor **Packaging** tab, select **Libraries** to display any jar files contained in the **lib** folder.

Programming Best Practices

The following sections include programming best practices applicable for writing Java implementations.

Using Default Values

Avoid hard coding default values in the Java methods. If there is a need to set a default value for one or more parameter the atomic action default configuration should be used (see **tbl_asdl_parm**).

Even if a default value has been configured in the cartridge (**tbl_asdl_parm**) for a particular parameter there is no guarantee that a default will be assigned in the customer specific service model (for example common service model), therefore the Java code cannot assume that the parameter will have a value and should therefore verify that it is not **NULL** before attempting to use it.

Enabling Value and Range Checking

The Java code must verify that parameters have a non-null value and log an error to the diagnostic file if such a parameter is missing (even if it is expected that it will be configured as a "required" parameter within the SARM) that are needed by an NE to ensure successful execution of the provisioning command. This checking is often used in common service modeling scenarios where it is not possible to perform error checking at the atomic action level. For example, a parameter may be required by one vendor, technology and software load and not another.

Perform value and range checking of atomic action parameters where possible within the Java code when an NE does not respond with a meaningful error message indicating which parameter has an invalid range/value.

If an NE expects a variable to be padded in some way the cartridge should perform the padding.

Logging Diagnostic Messages

Ensure that the ASAP core code (as well as cartridge code) does not write to **stdout** and **stderr** unless absolutely necessary. Instead, diagnostic messages should be written to the ASAP diagnostic files when required. For more information, see the Java **diag** method in the **Diagnosis** class in the *ASAP Java Online Reference*.

When ASAP is started, **stderr** and **stdout** messages are explicitly redirected to a file called **ASAP.Console**. For more information about the **start_control_sys** script that is called by the **start_asap_sys** script, see *ASAP System Administrator's Guide*. Writing to **stdout** and **stderr** can result in the **ASAP.Console** file dramatically increasing in size.

When logging optional parameters to the diagnostic files be sure to check if they have actually been defined first (including the **MCLI** parameter which is optional if **ID_ROUTING** is being used).

Do not log passwords of any kind (NE login passwords, database connection passwords etc.) to the ASAP diagnostic files.

Remove all internal debugging related diagnostic messages from the cartridge code when unit testing by the cartridge developer is complete.

Three diagnostic logging levels can be used within the cartridges. The developer can use **KERN** that should provide diagnostic messages more technical and debugging related. Use **LOW** for diagnostic message that are more cartridge related to show important information during development phase and test phase. Use **SANE** for diagnostic messages that are more informational. For more information about diagnostic levels, see *ASAP Administrator's Guide*.

Log messages which are stored in SARM database table **tbl_srq_log** to provide cartridge related information about work orders. For the telnet base cartridge, ASAP has already implemented that functionality, but for the CORBA, SOAP and another protocols you need to implement log messages, providing information about which method was run, and provide all atomic action parameters implemented in the method, log NE response, and error messages.

TCP/IP Message Parsing Options

When using the TCP/IP protocol you can take the following two approaches when parsing responses from the NE:

- parsing the raw response
- using the virtual screen in conjunction with ASAP core method calls.

Parsing the raw response means that more cartridge code is required, however it results in improved performance. In domains such as wireless where high volumes of work orders are expected, consider parsing the raw response from the NE.

The virtual screen mechanism extracts only the meaningful text strings from the responses and places them in the correct position on a two dimensional virtual screen where responses may be extracted using Cartesian coordinates. This approach results in less cartridge code however it decreases the performance of the cartridge. Use the virtual screen approach in low volume scenarios where ease of implementation is preferred.

Use of Journal Functionality

Some switches provide a journal ID as a response when a command is processed. If a subsequent error occurs on a later provisioning activity (either to the same switch or a different one) and rollback is therefore initiated, the journal ID can be used to undo commands that have previously been processed. This way, the cartridge does not have to keep track of exactly what commands were performed or query the switch in anticipation of rollback being performed (for example, to get the features on a line before a delete is performed so that they could be reapplied to the line at a later time). The journal IDs do however need to be remembered as each command is processed until the work order is completed.

Cartridges must support journaling capability where provided by the NE and should support use of this approach for rollback purposes.

13

Creating Java User Exit Types

This chapter describes how to create Java implementations for network element (NE) connections and atomic action scripts that implement MML commands for Oracle Communications ASAP.

Developing Return Parameters in Java Action Processors

The following sections provide information about the Java action processor:

About Return Parameters in Java Action Processors

Parameters are returned individually as a name value pair using the following API calls:

- returnCSDLParam
- returnRollbackParam
- returnInfoParam
- returnGlobalParam

Parameters can also be returned in a properties list which can contain multiple name value pairs using the following API calls:

- returnCompoundCSDLParam
- returnCompoundRollbackParam
- returnCompoundInfoParam
- returnCompoundGlobalParam

This API is available the Java JProcessor class described in the *ASAP Java Online Reference*.

Configuring Java Methods for Return Parameters to SARM

For Java methods that perform querying, depending on the value of the atomic action parameter **RET_PARM_TYPE** responses must be passed back to the SARM as either service action parameters, work order parameters, information parameters or some combination of these as follows:

- C - service action parameters
- W - work order parameters

- I - information parameters
- IC - information parameters and service action parameters
- IW - information parameters and WO parameters

If a default value is not provided for the **RET_PARM_TYPE** parameter or if it is left out of the atomic action parameter list then no parameters is returned from the query. The parameter names for service action and work order parameters must not conflict with the parameter names that come in on the work order therefore parameters of type "C" and "W" must be prefixed with the token "OLD_". Information parameters do not require this prefix.

Query responses should be parsed where possible rather than passing raw responses upstream including responses that are organized into columns. In general the format for the labels would be *feature tag_column header* = *value*. For example a query for feature information that results in the following response:

```
NAME PROV ACT NPI C-NUMBER
CFU ...CALL FWD Y A I 6742727
```

Should return the following:

```
CFU_PROV = Y
CFU_ACT = A
CFU_NPI = I
CFU_C-NUMBER = 6742727
```

If a column does not have a value then no parameter needs to be defined for that item.

Data extracted from a switch printout (for example, a query) must be passed back to the SARM as NE history for retrieval by clients such as OCA. For TCP/IP telnet the **startResponseLog** and **returnResponse** Java methods perform this automatically. For non-TCP/IP telnet protocols such as CORBA, the log method must be explicitly invoked to capture the name value pairs.

If an error occurs on the NE, an error text variable and an error code variable (if an error code is present) must be created and passed to the SARM as service action parameters. The value passed back for the exit text should be a meaningful alphabetic string created from the NE response. These variables are often used in customer defined atomic action spawning logic. The naming convention for the error text label and value is:

*technology_action*_entity_*EXIT_TEXT = generic error*

For example,

```
AUC_ADD_SUBS_EXIT_TEXT = SUBSCRIBER_ALREADY_EXISTS
```

The naming convention for the error code label and value is:

technology_action_entity_EXIT_CODE = <error code>

For example,

```
AUC_ADD_SUBS_EXIT_CODE = 00016
```

For all atomic actions that ran successfully on the NE, the error text and error code passed back to the SARM should be set to the value "SUCCEED" and "–1" respectively. For example:

```
AUC_ADD_SUBS_EXIT_TEXT = SUCCEED
AUC_ADD_SUBS_EXIT_CODE = -1
```

If it is not possible to determine the context for which the error occurred (for example for some NEs an error code and/or error text is not provided or cannot be interpreted) than the exit text and exit code should be set as appropriately as possible to reflect the error even if they may not be as visually meaningful.

```
AUC_ADD_SUBS_EXIT_TEXT = UNDEFINED
AUC_ADD_SUBS_EXIT_CODE = exception number
```

If an error code is not provided by the NE leave it without a value.

The presence of the generic error text will ensure that the cartridge implementation methods are compatible with a common service model. The presence of the technology token in the naming convention prevents collisions from occurring when similar atomic actions are being run on multiple NEs from a single service action (for example a service action to create a new subscriber may mean that the subscriber needs to be created on the FNR, AUC and HLR).

In common service modeling scenarios the error text code needs to follow the format described above; however, currently the team is using the actual user-defined exit type label (stored in the config file) as the label for the service action parameter. This includes the vendor, technology and software load which means that spawning logic implemented by customers would need to be implemented as follows:

```
A_DO_SOMETHING ( if ERIC-AUC_3-1_ADD_SUBS_EXIT_TEXT = SUBSCRIBER_ALREADY_EXISTS ||

                 if NOK-AUC_3-1_ADD_SUBS_EXIT_TEXT = SUBSCRIBER ALREADY EXIST ||

                 if NT-AUC_7-2_ADD_SUBS_EXIT_TEXT = SUBSCRIBER ALREADY EXIST )
```

The future guideline will likely be to provide both a vendor specific code and a generic code. There could be hundreds or thousands of codes coming back from a vendor and it may be difficult to map them uniquely across multiple vendors. The generic code is the cartridge interpretation and the vendor code gives the service modeler access to the precise code if desired.

When a hard failure is detected by the cartridge (this means a call will need to be made to return the core exit type), prior to exiting from the cartridge code the following two information parameters must be created:

```
USER_EXIT_TYPE = <the user defined exit type tag>
USER_EXIT_DESC = <the user defined exit type description - human readable description of
the error>
```

The description should be retrieved from **tbl_user_err** along with the base and core exit types. If the description is not available the actual error message from the switch should be provided.

An enhancement has been opened on the user-defined exit type mechanism to ensure that this is handled automatically by the core in the future. The core will automatically generate these labels, populate them with the user type and description and pass them back to the SARM as information parameters.

When a hard failure is detected on a rollback atomic action (this means a call will need to be made to return the core exit type), prior to exiting from the cartridge code the following two information parameters must be created:

```
ROLLBACK_USER_EXIT_TYPE = <the user defined exit type tag>
ROLLBACK_USER_EXIT_DESC = <the user defined exit type description - human readable
description of the error>
```

If the method is set to soft fail when a certain error is received the error code value should still be set to the generic error (because additional atomic actions may need to be spawned based on the error that has occurred).

Return Parameter Types

The following sections describes return parameter types and usage.

> **Note:**
>
> Any return parameter cannot exceed 255 characters. If it does, the return parameter value will be empty, and the information will not be returned.
>
> To avoid this situation, split large return values into multiple return messages when you implement your Java code.

Global Returned Parameter

Global parameters provide contextual information that different service actions can use. These global parameters are valid for the entire work order scope. Local parameters have precedence over Global parameters if the local parameters are defined.

Global and local work order parameters can be defined when you create a work order Activation Test Case, or an NE Template using Design Studio for ASAP. For more information, see *Design Studio for ASAP*.

Service Action Returned Parameter

Parameters defined in a service action overrides global parameters and there is no limit to the number of parameters you can associate with a service action.

Service action parameters are returned to give context between different atomic actions, and are valid in the service action scope. Returned service action parameters overwrites the previous parameter of the same type. Any subsequent atomic actions associated to the service action use the returned value.

Atomic Action Returned Parameter

Atomic action parameters are not returned explicitly; however, service action parameters are returned which may be implicitly be re-mapped to subsequent future forward atomic actions.

Returned Information for Upstream Purposes

SRP can only retrieve these parameters. Used usually for upstream information purposes. Error code and diagnostic information can be set in this type of return parameter. These parameters are not used parameter data for subsequent service model interactions at the service action or atomic action level.

Information parameters are returned for the upstream system only for future retrieval. These parameters are not available to any future forward or rollback atomic actions within the current or other service actions.

Indexed Rollback Returned Parameter

Rollback parameters may be returned as indexed scalar or compound parameters. However, each separate instance must be returned as a name-value pair with the name corresponding to

the correct index. For example, name=BASE[1] = AAA, BASE[2] = BBB. For a compound each element within the structure and its instance number must be explicitly returned.

Use Cases for Returning Parameters

The following section will outline some of the current best practice guidelines used in the field to address particular use cases implemented in existing cartridges.

Query for Rollback Information

The convention is to return service action parameters with the prefix ?OLD_?, which will then be the service action service parameters used for any rollback atomic actions.

There may be uses where it is desired to simply return the service action parameter with the same name as in the forward scenario if this simplifies the service model from having to re-map the service action to atomic action parameter in different context's.

Any parameter whether it is service action or ROLLBACK can be used in a Rollback atomic action; however CDSL parameters are persisted in the service action scope.

Error and Diagnostic Information

Generally error parameters are returned as Information parameters to the SRQ to be retrieved later by an upstream system. This error information can be returned as a series of one or more information parameters, or it can be custom encoded into a single value to be associated with one parameter name. For example, Name = ERRORINFO, Value = ?ERRORCODE | ERRORDESCRIPTION | MODULE?. This custom encoding needs to agree with the upstream system which will decode the single value.

If an error value is required as part of the evaluation expression in the service model for subsequent service logic, then the error value should be passed back to SARM as service action parameters and named appropriately in the parameter name.

Note that there are no specific Error parameters, it depends its uses.

Configuring Response Logging and Network Element History Capture

For stream-based protocols supported by core ASAP (for example TCP/IP Telnet) the Java startResponseLog and returnResponse methods should be called whether the virtual screen is being employed by the cartridge or not. This results in switch responses being stored in tbl_srq_log where they can be retrieved by upstream systems or viewed through OCA and hence explicit calls to the Java log method can be avoided.

Response logging can be activated/deactivated by setting the NE_CMD_LOG_ON option (which can be configured on a per NEP basis) in ASAP.cfg.

For stream-based protocols supported by core ASAP (for example TCP/IP Telnet), whenever confidential data must be sent to an NE the data should be prevented from being written into tbl_srq_log. There are two sets of methods that can be used as wrappers around the ?send? method calls to control core ASAP behavior in this manner:

- The disableCommandLog and enableCommandLog method calls result in no data being written into tbl_srq_log and should therefore only be employed if absolutely necessary (this impairs the ability to debug).
- The maskCommandLog and unMaskCommandLog result in asterisks being written to tbl_srq_log instead of the raw characters and is the preferred approach when secure data is being managed.

RULE: for non-stream based protocols such as CORBA it is not possible to use the core ASAP response logging functionality and therefore explicit calls to the log method (JProcessor class) must be made within the cartridge to capture NE history into tbl_srq_log. Ensure that for such protocols explicit calls are made to the log method to record the API call that is being made as well as the return code and/or return text received back from the NE if they are available. For performance reasons at this time do not place calls to log each parameter used in the API call (because these are available through OCA by querying on the work order and also through the diagnostics). In the case where XML documents are being constructed within the cartridge and transmitted using non-stream based protocols the entire XML document should be recorded using a call to the log method.

User Defined Exit Types

Where possible, user-defined exit types will be provided in the cartridge.

In the absence of a user-defined exit type configuration (for example the customer has removed those provided by the cartridge) the cartridge default should be to fail responses that lie outside the normal success detection criteria.

The mapping between a message received from the switch and its corresponding user-defined exit type (user_type) should be kept within a cartridge specific configuration file with the following naming convention:

```
<vendor>_<technology>_<swld>_UserExitTypes.xml
```

The mapping between the user-defined exit type and its corresponding base exit type (base_type) is contained in tbl_user_err.

Improvements to the user-defined exit type lookup mechanism in ASAP core are pending.

The following data columns in tbl_user_err must be populated:

- NE_VENDOR
- TECH_TYPE
- SFTWR_LOAD
- USER_TYPE
- BASE_TYPE
- DESCRIPTION
- SEARCH_PATTERN

The naming convention of user_type, <vendor>_<technology>_<swld>_<error tag>, is no longer required. user_type is for <error_tag> only.

tbl_user_err currently has a 20-character limit and therefore some truncation of the user_type may be required for it to be successfully loaded into the table. An issue is opened on core to increase the size of this table.

The data contained in tbl_user_err must be loaded into a RAM cache upon startup of the NEP.

When regular expressions (regex) are used to perform pattern searches on responses from NE, the following situations should be considered when defining a search pattern (to avoid exception and stack overflow). If the NE response is greater than 1400 characters, then suggest to use the search pattern as follow:

```
((?s).)*<search string>((?s).)*
```

Use the fail exit type when the NE indicates that an order cannot be processed due to incorrect parameter values.

In the case where numerous (for example hundreds) of responses/error codes are described in the NE specification, a subset of the most commonly occurring responses will be supported.

Apply the following guidelines when assigning exit types to error messages:

- Hard Fail—used for non-recoverable errors that cause the immediate failure of a work order. For example, when an invalid provisioning parameter has been used.
- Soft Fail—may be used when for minor errors that should not stop the provisioning of the order for example assigning a feature that has already been added to the subscriber line.
- Retry—used when an activation request fails due to reasons other than data errors. For example, if the NE is temporarily unavailable or too busy to handle the provisioning request.
- Fail—used when the NE indicates that an order cannot be processed due to incorrect parameter values

User defined exit types are most often configured without associated atomic actions (see tbl_user_err), however on occasion when the same Java method is associated with two different atomic actions it may be necessary to trigger different exit types. This is most often done as project configuration work and is not typically part of the cartridge.

The following is the naming convention for user-defined exit type:

```
<Severity Level>_<Error Label>
```

The following is the mapping between severity level and base type

Table 13-1 Severity Level Mapping to Base Type

Severity Level	Base Type
SUCCEED (S)	SUCCEED
CRITICAL (C)	FAIL
WARNING (W)	SOFT_FAIL
RETRY (R)	RETRY_DIS

The following are the common labels used in user-defined exit types:

Table 13-2 Common User-Defined Exit Type Labels

User Exit Type	Base Type	Search Pattern	Description
S_SUCCEED	SUCCEED	TBD	Succeed.
C_FAIL	FAIL	TBD	Fail.
C_INVALID-DATA-TYPE	FAIL	TBD	Data is specified in wrong data type.
C_DATA-OUT-OF-RANGE	FAIL	TBD	Data value is out of range.
C_INVALID-DATA-LNGTH	FAIL	TBD	The length of the given data exceeds the limit.
C_CMD-SYNTAX-ERR	FAIL	TBD	Command syntax error.
C_MISSING-PARAM	FAIL	TBD	Missing parameter in command (MML).
C_MISSING-DATA	FAIL	TBD	Expected data is missing.
C_SRV-NOT-IMPLMNT	FAIL	TBD	Service is not yet implemented.

Table 13-2 (Cont.) Common User-Defined Exit Type Labels

User Exit Type	Base Type	Search Pattern	Description
C_FEAT-NOT-IMPLMNT	FAIL	TBD	Feature is not yet implemented.
C_UNKNOWN-ERR	FAIL	TBD	Unknown response from the NE.
C_MATCH-NOT-FND	FAIL	TBD	No matching exit type is found.
S_DATA-NOT-FND	SUCCEED	TBD	No data/object is found in query.
C_DATA-NOT-FND	FAIL	TBD	No data/object is found in query.
W_DATA-NOT-FND	SOFT_FAIL	TBD	No data/object is found in query.
S_DATA-EXISTS	SUCCEED	TBD	Data/object already exists.
C_DATA-EXISTS	FAIL	TBD	Data/object already exists.
W_DATA-EXISTS	SOFT_FAIL	TBD	Data/object already exists.
S_WARNING	SUCCEED	TBD	Warning Message.
W_WARNING	SOFT_FAIL	TBD	Warning Message.
C_TIMEOUT	FAIL	TBD	Time out.
R_BUSY	RETRY_DIS	TBD	Network is busy.
R_CONNECTION-LOST	RETRY_DIS	((?s).)*Connection to the NE lost((?s).)*	Connection to the NE is lost. You must define this user exit type in your cartridge.
R_BROKEN-PIPE	RETRY_DIS	((?s).)*Broken pipe((?s).)*	Broken Pipe. Connection to the NE is lost. You must define this user exit type in your cartridge.
C_PROVCART-EXCEPTION	FAIL	((?s).)*ProvCartridgeException((?s).)*	Provisioning Cartridge Exception. You must define this user exit type in your cartridge.
CL_EXCEPTION	FAIL	((?s).)*Exception((?s).)*	General Exception. You must define this user exit type in your cartridge.

14

Documenting ASAP Cartridges

This chapter describes how to use Oracle Communications Service Catalog and Design - Design Studio to document Oracle Communications ASAP cartridges.

About Design Studio Cartridge Documentation

Design Studio provides a cartridge guide generation feature that simplifies the documentation process. The feature becomes available whenever you create a network cartridge project. Design Studio provides a template for the guide, and generates most of the cartridge documentation with information added to entities modeled in the project and the information entered in various editors during development process. An HTML version of the cartridge documentation can be found in the **doc/guide** folder.

The following list describes the components required to ensure that the cartridge contains all necessary information:

- Make sure all description fields for various entities, default values for parameters, and data restrictions are completed.
- The Document **Command Overview** tab in Action Processor editor should describe the MML command string that ASAP sends to the network element (NE). Whenever possible (for example, with the TCP/IP NE interfaces) document the logic of the conditional building of the MML command as pseudo-code, explaining the conventions/syntax used. For example:

  ```
  Router# configure terminal
  if(INTERFACE_TYPE="optical"){
  Router(config)# controller SONET {SLOT}/[MODULE]/{PORT}
  Router(config-controller)# au-4 {AU4_NUMBER} tug-3 {TUG3_Number}
  }else ……
  ```

  When this is not possible (for example, in the case of CORBA or web service NE interfaces), document the API calls and parameters and if the case provide the request XML (web services). For example:

  ```
  This is a sample of SOAP Activate service request:
  <?xml version='1.0' encoding='UTF-8'?> <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://
  schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <SOAP-ENV:Body>
  <ns1:submitSync xmlns:ns1="urn:ProvisioningRequestServer" SOAP-
  ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <request
  xmlns:ns2="java:provision.services.web.rpc.ejb" xsi:type="ns2:ProvisionRequest">
  <body xmlns:ns3="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns3:Array"
  ns3:arrayType="ns2:ProvisionReqEntity[1]"> <
  xmlns:ns5="http://schemas.xmlsoap.org/soap/encoding/" </header>
  </request> …….
  ……..etc
  </SOAP-ENV: Envelope>
  ```

- Document the output parameters, returned by the cartridge, under Action Processor **Output** tab. Provide the label and the value format for returned data as Service Action or INFO parameters. Such parameters are retrieved by the upstream system or can be used in the service model for conditional executions of the next mapped atomic actions. Make

sure that returned labels are unique and easy to be identified. It is a good practice to use as prefix the atomic action name. For example:

```
Return as Service Action parameter:
A_NT-DMS100_SN06_ADD_LINE_RETURNCODE=<user defined exit type>
Return as info parameter:
A_NT-DMS100_SN06_ADD_LINE_RETURNINFO=<NE error code>:<NE error description>
```

- Enter information under **Development Notes** tab. Describe the business logic, triggered by the atomic action execution, as implemented in the Java action processor class. For example:

```
Business Logic Flow:
- Get atomic action parameters from the work order
- Build the User object and set the content (parameters)
- Check if the system is in loopback or not
- If loopback, print the log the API call and parameters
- If not loopback send request to the NE by calling addSubscriber
- Set atomic action exit type by using user defined exit types
- Return out parameters (Service Action and INFO)
```

15

Work Order Processing and Sample Work Orders

This chapter describes how to create sample work orders for Oracle Communications ASAP cartridges.

Work Order Processing Overview

At a high level, the process of configuring the interaction between provisioning and activation includes the following steps.

1. Familiarize yourself with the order data definition that is associated with an automated task in the predefined provisioning model (the ASAP cartridge). In this case, refer to the Provisioning system's order schema.
2. If you are using an XML editor like Design Studio to view the schema, you can generate sample XML order data (see "About JSRP Sample OSS/J Work Orders").
3. Based on the automated task in the provisioning system, determine the service actions and atomic actions that are required to activate the service on the network elements (NEs). Table 15-1 shows the C-ADD_LINE mappings table and displays a sample service action with associated atomic actions.
4. Review the Activation schema **ServiceModel.xsd** file to review the components of a service model. For more information, see "About Cartridge XML Schemas."
5. To handle the order data from the Provisioning system, define one or more atomic actions and associated parameters of XML type (see "XML Parameters") so that the upstream order data is available to the network implementation (provisioning method) at run time. For an example of an atomic action parameter configuration, see Table 15-2 and Table 15-3.
6. You can define additional atomic action parameter labels of XPath type (see "XPath Parameters") using an XPath expression to control the spawning of ASDLs based on data of the XML document, provided that the evaluation of XPath expression resulted in a scalar value (see "About Atomic Action Spawning Logic"). You can also define atomic action parameter labels of XPath type to pass information of the XML so that the system integrators at the network element processor (NEP) level do not have to manipulate the XML document. Regardless of whether the you define atomic action parameter labels of XPath type, system integrators can retrieve information of the XML at the NEP level.

   If you want to directly manipulate the XML at the NEP level, you can retrieve the raw XML as a Document Object Model (DOM) object through Java-enabled NEP's public Java APIs. Refer to the ASAP Online Reference.

   You can use XPath technology to retrieve certain information of the XML document and then marshal the information to activate the NE.
7. Save the changes made to the service actions and atomic actions and then package the service model so that it can be deployed to the Activation system later through the SADT command line interface.

General Work Order Processing

A work order that has been submitted for provisioning is processed as follows:

1. A work order enters the service request processor (SRP) or Java SRP (JSRP) server, where it is initialized. Information such as the due date, order number, and priority of the work order is fed into ASAP.
2. The SRP converts work order information such as universal service order codes (USOCs) and service offerings (SOFFs) into service requests appropriate for the Service Activation Request Manager (SARM). During this process, the SRP translates work order components to their corresponding Common Service Description Layer (CSDL) commands and parameters that are sent to the SARM.
3. After the work order enters the SARM, CSDL commands are mapped into a set of corresponding Atomic Service Description Layer (ASDL) commands.
4. ASDL commands and parameters are translated to switch-specific commands for execution on the network element (NE).
5. ASDL commands with their corresponding parameters are directed to the appropriate switch (network) elements.
6. Execution on the NE results in various responses that are communicated back to the SARM.

OSS/J or Web Service Work Order Processing with XML or XPath Parameters

ASAP has the ability for structured XML order data to be passed from an upstream system through the Activation system and to be flowed back to Provisioning or another upstream system. Using this mechanism, the upstream system can include structured XML data on an Activation order. The process is as follows:

1. The user enters multi-instance data in the Provisioning application or using the XML API as part of a process flow.
2. The upstream system sends some or all of the order data. The upstream system performs the following actions:
    a. Creates an OSS/J XML or web service order request containing the order data information. Instead of marshalling the multi-instance order data into name/value pairs, the multi-instance XML order data is included as a parameter of the provisioning work order. See "Sample OSS/J Work Order with Conditional Logic Using XML Parameters" and "Sample OSS/J Work Order with Conditional Logic using XPath Parameters" for a sample activation work order with multi-instance XML order data.
    b. Submits the OSS/J order request to the JSRP of the Activation system for provisioning.
3. When the incoming order request comes into the JSRP, the JSRP determines whether XML data is present as part of the request. The JSRP sends a commit acknowledgement to the Provisioning system if the work order is accepted in the Activation system.

> **Note:**
>
> If a work order already exists in the Activation system with the same work order ID, the new work order is rejected.
>
> Any work order data that exceeds allowed data sizes will cause the work order to be rejected.

4. The JSRP saves the XML order data to the SARM database and updates the order request with a reference ID that is returned by the database.
5. The SARM periodically picks up work orders and starts provisioning the work order.
6. When the work order is provisioned, the SARM loads the XML data from the database table by using the reference ID. The work order can contain the following types of XML parameters:
   - XML – If passing complex structured data downstream.
   - XPath – The SARM runs an XPath expression against the XML data in the following cases:
     - An XPath parameter is included in the CSDL spawning logic to determine whether an ASDL should be spawned or not.
     - An XPath parameter is used to spawn multiple instances of the same ASDLs depending on how many instances of XML elements are present in the work order.

   If an XPath parameter is used as part of spawning logic, the evaluation of the XPath expression must result in a scalar value. The XPath parameter can also be used to conditionally run ASDLs multiple times depending on how many instances of XML elements present in the XML document. See "Sample OSS/J Work Order with Conditional Logic using XPath Parameters."
7. The SARM examines each ASDL and evaluates its corresponding spawning expression based on the work order parameters and CSDL parameters. This examination encompasses the data within the multi-instance XML order data from the Provisioning system by referring to the CSDL label of an XPath type parameter. In the case of an ASDL, if the spawning expression is evaluated to be true, the ASDL is scheduled for provisioning with its mapped ASDL parameters. For indexed ASDL parameters, the ASDL may be provisioned multiple times for multi-instance order data (see "Indexed Parameters" for more information). If the ASDL spawning expression evaluates to false, the ASDL is omitted from provisioning. See "About Atomic Action Spawning Logic."

   Spawning can fail under the following circumstances:
   - The XML parameter that the XPath evaluates on is missing from the work order
   - The XML document that the XPath evaluates on is not well structured
   - The work order contains an invalid XPath expression
   - The evaluation of XPath expression results fails
   - The evaluation of an XPath expression results in a non-scalar value
8. The SARM provisions the ASDL to the NEP.
9. The NEP sends all ASDL parameters to the Java-enabled NEP.
10. If the ASDL contains an XML parameter, the Java-enabled NEP loads the XML data from the SARM database table and makes the raw XML available as the value of the XML parameter.

11. If the ASDL contains an XPath parameter, the Java-enabled NEP evaluates the associated XPath expression when the value of the XPath parameter is requested.

    For more information on the XML and XPath parameter types, refer to the ServiceModel schema reference material, accessible through the ASAP Online Reference, "XML Parameters," and "XPath Parameters."

12. After the MML command is sent to the NE, custom provisioning code may optionally update the XML data with the NE's response by calling the Java-enabled NEP API (for example, **returnXMLCSDLParm**(name, value)) so that subsequent ASDLs may make use of this information. For more information, refer to the ASAP Online Reference.

13. If there is an update to the XML parameter, the Java-enabled NEP saves the modified XML data to a SARM database table and updates the parameter value with a new reference ID before returning the exit status of ASDL along with return parameters. These parameters include updated global, CSDL, and information parameters that are returned to the NEP (see "Return Parameter Types" for more information).

14. The NEP sends the ASDL's exit value and all returned parameter values for each returned parameter (see "Configuring Base Exit and User Exit Types ").

15. The SARM continues to provision the next ASDL until the work order completes successfully or fails.

16. After the work order is finished provisioning, SARM publishes the work order event to various SRP servers (such as the JSRP) to indicate whether the work order has completed.

17. Upon receiving the work order event from the SARM server, the JSRP server publishes appropriate events (such as orderCompleteEvent, orderFailEvent) so that the automated task in the Provisioning system can update the state of the task and transition to the next task in the process flow (see "Configuring Service Action Fail and Complete Events").

About Testing Cartridge Elements with Sample Work Orders

Developer unit testing should occur as the cartridge is created so that all service actions, atomic actions, and code are fully tested. When unit testing is complete, run all of the sample work orders that have been created against the cartridge to ensure that the desired outcome is achieved.

There are two ways to test a cartridge: using the SRP Emulator or the JSRP. Both of these components are available with the ASAP installation.

About SRP Emulator Sample Work Orders

The SRP Emulator is an ASAP application server that fully emulates the complete behavior of any SRP application. It is used to create and transmit work orders to the SARM generally in the development and service modeling phases of project implementation. The SRP Emulator has no external system interface. Instead of externally generated work orders, the SRP Emulator employs user-defined test suites of work order definitions created in the SRP Emulator database for execution by the emulator. Orders in such test suites are created in Service Action format together with the appropriate parameters.

The file format is described using the following symbols and is detailed below.

```
WO <WO_ID> [WO Description]  [ORG_UNIT <WorkGroup>]  [ORIGIN <Originator>]  [SRP_STAT
<SRP Status>]  [PRIORITY <Priority>]  [SRQ_TYPE <Service Request Type>]  [USERID
<Security Userid on WO>]  [PASSWORD <Security Password on WO>]  [ASDL_TIMEOUT <atomic
action Timeout Value>]  [PARENT_WO <Parent WO (Related Order)>]  [WO_TIMEOUT <WO Timeout
Value>]  [ASDL_RETRY_NUM <Number of atomic action Retries on WO>]  [ASDL_RETRY_INT
```

```
<Interval between atomic action Retries>]  [WO_RBACK <Rollback WO upon Failure>]
[ASDL_DELAY_FAIL <Treat Failures as Delayed Failures>]  [DELAY_THRESHOLD <Delayed
Failure Threshold>]  [BATCH_GROUP <Batch group to which order belongs>]

  [WO_PARM <ParmLbl> <ParmVlu>]...  [WO_PARM <ParmLbl> <ParmVlu>]

  [BATCH_WO_PARM <BatchNum> <ParmLbl> <ParmVlu>]...  [BATCH_WO_PARM <BatchNum> <ParmLbl>
<ParmVlu>]

  [[Service Action <Service Action Command>]       [PARM <ParmLbl> <ParmVlu>]...
[PARM <ParmLbl> <ParmVlu>]       [BATCH_PARM <BatchNum> <ParmLbl> <ParmVlu>]...
[BATCH_PARM <BatchNum> <ParmLbl> <ParmVlu>]]...  [[Service Action <Service Action
Command>]       [PARM <ParmLbl> <ParmVlu>]...       [PARM <ParmLbl> <ParmVlu>]
[BATCH_PARM <ParmLbl> <ParmVlu>]...       [BATCH_PARM <ParmLbl> <ParmVlu>]][SUITE <Suite
Name> [Suite description]       [[WO_ID <WO_ID> [WO Delay] [WO Operation] [WO Due Date
Offset] [Parent WO]
                     [WO Batch Group]]...       [[WO_ID <WO_ID> [WO Delay] [WO
Operation] [WO Due Date Offset] [Parent WO]
                       [WO Batch Group]]]
\
```

where:

- **<>**: indicates a mandatory parameter
- **[]**: indicated an optional parameter
- **...**: indicates multiple occurrences of a parameter

SRP Emulator sample orders are created under a project test folder (Package Explorer view). As the tokens from the test file are positional parsed during loading, a very strict format is imposed (assuming space separation between tokens). To avoid mistakes, it is easier to build a sample test file from an old template, replacing the tokens and values with the new ones.

For more information about running a work order through the SRP Emulator, see ASAP Installation Guide.

About JSRP Sample OSS/J Work Orders

The ASAP Java SRP (JSRP) component supports upstream requests in XML format bound by OSS/J standards. Using Design Studio, you can create activation OSS/J test cases, which generate work orders targeting the JSRP. These sample work orders can be sent from Design Studio, after connecting to an ASAP environment where the cartridge to be tested has been deployed. For details on how to create and run test cases from the Activation Test Cases editor, see the Design Studio Help.

Sample OSS/J Work Order with Conditional Logic Using XML Parameters

This section describes the structure of a work order that contains XML data.

Table 15-1 shows a CSDL (C-ADD_LINE) mapped to three ASDLs.

Table 15-1 C_ADD_LINE mappings

Seq	ASDL	Condition	Condition Label	Condition Value	Expression
1	A-ADD_POTS_LINE	Always	-	-	-

Table 15-1 (Cont.) C_ADD_LINE mappings

Seq	ASDL	Condition	Condition Label	Condition Value	Expression
2	A-ADD_CODES	Equals	A1141	BD/U2B	-
3	A-ADD_OPTIONS	Equals	A1141	BD/U2B	-

ASDLs 2 and 3 have spawning logic associated with them. A-ADD_CODES and A-ADD_OPTIONS are spawned only if A1141 is present in the upstream order with the value of BD/U2B.

For A-ADD_CODES (see Table 15-2), in addition to the standard parameters (NE_ID and DN), this ASDL expects the CSDL to contain parameters MY_OMS_DATA and MY_XML_DATA. These two parameters are of XML type.

Table 15-2 A_ADD_CODES

CSDL Label	ASDL Label	Default[1]	Parameter Type	Required
NE_ID	MCLI	NA	Scalar	Yes
MY_OMS_DATA	OMS_DATA	NA	XML	Yes
MY_XML_DATA	XML_DATA	NA	XML	No
DN	DN	NA	Scalar	Yes

1 The default is not applicable to XML and XPath

The CSDL parameter MY_OMS_DATA maps to the OMS_DATA ASDL parameter.

The following is a sample OSS/J XML work order createOrderByValueRequest that shows the configuration described in Table 15-1 and Table 15-3:

```
<?xml version="1.0" encoding="UTF-8"?>
<createOrderByValueRequest xmlns="http://java.sun.com/products/oss/xml/
ServiceActivation" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mslv-
sa="http://www.metasolv.com/oss/ServiceActivation/2003" xmlns:co="http://java.sun.com/
products/oss/xml/Common" xsi:schemaLocation="http://java.sun.com/products/oss/xml/
ServiceActivation ../../xsd/XmlServiceActivationSchema.xsd http://www.metasolv.com/oss/
ServiceActivation/2003 ../../xsd/ASAPServiceActivation.xsd">
<orderValue xsi:type="mslv-sa:ASAPOrderValue">
<apiClientId>SRL</apiClientId>
<orderKey>
<co:applicationContext>
<co:factoryClass/>
<co:url/>
<co:systemProperties/>
</co:applicationContext>
<co:type/>
<primaryKey>POTS-60</primaryKey>
</orderKey>
<priority>3</priority>
<requestedCompletionDate>2005-12-01T12:00:00</requestedCompletionDate>
<services>
<item xsi:type="mslv-sa:ASAPService">
```

```
<serviceKey xsi:type="mslv-sa:ASAPServiceKey">
<co:applicationContext>
<co:factoryClass/>
<co:url/>
<co:systemProperties/>
</co:applicationContext>
<co:applicationDN>System/DEV1/ApplicationType/ServiceActivation/Application/1-0;5-0;ASAP/
Comp/</co:applicationDN>
<co:type/>
<primaryKey>C-ADD_LINE</primaryKey>
</serviceKey>
<mslv-sa:asdlRoute>TO_BE_DETERMINED</mslv-sa:asdlRoute>
<mslv-sa:serviceValues>
<mslv-sa:serviceValue xsi:type="mslv-sa:ASAPServiceValue">
<mslv-sa:name>NE_ID</mslv-sa:name>
<mslv-sa:value>TOR_REM1</mslv-sa:value>
</mslv-sa:serviceValue>
<mslv-sa:serviceValue>
<!-- Xpath type parameter with CSDL parameter name "A1141"
                                    and value "/exchange/a1141" -->
<mslv-sa:name>A1141</mslv-sa:name>
<mslv-sa:serviceValue xsi:type="mslv-sa:ASAPServiceValue">[1]
 <mslv-sa:name>MY_OMS_DATA</mslv-sa:name>
 <mslv-sa:xmlValue>
 <exchange xmlns="">
 <a1141>BD/U2B</a1141>
</mslv-sa:serviceValue>
<mslv-sa:serviceValue>
<!-- Xml type parameter with CSDL parameter name "MY_OMS_DATA" and value of order data
from Provisioning -->
<mslv-sa:name>MY_OMS_DATA</mslv-sa:name>[2]
<mslv-sa:xmlValue>
<exchange>[3]
<a1141>BD/U2B</a1141>
<codes>
<code>[4]
<poe>984</poe>
<decode>01246811</decode>
<pds_list>
<pds>2134</pds>
<pds>3265</pds>
<pds>1234</pds>
<pds>2345</pds>
<pds>4321</pds>
</pds_list>
</code>
<code>
<poe>984</poe>
<decode>01246812</decode>
<pds_list>
<pds>6789</pds>
<pds>9876</pds>
<pds>5432</pds>
<pds>2345</pds>
<pds>2354</pds>
```

1 Identifies an XPATH, where the associated ASDL is spawned only if A1141 is present in the XML order data with the value of BD/U2B.

2 A CSDL parameter on the order that references an ASDL parameter label. This ASDL parameter label of type X (required XML parameter) is associated with.

3 Root element of the XML fragment.

4 The data from the XML order. In this example, found between <code> and </code>.

```
</pds_list>
</code>
<code>
<poe>984</poe>
<decode>01246813</decode>
<pds_list>
<pds>3421</pds>
<pds>5632</pds>
<pds>1020</pds>
</pds_list>
</code>
…
</codes>
<errors>
<error>
<error_priority>1</error_priority>
<error_name>BROKEN PIPE</error_name>
<error_description>Lost switch connection</error_description>
</error>
…
</errors>
</exchange>
</mslv-sa:xmlValue>
</mslv-sa:serviceValue>
<mslv-sa:serviceValue xsi:type="mslv-sa:ASAPServiceValue">
<mslv-sa:name>DN</mslv-sa:name>
<mslv-sa:value>6742727</mslv-sa:value>
</mslv-sa:serviceValue>
<mslv-sa:serviceValue xsi:type="mslv-sa:ASAPServiceValue">
<mslv-sa:name>LATA</mslv-sa:name>
<mslv-sa:value>236</mslv-sa:value>
</mslv-sa:serviceValue>
<mslv-sa:serviceValue xsi:type="mslv-sa:ASAPServiceValue">
<mslv-sa:name>LCC</mslv-sa:name>
<mslv-sa:value>1</mslv-sa:value>
</mslv-sa:serviceValue>
<mslv-sa:serviceValue xsi:type="mslv-sa:ASAPServiceValue">
<mslv-sa:name>LTG</mslv-sa:name>
<mslv-sa:value>1</mslv-sa:value>
</mslv-sa:serviceValue>
<mslv-sa:serviceValue xsi:type="mslv-sa:ASAPServiceValue">
<mslv-sa:name>LEN</mslv-sa:name>
<mslv-sa:value>1010101</mslv-sa:value>
</mslv-sa:serviceValue>
</mslv-sa:serviceValues>
</item>
</services>
<mslv-sa:parentKey>
<co:applicationContext>
<co:factoryClass/>
<co:url/>
<co:systemProperties/>
</co:applicationContext>
<co:applicationDN/><co:applicationDN/>
<co:type/>
<primaryKey/>
</mslv-sa:parentKey>
<mslv-sa:origin>ASC Test Orders</mslv-sa:origin>
<mslv-sa:organizationUnit>POTS</mslv-sa:organizationUnit>
<mslv-sa:timeout>-1</mslv-sa:timeout><!-- Use Default -->
<mslv-sa:secureData>true</mslv-sa:secureData>
<mslv-sa:maximumDelayFail>0</mslv-sa:maximumDelayFail>
```

```
<mslv-sa:rollbackIfFail>false</mslv-sa:rollbackIfFail>
<mslv-sa:batchGroup/>
<mslv-sa:asdlTimeout>-1</mslv-sa:asdlTimeout> <!-- Use Default -->
<mslv-sa:asdlRetry>5</mslv-sa:asdlRetry>
<mslv-sa:asdlRetryInterval>120</mslv-sa:asdlRetryInterval>
<mslv-sa:asdlDelayFail>false</mslv-sa:asdlDelayFail>
<mslv-sa:externalSystemId/>
<mslv-sa:srqAction>ADD</mslv-sa:srqAction>
<mslv-sa:command>UPDATE</mslv-sa:command>
<mslv-sa:orderParameters>
<mslv-sa:orderParameter>
<mslv-sa:name>ACCT</mslv-sa:name>
<mslv-sa:value>1764571</mslv-sa:value>
</mslv-sa:orderParameter>
</mslv-sa:orderParameters>
<mslv-sa:infoParms/>
<mslv-sa:extendedWoProperties/>
</orderValue>
</createOrderByValueRequest>
```

After the MML command is sent to the NE, a system integrator can persist the response from the NE in the Activation system so that the upstream system can retrieve information from the NE's response. The response from the NE may also update the XML order data as part of Activation work order. The response from the NE can be used by:

- Subsequent ASDLs

  To send information between ASDLs, system integrators can call JNEP's API (such as returnXMLCSDLParm(name, value), returnGlobalParam(name, value)) to return CSDL parameters or global work order parameters.

  Consider the sample work order: A-ADD_CODES can pass an XML document to subsequent ASDL A-ADD_OPTION by calling returnXMLCSDLParm() function with the MY_XML_DATA parameter, where the value of parameter is the XML document. For example:

  ```
  <switch>
  <a1141>BD/U2B</a1141>
  <options>
  <feature>1100</feature>
  <feature>3232</feature>
  <feature>2000</feature>
  </options>
  </switch>
  ```

- The Provisioning system to update the order in Provisioning
- Any upstream system to update the NEs status

Sample OSS/J Work Order with Conditional Logic using XPath Parameters

This section describes the structure of a work order that contains XPath parameters.

Table 15-3 describes the same ASDL using XPath parameters.

Table 15-3 A_ADD_CODES Parameters

CSDL Label	ASDL Label	Default[1]	Parameter Type	Required	Dependent ASDL Label[2]
NE_ID	MCLI	NA	Scalar	Yes	-

Table 15-3 (Cont.) A_ADD_CODES Parameters

CSDL Label	ASDL Label	Default[1]	Parameter Type	Required	Dependent ASDL Label[2]
MY_OMS_DATA	OMS_DATA	NA	XML	Yes	-
MY_XML_DATA	XML_DATA	NA	XML	No	-
A1141	A1141	NA	XPath	Yes	OMS_DATA
CODE++	CODE	NA	XPath	No	OMS_DATA
DN	DN	NA	Scalar	Yes	-

1 The default is not applicable to XML and XPath

2 Applies only to the XPath type

The following sample code illustrates how the service model for this ASDL may appear. Note the parameter types.

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:sam="http://
www.metasolv.com/ServiceActivation/2003/ServiceModel" xmlns="http://www.metasolv.com/
ServiceActivation/2003/ServiceModel" xmlns:fo="http://www.w3.org/1999/XSL/Format">
<description>Adds a line.</description>
<atomicService name="A_ADD_CODES" xsi:type="AtomicServiceType">
<description>Adds codes.</description>
<rollbackService enable="false">
</rollbackService>
<sendParameterCount>false</sendParameterCount>
<parameter name="MCLI" xsi:type="SimpleParameterType">
<description>Host NE identifier.</description>
<required>true</required>
<default/>
<parameterValueMap>NE_ID</parameterValueMap>
</parameter>
<parameter name="OMS_DATA" xsi:type="XMLParameterType">
<description>OMS data.</description>
<required>true</required>
<default/>
<parameterValueMap>MY_OMS_DATA</parameterValueMap>
</parameter>
<parameter name="XML_DATA" xsi:type="XMLParameterType">
<description>XMLDATA.</description>
<required>false</required>
<default/>
<parameterValueMap>MY_XML_DATA</parameterValueMap>
</parameter>
<parameter name="A1141" xsi:type="XPathParameterType">
<description>Description.</description>
<required>true</required>
<default/>
<parameterValueMap>A1141</parameterValueMap>
<dependentXMLParameter>OMS_DATA</dependentXMLParameter>
</parameter>
<parameter name="CODE" xsi:type="XPathParameterType">
<description>Code.</description>
<required>false</required>
<default/>
<parameterValueMap>CODE#</parameterValueMap>
```

```
</parameter>
<parameter name="DN" xsi:type="SimpleParameterType">
<description>DN.</description>
<required>true</required>
<default/>
<parameterValueMap>DN</parameterValueMap>
</parameter>
```

In A_ADD_CODES Parameters table, OMS_DATA is an ASDL parameter that is associated with two network action labels that invoke the evaluation of an XPath expression. As a result, if the CSDL contains a parameter A1141, the SARM attempts to locate a set of data in the order designated by <a1141>. The incoming XML must identify that specific CSDL parameter name and its XPath value, an example of which follows:

```
<mslv-sa:serviceValue>
<!-- Xpath type parameter with CSDL parameter name "A1141" and value "/exchange/a1141" --
>
<mslv-sa:name>A1141</mslv-sa:name>
<mslv-sa:xpathValue>/exchange/a1141</mslv-sa:xpathValue>
```

The XML should have an XPath name and value declaration for each CSDL parameter that is subject to an XPath expression.

If the A1141 parameter exists on the order, ASAP will apply the data at the specified location in the file, in the **<a1141>** element:

```
<exchange>
<a1141>
...
</a1141>
```

The spawning logic for A-ADD_CODES (described in C-ADD_LINE mappings table) requires that a condition value of BD/U2B be defined for parameter a1141 for that ASDL to be spawned. For this ASDL to be spawned, the incoming XML must contain data formulated as follows:

```
<exchange>
<a1141>BD/U2B</a1141>
<codes>
<code>
...
</code>
...
</codes>
```

For parameter CODE, the ++ at the end of service action label CODE++ indicates that at run time, the current network action may be spawned multiple times depending on how many instances of "exchange/codes/code" are present in the work order. In addition, the network action label CODE for each A-ADD_CODE, execution will have a different value.

In the following example, the ASDL is spawned based on the evaluation of the XPath expression, and the order data contained in the exchange/codes/code location is passed to the NE.

```
<exchange>
<a1141>BD/U2B</a1141>
<codes>
<code>
<poe>984</poe>
<decode>01246811</decode>
<pds_list>
<pds>2134</pds>
<pds>3265</pds>
```

```
<pds>1234</pds>
<pds>2345</pds>
<pds>4321</pds>
</pds_list>
</code>
<code>
…
```

About Web Service Sample Work Orders

You can create web service work orders by taking the OSS/J work order information generated using Design Studio and placing it within this web service sample wrapper:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
<env:Header>
        <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-secext-1.0.xsd" env:mustUnderstand="1">
            <wsse:UsernameToken xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="unt_AF6po7ocfkMUDzde">
                <wsse:Username>username</wsse:Username>
                <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-username-token-profile-1.0#PasswordText">password</wsse:Password>
            </wsse:UsernameToken>
        </wsse:Security>
    </env:Header>
    <env:Body>
        <m:order_type xmlns:m="http://xmlns.oracle.com/communications/activation/asap/
webservices">

OSS/J_work_order

        </m:order_type>
    </env:Body>
</env:Envelope>
```

where:

- *username* is the user name for the web service user-defined in the ASAP WebLogic Server instance.
- *password* is the password for the web service user-defined in the ASAP WebLogic Server instance.
- *order_type* is the type of work order sent.
- *OSS/J_work_order* is the OSS/J work order information. When you add the work order information, do not include the XML header information (**<?xml version="1.0" encoding="UTF-8"?>**) because this has already been provided in the sample. Ensure that there are no namespace conflicts.

Guidelines for Creating Sample Work Orders

Always create sample work orders that test all of the service action and atomic actions in the cartridge. This includes work orders that test for sunny day as well as rainy day scenarios. Though it may not be possible to exercise all of the method logic (for example, all NE response combinations), mainstream paths (both common success and failure paths) should be invoked by the test suite.

Use consistent values for the parameters when creating sample work orders. This is useful when using the OCA GUI to query on orders by the field names and distinguish other orders from the sample cartridge orders.

```
ORG_UNIT MetaSolv
ORIGIN TEST
```

The value of the **SRQ_TYPE** variable should be appropriately set depending on the context of the work order (A – add, R – remove, C – change/update, Q - query).

```
SRQ_TYPE = A/R/C/Q
```

Troubleshooting Atomic Actions

Each atomic action-command has associated parameters that are sent to the NEP by the SARM when the ASAP is selected to provision.

> **Note:**
>
> An atomic action parameter value must be specified at the time of provisioning. If the parameter is not supplied at the time of provisioning, a SARM translation error results.

The following errors can occur during atomic-action-to-service-action translation:

- **No service action configuration in database** – The SARM receives a service action whose configuration is unknown and rejects the work order.
- **No service action To atomic action translation in database** – The service action is skipped because there is no work order to be done.
- **No atomic action configuration in database** – The SARM shuts down upon finding that an atomic action configuration is missing while loading the atomic action configurations associated with each service action. If all atomic action configurations are not defined, the work order being provisioned is failed.
- **Atomic action parameters not specified** – The SARM treats mandatory and optional parameters differently. If a mandatory atomic action parameter is not translated successfully with a name-value pair in the work order (empty string for the value is allowed), and no default value is available in the database, the work order is failed. The reason can be the label is missing in the work order. If the parameter is optional, order provisioning continues.

On startup, the SARM ensures that the following tables exist and have been populated:

- Service action configuration table
- Service-action-to-atomic-action translation table
- Atomic action configuration table
- Atomic action parameter table

If any of these tables is empty, the SARM shuts down.

Troubleshooting Service-Action-to-Atomic-Action Translation Errors

The following errors can occur during atomic action to service action translation:

- **No Service Action configuration in database** – The SARM receives a service action whose configuration is unknown and rejects the work order.
- **No Service Action To Atomic Action translation in database** – The service action is skipped because there is no work order to be done.
- **No Atomic Action configuration in database** – The SARM shuts down upon finding that an atomic action configuration is missing while loading the atomic action configurations associated with each service action. If all atomic action configurations are not defined, the work order being provisioned is failed.
- **Atomic Action parameters not specified** – The SARM treats mandatory and optional parameters differently. If a mandatory atomic action parameter is not translated successfully with a label-value pair in the work order (empty string for the value is allowed), and no default value is available in the database, the work order is failed. The reason can be the label is missing in the work order. If the parameter is optional, order provisioning continues.

16

Creating and Deploying a SAR File (ASAP Cartridge)

This chapter describes how to create and deploy a service activation archive (SAR) file (Oracle Communications ASAP cartridge).

SAR File Creation and Deployment Options

ASAP provides the following SAR file creation tools:

- Design Studio: Design Studio automatically generates a SAR file when you build an ASAP cartridge project without errors. Design Studio generates SAR files that support the ASAP 4.7 folder structure (see "ASAP 4.7 SAR File Folder Structure").
- **CreateSar**: The *ASAP_Home*/**programs/CreateSar** script generates ASAP 4.6 SAR files. This script generates SAR files that support the ASAP 4.6 folder structure (see "ASAP 4.6 SAR File Folder Structure") and is included for backward compatibility. For more information about the **CreateSar** script, see "Creating an ASAP 4.6 SAR File."

ASAP provides several SAR file deployment tools:

- The Service Activation Deployment Tool (SADT): This tool can be used to deploy the ASAP service model contained in the SAR file. For more information, see "Deploying Service Models with the Service Activation Deployment Tool."
- The Service Activation Configuration Tool (SACT): This tool is primarily used to configure ASAP servers using XML; however, SACT can also be used to deploy server-specific configuration changes within a SAR file contained in **SarPatch_configure** and **SarPatch_unconfigure** files. Design Studio cannot be used to generate these files: You must create them manually. For more information about this tool, see *ASAP Server Configuration Guide*.
- The ASAP_Home**/scripts/PostDeploySarFile**: This tool can be used to deploy SQL*Plus-specific configuration changes within a SAR file contained in **SarPatch** and **SarPatch_undeploy** files. In addition, this tool searches SAR file directories for SQL files with no **undeploy**_ prefix and work order TST files and commits them to the database. Design Studio cannot be used to generate these files: You must create them manually.
- The *ASAP_Home*/**samples/DIT/scripts/installCartridge** file: This sample script consolidates the SACT, SADT and **PostDeploySarFile** deployment options and should be customized for cartridges deployed in production environments.
- Design Studio: Design Studio provides the same functionality as SADT, but does not support **PostDeploySarFile** or SACT functionality. This tool should be used in development environments. For more information about the Design Studio cartridge deployment feature, see the Design Studio Help.

SAR File Folder Structure Options

ASAP supports the following SAR file folder structure options:

- ASAP 4.7 SAR File Folder Structure

ASAP 4.7 SAR File Folder Structure

Table 16-1 lists and describes the ASAP 4.7 SAR file folder structure used by Design Studio and supported for XML-based cartridges. You can enhance this directory structure with additional directories based on your requirements and deliverables.

Table 16-1 Design Studio ASAP 4.7 Folder Structure

Directory	Description
ActionProcessor	This folder contains action processor XML information used by Design Studio. This information is stored in the SAR file but is not used by ASAP.
doc	This folder contains Design Studio autogenerated cartridge documentation. For more information about autogenerated cartridge documentation, see "Documenting ASAP Cartridges ."
lib	This folder contains a **.jar** that provides the Java classes (autogenerated, or non-autogenerated) created to implement connections, or send network element (NE) commands as MML or API calls.
META-INF	This folder contains the following files: • **activation-model.xml**: For more information about this file, see "SA_archive.xsd." • **cartridge.xml**: This file defines the ASAP cartridge version, target platform, and the packaged deployment list.
NetworkElements	This folder contains NE XML information that ASAP uses to make NE connections.
ServiceModel	This folder contains service model XML information, such as atomic actions, service actions, connection handlers, user-defined exit types, and so on.
src	This folder contains the source files for the Java classes compiled in the **.jar** file.

ASAP 4.6 SAR File Folder Structure

When creating an ASAP 4.6 SAR file, you must use a fixed directory structure. This folder structure was developed in ASAP 4.6 and was replaced by the ASAP 4.7 folder structure (see "ASAP 4.7 SAR File Folder Structure"). ASAP supports the ASAP 4.6 folder structure for backward compatibility.

> **Note:**
>
> Design Studio uses the ASAP 4.7 SAR file folder structure when generating SAR files; however, you can import ASAP SAR files with the 4.6 folder structure into Design Studio.

This section describes the minimum required structure; you can enhance this directory structure with additional directories based on your requirements and deliverables.

```
META-INF/activation-model.xml
vendor/
     NE technology/
         service pack/
              sample_wo/
              sarm/
                  ne_progs/
                  PLSQL/
              control/
                  PLSQL/
              nep/
                  PLSQL/
              java/
                  lib/
              cpp/
                  lib/
              service_model/{at least one .xml file}
              application_config/ {optional}
     common/
              sarm/
                  ne_progs/
                  PLSQL/
              control/
                  PLSQL/
              nep/
                  PLSQL/
              java/
                  lib/
              cpp/
                  lib/
              service_model/ {optional}
              application_config/ {optional}
              scripts/ {optional}
vendor
…
```

The elements that uniquely identify an archive are a combination of the following:

- NE or EMS/NMS vendor name
- NE name and technology/software
- Service provided by the service model

The directory format of *vendor*/*NE technology*/*service pack*/ avoids collisions with other activation model directory structures.

- *vendor* directory – All ASAP service activation models developed for the same NE/EMS/NMS vendor reside in this directory.
- *vendor*/*NE Technology* – All ASAP service activation models for the same NE/EMS/NMS vendor and the same software load reside in this directory.
- *vendor*/*NE Technology*/*service pack* – The base directory for a specific service activation model. The following are examples of services: ADSL_ATM, SDSL_FR and Mail_Box.

An example of the directory format is Nortel/UEIMAS_5_2/ADSL_ATM.

Table 16-2 lists the directories supported for ASAP 4.6 SAR files.

Table 16-2 ASAP 4.6 SAR File Directory Structure

Directory	Description
sample_wo	Contains sample work order test files.
sarm	Contains files specific to the current activation model and targeted for service in the SARM database. • PLSQL – Contains files with sample ASAP configuration data specific to the SARM database.
control	Contains files specific to the current activation model and targeted for service in the CONTROL database. • PLSQL – Contains files with sample ASAP configuration data specific to the CONTROL database.
nep	Contains files specific to the current activation model and targeted for service in the NEP database. • PLSQL – Contains files with sample ASAP configuration data specific to the NEP database.
java	Contains all implementation files for the JInterpreter. • lib – Contains JAR files for JInterpreter provisioning implementations and third-party libraries.
service_model	Contains the XML documents that define the service models for this activation model. There must be at least one XML file in this directory. All documents in this service model directory must conform to the ServiceModel schema (refer to the following section).
application_config	Contains the XML documents that define configurations other than the service models for this activation model. There is no restriction on the number of XML files in this directory. All XML documents in this directory must conform to the activationConfig schema. The XML file in this directory can be an alternative to the SQL file in PLSQL directories above. This subdirectory is optional.
common/scripts	Contains the user patch script file **SarPatch** that is invoked by other utilities, such as **PostDeploySarFile** and **asapConfig**. These scripts customize the content within the SAR file, such as replacing fixed strings in the SQL or non-service-model XML files with relevant environment variables. The creator of the SAR file is responsible for providing this customization script file. The scripts and **SarPatch** are both optional. **SarPatch** is invoked after a SAR file has been deployed. It seeks any SQL file in PLSQL/ and WO test file in the sample_wo directory, and populates data into the relevant ASAP databases. Before populating, the utility tries to run **SarPatch** in the common/scripts directory, if it exists in the SAR file, to make generic SQL data specific to the current activation model. This utility is invoked as follows: PostDeploySarFile [-b] *sar_file_with_path* This directory also contains the **SarPatch_configure** and **SarPatch_unconfigure** scripts. These scripts are invoked by the script asapConfig to perform customizations or patching against the application configuration within the SAR file.

The common directory has a directory structure similar to *service pack*, with one more scripts subdirectory. The common directory contains all common files across different cartridges that share the same *vendor*/*NE technology*/ but offer different services. These files can include common definitions such as connect/disconnect classes, and so on. It also contains some supplementary files such as the SarPatch script.

The SARM, Java, NEP, Control and CPP directories under the common directory have a similar structure and meaning as the directories located under the <service_pack> directory.

Creating an ASAP 4.6 SAR File

You can archive the directory using the assembly tool. The assembly tool:

- Validates the directory structure
- Ensures that an **activation-model.xml** file and at least one service model XML file exist
- Validates the a**ctivation-model.xml** file and all service **model xml** files against their respective schemas. This validation is performed using the Oracle9_0_2_0_0D XML parsers.
- Picks up the activation model ID from the **activation-model.xml** file

You can also archive the directory using the **jar** command. Components other than service model components can be packaged in an archive file; for example, Design guidelines, API documentation, and source files.

After all validation is successfully completed, the assembly tool assembles the components to generate an archive file with a base file name that is the same as the ID, and supplies the **sar** extension. The assembly tool accepts the directory where the SAR file is to be placed as a parameter.

To run the assembly tool, enter the following command:

```
CreateSar [-help] [-v] <sar_file_dir>
```

The current directory must be the base from which the **SAR** file is made and it is the parent directory of the **<vendor>** directory. The **<sar_file_dir>** specifies where you want to put the **SAR** file you have created. The -v parameter enables directory structure validation. If you omit this parameter, no directory structure validation is performed. The SAR file name is generated based on the name in **activation-model.xml**.

Deploying Service Models with the Service Activation Deployment Tool

The SADT deploys the SAR file. The SADT has three interfaces:

- Using the SADT Command Line Interface
- Using the SADT Web Interface
- Using the SADT JMX Interface

You can use the SADT to assemble and deploy generic service models and cartridge-specific service models.

Using the SADT Command Line Interface

You can invoke the SADT using a command-line interface. The command-line interface must be invoked by passing in the WebLogic Server URL, a user name and a password.

The command-line interface supports two modes of invocation:

- Interactive – You can select options and enter data
- Script-based – You can start the SADT from within scripts by passing in all parameters on the command-line.

> **Note:**
>
> A customizable script **sadtclient** is available in *ASAP_Home***/scripts**. This script enables you to pre-populate the information required in Step 2 for up to four working environments. The first time you use the script, you will be prompted to modify the script.

You can choose from the following actions:

- List all deployed Service Activation Archive model
- Deploy a Service Activation Archive model
- Undeploy a Service Activation Archive model
- Query a Service Activation Archive model

Using the SADT Command Line Interface in Interactive Mode

To access the command-line utility in interactive mode, do the following.

1. From within a UNIX or Linux script, enter the following:

```
java -classpath $CLASSPATH com.mslv.activation.management.application.sadtClient
```

> **Note:**
>
> If sadtClient.jar, asaplibcommon.jar or weblogic.jar are not in $CLASSPATH, add them.

2. A login screen appears. Type the information that appears in italics.

```
Welcome to Service Activation Deployment Tool
Please enter WebLogic login information
WebLogic host:port -> myhost:1234
Username -> username
Password -> password
JNDI Context -> (long JNDI string)
Replace ('t' for true, else false) -> (t or others)
Connecting to WebLogic server...
```

An example of a long JNDI string is System/S123/ApplicationType/ServiceActivation/Application/1-0;4-7;ASAP/Comp/

When you have defined your JNDI prefix, replace "S123" above with the appropriate $ENV_ID value in your ASAP environment, for example, "TST1".

1. After you have logged in, the following menu appears. Select the option you require:

```
***** Service Activation Deployment Tool *****
1. Deploy an activation model
2. Undeploy an activation model
3. Query an activation model
4. List all deployed activation models
5. Export existing service model
Enter Choice, <Q - Quit): 1
```

Option 5 enables you to save the activation model to a SAR file.

Deploying a Service Activation Model Archive

This menu option prompts you to do one of the following:

- Type the absolute file path for a Service Activation Model Archive to be installed on the ASAP instance
- Specify the SAR ID if the SAR already exists in ASAP but has not yet been deployed.

The status of deployment is displayed on screen and the menu option does not return until the SAR is successfully or unsuccessfully deployed.

```
***** 1. Deploy an activation model *****
Enter the file path or ID of the SAR you want to deploy
-> /sunenv123/samples/sadt/sar/Nortel_HLR_GEM14_MSP.sar
Deploying model...
Activation model </sunenv123/samples/sadt/sar/Nortel_HLR_GEM14_MSP.sar> has been deployed
Press ENTER to continue ...
```

Undeploying a Service Activation Model Archive

This menu option prompts you to type the ID of an activation model to uninstall from an ASAP instance. If you do not know the SAR ID, refer to "List All Deployed Activation Models."

```
*****2. Undeploy an activation model *****
Enter the ID of the model you want to undeploy
-> Nortel_DMS_POTS
Activation model Nortel_DMS_POTS has been undeployed
```

Querying an Activation Model

This menu option prompts you to type the ID of an activation model to query. It only queries models that are deployed in ASAP. If the model is undeployed, it returns with a message stating the requested model is not deployed.

If you select **Query an Activation Model**, the following appears:

```
***** 3. Query an Activation Model *****
Enter the ID of the activation model you want to query
-> Nortel_HLR_GEM14_MSP
Querying activation model <Nortel_HLR_GEM14_MSP> ...
id: Nortel_HLR_GEM14_MSP
deployed: Yes
description: Nortel 3G Wireless GEM14 Cartridge
vendor: Nortel
technology: HLR
softwareLoad: GEM14
version:
author: Nortel Networks
label: 1.2
majorVersion: 1
minorVersion: 2
createDate: Sun Aug 13 00:00:00 GMT-05:00 2000
validDuration: P1Y2M3DT10H30M
```

All information in the deployment descriptor is returned except for the icon and all component elements.

List All Deployed Activation Models

This action lists IDs for all deployed models.

```
***** 4. List all deployed activation models *****
IDs of all deployed activation models are:
Nortel_HLR_GEM14_MSP
Nortel_PASSPORT_3_0_ATM_FR
```

Using the SADT Command Line Interface in Script Mode

You can access SADT using scripting through the command line interface. All functions in the command line interface are accessible from a single invocation of the tool. All parameters to a function can be passed on the command line.

Examples of invoking the command line tool for script-based usage:

- Deploying a service activation model

  This function call deploys the activation model specified by the file name:

  ```
  java sadtClient -url myhost:1234 -username system -password admin - jndiContext
  "System/S123/ApplicationType/ServiceActivation/Application/1-0;4- 6;ASAP/Comp/"
  deploy /sunenv123/samples/sadt/sar/Nortel_HLR_GEM14_MSP.sar
  ```

  This function call returns the same status as AMDT-450-1.2:

  ```
  Activation model </sunenv123/samples/sadt/sar/Nortel_HLR_GEM14_MSP.sar> has been
  deployed
  ```

- Undeploying an activation model

  ```
  java sadtClient -url myhost:1234 -username system -password admin -jndiContext
  System/S123/ApplicationType/ServiceActivation/Application/1-0;4-7;ASAP/Comp/ -
  replace false undeploy Nortel_HLR_GEM14_MSP
  ```

  This function call returns the same status as AMDT-450-1.3:

  ```
  Activation model <Nortel_HLR_GEM14_MSP> has been undeployed
  ```

- Querying for an activation model

  ```
  java sadtClient -url myhost:1234 -username system -password admin -jndiContext
  System/S123/ApplicationType/ServiceActivation/Application/1-0;4-7;ASAP/Comp/ -
  replace false query Nortel_HLR_GEM14_MSP
  ```

This function call returns the information about the model, same as ADMT-450-1.4.

```
id: Nortel_HLR_GEM14_MSP
deployed: Yes
description: Nortel 3G Wireless GEM14 Cartridge
vendor: Nortel
technology: HLR
softwareLoad: GEM14
version:
author: Nortel Networks
label: 1.2
majorVersion: 1
minorVersion: 2
createDate: Sun Aug 13 00:00:00 EDT 2000
validDuration: P1Y2M3DT10H30M
```

- List all deployed activation models, by doing one of the following:

- Enter **java sadtClient** and follow the prompts
- Enter the following:

```
java sadtClient -url myhost:1234 -username system -password admin -jndiContext
System/S123/ApplicationType/ServiceActivation/Application/1-0;4-7;ASAP/Comp/  -
replace false list
```

This function call returns a list of all model IDs that are deployed, same as ADMT-450-1.1.

```
IDs of deployed activation models are:
Nortel_DMS_POTS       Deployed
Nortel_HLR_GEM14_MSP  Undeployed
```

The following is a sample script to invoke the commands:

```
#!/bin/ksh
if [ -f /sunenv123/samples/sadt/sar/Nortel_HLR_GEM14_MSP.sar ]; then
java sadtClient -url myhost:1234 -username system -password admin -jndiContext System/
S123/ApplicationType/ServiceActivation/Application/1-0;4-7;ASAP/Comp/ -replce true
deploy /sunenv123/samples/sadt/sar/Nortel_HLR_GEM14_MSP.sar
fi
```

> **Note:**
>
> In an actual script line, all the semicolons above must be preceded by an escape character "\".

To simplify the use of the command scripts, ASAP includes scripts that prompt you to personalize them when invoked. Edit this script and follow the instructions placed at the beginning of the script to change required strings. After the script is personalized, you will not have to type the host name, port number, long JNDI string, user name, and password, which seldom need to be changed in a working environment.

For more information on personalized scripts, see *ASAP Server Configuration Guide*.

Using the SADT Web Interface

The web-based SADT GUI is a standalone client application. Using the SADT GUI, you can:

- Viewing Deployed Service Activation Models

  Deploying a service activation archive file
- Undeploying a Service Activation Model
- Deploying Multiple Cartridges

Viewing Deployed Service Activation Models

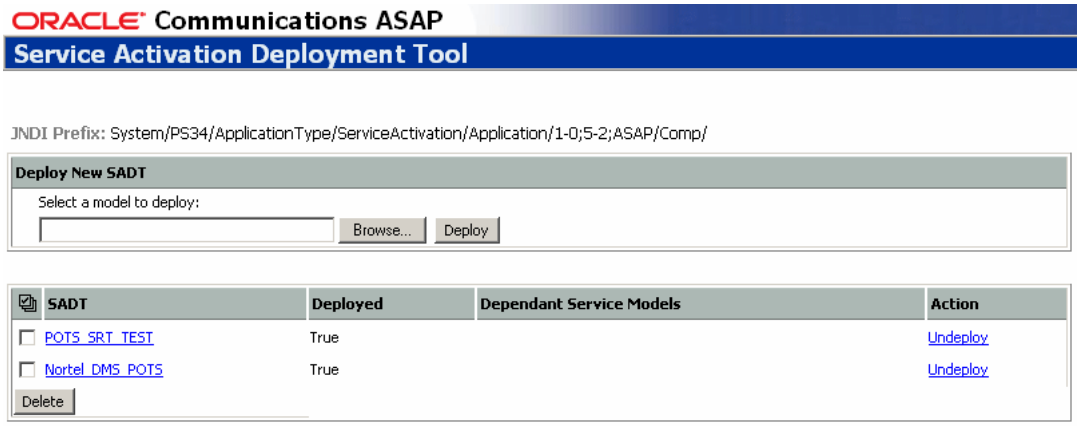
To view deployed service activation models:

1. In the Address field of your web browser, type the login URL (for example, http://<BEA_HOST>:<BEA_PORT>/<ENV_ID>/sadtConsole), and press **Enter**.

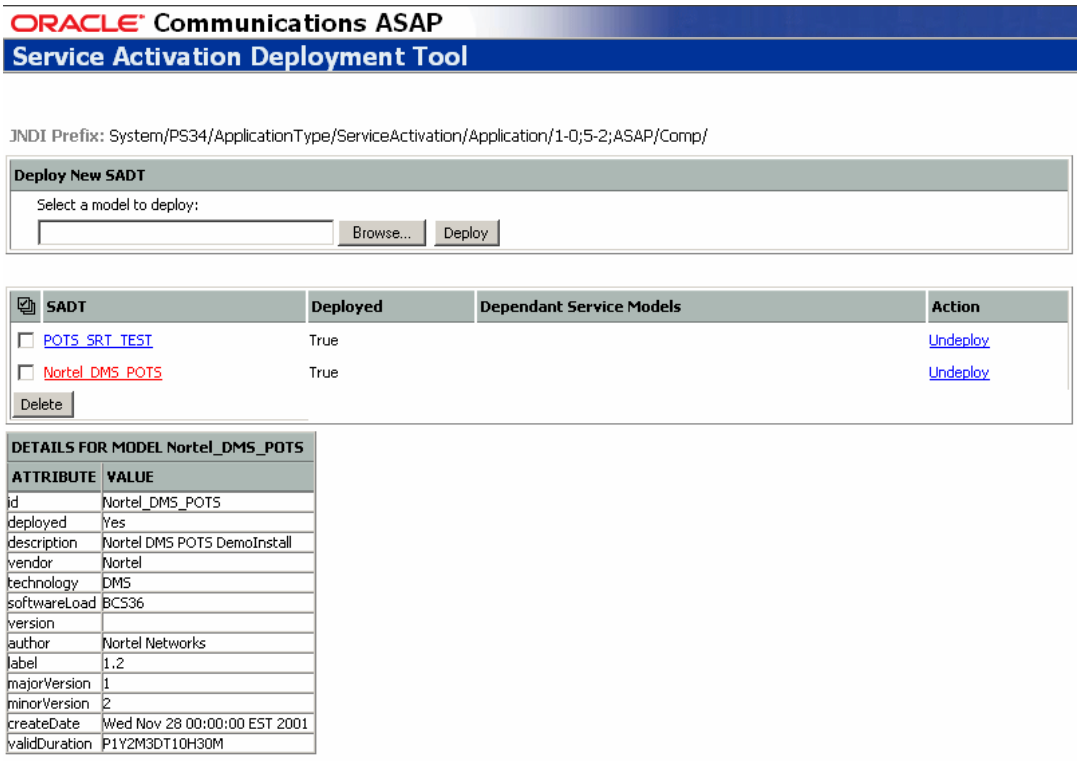
   The Enter Network Password dialog box appears.

> **Note:**
>
> <ENV_ID> represents the environment ID chosen in the installer. See the table of installation values in the *ASAP Installation Guide*.

2. In the **User Name** field, enter your user name.
3. In the **Password** field, enter your password, and then click **OK**.
4. Click **OK**. The Service Activation Deployment Tool view appears.



5. View the details of a service activation model, by clicking the appropriate service activation label. The service activation model details are displayed in the lower part of the window.



Deploying a service activation archive file

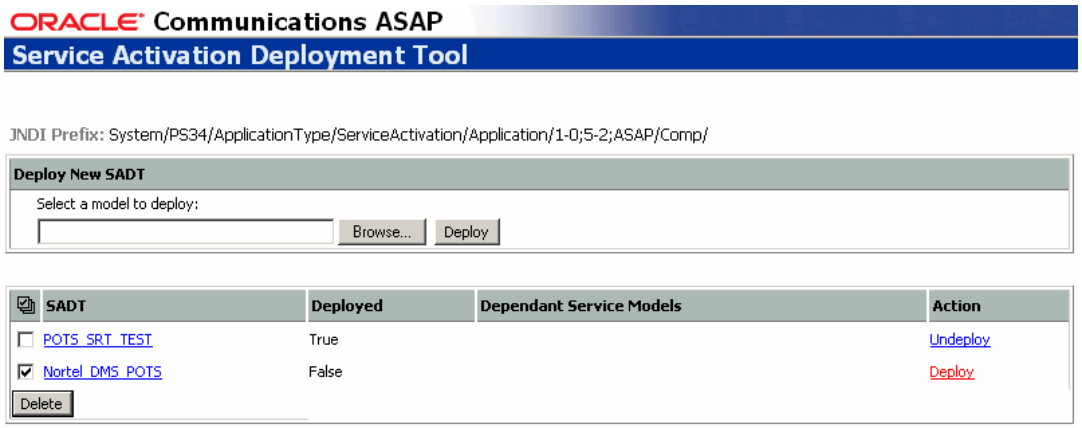
To deploy a service activation archive file, select a service activation archive file and then deploy the selected SAR file.

When deploying service models, you must pay special attention to dependencies (displayed in the **Dependent Service Models** column of the SADT console). If service model B is dependent on service model A, service model A must be deployed before service model B.

To deploy a service activation archive file:

1. In the Service Activation Deployment Tool view, do one of the following:
   - If the service activation archive file already appears in the list, click **Deploy**.

     The Deploy Service Activation Models view appears.
   - If the service activation archive file that you want to deploy does not appear in the list, click **Browse** and navigate to the location of the SAR files. The SAR file name appears in the **Select a model to deploy** field.



2. Depending on your action above, click Deploy either in the list or in the **Deploy New SADT** section. All successfully deployed service activation models are flagged with **True** in the **Deployed** column.

> **Note:**
>
> You can redeploy a service activation model that has already been deployed. If you change the contents of the deployment descriptor (**activation-model.xml**) of a SAR file, the modified SAR file is considered as a different version of the service activation model. In this case, you must undeploy the existing service activation model and then deploy the modified SAR file.

Undeploying a Service Activation Model

You can undeploy one or more service activation models from an ASAP instance.

If you are undeploying a service model, you must undeploy the dependent service model before undeploying its parent. The SADT console displays an error message if you attempt to undeploy a service model that has dependent service models.

To undeploy a service activation model:

1. In the Service Activation Deployment Tool view, click **Undeploy**.

   The screen refreshes and the **Deployed** column displays **False**.

Deploying Multiple Cartridges

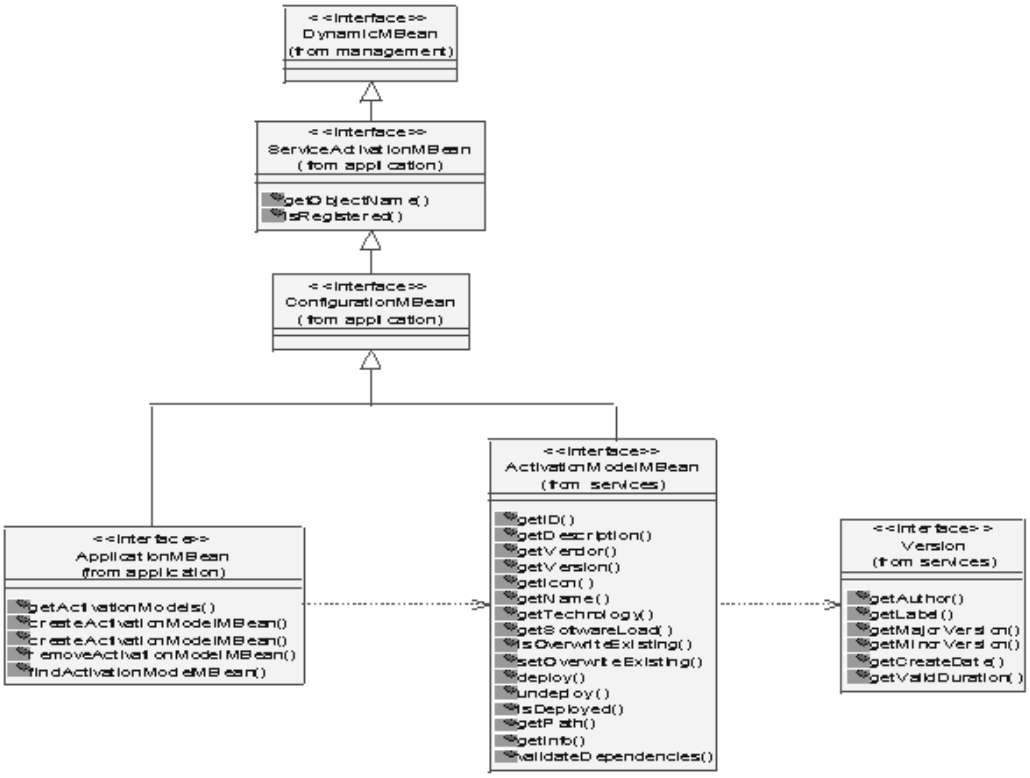
You can deploy (and undeploy) SAR cartridge files without conflicts, even if multiple cartridges with the same content are deployed. When a SAR file is deployed through either the command line, GUI, or JMX-based interfaces, a target directory is created using the cartridge ID as a component in the directory name. The Java provisioning files are placed into this unique target directory where they will not be overwritten by future deployments.

After the Java provisioning files are placed in the target directory, the CLASSPATH is modified to contain references to the newly added Java JAR files.

Using the SADT JMX Interface

With the Java Management Extension based interface, you can access all deployment functionality programmatically.

Figure 16-1 JMX-based Interface



The ServiceActivationMBean is the base interface for all MBeans in the system. Every MBean is registered into the WebLogic Server (as MBeanServer) with an object name. ApplicationMBean represents an ASAP instance. Currently, its interface only defines some simple management functionality to retrieve and create new ActivationModelMBeans. An

ActivationModelMBean represents an activation model archive. To create a new ActivationModelMBean, invoke the createActivationModelMBean method on an ApplicationMBean. After an ActivationModelMBean has been created, the archive can be deployed or undeployed.

Configuring JMX Interfaces to Validate XML Documents

You can configure JMX interfaces to validate all XML documents against their respective schemas within the WebLogic server. To enable validation, you must change the **VALIDATE** option to **True** in the **web.xml** deployment descriptor, through the WebLogic Administration Console.

To enable the Validate option:

1. Navigate to *ASAP domain name* > **Deployments** > **Web Applications**, right-click sadtConsole and choose **Edit Web Application Deployment Descriptor**. A new browser window opens in which you can edit the deployment descriptors.
2. Navigate to **Web Descriptor** > **Web App Descriptor**. Right-click **Env Entries** and choose **Configure a new EnvEntry**.
3. In the screen that appears, enter the following:
   - **Description** – An optional description
   - **Env Entry Name** – VALIDATE
   - **Env Entry Value** – true
   - **Env Entry Type** – java.lang.String
4. Click **Create**.

The EnvEntry portion of **web.xml** appears as follows:

```
<env-entry>
    <env-entry-name>ASAP_BASE</env-entry-name>
                <env-entry-value>ASAPDEV_ASAP_BASE</env-entry-value>
    <env-entry-type>java.lang.String</env-entry-type>
  </env-entry>

  <env-entry>
    <env-entry-name>SYBASE</env-entry-name>
                <env-entry-value>ASAPDEV_SYBASE</env-entry-value>
    <env-entry-type>java.lang.String</env-entry-type>
  </env-entry>

<env-entry>
    <env-entry-name>VALIDATE</env-entry-name>
                <env-entry-value>false</env-entry-value>
    <env-entry-type>java.lang.String</env-entry-type>
  </env-entry>

<env-entry>
        <env-entry-name>SA-jndi-context</env-entry-name>
        <env-entry-value>System/SAAS-1/ApplicationType/ServiceActivation/Application/
1-0;4-7
;ASAP/Comp/</env-entry-value>
        <env-entry-type>java.lang.String</env-entry-type>
  </env-entry>
```

Loading ASAP Services Dynamically

When an ASAP server such as a SARM or NEP is initialized, configuration data groups are loaded from the database into memory. These configuration data groups are:

- **ASAP service definitions** – Includes all service-action and atomic-action related configurations.
- **ASAP network interface configurations** – Includes hosts, NE resources.

In some cases, configuration data can be loaded into memory after initialization (namely, NE communication parameters). For all other new service definitions or network interface configurations to take effect, you must do one of the following:

- Restart the SARM and NEP
- Dynamically add service definitions using **asap_utils**

For more information on **asap_utils**, see *ASAP Server Configuration Guide*

With **asap_utils**, you can dynamically add new items to the configuration without having to restart the ASAP servers. You can dynamically add service actions, atomic actions, mapping configuration, NEPs, hosts, routing, and NE resources by:

- Updating the ASAP databases through **asap_utils**
- Synchronizing the ASAP servers with the databases

> **Note:**
>
> Only new configuration entries are added to the in-memory caches. Any updates and deletions of existing entries are ignored by the update procedure. These still require a restart of the affected ASAP servers.

To dynamically configure service actions, atomic actions, mapping configuration, or NEPs, you must set up the ASAP system and its components as follows.

1. Make the configuration addition in the ASAP database manually by scripting (SQL inserts). You can add the configuration entries in Table 14-3 dynamically to ASAP. All other changes are ignored by the sychronization process.

Table 16-3 Configuration Entries and Synchronization

Configuration Entry	Description	Synchronization Details
Service Definition	Service actions, atomic actions, service-action-TO-atomic mappings and service action-to-atomic action parameters.	Internal cache update with new entries in SARM and NEP servers.
Network Element Definition	NE definitions and associated NE connection properties, including primary and secondary devices, and host-to-remote NE mappings.	Internal cache and thread updates in SARM (atomic action queues) and NEP (session managers, command processors) application servers.
NEP	Entirely new NEP added to ASAP.	Internal cache and thread updates in SARM (NEP drivers, atomic action queues).

Table 16-3 (Cont.) Configuration Entries and Synchronization

Configuration Entry	Description	Synchronization Details
Secondary NE Devices	Communication devices in the auxiliary resource pool in existing NEPs.	Internal cache and thread updates in NEP (command processors).

> **Note:**
>
> A service definition consists of a service action and all its associated atomic actions and parameters. Adding a new atomic action to an existing service action or adding a new service action to an existing atomic action is considered a change to the service definition. The SARM server must be restarted before the new service definition takes effect.

2. You can request the SARM and NEP servers to refresh the configuration in memory through the command options available in **asap_utils**. There are two commands in the **asap_utils** client application for the addition of new configuration entries. You use these commands to request that the ASAP servers synchronize the in-memory configuration with the configuration of the databases. You are responsible for ensuring that the database configurations have been applied (for example, through database insert scripts).

   By sending RPCs to notify ASAP servers of the additional configuration entries, a synchronization process is performed in each of the ASAP servers to update the in-memory caches with the latest data of the configurations in the database. The synchronization reloads the relevant configuration information from the database into the memory. With the synchronization, you can add new service actions, atomic actions, mapping configurations, NEPs, host NEs, NE routings, and NE resources dynamically.

   > **Note:**
   >
   > You can only dynamically add new items, you cannot modify or delete new configurations.

   In **asap_utils**, the command choices for the newly added configuration are:

   - 112. Load New Service Configuration into Cache
   - 113. Load New NE Configuration into Cache

   You are asked to provide the SARM and NEP names to which the updates apply. The defaults are the **$SARM** and **$NEP** environment variables. If multiple NEPs are to be updated, you can enter a list of NEP server names and the utility issues the appropriate request to each listed NEP. You are notified whether each RPC was successful or not. In all cases, the NEP update requests are issued.

3. After receiving the synchronization RPC, the ASAP server works with a specific handler to reload the relevant configuration data from the ASAP database into memory. The RPC returns a status to the client application, indicating whether reloading was successful or not.

4. The synchronization sequence is the reverse of the normal processing flow; in other words, the NEP is synchronized first, then the SARM.

You must ensure that the time interval between the database modification and the memory flush is as short as possible.

> **Note:**
>
> Only SARM and NEP servers support the synchronization.

A

Configuring Services Using XML

This appendix describes how to configure Oracle Communications ASAP services using XML schemas and deploy these services using the Service Activation Deployment Tool (SADT).

This service configuration method supersedes the use of stored procedures to configure services.

> **Note:**
>
> Schema validation for XML data processed by the Service Activation Configuration Tool (SACT) and the SADT is turned off by default. If you turn on schema validation and use these tools to deploy ASAP configuration data and service models, and you upgrade ASAP to version 7.2 or later, errors may be reported where previously none were reported.

Configuration Restrictions and Limitations

You can add, update, or delete an entity within the provisioning translation configuration. Before doing so, however, Oracle recommends that you review the structure of the existing configuration to ensure that real-time translation is carried out accurately and successfully.

Specifically, consider the following when you configure provisioning translation:

- Constraint conditions, such as switch technology and software load (the software version of the specific switch).
- Prerequisite information, such as translation mappings from service actions to atomic actions, that cannot be defined unless the atomic action is already defined.

You can add configurations at run time without restarting the SARM or network element processor. For more information, see "Loading ASAP Services Dynamically."

Configuring ASAP Services

This section describes the steps for configuring ASAP services.

The configuration of ASAP services using XML consists of the following steps:

- Planning
- Configuring Atomic Actions
- Adding Supporting Data
- Configuring Service Actions
- Mapping Atomic Actions to Service Actions
- Mapping User Exit Types to Base Exit Types
- Creating Activation-Model.xml

- Configuring Network Element Throughput Using XML

Planning

Based on the services and network elements to be supported, determine the NE-specific commands used for identified service (API calls, MML commands).

Create the service model components and scripts to support the service requirements. Specifically:

- Create a Java method for each NE-specific API call or MML command.
- Identify an atomic action for each script (Java method).
- Identify the atomic action parameters and values for atomic action.
- Identify the mapping between each atomic action.
- Identify service action commands required for each supported service.
- Identify service-action-to-atomic-action mappings.
- Plan enhanced service-action-atomic-action translation.

Configuring Atomic Actions

A typical atomic action XML definition appears as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:sam="http://
www.metasolv.com/ServiceActivation/2003/ServiceModel" xmlns="http://www.metasolv.com/
ServiceActivation/2003/ServiceModel" xmlns:fo="http://www.w3.org/1999/XSL/Format">
<description>Nortel NT-DMS10 atomic action Services</description>
<atomicService name="A_NT-DMS10_503-10_ADD_POTS-LINE" xsi:type="AtomicServiceType">
<description>Adds a POTS line.</description>
<timeout>20</timeout>
<retryCount>3</retryCount>
<retryInterval>6</retryInterval>
<rollbackService enable="true">
<rollbackService>A_NT-DMS10_503-10_ADD_POTS-LINE-RB</rollbackService>
</rollbackService>
<sendParameterCount>false</sendParameterCount>
<parameter name="MCLI" xsi:type="SimpleParameterType">
<description>Host NE identifier.</description>
<required>true</required>
<default/>
<parameterValueMap>NE_ID_NT-DMS10</parameterValueMap>
</parameter>
<parameter name="NPA" xsi:type="SimpleParameterType">
<description>3 digit area code.</description>
<required>false</required>
<default/>
<parameterValueMap>NPA</parameterValueMap>
</parameter>
<parameter name="NXX" xsi:type="SimpleParameterType">
<description>First 3 digits of the line number.</description>
<required>true</required>
<default/>
<parameterValueMap>NXX</parameterValueMap>
</parameter>
<parameter name="LINE" xsi:type="SimpleParameterType">
<description>4 digit line extension.</description>
<required>true</required>
```

```
<default/>
<parameterValueMap>LINE</parameterValueMap>
</parameter>
<parameter name="LEN" xsi:type="SimpleParameterType">
<description>Line equipment number.</description>
<required>true</required>
<default/>
<parameterValueMap>LEN</parameterValueMap>
</parameter>
<parameter name="LOC" xsi:type="SimpleParameterType">
<description>Equipment location.</description>
<required>false</required>
<default/>
<parameterValueMap>LOC</parameterValueMap>
</parameter>
<parameter name="LCC" xsi:type="SimpleParameterType">
<description>The line or agent class code. 1FR, 1MR, 2FR, 8FR, 10FR, 1MB, 2MR, 4MR.</
description>
<required>true</required>
<default/>
<parameterValueMap>LCC</parameterValueMap>
</parameter>
<parameter name="RINGCODE" xsi:type="SimpleParameterType">
<description>The ringing code input for two-party or multiparty services.</description>
<required>false</required>
<default/>
<parameterValueMap>RINGCODE</parameterValueMap>
</parameter>
<parameter name="ZONE" xsi:type="SimpleParameterType">
<description>The OUTWATS zone identification number.</description>
<required>false</required>
<default/>
<parameterValueMap>ZONE</parameterValueMap>
</parameter>
<parameter name="LTG" xsi:type="SimpleParameterType">
<description>Line treatment group.</description>
<required>false</required>
<default/>
<parameterValueMap>LTG</parameterValueMap>
</parameter>
<parameter name="OPT" xsi:type="CompoundParameterType">
<description>Options associated with a service to be established or deleted. A maximum
of 20 options can be specified in any single command.</description>
<required>false</required>
<default/>
<parameterValueMap>OPT</parameterValueMap>
</parameter>
<parameter name="IS_OPTIMIZED" xsi:type="SimpleParameterType">
<description>A flag to indicate MML optimization of the command. The values are false
and true.</description>
<required>false</required>
<default>true</default>
<parameterValueMap>IS_OPTIMIZED</parameterValueMap>
</parameter>
<parameter name="ACT_CFB" xsi:type="SimpleParameterType">
<description>Conditional flag to activate the CFB feature.</description>
<required>false</required>
<default/>
<parameterValueMap>ACT_CFB</parameterValueMap>
</parameter>
<parameter name="ACT_CFD" xsi:type="SimpleParameterType">
<description>Conditional flag to activate the CFD feature.</description>
```

```
<required>false</required>
<default/>
<parameterValueMap>ACT_CFD</parameterValueMap>
</parameter>
<parameter name="ACT_CFDA" xsi:type="SimpleParameterType">
<description>Conditional flag to activate the CFD feature.</description>
<required>false</required>
<default/>
<parameterValueMap>ACT_CFDA</parameterValueMap>
</parameter>
<parameter name="CUSTOM" xsi:type="CompoundParameterType">
<description>Customer-specific parameter.</description>
<required>false</required>
<default/>
<parameterValueMap>CUSTOM</parameterValueMap>
</parameter>
</atomicService>
<atomicService name="A_NT-DMS10_503-10_DEL_POTS-LINE" xsi:type="AtomicServiceType">
<description>Deletes a POTS line.</description>
<sendParameterCount>false</sendParameterCount>
<parameter name="MCLI" xsi:type="SimpleParameterType">
<description>Host NE identifier.</description>
<required>true</required>
<default/>
<parameterValueMap>NE_ID_NT-DMS10</parameterValueMap>
</parameter>
<parameter name="NPA" xsi:type="SimpleParameterType">
<description>3 digit area code.</description>
<required>false</required>
<default/>
<parameterValueMap>NPA</parameterValueMap>
</parameter>
<parameter name="NXX" xsi:type="SimpleParameterType">
<description>First 3 digits of the line number.</description>
<required>true</required>
<default/>
<parameterValueMap>NXX</parameterValueMap>
</parameter>
<parameter name="LINE" xsi:type="SimpleParameterType">
<description>4 digit line extension.</description>
<required>true</required>
<default/>
<parameterValueMap>LINE</parameterValueMap>
</parameter>
<parameter name="CUSTOM" xsi:type="CompoundParameterType">
<description>Customer-specific parameter.</description>
<required>false</required>
<default/>
<parameterValueMap>CUSTOM</parameterValueMap>
</parameter>
</atomicService>
```

Adding Supporting Data

For atomic actions that map to the JInterpreter device type, the supporting data will consist of Java classes. Java classes are placed in the **..\java\lib** directory within a .jar file. The deviceMap section of the atomic action definition appears as follows:

```
<atomicDeviceMap name="A_NT-DMS10_503-10_ADD_POTS-LINE">
<deviceMap>
<description>(user_msg_only)</description>
```

```
<type>NT-DMS10</type>
<version>503-10</version>
<implementation>com.metasolv.cartridge.oss.nt_dms10_503_10.
  DMS10PotsLineProv.addNewLine</implementation>
<interpreter>JINTERPRETER_PROGRAM</interpreter>
</deviceMap>
</atomicDeviceMap>
<atomicDeviceMap name="A_NT-DMS10_503-10_DEL_POTS-LINE">
<deviceMap>
<description>(user_msg_only)</description>
<type>NT-DMS10</type>
<version>503-10</version>
<implementation>com.metasolv.cartridge.oss.nt_dms10_503_10.
  DMS10PotsLineProv.delPotsLine</implementation>
<interpreter>JINTERPRETER_PROGRAM</interpreter>
</deviceMap>
</atomicDeviceMap>
```

Configuring Service Actions

A service action command (referred to as a CommonServiceType in the XML schema) is an ASAP command that is associated with a particular work order. The service action command is associated with one or more operations on one or more NEs.

Each service action command within the SARM has a configuration record that you can set up. This record contains the following attributes:

A service action definition appears as follows:

```
<commonService name="C_NT-DMS10_503-10_ADD_POTS-LINE">
<description>Adds a POTS line.</description>
<rollbackOnFailure>false</rollbackOnFailure>
<priority>30</priority>
<failEvent>
<customEvent/>
</failEvent>
<completeEvent>
<customEvent/>
</completeEvent>
```

Mapping Atomic Actions to Service Actions

Following the service action configuration parameters, add one or more atomic actions within **<serviceMap>** element. Atomic-action-to-service-action mappings consist of the **<atomicService>** identifier and one or more optional conditions.

- **atomicService** – The atomic action identified in the atomicService name.
- **pointOfNoReturn** – The 'point of no return' value for partial rollbacks. Values are:
  - 0 (default) – This atomic action is not the 'point of no return' for rollback purposes
  - 1 – This atomic action is the 'point of no return' for partial rollback. If rollback occurs, and execution has continued beyond this point, roll back to this atomic action but no further.
  - 2 – 'point of no return' for no rollback. After past this atomic action, no rollback can occur.

  For more information, see "About Configuring a Rollback Point (Point of No Return)."
- **description** – You can optionally provide a description of the atomic action.

- **condition** – Can be one of four types "A" (AlwaysConditionType),"D" (DefinedConditionType),"N" (NotDefinedConditionType) and "E" (EqualConditionType).
  - If the condition is A, the SARM always generates the network action for this service action. For example:

```
...
<condition xsi:type="AlwaysConditionType">
<expression>true</expression>
</condition>
```

or

```
<condition xsi:type="AlwaysConditionType"/>
```

- If the condition is D, the SARM only generates a particular atomic action if the stated service action parameter is defined on the current service action. For example:

```
...
<condition xsi:type="DefinedConditionType">
<expression/>
<parameterLabel>CCC</parameterLabel>
</condition>
```

- If the condition is N, the SARM only generates a particular atomic action if the stated service action parameter is not defined on the current service action. For example:

```
...
<condition xsi:type="NotDefinedConditionType">
<expression/>
<parameterLabel>DDD</parameterLabel>
</condition>
```

- If the condition is E, the SARM only generates a particular atomic action if the stated service action parameter is defined on the current service action and has a particular parameter value. For example:

```
...
<condition xsi:type="EqualConditionType">
<expression>ABC LIKE "BCS%"</expression>
<parameterLabel>AAA</parameterLabel>
<parameterValue>12345</parameterValue>
</condition>
...
```

A complete service action definition, with the atomic action mappings highlighted in bold, appears as follows:

```
<commonService name="C_NT-DMS10_503-10_ADD_POTS-LINE">
<description>Adds a POTS line.</description>
<rollbackOnFailure>false</rollbackOnFailure>
<priority>30</priority>
<failEvent>
<customEvent/>
</failEvent>
<completeEvent>
<customEvent/>
</completeEvent>
<serviceMap>
<atomicService>A_NT-DMS10_503-10_ADD_POTS-LINE</atomicService>
</serviceMap>
<serviceMap>
<atomicService>A_NT-DMS10_503-10_ACT_CFB-OPT</atomicService>
<pointOfNoReturn>1</pointOfNoReturn>
```

```
<condition xsi:type="AlwaysConditionType">
<expression>(ACT_CFB LIKE "Y%")</expression>
</condition>
</serviceMap>
<serviceMap>
<atomicService>A_NT-DMS10_503-10_ACT_CFD-OPT</atomicService>
<condition xsi:type="AlwaysConditionType">
<expression>(ACT_CFD LIKE "Y%")</expression>
</condition>
</serviceMap>
<serviceMap>
<atomicService>A_NT-DMS10_503-10_ACT_CFDA-OPT</atomicService>
<condition xsi:type="AlwaysConditionType">
<expression>(ACT_CFDA LIKE "Y%")</expression>
</condition>
</serviceMap>
</commonService>
```

Mapping User Exit Types to Base Exit Types

The **ServiceModel.xsd** XML schema file contains the following definitions:

```
<xsd:simpleType name="RegexPattern">
<xsd:annotation>
<xsd:documentation>Simple data type for representing regular
expression search pattern</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:string">
<xsd:maxLength value="255"/>
<xsd:minLength value="1"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="BaseType">
<xsd:annotation>
<xsd:documentation>Simple data type for representing base
atomic action exit types</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:string">[1]
<xsd:enumeration value="SUCCEED"/>
<xsd:enumeration value="FAIL"/>
<xsd:enumeration value="RETRY"/>
<xsd:enumeration value="RETRY_DIS"/>
<xsd:enumeration value="MAINTENANCE"/>
<xsd:enumeration value="SOFT_FAIL"/>
<xsd:enumeration value="DELAYED_FAIL"/>
<xsd:enumeration value="STOP"/>
</xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="UserType">
<xsd:annotation>
<xsd:documentation>Simple data type for representing user
defined atomic action exit types</xsd:documentation>
</xsd:annotation>
<xsd:restriction base="xsd:string">
<xsd:minLength value="1"/>
<xsd:maxLength value="20"/>
</xsd:restriction>
```

[1] The base types supported by this service model

```
</xsd:simpleType>

<xsd:complexType name="NEDescriptor">
<xsd:annotation>
<xsd:documentation>
Identifier used for representing network element software load
and technology software load. Put in place so that
this information is represented as one logical unit of data.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="softwareLoad" type="sam:SoftwareLoadType"
minOccurs="0"/>
<xsd:element name="technology" type="sam:TechnologyType"/>
<xsd:element name="neVendor" type="sam:VendorType"/>
</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="UserDefinedExitType">
<xsd:annotation>
<xsd:documentation>
A logical representation of a user defined exit type to base type
mapping.
</xsd:documentation>
</xsd:annotation>
<xsd:sequence>
<xsd:element name="CSDL" type="sam:CommandType" minOccurs="0"/>
<xsd:element name="ASDL" type="sam:CommandType" minOccurs="0"/>
<xsd:element name="neDescriptor" type="sam:NEDescriptor"
minOccurs="0"/>
<xsd:element name="searchPattern" type="sam:RegexPattern" [2]minOccurs="0"/>
<xsd:element name="userType" type="sam:UserType"/>
<xsd:element name="baseType" type="sam:BaseType"/>
<xsd:element name="description" type="sam:DescriptionType"
minOccurs="0"/>
</xsd:sequence>
</xsd:complexType>
```

A sample instance document is illustrated below:

```
<serviceModel xmlns=http://www.metasolv.com/ServiceActivation/2003/ServiceModel
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.metasolv.com/ServiceActivation/2003/
ServiceModel
X:\...\Program\ASAP\4.7.1\920_Analysis+Design\Schema\ServiceModel.xsd">
…
…
</commonService>
<userDefinedExitType>
<CSDL>C-ADD_POTS_LINE</CSDL>
<ASDL>A-ADD_POTS_LINE</ASDL>
<neDescriptor>
<softwareLoad>DYNAMIC_SL</softwareLoad>
<technology>DYNAMIC_VENDOR-DYNAMIC_TECH</technology>
</neDescriptor>
<searchPattern>SUCCESS.</searchPattern>[3]
<userType>U_SUCCEED</userType>[4]
```

[2] The mapping between base and user types, with an optional search pattern and description.

[3] Pattern searches accommodate situations in which responses from the device contain small variants that represent the same meaning. The user type contains an associated search pattern that is applied at runtime. Using regular expressions, you can default a series of responses. For example a regular expression "90." can specify a pattern where any response

```
<baseType>SUCCEED</baseType>[5]
<description>The Atomic Action provisioning was successful</description>
</userDefinedExitType>
<userDefinedExitType>
<searchPattern>90.</searchPattern>
<userType>U_FAIL</userType>
<baseType>FAIL</baseType>
<description>The Atomic Action failed - fail the current order
and stop processing.</description>
</userDefinedExitType>
<userDefinedExitType>
<searchPattern>101-110[201-215]</searchPattern>[6]
<userType>U_SOFT_FAIL</userType>
<baseType>SOFT_FAIL</baseType>
<description>The Atomic Action has encountered a soft failure. Processing will
continue.</description>
</userDefinedExitType>
<userDefinedExitType>
<searchPattern>801-850</searchPattern>[7]
<userType>U_MINOR_ERROR</userType>
<baseType>SOFT_FAIL</baseType>
<description>The Atomic Action has encountered a soft failure. Processing will
continue.</description>
</userDefinedExitType>
<userDefinedExitType>
<searchPattern>251-275&&[^261-265]</searchPattern>[8]
<userType>U_DELAYED_FAIL</userType>
<baseType>DELAYED_FAIL</baseType>
<description>The Atomic Action has failed during provisioning.</description>
</userDefinedExitType>
<userDefinedExitType>
<CSDL>C-DEL_POTS_LINE</CSDL>
<ASDL>A-DEL_POTS_LINE</ASDL>
<neDescriptor>
<softwareLoad>BCS36</softwareLoad>
<technology>NORTEL_DMS</technology>
<neVendor>Nortel</neVendor>
</neDescriptor>
<searchPattern>*.</searchPattern>
<userType>U_MAINTAIN</userType>
<baseType>MAINTENANCE</baseType>
<description>The Atomic Action will Wait until the NE comes out of
Maintenance Mode</description>
</userDefinedExitType>
</serviceModel>
```

The previous code sample shows some typical search pattern examples.

Creating Activation-Model.xml

The deployment descriptor must be named **activation-model.xml** and must reside in the top level of the **META-INF** directory of the service activation archive (SAR) file. The deployment

with the character "90" followed by any character will translate to base type of FAIL. If the regular expression is defined as "90*", then any response with the character "90" followed by any number of characters will translate to base type of FAIL

4 The user type that the search pattern maps to.

5 The base type that maps to the user type.

6 101 to 110 and 201 to 215 will translate to a base type of SOFT_FAIL

7 801-850 will translate to a base type of SOFT_FAIL. Note that the user type differs from the previous range.

8 251 to 275 but not 261 to 265 will translate to a base type of DELAYED_FAILURE.

descriptor must be a valid XML document according to the schema for an activation model deployment descriptor XML document.

When you define an activation model, Oracle recommends that you define your own unique namespace and corresponding namespace prefix for your names. For example:

```
<activationModel targetNamespace="Nortel,UEIMAS,5.2,ADSL/ATM,DSL 2.0.7"
xmlns="http://www.metasolv.com/2003/ServiceActivation/ActivationModel" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xmlns:imas="Nortel,UEIMAS,5.2,ADSL/ATM,DSL 2.0.7"
xmlns:foo="foo,bar"
xsi:schemaLocation="http://www.metasolv.com/2003/ServiceActivation/ActivationModel
D:\ccm_databases\ASAP~smith_windows\ASAP\jmx\xsd\SA_Archive.xsd">
     <vendor>String</vendor>
     <version/>
     <name>String</name>
     <components>
          <component>
               <serviceModel>my_service_model.xml</serviceModel>
          </component>
     </components>
</activationModel>
```

In this example, any new IDs (commonServices, atomicServices, etc.) defined in the activation model are scoped by the target namespace. Any references to IDs in other activation models must be prefixed by the appropriate namespace (for example, **foo:name**). Any references to IDs in the current activation model must also be prefixed because the default namespace (**xmlns**=) refers to the activation model schema namespace. The semantics for the XML schema are the same.

Any service models defined in this SAR must also define the IMAS namespace because any common services, atomic services, and so forth, defined in the service model are scoped by the target namespace. Any references to IDs in other cartridges must be prefixed by the appropriate namespace.

Configuring Network Element Throughput Using XML

NE instance throughput (expressed as the number of milliseconds per transaction on an NE) can be configured using XML through the SACT or using **asap_utils**. (See **asap_utils** option **18. Set NE instance throughput** in the *ASAP Server Configuration Guide* for more information).

For general information on NE throughput configuration, see the *ASAP Server Configuration Guide*.

The Activation Configuration XML schema supports configuration of NE instance throughput through the ElementType and DynamicRoutingTemplateType schema definitions, illustrated below:

Figure A-1 ElementType Schema

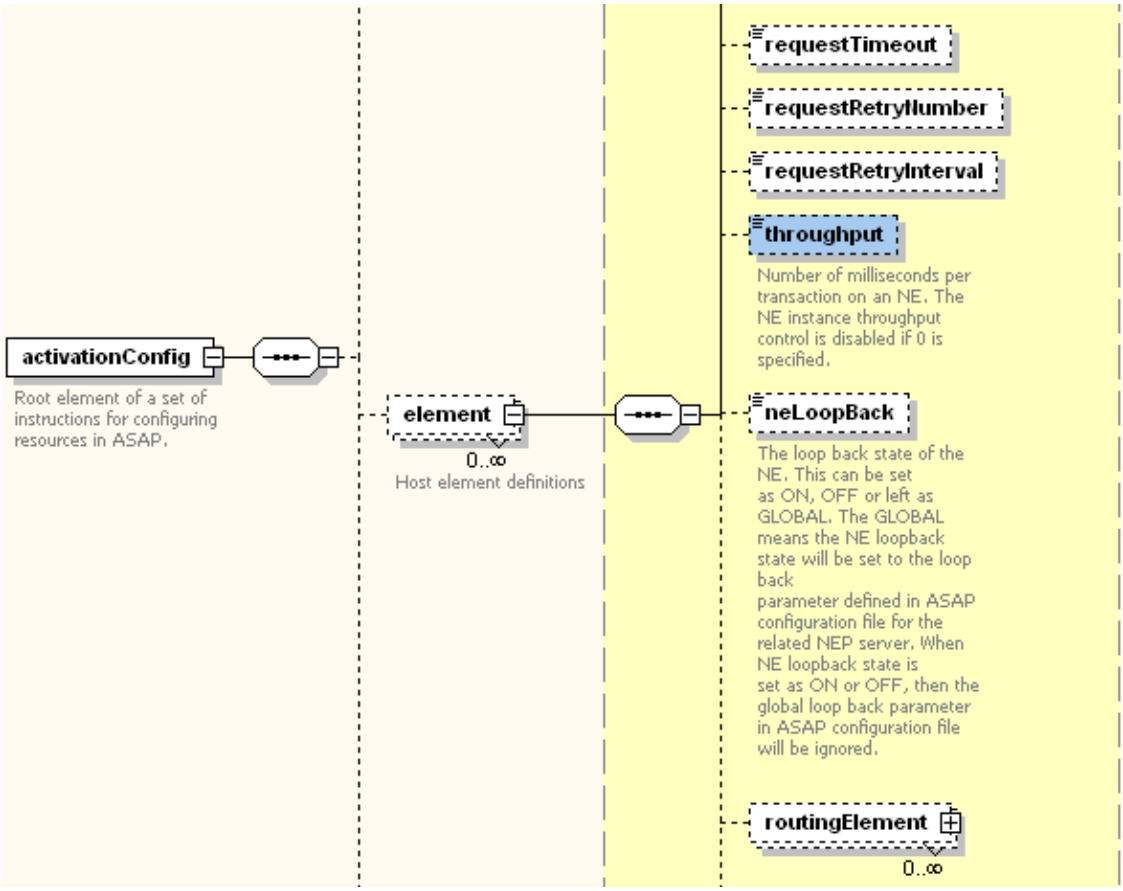
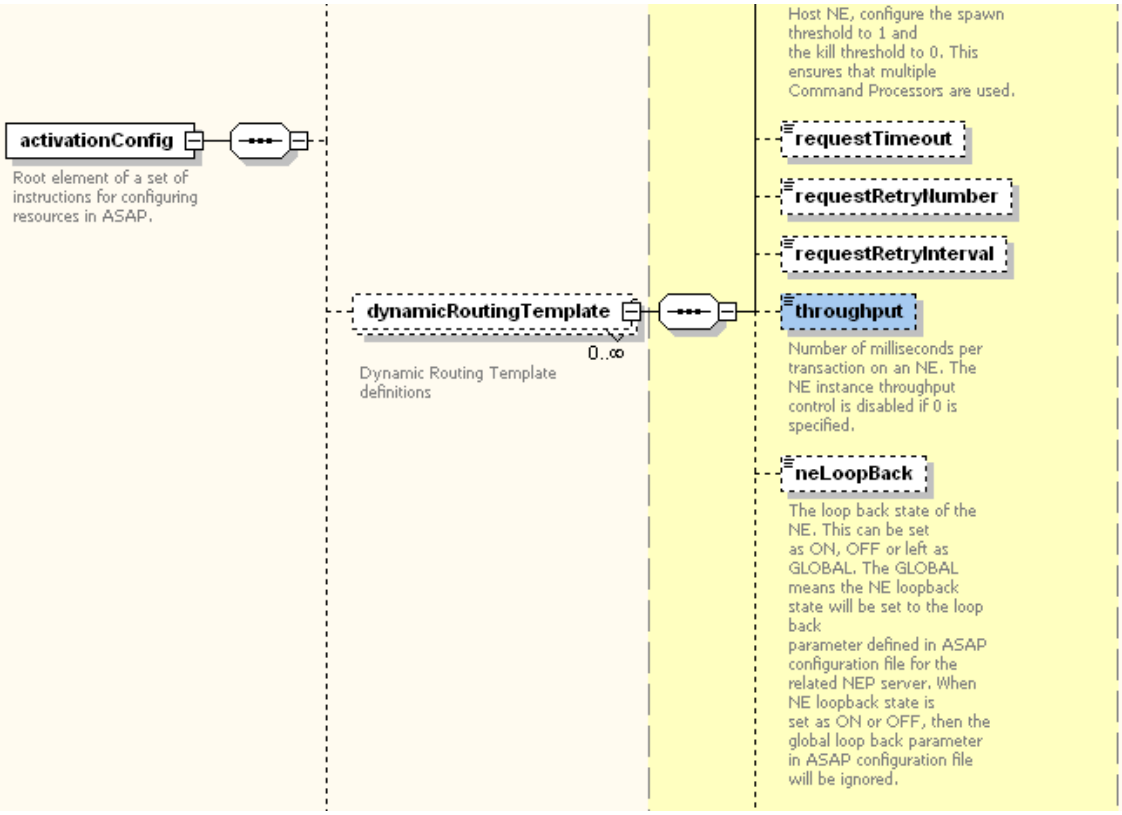


Figure A-2 DynamicRoutingTemplateType Schema



B

Configuring Services Using Stored Procedures

This appendix describes stored procedures used to create Oracle Communications ASAP service models.

> **Note:**
>
> Stored procedures have been deprecated.

Configuring ASAP Services Using Stored Procedures

The configuration of ASAP services using stored procedures consists of the following steps:

- Configuring Service Actions
- Configuring Atomic Actions
- Configuring Atomic Action Parameters
- Configuring Service Action-to-Atomic Action Mappings
- Configuring Atomic Action-to-Program Mappings

Configuring Service Actions

Use the following stored procedures to define, list, and delete service action commands:

- **SSP_new_csdl_defn** – Defines a service action command in the SARM database.
- **SSP_list_csdl_defn** – Lists configuration information for the service action command you specify from the SARM database. This information includes the rollback flag, service action command service action level, fail and completion events, and a description of the command. If you do not specify a service action command, the procedure returns information on all service action commands currently defined in the SARM.
- **SSP_del_csdl_defn** – Deletes service action definitions from the SARM database.

For more information on these stored procedures and **tbl_csdl_config**, refer to the *ASAP Developer's Guide*.

Configuring Atomic Actions

Use the following stored procedures to define, list, and delete atomic actions:

- **SSP_new_asdl_defn** – Defines an atomic action configuration record in the SARM database.
- **SSP_list_asdl_defn** – Lists all or specific atomic action definitions from the SARM database. You can use wildcards in this procedure. If you do not specify a parameter, all atomic action definitions are listed.
- **SSP_del_asdl_defn** – Deletes atomic action definitions from the SARM database.

For more information on these stored procedures and tbl_asdl_config, refer to the *ASAP Developer's Guide*.

Configuring Atomic Action Parameters

Use the following stored procedures to define, list, and delete atomic action parameters:

- **SSP_new_asdl_parm** – This stored procedure defines up to nine atomic action parameters in a single stored procedure call for the specified atomic action starting at base_seq_no.
- **SSP_list_asdl_parm** – This stored procedure lists atomic action parameters from the SARM database by atomic action name and/or atomic action parameter label.
- **SSP_del_asdl_parm** – This stored procedure deletes an atomic action parameter from the specified atomic action.

These stored procedures populate tbl_asdl_parm. For more information, refer to the *ASAP Developer's Guide*.

Configuring Service Action-to-Atomic Action Mappings

The following stored procedures update **tbl_csdl_asdl**. **tbl_csdl_asdl** is a static table that is used by the SARM and contains the mapping between service action commands and atomic actions. For each atomic action associated with a service action, the SARM verifies whether the atomic action should be spawned for the specified service action. The final determination of whether the atomic action is spawned depends on the atomic action parameter translation process specified by **tbl_asdl_parm**.

Use the following stored procedures to define, list, and delete service action to atomic action mappings:

- **SSP_new_csdl_asdl** – This stored procedure defines up to nine service action-to-atomic action mappings at a time from a service action command to one or more atomic actions with consecutive numbers starting at base_seq_no=1 in the SARM database.
- **SSP_new_csdl_asdl_idx** – This stored procedure allow multiple conditions to be inserted into tbl_csdl_asdl_eval.
- **SSP_list_csdl_asdl** – This stored procedure lists service action-to-atomic action mapping definitions.
- **SSP_del_csdl_asdl** – This stored procedure deletes service action-to-atomic action mapping definitions from the SARM database.

These stored procedures populate **tbl_csdl_asdl** and **tbl_csdl_asdl_eval**. For more information, refer to the *ASAP Developer's Guide*.

Configuring Atomic Action-to-Program Mappings

Use the following stored procedures to define, list, and delete atomic action-to-program mappings:

- **SSP_new_asdl_map** – This stored procedure defines a mapping from an atomic action to a program based on the technology and software load in the SARM database.
- **SSP_list_asdl_map** – This stored procedure lists atomic action-to-program mappings according to various criteria.
- **SSP_del_asdl_map** – This stored procedure deletes atomic action-to-program mappings. The mapping is based on the technology and software load.

For more information on these stored procedures and tbl_nep_asdl_prog, refer to the *ASAP Developer's Guide*.

Configuring Network Elements Using Stored Procedures

The definition of host network elements (NEs) consists of the following procedures:

- Configuring Host Network Elements
- Configuring Host to Remote Network Element Mappings
- Configuring NEP-to-Host NE Mappings
- Configuring Resource Pools
- Configuring Communication Parameters
- Configuring Network Element Error Thresholds
- Configuring User Errors and Thresholds
- Configuring Static Routing
- Configuring Network Element Blackout Periods (optional)

Configuring Host Network Elements

tbl_host_clli is a static table that contains the host NE, the technology, and the software load of each NE in the ASAP system. It also contains records for each host NE to which the NEPs interface. You must populate this table to determine which NEs the NEP interfaces with.

Use the following stored procedures to define, list, and delete host NEs:

- **SSP_new_ne_host** – This stored procedure defines a host NE with its technology type, software version, and inventory manager in the SARM database.
- **SSP_list_ne_host** – This stored procedure lists host NE definitions.
- **SSP_list_host** – This stored procedure retrieves host-related information from **tbl_resource_pool**, **tbl_ne_config**, and **tbl_clli_route**.
- **SSP_del_ne_host** – This stored procedure deletes a host NE definition from the SARM database.

> **Note:**
>
> You cannot delete a host NE that has a mapping relationship with either an NEP or a remote NE. Any mapping relationship must therefore be deleted prior to deleting the host NE.

For more information on these stored procedures and **tbl_host_clli**, refer to the *ASAP Developer's Guide*.

Configuring Host to Remote Network Element Mappings

tbl_clli_route is a static table that contains the mapping between a remote NE and its host NE. You must populate this table to specify remote NE to host NE mappings.

Use the following stored procedures to define, list, and delete mappings from a remote NE to a host NE:

- **SSP_new_clli_map** – This stored procedure defines a mapping from a remote CLLI to a Host CLLI in the SARM database.
- **SSP_list_clli_map** – This stored procedure lists remote CLLI-to-Host CLLI mapping definitions.
- **SSP_del_clli_map** – This stored procedure deletes a remote CLLI to host CLLI mapping.

For more information on these stored procedures and **tbl_clli_route**, refer to the *ASAP Developer's Guide*.

Any changes you make to the mapping relationships between host NEs to remote NEs, only take effect at runtime. All other changes require that you restart the SARM.

Configuring NEP-to-Host NE Mappings

NEP to host NE mappings are defined in **tbl_ne_config**.

Use the following stored procedures to configure NEP to host NE mappings.

- **SSP_new_net_elem** – This stored procedure defines a host NE in the SARM database and identifies the logical name of the NEP that connects to this host NE. It also defines the loopback setting for the NE.
- **SSP_list_net_elem** – This stored procedure lists NE definitions based on the host NE and/or NEP server you specify.
- **SSP_del_net_elem** – This stored procedure deletes an NE definition for an NEP from the SARM database.
- **SSP_set_ne_loopback** – This stored function is called by NEP server to update the table **tbl_ne_config** when the loop back state is set to ON, OFF, or GLOBAL through the utility tool **asap_utils**.

For more information on these stored procedures and **tbl_ne_config**, refer to the *ASAP Developer's Guide*.

Configuring Resource Pools

tbl_resource_pool is a static table that defines the collection of command processors (devices) that the NEP uses to establish connections to NEs. Groups of command processors are called resource pools. Each NE configuration determines a primary resource pool that defines one or more devices the NEP uses to connect to that NE. These devices are not used to connect to other NEs. Each NEP has an auxiliary resource pool that contains devices used by the NEP to establish connections to any NE managed by the NEP. These primary and auxiliary resource pools are defined in this table. You must populate this table to add command processors.

Use the following stored procedures to define, delete, and list command processors:

- **SSP_new_resource** – This stored procedure defines an NEP resource (“device") to be used for NE access in the SARM database.
- **SSP_del_resource** – This stored procedure deletes an NEP resource record from the SARM database.
- **SSP_list_resource** – This stored procedure lists NEP resource records.

For more information on these stored procedures and **tbl_resource_pool**, refer to the *ASAP Developer's Guide*.

Configuring Communication Parameters

tbl_comm_param contains communication parameters required for the NEP to communicate with various external systems. You must populate this table to configure communication parameters.

For more information on **tbl_comm_param**, refer to the *ASAP Developer's Guide*.

Use the following stored procedures to define, list, and delete communication parameters:

- **SSP_new_comm_param** – This stored procedure defines a communication parameter for a specified device type, host, and device into the SARM database.
- **SSP_list_comm_param** – This stored procedure lists communication parameter information for dev_type, host, device, **param_label**, or for all of them.
- **SSP_del_comm_param** – This stored procedure deletes communication parameter information from the SARM database.

Configuring Network Element Error Thresholds

Use the following stored procedures to define, list, and delete error thresholds.

- **SSP_new_err_threshold** – This stored procedure defines error thresholds for a specific NE and atomic action.
- **SSP_list_err_threshold** – This stored procedure lists the error thresholds for a specific NE and atomic action.
- **SSP_del_err_threshold** – This stored procedure deletes error thresholds for a specific NE and atomic action.

For more information on these stored procedures, refer to *ASAP Developer's Guide*.

Configuring User Errors and Thresholds

Use the following stored procedures to define, list, and delete user errors.

- **SSP_new_err_type** – This function configures the mapping between user-defined error types and the base-error types.
- **SSP_list_err_type** – This function lists the mapping between user-defined error types and the base-error types.
- **SSP_del_err_type** – This function deletes the mapping of user-defined error types.

Use the following stored procedures to define, list, and delete user error thresholds.

- **SSP_new_user_err_threshold** – This stored procedure creates a new user-defined error threshold in the system for the specified NE, atomic action, and the user-defined error type.
- **SSP_list_user_err_threshold** – This stored procedure is used to list the user-defined error thresholds for a specific NE, atomic action, and user error type.
- **SSP_del_user_err_threshold** – This stored procedure deletes a user-defined error threshold or set of thresholds.

For more information on these stored procedures, refer to *ASAP Developer's Guide*.

Configuring Static Routing

Configuring Atomic Action Routings by ID_ROUTING Using Stored Procedures

The stored procedures that you can use as external interfaces are the following:

- **SSP_list_id_routing** (RC1, host_clli) – Lists the host NE and ID_ROUTING mapping records in the SARM database.
- **SSP_new_id_routing** (host_clli, asdl_cmd, id_routing_from, id_routing_to) – Defines the host NE and ID_ROUTING mapping records in the SARM database.
- **SSP_del_id_routing** (host_clli, asdl_cmd, id_routing_from, id_routing_to) – Deletes the host NE and ID_ROUTING mapping records from the SARM database.

For more information on these stored procedures, refer to the *ASAP Developer's Guide*.

The following steps must be followed when routing by ID_ROUTING:

1. Populating the routing table (tbl_id_routing).
2. Defining the atomic action parameter. A sample is located in ..\samples\ASDL_ROUTE\oraRoutingServices.
3. Defining the work order. A sample is located in ..\samples\ASDL_ROUTE\RoutingSrpInput.
4. Starting ASAP and submitting the work order.

The following examples provide samples of how each step can be configured.

The following example displays how to populate tbl_id_routing.

```
sqlplus -s $SARM_USER/$(GetPassword $SARM_USER 2)
<<HERE | grep -v "successfully completed"

set serveroutput on
var retval number

prompt Defining the ID_ROUTING Configurations

exec :retval := SSP_del_id_routing ;

exec :retval := SSP_new_id_routing ('BALTIMORE', '', 'BAL', 'CAL');
exec :retval := SSP_new_id_routing ('BALTIMORE', '', 'DEL', 'FAL);
exec :retval := SSP_new_id_routing ('BOSTON', '', '120000', '220000');

HERE
```

You can add new records to the database dynamically without downtime on the server by using the “Add new NE Configuration" command (113) of asap_utils. This command must be used after loading the ASAP database.

For more information on asap_utils, see the *ASAP Server Configuration Guide*.

For more information on the tbl_id_routing table, see the *ASAP Developer's Guide*.

Configuring Atomic Action Routings by USER_ROUTING

You can perform atomic action routing by using a user-defined procedure. Routing by user-defined procedure provides the following:

- Allows for custom provided logic for atomic action routing.
- Uses the atomic action parameter USER_ROUTING.
- Uses the external interface SSP_get_user_routing.
- Allows you to write your own routing logic using the predefined external user interface.

The USER_ROUTING parameter can be represented as any string of characters to a maximum of 255 characters. You can define it as part of a work order, or as a service action parameter.

If the atomic action parameter USER_ROUTING information is provided in the work order, then the user-defined stored procedure is called. The user-defined procedure takes the asdl_cmd and the value of USER_ROUTING as input arguments, and returns the host NE to be routed.

You can use the following stored procedure as an external interface:

- **SSP_get_user_routing** (user_routing, asdl_cmd, host_clli, ret_val) – Returns a host NE (host_clli) that is used to route the atomic action. You must provide your own routing logic in the body of SSP_get_user_routing to find the host NE (CLLI) using the USER_ROUTING atomic action parameters, and the asdl_cmd if required.

For more information on the above stored procedure, refer to the *ASAP Developer's Guide.*

To use USER_ROUTING, perform the following steps:

1. Write the stored procedure SSP_USER_ROUTING. A sample is located in ..\samples\ASDL_ROUTE\user_routing_proc.sp.
2. Define and populate the routing table, if required. A sample is located in ..\samples\ASDL_ROUTE\user_routing_table.tbl and ..\samples\ASDL_ROUTE\oraLoadRouting.
3. Define the atomic action parameter. A sample is located in ..\samples\ASDL_ROUTE\oraRoutingServices.
4. Define the work order. A sample is located in ..\samples\ASDL_ROUTE\RoutingSrpInput.
5. Run ASAP and submit the work order.

When you choose a user-defined procedure with a database table, the database must be accessed every time the routing is requested. Consequently, there will be a slight performance degradation.

Configuring Atomic Action Routings by Distinguished Name

You can edit routing definitions provided the new routing definition does not already exist in ASAP.

- **SSP_new_dn_map** – This stored procedure defines atomic action routings by directory number.
- **SSP_list_dn_map** – This stored procedure lists directory mappings for atomic action, directory, exchange number, or for all of them.
- **SSP_del_dn_map** – This stored procedure deletes a directory number mapping from the SARM database.

Configuring Network Element Blackout Periods (optional)

Use the following stored procedures to define, list, and delete blackout definitions.

- **SSP_add_blackout** – This procedure configures the static and dynamic blackout periods for a specific NE host.
- **SSP_list_blackout** – This procedure lists blackout periods for a specific NE host.
- **SSP_del_blackout** – This procedure removes blackout periods for a specific NE host.

For more information on these stored procedures, refer to *ASAP Developer's Guide*.

Checking Network Element Blackout Periods

The stored procedure **SSP_check_blackout** enables you to check whether or not the specified NE is currently blacked out.

For more information on this stored procedures, refer to *ASAP Developer's Guide*.