Oracle® Communications MetaSolv Solution

Custom Extensions Developer's Reference

Release 6.2.1

E48741-01

August 2014

ORACLE®

Oracle Communications MetaSolv Solution Custom Extensions Developer's Reference, Release 6.2.1

E48741-01

Contents

B Extensions Sample Code

Preface

This document explains how to extend the Oracle Communications MetaSolv Solution business logic with custom business through the use of custom extensions.

Audience

This document is for individuals who are responsible for developing software to integrate an external application with MetaSolv Solution. This document assumes the reader has a working knowledge of Oracle DBMS 11g, Windows XP, Oracle WebLogic 10.3, and Java JEE.

Related Documents

For more information, see the following documents in Oracle Communications MetaSolv Solution 6.2.1 documentation set:

- *MSS Planning Guide*: Describes information you need to consider in planning your MetaSolv Solution environment prior to installation.
- *MSS Installation Guide*: Describes system requirements and installation procedures for installing MetaSolv Solution.
- *MSS System Administrator's Guide*: Describes post-installation tasks and administrative tasks such as maintaining user security.
- *MSS Database Change Reference*: Provides information on the database changes for the MetaSolv Solution 6.2.1 release. Database changes for subsequent maintenance releases will be added to this guide as they are released.
- *MSS Network Grooming User's Guide*: Provides information about the MSS Network Grooming tool.
- *MSS Address Correction Utility User's Guide*: Provides information about the MSS Address Correction utility.
- *MSS Technology Module Guide*: Describes each of the MetaSolv Solution technology modules.
- *MSS Data Selection Tool How-to Guide*: Provides an overview of the Data Selection Tool, and procedures on how it used to migrate the product catalog, equipment specifications, and provisioning plans from one release of your environment to another.
- *MSS Operational Reports*: Provides an overview of using Operational Reports and Business Objects with MSS, and procedures for running reports, updating universes, and simple maintenance.

- *MSS CORBA API Developer's Reference*: Describes how MetaSolv Solution APIs work, high-level information about each API, and instructions for using the APIs to perform specific tasks.
- *MSS XML API Developer's Reference*: Describes how to integrate MetaSolv Solution with other Oracle products, or with external applications, through the use of APIs.
- *MSS Flow-through Packages Guide*: Describes information and procedures you need to install and work with the flow-through packages provided by Oracle as an example of how to integrate MetaSolv Solution with ASAP for flow-through activation.

For step-by-step instructions for tasks you perform in MetaSolv Solution, log into the application to see the online Help.

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 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.

1

Extensions Overview

This chapter provides basic information about custom extensions and how you can use them to invoke API calls and send messages that support your business processes.

About Custom Extensions

A custom extension enables you to extend Oracle Communications MetaSolv Solution functionality with additional business logic specific to your organization. In other words, extensions provide the ability to make calls to external systems and to send email and JMS messages at predefined execution points, over and above the functionality supported by the MetaSolv Solution application and APIs.

You can develop custom extensions that simply send data to another system, or that send and receive data. An extension that sends data and does not expect a response from an external system is called **asynchronous.** An example of an asynchronous extension is an email message. You may choose to develop an asynchronous extension that sends an email when a particular process or event occurs in MetaSolv Solution.

An extension that sends data and expects a response from an external system is called **synchronous.** An example of an execution point that can be used to develop a synchronous extension is Assign Queues. You may choose to develop a synchronous extension that executes a custom Java class when a particular process occurs in MetaSolv Solution. The Java class executes as its own transaction, separate from the process that initiated it.

Developing a custom extension involves several tasks. These tasks, listed below, appear in a conceptual order to help you understand extensions. In reality, these tasks would probably be performed by different people, and at varying times.

1. Define the extension.
2. Identify execution points.
3. Code the extension logic.

Extensions

The first step in developing a custom extension is to define the extension in the MetaSolv Solution user interface (UI). The extension name that you define is the name of the Java class that will contain your custom logic.

Execution Points

The second step in developing a custom extension is to define the point at which you want the custom extension logic to execute; that is, the process or action that triggers

the invocation of your custom code. You define this execution point by identifying three key pieces of information:

- Building Block
- Process Point
- Action Type

Building Block

A building block type is a predefined item in MetaSolv Solution, such as a gateway event, with which you can associate an extension. Building blocks further describe building block types. For example, using the building block type of Gateway Event enables you to associate an extension with a gateway event. You then further define this item by selecting the building block of All Gateway Events. This means you can associate the extension with all gateway events, as opposed to specific events.

Process Point

A process point describes general processing that takes place in MetaSolv Solution, for example, gateway event maintenance. To continue with the example used for building blocks, you can associate a process point of GW (gateway) Event Maintenance with the extension. This means the extension logic is triggered when MetaSolv Solution processes some type of gateway event maintenance.

Like building blocks, process points are predefined in MetaSolv Solution.

Action Type

An action type is a specific task or process that takes place in MetaSolv Solution. When you associate an action type with an extension, you are identifying the specific action that triggers the extension logic to execute for a particular extension. To conclude the previous example, you can associate the action type of GW (gateway) Event Failed with the extension. This means the extension logic is triggered when MetaSolv Solution processes a gateway event and it fails to successfully complete.

Like building blocks and process points, action types are predefined in MetaSolv Solution.

Extension Logic

The next step in developing a custom extension is to code a free-form Java class that provides additional functionality to support your business processes. As examples, you can code a Java class to:

- Make calls to external systems
- Send email notifications
- Send JMS messages
- Invoke other MetaSolv Solution API calls

Invocation Methods

This section is not listed as a step in the above "About Custom Extensions" because identifying the execution points is what defines the invocation methods. Therefore, this is not actually a step that you need to perform. However, it is important to

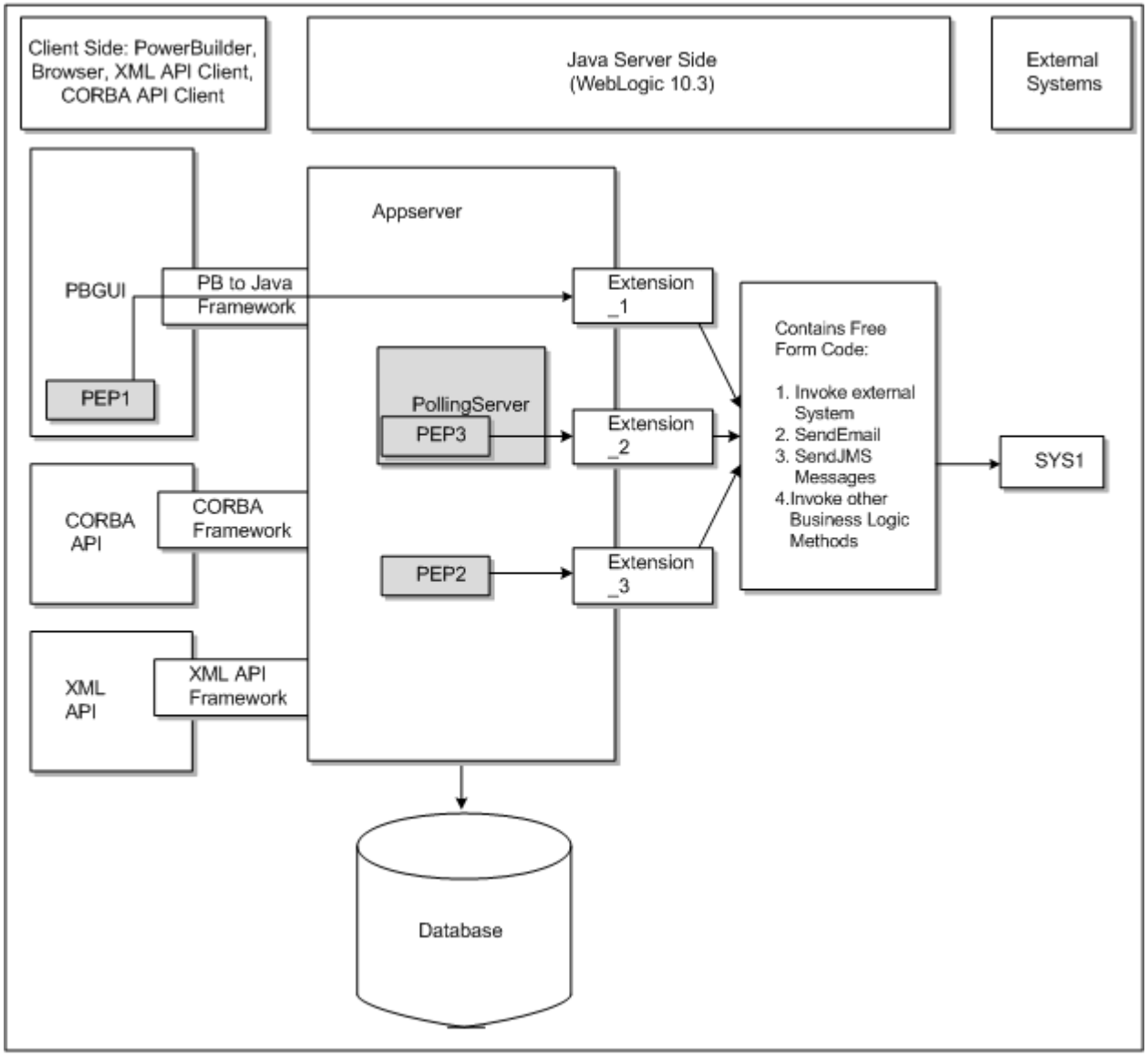
understand the information contained in this section, so it is included in the overview because it addresses, at a high level, how custom extension logic is invoked. See "Supported Execution Points" for specific information regarding invocations for supported execution points.

After you define the extension, associate the execution point, and code the logic for your custom extension, it is invoked from one or more of the places listed below. The invocations are dependent upon the execution points associated with your extension.

- MetaSolv Solution UI
- XML API Clients
- CORBA API Clients
- Polling Servers

Figure 1–1 shows the architecture of MetaSolv Solution and how the various system components interact to support custom extension functionality.

Figure 1–1 Architecture Supporting Extension Functionality



MetaSolv Solution UI

You can invoke extension logic through the UI when the specified action, defined by an execution point (combination of building block, process point, and action type), occurs. For example, a user assigning a jeopardy code to a task is a specific action that can invoke an extension, if that action is defined as an execution point. Specifically, you would choose the execution point combines the building block type of Task Type, process point of Task Maintenance, and action type of Assign Jeopardy.

XML API Clients

You can invoke extension logic through a call to an XML API method when the specified action, defined by an execution point (combination of building block, process point, and action type), occurs. For example, a third party calling the addTaskJeopardyRequest method to assign a jeopardy code to a task is a specific action that can invoke an extension, if that action is defined as an execution point. Specifically, you would choose the execution point that combines the building block type of Task Type, process point of Task Maintenance, and action type of Assign Jeopardy.

CORBA API Clients

You can invoke extension logic through a call to a CORBA API method when the specified action, defined by an execution point (combination of building block, process point, and action type), occurs. For example, a third party calling the deleteTaskJeopardy method to remove a jeopardy code from a task is a specific action that can invoke an extension, if that action is defined as an execution point. Specifically, you would choose the execution point that combines the building block type of Task Type, process point of Task Maintenance, and action type of Assign Jeopardy.

Polling Servers

You can invoke extension logic through polling servers as well. These servers, which need to be configured in the **gateway.ini** file, are listed on the following page. See "Additional Configurations" for detailed information regarding these configurations.

Polling servers can invoke extension logic if the action of the polling server is defined as an execution point. For example, a task that is defined as a system task with a task execution point of Ready is automatically picked up by the System Task Server when the task status becomes Ready. If the task completion logic that runs on the server fails, extension logic can be invoked if it defines that as an execution point. Specifically, you would choose the execution point that combines the building block type of Task Type, process point of Task Maintenance, and action type of System Task Failure.

Polling Servers and Supported Execution Points

The following polling servers can invoke an extension that is defined with the specified execution points. See "Supported Execution Points" for more information about the supported execution points.

- Background Processor
  - System Task Failure
- Gateway Event Server
  - Gateway Event Failure

- Integration Server
  - Gateway Event Failure
  - Late Task
  - Potentially Late Task
- System Task Server
  - System Task Failure

Note: The Background Processor is not a Java-based polling server. Rather, it is a PowerBuilder application that runs in the background.

2

Defining An Extension

This chapter explains how to define a custom extension in the user interface (UI). Online Help for defining an extension is available in the help topics listed below.

Open the online Help and type the following window or procedure names in the Search field:

- Extensions window
- Extension Summary window
- Extension Parameters window
- Opening the Extension Summary window
- Creating a new Extension
- Editing an existing Extension
- Deleting an existing Extension
- Associating an Execution Point to an Extension
- Disassociating an Execution Point from an Extension
- Editing an Extension Parameter
- Filtering the Extensions list

Defining an Extension in the UI

For specific UI instructions on how to define the extension, see the online Help procedures Creating a new Extension, Associating a Process Point to an Extension, and Editing an Extension Parameter.

Type of Extension

When defining an extension, you must select the Type from a drop-down. The following types display in the drop-down, which is defaulted to Logic.

- Logic

  Logic is the only type of extension that is supported at this time. Logic extensions define associated execution points that, when triggered, invoke the custom extension logic Java class defined by the extension name.

- Viewable

  Viewable extensions are not supported at this time.

Name of Extension

When defining an extension, you must define the name of the extension. The name of the extension is the name of the Java class that is to be invoked when an associated execution point is triggered. When naming your extension, be sure to follow Java class naming standards such as starting with an upper case letter, using upper and lower case letters to distinguish words, no spaces, etc. Also, do not include the **.java** file type extension in the name of the extension. For example, if you are defining an extension to call the Java class MySpecificLogic.java, name the extension MySpecificLogic.

Execution Mode

When defining an extension, you must select the **Execution Mode** from a drop-down. The following execution modes display in the drop-down, which is defaulted to **Synchronous**.

- Synchronous

  A synchronous extension executes and returns specified data. The calling process must wait for the extension to finish before continuing.

- Asynchronous

  An asynchronous extension executes and does not return any data, allowing processing to continue without waiting for the extension to finish.

Associating an Execution Point With an Extension

When defining an extension, you must associate one or more execution points with the extension. Execution points are predefined combinations of a building block, process point, and action type. These execution points have "hooks" in the code that, when triggered, invoke the extension Java class. See "Identifying An Execution Point" for more information.

Defining the Extension Parameters

When defining an extension, the parameter IDs and their corresponding default names are displayed on the Extension Parameters window. The types of extension parameters are predefined for each execution point, such as String, int, etc. The corresponding default parameter names may be edited so that is has meaning to your particular usage of it.

Configuring an Extension

This section describes how to configure a custom extension in the **gateway.ini** file and additional information regarding configuration requirements.

Configuring Gateway.ini

To enable custom extensions, the following changes must be made in the **gateway.ini** file located in the *MSLV_Home*/*server*/**appserver/gateway** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

Specifying the CLASSPATH tells the framework where to find your custom extension Java class, which must reside in the path specified in the **gateway.ini**.

1. Save a copy of the **gateway.ini** file.

2. Open the original **gateway.ini** file for editing.
3. Add the following line at the end of [Custom] section within the file. (If your **gateway.ini** file does not have the [Custom] section, you need to add it.)
   - For Windows operating systems

     CLASSPATH=*MSLV_Home***/***server***/appserver/samples/customExtension**;
   - For Unix operating systems

     CLASSPATH=*MSLV_Home***//***server***//appserver//samples//customExtension**;

     where:

     MSLV_Home is the directory in which the MetaSolv Solution software is installed

     server is the name of the WebLogic server
4. Save and close the file.

Additional Configurations

See *MetaSolv Solution Installation Guide* for more information on the configuration requirements for using custom extensions.

> **Note:** You need to manually modify the **loggingconfig.xml** file and **integration.XML** file to avoid encountering an error that appears on your appserver console. Additionally, when using custom extensions, you must manually modify the **gateway.ini** file. See *MetaSolv Solution Installation Guide* for more information.

If you have performed a full installation, these configurations are already in place. The configurations described in *MetaSolv Solution Installation Guide* need to be set up only if you have upgraded from a release prior to 6.0.12.

> **Note :** Regarding step 3 in "Configuring Gateway.ini", a full installation puts the classpath for custom extensions in the **gateway.ini** file, but you still need to specify the correct path to your server. For an upgrade, you need to add the classpath for custom extensions to **gateway.ini** file as well as specify the correct path to your server.

Invoking an Extension

Certain execution points are invoked by polling servers. See "Polling Servers" for more information. Three of these servers are Java-based servers that need to run as part of the appserver. This is accomplished by configuring the **gateway.ini** file to define the appropriate servers within the [Servers] section as follows:

- Gateway Event Server

  EVENTPROC=MetaSolv.eventServer.S3Startup
- Integration Server

  INTEGRATIONSERVER=com.mslv.integration.integrationServer.S3Startup
- System Task Server

SYSTEMTASKSERVERPROC=com.mslv.core.api.internal.WM.systemTaskServer.SystemTaskServer

The remaining server, the Background Processor, is not part of the appserver and, therefore, is not configured through the **gateway.ini** file. To start the background processor, run **jmaster.exe** located in the MSS directory.

3

Identifying An Execution Point

This chapter explains how to identify an execution point. Once identified, you can then associate execution points with an extension. See "Defining An Extension" for more information. Online Help for identifying execution points is available in the help topics listed below.

Open the online Help and type the following window or procedure names in the **Search** field:

- Execution Point Search and Results window
- Execution Points window
- Searching for an Execution Point
- Toggling between Execution Point Search and Results
- Filtering the Execution Points list

Component Options

An execution point is defined by a combination of three components: its building block, process point, and action type. Oracle Communications MetaSolv Solution predefines a number of options for each of these components, along with the combinations of options that represent valid execution points. This section describes the options that are available for each component.

Building Block Options

Building blocks are grouped into building block types. Both building blocks and building block types are MetaSolv Solution defined data. The following table lists building block types that appear in the drop-down list on the Execution Point Search window.

Building block type options:

- Task Type
- Gateway Event
- Connection
- Network System
- Address

Table 3–1 lists the building blocks defined by MetaSolv Solution that can be used with extensions. The building blocks available for selection depend on the building block type chosen. The building block ID, an Oracle generated number, is included in the

information because it is part of the data that is passed from an execution point to an extension Java class.

Table 3–1 Building Block Options

Building block	Building block ID
All Task Types	1001
All Gateway Events	1002
[specific task type]	[depends on task type]
All Connections	409
All Network Systems	410
All End User Locations	-

Specific task types are user defined data stored on the TASK_TYPE table. To support the Complete Task execution point for individual task types, the building block id field (ms_bb_id) was added to the TASK_TYPE table. A row is inserted into the TASK_TYPE table when a new task type is created in Work Management. However, the ms_bb_id field is not populated with the row insertion, rather, it is populated when the task is selected from the **Name** list in the Execution Point Search window. The **Name** list displays all task types when you select **Task Type** in the **Building Block Type** list.

Note: This document does not provide the building block ids for each task type because they are based on user data. Building block ids are not displayed in the application, therefore, they must be manually looked up on the TASK_TYPE table.

Process Point Options

Table 3–2 lists the process points defined by MetaSolv Solution that can be used with extensions. The process points available for selection depend on the building block chosen. The process point ID, an Oracle generated number, is included in the information because it is part of the data that is passed from an execution point to an extension Java class.

Table 3–2 Process Point Options

Process point	Process point ID
Task Generation	1
Task Maintenance	101
GW Event Maintenance	102
PCONDES Maintenance	103
VCONDES Maintenance	105
Network System Design	107
Connection Design	108
Print	140
EUL Maintenance	123
PSR	124

Action Type Options

Table 3–3 lists the action types defined by MetaSolv Solution that can be used with extensions. The action types available for selection depend on the process point chosen.The action type ID, an Oracle generated number, is included in the information because it is part of the data that is passed from an execution point to an extension Java class.

Table 3–3 Action Type Options

Action type	Action type ID
Generate	32
Assign Jeopardy	41
Reject	42
Assign Queues	43
Change Completion Date	44
System Task Failure	45
Late	46
Potentially Late	47
GW Event Failed	51
Provision Plan Default	52
Complete	53
Select Component or Element	54
Select Port Address	55
Email	56
Select Network System	60
Select Customer Edge Component	61
Select End Component For Physical Connection	62
Select Equipment For CE	63
DS0/DS1 Automated Design	70
Connection Id Automation	71
Select Dedicated Plant	-
Update	91
Create	92

Component Combinations

As explained in each of the previous component sections, there are dependencies between the components. Specifically, action types are dependent on process points, which are dependent on building blocks, which are dependent on building block types.

Table 3–4 shows the current valid combinations that result from these dependencies. For example, if you choose a building block type of Task Type, your only choice of building block is currently All Task Types. If you then choose the process point of Task Generation, your only action type choices are Generate or Provision Plan Default.

Table 3–4 Valid Combinations

Building block type	Building block	Process point	Action type
Task Type	All Task Types	Task Generation	■ Generate ■ Provision Plan Default
Task Type	All Task Types	Task Maintenance	■ Assign Jeopardy ■ Reject ■ Assign Queues ■ Change Completion Date ■ System Task Failure ■ Late ■ Potentially Late ■ Complete
Task Type	[specific task type]	Task Maintenance	■ Complete
Gateway Event	All Gateway Events	GW Event Maintenance	■ GW Event Failed
Connection	All Connections	Print	■ Email
Connection	All Connections	PCONDES Maintenance	■ Select Component or Element ■ Select Port Address ■ Select Dedicated Plant
Connection	All Connections	VCONDES Maintenance	■ Select Component or Element
Connection	All Connections	Connection Design	■ DS0/DS1 Automated Design ■ Connection Id Automation ■ Select Dedicated Plant
Network System	All Network Systems	Network System Design	■ Select Network System ■ Select Customer Edge Component ■ Select End Component For Physical Connection ■ Select Equipment For CE
Address	All End User Locations	PSR	■ Create ■ Update
Address	All End User Locations	EUL Maintenance	■ Create ■ Update

4

Coding The Extension Logic

This chapter provides information about coding the extension Java class. Sample code is provided with your installation of Oracle Communications MetaSolv Solution, and the sample code provides concrete code examples of extension Java classes. See "Extensions Sample Code" for detailed information about the sample code.

Inheriting From the Extension Framework

All extension Java classes must extend the extension framework through the class ExtensionRoot located in the package **com.metasolv.custom.common.extension.** Extending the extension framework is necessary to access the data passed from the execution point. Therefore, all new extension Java classes should contain the following lines of code, or some derivation of them:

```
import com.metasolv.custom.common.extension.ExtensionRoot
public class MyExtension extends ExtensionRoot
```

A derivation of the code could be that the extension Java class directly, or indirectly, extends ExtensionRoot. For example, all of the sample source code extends SampleExtensionRoot rather than ExtensionRoot. That is because SampleExtensionRoot extends ExtensionRoot, adding a middle layer to the inheritance that provides common functionality used by all the sample classes. You may wish to create a similar class, or even use the SampleExtensionRoot class, depending on what you are developing.

All of the sample source code implements the class Extension. This is really not necessary because ExtensionRoot implements Extension. Therefore, by inheritance, any class that extends ExtensionRoot implements Extension.

Accessing Data Passed From the Execution Point

This section provides an overview about methods of accessing data passed from the execution point, and provides class details.

Overview

Extension Java classes cannot define input parameters. Rather, data passed from the execution point can be accessed by the extension Java class through the extension framework. Specifically, the class ExtensionRoot defines the following methods:

```
protected final Policy getPolicy()
protected final Entity[] getParameter()
```

Though these methods are defined as protected, they are available to the extension Java class because it inherits from the class in which the methods are defined (ExtensionRoot). From these two methods, the following data can be retrieved:

- Execution mode

  The execution mode tells you if the execution point that invoked the extension class is defined as synchronous or asynchronous. This information was entered in the UI when defining the extension.

- Execution point

  The execution point tells you the point at which the extension class was invoked. This information is passed in the form of building block ID, process point ID, and action type ID. The unique combination defines a specific execution point such as Assign Queues or Reject Task.

- Execution point data

  The execution point data is the specific data that is associated with each supported execution point. This information is passed in the form of a name/value pair array. See "Supported Execution Points" for the specific data that is passed from each execution point.

Class Details

This section provides details about the policy and entity class.

Policy Class

As mentioned in the "Overview" section, the method getPolicy() returns Policy. However, it actually returns an instance of the class PlugInPolicy, which extends Policy. Therefore, you can caste the returned Policy to PlugInPolicy, which makes an instance of the class PlugInPolicy available to the extension Java class.

The class PlugInPolicy defines the following methods:

```
public String getExecutionMode();
public PlugInExecutionPoint getExecutionPoint();
```

Calling the method getExecutionMode() from the extension Java class returns a String that indicates if the execution mode is synchronous or asynchronous. Calling the method getExecutionPoint() returns an instance of the class PlugInExecutionPoint.

The class PlugInExecutionPoint defines the following methods:

```
int getBuildingBlock();
int getProcessPoint();
int getActionType();
```

Calling these methods returns the combination of building block ID, process point ID, and action type ID that defines an execution point. See "Supported Execution Points" for more information.

Entity Class

As mentioned in the "Overview" section, the method getParameter() returns an Array of Entity classes. Another class, ExtensionData, extends the class Entity. Since ExtensionData is a child of Entity, Entity can be casted to ExtensionData. Casting Entity to ExtensionData makes the Array of ExtensionData available to the extension Java class.

The class ExtensionData defines the following method:

```
public NameValuePair[]  getNameValuePairs()
```

Calling this method from the extension Java class returns an Array of NameValuePair classes. The name/value pairs represent the specific data that is defined for each supported execution point. See "Supported Execution Points" for more information.

Finally, the class NameValuePair defines the following methods:

```
public String getName();
public String[] getValue();
```

Calling these methods returns the String name and the String values. It is important to note that all value data is of type String.

A

Supported Execution Points

The preceding chapters described what custom extensions are and how to create them. As mentioned earlier, Oracle Communications MetaSolv Solution predefines the components used to define execution points: the building blocks, process points, and action types. This means there are specific execution points that are available for your use.

In addition to predefining "Component Combinations" associated with each execution point, MetaSolv Solution provides functionality that supports the invocation of a custom extension Java class for each valid combination. This functionality includes:

- "Hooks" that are triggered by an execution point. These "hooks" call the extension framework, which determines what extension class to invoke based on which extensions the execution point is associated with.
- Parameters for each execution point. The parameters are used to pass data that is pertinent to the execution point to the extension class. This data is then available to the extension class and can be used to code your specific business logic.

The supported execution points are listed in Table A–1. The execution points are grouped by building block, and ordered alphabetically. The number of supported execution points correlates to the number of valid component combinations, and the execution point names correlate to the action type of each valid combination.

Table A–1 Supported Execution Points

Building Block	Execution Point
Task	▪ Assign Queues ▪ Assign Task Jeopardy ▪ Change Task Completion Date ▪ Complete Task ▪ Generate Tasks ▪ Late Task ▪ Potentially Late Task ▪ Provisioning Plan Default ▪ Reject Task ▪ System Task Failure
Gateway Event	Gateway Event Failure

Table A–1 (Cont.) Supported Execution Points

Building Block	Execution Point
Connection	■ Email CLR/DLR/TCO ■ Select Port Address ■ Select Component or Element for Physical Connection ■ Select Component or Element for Virtual Connection ■ DS0/DS1 Automated Design ■ Connection Id Automation ■ Select Dedicated Plant
Network System	■ Select Network System ■ Select Customer Edge Component ■ Select End Component For Physical Connection ■ Select Equipment For CE
Address	■ Create ■ Update

This appendix provides detailed information for each supported execution point, which includes:

- A brief description of the execution point.
- A business example of how you might use the execution point.
- The options you should choose when searching for the execution point to associate it with an extension.
- The data that is sent from the execution point to the extension Java class, and, in the case of a synchronous call, the data that is returned from the extension Java class to the execution point. The data is housed in an Array of name/value pairs. All value data in the name/value pair is of type String.
- How the extension Java class is invoked by the execution point, whether it is by the UI, XML APIs, CORBA APIs, or polling servers.

Execution Points

This section provides information about the following execution points:

- Gateway Event Failure
- Email CLR/DLR/TCO
- Select Port Address
- Select Component or Element for Physical Connection
- Select Component or Element for Virtual Connection
- Select Network System
- Select Customer Edge Component
- Select End Component For Physical Connection
- Select Equipment For CE
- DS0/DS1 Automated Design
- Connection Id Automation
- Select Dedicated Plant
- Create/Update End User Location

Assign Queues

MetaSolv Solution provides the ability to assign a provisioning plan to an order. A provisioning plan defines tasks, and assigns work queues to the tasks within the provisioning plan. This execution point enables you to extend logic in the way the work queues are assigned to tasks within a provisioning plan when tasks are generated for an order.

Business Example

You built provisioning plans and assigned default work queues to the tasks in every plan. However, for a specific task type, you would like to do the following:

- Assign it to the ABC queue at certain hours of the day, depending on the workload.
- Assign it to the XYZ queue at certain hours of the day, depending on the workload.
- Send an email notification to the owner of each work queue when a task is assigned to them.

You can use the Assign Queues execution point to extend logic to accomplish those tasks.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–2 when searching for an execution point to associate with the extension:

Table A–2 Assign Queues Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Assign Queues (43)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–3 shows the data that is passed to the extension Java class.

Table A–3 Assign Queues Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task type Array	taskType
Task number Array	taskId

Table A–4 shows the data that is returned by the extension Java class.

Table A–4 Assign Queues Name/Value Pair Return Data

Data Description	Data Name
Work queue ID Array	workQueueId

The work queue ID Array is returned in the same order as the input Arrays of task types and corresponding task numbers.

UI Invocation

After you assign a provisioning plan to an order, you click the **Queues** button to assign the tasks to the appropriate work queues. The execution point is triggered when you click the **Queues** button on the **Task List** tab of the Tasks window.

When you click the **Queues** button, the task list is sent to the extension. The data received back populates the **Work Queue** field for each task. This logic overrides the default work queues that were assigned to the provisioning plan when it was established. However, you can still select a different work queue for any or all tasks, should you need to do so after the extension logic executes.

XML API Invocation

The XML API method through which the Java class extension is invoked is:

OrderManagement - > assignProvisionPlanProcedureRequest

CORBA API Invocation

The CORBA API method through which the Java class extension is invoked is:

WorkManagement -> generateAndSaveTasks

Assign Task Jeopardy

MetaSolv Solution provides the ability to add, change, and delete jeopardy information for tasks. This execution point enables you to extend logic that executes when jeopardy information on a task changes (in the form of add, change, or delete).

Business Example

You assigned a provisioning plan and, from your Work Queue, set up a jeopardy code on a task. The task ends up going into jeopardy. When the jeopardy status changes, the

extension logic executes and sends an email notification to the appropriate person regarding the task jeopardy status.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–5 when searching for an execution point to associate with the extension.

Table A–5 Assign Task Jeopardy Execution Point

Field Name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Assign Jeopardy (41)

Data Passed

This is a recommended asynchronous call, therefore no data should be returned from the extension Java class.

Table A–6 shows the data passed to the extension Java class.

Table A–6 Assign Task Jeopardy Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task type	taskType
Task number	taskId
Work Queue ID	workQueueId

UI Invocation

From the Work Queue window, select a task, right-click, and select **Jeopardy Status**. This opens the Task Jeopardy Codes window where jeopardy codes can be added, changed, or deleted. Click **OK** or the **Apply** button to trigger the Assign Task Jeopardy execution point.

XML API Invocation

The XML API method through which the Java class extension is invoked is:

OrderManagement > addTaskJeopardyRequest

CORBA API Invocation

The CORBA API methods through which the Java class extension is invoked are:

- Work Management > addTaskJeopardy
- Work Management > deleteTaskJeopardy
- Work Management > updateTaskJeopardy

Change Task Completion Date

MetaSolv Solution provides the ability to change a task due date. This execution point enables you to extend logic that executes when a task due date is changed.

Business Example

You entered an order, assigned a provisioning plan, and then supplemented the order to change the due date. The extension logic executes and sends an email notification to the appropriate person regarding the task due date change.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–7 when searching for an execution point to associate with the extension.

Table A–7 Change Task Completion Date Execution Point

Field Name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Change Completion Date (44)

Data Passed

This is a recommended asynchronous call, therefore no data should be returned from the extension Java class.

Table A–8 shows the data passed to the extension Java class.

Table A–8 Change Task Completion Date Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task Type	taskType
Task number	taskId
Work Queue ID	workQueueId
New revised completion date	newRevisedCompletionDate

UI Invocation

From the Work Queue window, select a task, right-click, and select **Service Request Tasks**. This opens the **Task List** tab of the Tasks window, where task due dates can be changed. Click **OK** or the **Apply** button to trigger the Change Task Completion Date execution point, which only executes if any task due dates were actually changed.

Additionally, you can supplement an order to bring up the Tasks window where task due dates can be changed.

XML API Invocation

The XML API method through which the Java class extension is invoked is:

Order Management > processSuppOrder

CORBA API Invocation

The Change Task Completion Date execution point is not triggered by the CORBA API.

Complete Task

MetaSolv Solution provides the ability to complete a task assigned to an order. This execution point enables you to extend logic that executes when a task completes, either manually from the UI or automatically from the System Task Server.

Business Example

You entered a PSR order and assigned a provisioning plan comprised of three tasks. The second task is defined as an execution point and associated to an extension. When the task completes, the extension logic executes and sends an email notification to the appropriate person regarding the task completion.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shows in Table A–9 when searching for an execution point to associate with the extension.

Table A–9 Complete Task Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001) or [specific task type] (dynamic)
Process Point	Task Maintenance (101)
Action Type	Complete (53)

Data Passed

This is required to be a synchronous call because existing logic must know if the extension logic executed successfully before continuing. While no task related data needs to be returned from the extension Java class, it must indicate success or failure.

Table A–10 shows the data passed to the extension Java class.

Table A–10 Complete Task Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task number	taskId

UI Invocation

From the Work Queue window within Work Management, select a task, right-click and select **Complete**. The extension logic executes after the task completion logic runs successfully, but before the commit. If the task completion logic fails, the extension logic does not execute. If the extension logic fails, the task does not complete and a rollback occurs.

XML API Invocation

The XML API method through which the Java class extension is invoked is:

Order Management > updateOrderManagementRequest

The updateOrderManagementRequest method defines a choice of input structures. To complete a task, use the input structure CompleteTaskProcedureValue.

CORBA API Invocation

The CORBA API methods through which the Java class extension is invoked are:

- Work Management > completeTask
- Work Management > completeTaskOnDate

Additional Invocations

This execution point can also be triggered by the System Task Server for cases where the task is defined as a System Task.

For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Generate Tasks

MetaSolv Solution provides the ability to generate tasks for an order. This execution point enables you to extend logic that executes after tasks are generated. Order management also provides the ability to split a PSR order, a process that also generates tasks for the new order created as a result of the split. This execution point also enables you to extend logic that executes after tasks are generated as a result of a split.

Business Example

You entered a PSR order and assigned a provisioning plan. Two of the service items on the order are delayed, and you split the order so the remaining items can be completed. When the order is split, tasks are generated for the new order that is created as a result of the split. The extension logic executes and sends an email notification to the appropriate person regarding the tasks being generated due to the split. Both the original order and the split order information is made available to the extension.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–11 when searching for an execution point to associate with the extension.

Table A–11 Generate Tasks Execution Point

Field Name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Generation (1)
Action Type	Generate (32)

Data Passed

This is a recommended asynchronous call, therefore no data should be returned from the extension Java class.

Table A–12 shows the data passed to the extension Java class.

Table A–12 Generate Tasks Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Split document number	splitDocumentNumber

The document is always passed to the extension Java class, but the split document number may or may not be passed, depending on what triggered the task generation. If a split order triggered the task generation, then the split document number, in addition to the document number, is passed to the extension Java class.

UI Invocation

From the Product Service Request window within Order Management, select **Options** from the menu bar, and then select **Task Generation Maintenance**. This opens the **Plan Selection** tab of the Tasks window. Select a provisioning plan from the list. Click the **Task List** tab, and select work queues for each task. Click **OK** or the **Apply** button to trigger the Generate Tasks execution point, which happens immediately following the creation of the tasks for the order.

XML API Invocation

The Generate Tasks execution point is not triggered by the XML API.

CORBA API Invocation

The Generate Tasks execution point is not triggered by the CORBA API.

Late Task

MetaSolv Solution considers a task late when the current GMT date is greater than the revised completion date on the task. This execution point enables you to extend logic that executes when a task becomes late.

This execution point is triggered only once when the task is determined to be late. It may be triggered again if the revised completion date is updated on the task. There are new fields on the Task table that indicate if an extension has been invoked.

At the point you define this extension, there could be a large number of late tasks already existing in the database. Invoking this extension for each of these tasks can affect system performance. You can manage the system load by modifying the setup values in the **integration.xml** file. The maxThreads should always be set to 1. However, the queueMaxCapacity can be lowered and the dbPollingInterval increased to allow breaks in the system processing so the late task extensions can be invoked. The following excerpt from the **integeration.xml** file illustrates this concept:

```
<LateTaskExtensionEvent event_name="LateTaskExtensionEvent">
<maxThreads>1</maxThreads>
<queueMaxCapacity>100</queueMaxCapacity>
<dbPollingInterval>5</dbPollingInterval>
</LateTaskExtensionEvent>
```

Business Example

You entered an order and assigned a provisioning plan. One of the tasks becomes late. The extension logic executes and sends an email notification to the appropriate person regarding the late task.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–13 when searching for an execution point to associate with the extension.

Table A–13 Late Task Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Late (46)

Data Passed

This is required to be a synchronous call because existing logic must know if the extension logic executed successfully before continuing. While no task related data needs to be returned from the extension Java class, it must indicate success or failure.

Table A–14 the data passed to the extension Java class.

Table A–14 Late Task Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task number or identifier	taskId
Task type	taskType
Work queue identifier	workQueueId
Organization for employee	organizationName
Employee name	employeeName
Error text for failure	errorText

UI Invocation

The Late Task execution point is not triggered by the UI.

XML API Invocation

The Late Task execution point is not triggered by the XML API.

CORBA API Invocation

The Late Task execution point is not triggered by the CORBA API.

Additional Invocations

This execution point is triggered by the Integration Server.

For this to occur, the Integration Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Potentially Late Task

MetaSolv Solution provides the ability to define the potentially late window of time for each task type. MetaSolv Solution considers a task potentially late when the revised completion date on the task, minus the time defined as the potentially late window, is

less than the current GMT date. This comparison takes into account the calendar that is set up by the organization. The calendar relationship is determined from the task's work queue, which is then associated with an employee, and each employee is associated with organization. For an organization, the calendar may reflect non-work days, which would be considered in determining if a task was potentially late.

This execution point enables you to extend logic that executes when a task becomes potentially late. Note the following regarding the Potentially Late Task execution point:

- This execution point is triggered only once when the task is determined to be potentially late. It may be triggered again if the revised completion date is updated on the task. There are new fields on the Task table that indicate if an extension has been invoked.
- If the potentially late server event is disabled during the window of time for a potentially late task, and the task passes from a potentially late task to a late task, the Potentially Late Task execution point trigger does not execute. When the server event is enabled, and the task is now late, then the Late Task execution point is triggered.

Business Example

You entered an order and assigned a provisioning plan with a task that defines a potentially late window. The task becomes potentially late. The extension logic executes and sends an email notification to the appropriate person regarding the potentially late task.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–15 when searching for an execution point to associate with the extension.

Table A–15 Potentially Late Task Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Potentially Late (47)

Data Passed

This is required to be a synchronous call because existing logic must know if the extension logic executed successfully before continuing. While no task related data needs to be returned from the extension Java class, it must indicate success or failure.

Table A–16 shows the data passed to the extension Java class.

Table A–16 Potentially Late Task Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task number or identifier	taskId
Task type	taskType
Work queue identifier	workQueueId

Table A–16 (Cont.) Potentially Late Task Name/Value Pair Input Data

Data Description	Data Name
Organization for employee	organizationName
Employee name	employeeName
Error text for failure	errorText

UI Invocation

The Potentially Late Task execution point is not triggered by the UI.

XML API Invocation

The Potentially Late Task execution point is not triggered by the XML API.

CORBA API Invocation

The Potentially Late Task execution point is not triggered by the CORBA API.

Additional Invocations

This execution point is triggered by the Integration Server.

For this to occur, the Integration Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Provisioning Plan Default

MetaSolv Solution provides the ability to assign a provisioning plan to an order. This execution point enables you to extend logic to default the appropriate provisioning plan to an order, rather than having to specify a particular provisioning plan.

Business Example

You built provisioning plans and assigned default work queues to the tasks in every plan. An extension could be added for defaulting a provisioning plan, allowing you to put logic around the default. For example, you can reduce the number of errors that are made in assigning a provisioning plan to an order by basing the assignment on specific data. Additionally, when the extension logic executes, you can send an email notification to the appropriate person regarding the defaulted provisioning plan.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–17 when searching for an execution point to associate with the extension.

Table A–17 Provision Plan Default Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Generation (1)
Action Type	Provision Plan Default (52)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–18 shows the data passed to the extension Java class.

Table A–18 Provisioning Plan Default Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Organization	organization
Jurisdiction	jurisdiction
Service type group	serviceTypeGroup
Order status	status

Table A–19 shows the data returned by the extension Java class.

Table A–19 Provisioning Plan Default Name/Value Pair Return Data

Data Description	Data Name
Provision plan ID	provisionPlanId

UI Invocation

From a Service Request window (ISR, PSR, etc.) within Order Management, select **Options** from the menu bar, and then select **Task Generation Maintenance**. This opens the **Plan Selection** tab of the Tasks window. The Provisioning Plan Default execution point is triggered just prior to the Tasks window being displayed. If custom logic is executed, and a valid provisioning plan is returned from the extension, that plan is automatically populated in the drop-down list and the display proceeds to the **Task Gantt** tab. The user may return to the **Plan Selection** tab to change the selected plan.

XML API Invocation

The Provisioning Plan Default execution point is not triggered by the XML API.

CORBA API Invocation

The Provisioning Plan Default execution point is not triggered by the CORBA API.

Reject Task

MetaSolv Solution provides the ability to reject a task. This execution point enables you to extend logic that executes when a specified task is rejected.

Business Example

You assigned a provisioning plan and, from your Work Queue, reject a task. The extension logic executes and sends an email notification to the appropriate person regarding the rejected task.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–20 when searching for an execution point to associate with the extension.

Table A–20 Reject Task Execution Point

Field Name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Reject (42)

Data Passed

This is a recommended asynchronous call, therefore no data should be returned from the extension Java class.

Table A–21 shows the data passed to the extension Java class.

Table A–21 Reject Task Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task type	taskType
Task number	taskId
Work Queue ID	workQueueId
Previous task status	priorTaskStatus
Reject reason	note

UI Invocation

From the Work Queue window, select a task, right-click, and select **Reject Task**. This opens the Reject Task window where you select, from a list of predecessor tasks, the task to be set back to Ready status. All tasks between the initial selection and this second selection (tasks in that provisioning plan for that order) are set back to Pending status. Click **OK** to trigger the Reject Task execution point. A list of affected tasks is sent to the extension.

XML API Invocation

The Reject Task execution point is not triggered by the XML API.

CORBA API Invocation

The CORBA API method through which the Java class extension is invoked is:

Work Management > rejectTask

System Task Failure

MetaSolv Solution provides the ability to define a task as a system task. This indicates that the task's completion logic automatically runs on the System Task Server when the task becomes Ready or when the task start date is reached. However, the system task's completion logic may fail. When a system task cannot be completed, the System Task Server rolls back the transaction, transfers the task to the Exception queue, and logs information to the Server Log table. The server log entries associated with a task can be viewed from the work queue by selecting the task, and then clicking the **Server Log** tab. Tasks are not completed if a gateway event is in error or if a why-missed code cannot be defaulted.

This execution point enables you to extend logic that executes when a system task fails to complete. This execution point is asynchronous so that the continuation of the System Task Server process is not jeopardized.

Business Example

You entered an order and assigned a provisioning plan with a system task. The task becomes Ready, the System Task Server picks up the task and attempts to complete it, but fails. The extension logic executes and sends an email notification to the appropriate person regarding the failed system task.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–22 when searching for an execution point to associate with the extension.

Table A–22 System Task Failure Execution Point

Field Name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	System Task Failure (45)

Data Passed

This is a recommended asynchronous call, therefore no data should be returned from the extension Java class.

Table A–23 shows the data passed to the extension Java class.

Table A–23 System Task Failure Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Task number or identifier	taskId
Task type	taskType
Work queue identifier	workQueueId
Error text for failure	errorText or note

UI Invocation

The System Task Failure execution point is not triggered by the UI.

XML API Invocation

The System Task Failure execution point is not triggered by the XML API.

CORBA API Invocation

The System Task Failure execution point is not triggered by the CORBA API.

Additional Invocations

- This execution point is triggered by the System Task Server.

For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

- This execution point is triggered by the Background Processor.

  For this to occur, the Background Processor must be running. See "Invoking an Extension" for specific information on how to run the Background Processor.

Gateway Event Failure

MetaSolv Solution provides the ability to change the status of a gateway event to Error. This execution point enables you to extend logic that executes after the gateway event status change has completed. This execution point is asynchronous so the continuation of the Gateway Event Server process is not jeopardized.

Business Example

You entered an order and assigned a provisioning plan with a task that has an auto-complete gateway event associated with it. When the task becomes Ready, the gateway event automatically fires, but fails. The gateway event status is set to Error, and the extension logic executes and sends an email notification to the appropriate person regarding the failed gateway event.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–24 when searching for an execution point to associate with the extension.

Table A–24 Gateway Event Failure Execution Point

Field Name	Option
Execution Mode	Asynchronous
Building Block	All Gateway Events (1002)
Process Point	GW Event Maintenance (102)
Action Type	GW Event Failed (51)

Data Passed

This is required to be an asynchronous call. Data cannot be returned from the extension Java class.

The data passed to the Gateway Event Failure extension depends on the gateway event type. There are four types of gateway events listed below. Table A–25 shows all the data inputs, but these vary based on gateway event type.

- Service Request or Order Type
- Service Item or Item Level Type
- Equipment Type
- Design Type

Table A–25 shows the data passed to the extension Java class.

Table A–25 Gateway Event Failure Data Value Input by Event Type

data value	Order Type	Item Level Type	Equipment Type	Design Type
documentNumber	yes	yes	no	no
taskId	yes	yes	no	no
taskType	yes	yes	no	no
gatewayName	yes	yes	yes	yes
gatewayEventType	yes	yes	yes	yes
gatewayEventId	yes	yes	yes	yes
gatewayEventName	yes	yes	yes	yes
gatewayEventVersion	yes	yes	yes	yes
serviceItemId	yes	yes	no	no
errorText	yes, if exists	yes, if exists	yes, if exists	yes, if exists

UI Invocation

The Gateway Event Failure execution point is not triggered by the UI.

XML API Invocation

The XML API method through which the Java class extension is invoked is:

Order Management > updateOrderManagementRequest

> **Note:** The updateOrderManagementRequest method defines several choices of input structures. The invocation is applicable only when the input structure chosen is TaskGWEventValue.

CORBA API Invocation

The CORBA API method through which the Java class extension is invoked is:

Work Management > updateGWEvent

Additional Invocations

- This execution point is triggered by the Gateway Event Server.

  For this to occur, the Gateway Event Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

- This execution point is triggered by the Integration Server.

  For this to occur, the Integration Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Email CLR/DLR/TCO

MetaSolv Solution provides the ability to perform a process from the connection print window. This execution point enables you to extend logic that activates upon clicking of the **OK** button on the print window after closing the email recipient's window. To open the email recipient's window, in the Preference window, set the **Enable HTML Email** option to true and select the **Email** option in the Print window. The **Enable**

HTML Email preference is located under Preferences > Inventory Management > Connection Design.

You can modify the sample code to fit the email protocol used at a customer site. The sample extension uses the ByteArrayDataSource method in the **mailapi.jar** file. The sample email extension exists in the **SendEmailAttachment** folder.

If required, download the **mailapi.jar** file from the Oracle Web site. After downloading, you can include the JAR in the CLASSPATH of the appserver environment.

Business Example

You can use this custom extension in several ways. One possible use of this extension is to retrieve the saved HTML files from the database and email the files to the appropriate recipients. Other possibilities include displaying the HTML files on an Intranet or providing access to the HTML files from other applications. The HTML attachment exists as a CLOB in the Email_Job_Attachment table.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–26 while searching for an execution point to associate with the extension.

Table A–26 Email CLR/DLR/TCO Execution Point

Field Name	Option
Execution Mode	Asynchronous
Building Block	All Connections (409)
Process Point	Print (140)
Action Type	Email(56)

Data Passed

As this is an asynchronous call, therefore extension Java class does not return data.

Table A–27 shows the data passed to the extension Java class.

Table A–27 Email CLRD/DLR/TCO Name/Value Pair Input Data

Data Description	Data Name
JobId	jobid

UI Invocation

From the Connection print window, select the **Email** check box and click **OK**. The execution occurs on the Print window but the logic waits until the user clicks **OK** on the Recipient window and the Recipient window closes. If the user clicks **Cancel** on the Recipients window, the extension does not execute.

XML API Invocation

The Email CLR/DLR/TCO execution point is not triggered by the XML API.

CORBA API Invocation

The Email CLR/DLR/TCO execution point is not triggered by the CORBA API.

Additional invocations

This execution point is not triggered anywhere else.

Select Port Address

MetaSolv Solution provides the ability to automatically design physical connections through the PCONDES task. This execution point enables you to extend logic that is triggered when the PCONDES task is executed, either manually from the UI or automatically from the System Task Server. The extension logic enables you to select the appropriate port address to use in the physical design of the connection. It executes prior to the existing PCONDES auto-provisioning logic. If a port address is successfully selected by the extension logic, the existing PCONDES auto-provisioning logic is bypassed. If a port address is not selected by the extension logic, the existing PCONDES auto-provisioning logic still executes.

Business Example

You enter a PSR order and assign a provisioning plan that defines the PCONDES task as a system task. The PCONDES task is used to automatically design physical connections. When the status of the PCONDES task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate port address is selected for the design of the physical connection.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–28 when searching for an execution point to associate with the extension.

Table A–28 Select Port Address Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	PCONDES Maintenance(103)
Action Type	Select Port Address Element(55)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–29 shows the data that is passed to the extension Java class.

Table A–29 Select Port Address Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Circuit design ID	circuitDesignId
Service item ID	servItemId
End user location ID	endUserLocationId
Rate code	rateCode
Network system component ID	nsCompId

Table A–29 (Cont.) Select Port Address Name/Value Pair Input Data

Data Description	Data Name
Network system ID	nsId
Network system component key Array	nsCompKey (String Array comprised of nsCompId and nsId)

Pass nsCompId and nsId, or pass an Array of nsCompKeys; do not pass both sets of data. If the input data is comprised of the Array of nsCompKeys, custom logic can be written to select which component id is used. Having this option of input data allows for you to customize your extension code to account for things like load balancing between different elements. For example, if there are three valid elements from which to choose, custom code can select the element which has the most or least capacity available, depending on your specific business requirements.

Table A–30 shows the data that is returned by the extension Java class.

Table A–30 Select Port Address Name/Value Pair Return Data

Data Description	Data Name
Equipment ID	equipmentId
Port Address Sequence	portAddrSeq

UI Invocation

From the Work Queue window within Work Management, select a PCONDES task, right-click and select **Auto Provision**. The extension logic executes prior to the existing PCONDES auto provision logic. If a port address is successfully selected by the extension logic, the existing PCONDES auto provision logic is bypassed. However, if a port address is not selected by the extension logic, the existing PCONDES auto provision logic still executes.

XML API Invocation

The Select Port Address execution point is not triggered by the XML API.

CORBA API Invocation

The Select Port Address execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the PCONDES task is defined as a System Task.

For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Select Component or Element for Physical Connection

MetaSolv Solution provides the ability to automatically design physical connections through the PCONDES task. This execution point enables you to extend logic that is triggered when the PCONDES task is executed, either manually from the UI or automatically from the System Task Server. The extension logic enables you to select the appropriate component or element to use in the physical design of the connection. It executes prior to the existing PCONDES auto-provisioning logic. If a component or element is successfully selected by the extension logic, the existing PCONDES

auto-provisioning logic is bypassed. If a component or element is not selected by the extension logic, the existing PCONDES auto-provisioning logic still executes.

Business Example

You enter a PSR order and assign a provisioning plan that defines the PCONDES task as a system task. The PCONDES task is used to automatically design physical connections. When the status of the PCONDES task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate component or element is selected for the design of the physical connection.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–31 when searching for an execution point to associate with the extension.

Table A–31 Select Component or Element for Physical Connection Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	PCONDES Maintenance(103)
Action Type	Select Component or Element(54)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–32 shows the data that is passed to the extension Java class.

Table A–32 Select Component or Element for Physical Connection Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Circuit design ID	circuitDesignId
Service item ID	servItemId
End user location ID	endUserLocationId

Table A–33 shows the data that is returned by the extension Java class.

Table A–33 Select Component or Element for Physical Connection Name/Value Pair Return Data

Data Description	Data Name
Network system component key Array	nsCompKey (String Array comprised of nsCompId and nsId)

UI Invocation

From the Work Queue window within Work Management, select a PCONDES task, right-click and select **Auto Provision**. The extension logic executes prior to the existing

PCONDES auto provision logic. If a component or element is successfully selected by the extension logic, the existing PCONDES auto provision logic is bypassed. However, if a component or element is not selected by the extension logic, the existing PCONDES auto provision logic still executes.

XML API Invocation

The Select Component or Element for Physical Connection execution point is not triggered by the XML API.

CORBA API Invocation

The Select Component or Element for Physical Connection execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the PCONDES task is defined as a System Task.

For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Select Component or Element for Virtual Connection

MetaSolv Solution provides the ability to automatically design virtual connections through the VCONDES task. This execution point enables you to extend logic that is triggered when the VCONDES task is executed, either manually from the UI or automatically from the System Task Server. The extension logic enables you to select the appropriate component or element to use in the virtual design of the connection. It executes prior to the existing VCONDES auto-provisioning logic. If a component or element is successfully selected by the extension logic, the existing VCONDES auto-provisioning logic is bypassed. If a component or element is not selected by the extension logic, the existing VCONDES auto-provisioning logic still executes.

Business Example

You enter a PSR order and assign a provisioning plan that defines the VCONDES task as a system task. The VCONDES task is used to automatically design virtual connections. When the status of the VCONDES task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate component or element is selected for the design of the virtual connection.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–34 when searching for an execution point to associate with the extension.

Table A–34 Select Component or Element for Virtual Connection Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	VCONDES Maintenance (105)
Action Type	Select Component or Element (54)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–35 shows the data that is passed to the extension Java class.

Table A–35 Select Component or Element for Virtual Connection Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Circuit design ID	circuitDesignId
Service item ID	servItemId
Connection Spec	nstCompTypeConId
Network Configuration Type	nstConfigTypeId
Component Type	networkComponentType

Table A–36 shows the data that is returned by the extension Java class.

Table A–36 Select Component or Element for Virtual Connection Name/Value Pair Return Data

Data Description	Data Name
Network system component key Array	nsCompKey (String Array comprised of nsCompId and nsId)

Returned data validation

The data returned by the VCONDES Maintenance - Select Component custom extension must adhere to certain rules. All components (NS_ID/NS_COMP_ID combination) must pass the following validation logic:

- The NS_COMP_ID must exist in the database.
- The component type of the returned NS_COMP_ID must match the networkComponentType input parameter.
- The NS_ID must exist in the database.
- The network configuration type of the returned NS_ID must match the nstConfigTypeId input parameter.

UI Invocation

From the Work Queue window within Work Management, open the Service Request Virtual Connections window by double-clicking a VCONDES task and then select **Auto Provision** from the **Options** menu. The extension logic executes prior to the existing VCONDES auto provision logic. If a component or element is successfully selected by the extension logic, the existing VCONDES auto provision logic is bypassed. However, if a component or element is not selected by the extension logic, the existing VCONDES auto provision logic still executes.

XML API Invocation

The Select Component or Element for Virtual Connection execution point is not triggered by the XML API.

CORBA API Invocation

The Select Component or Element for Virtual Connection execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the VCONDES task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Select Network System

MetaSolv Solution provides the ability to automatically design physical connections through the NETDESIGN task. This execution point enables you to extend logic that is triggered when the NETDESIGN task is executed automatically from the System Task Server. The extension logic enables you to select the appropriate network system to use in the physical design of the connection. It executes prior to the NETDESIGN task. If a network system is successfully selected by the extension logic, the existing NETDESIGN auto-provisioning logic is bypassed. If a network system is not selected by the extension logic, the existing NETDESIGN auto-provisioning logic still executes.

Business Example

You enter a PSR order and assign a provisioning plan that defines the NETDESIGN task as a system task. The NETDESIGN task is used to automatically design physical connections. When the status of the NETDESIGN task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate network system is selected for the design of the physical connection.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–37 when searching for an execution point to associate with the extension.

Table A–37 Select Network System Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Network Systems (410)
Process Point	Network System Design (107)
Action Type	Select Network System (60)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–38 shows the data that is passed to the extension Java class.

Table A–38 Select Network System Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber

Table A–39 shows the data that is returned by the extension Java class.

Table A–39 Select Network System Name/Value Pair Return Data

Data Description	Data Name
Activity code	activityCd
ArrayList of CaContainer objects, which contains CA names and values	CaList
Short name of network system	nsNmShort
Long name of network system	nsNmLong
Description	desc100
Network system ID	nsId
Hard/soft code	hardSoftCdExtension
Network system customer system ID	nsCustomerSysId
Network system provider system ID	nsProviderSysId
template name	templateName

Returned data validation

The data returned by the NETDESIGN Maintenance - Select Network System custom extension must adhere to certain rules. Network System details returned by the extension must pass the following validation logic:

- SHORT_NAME is mandatory, and the length of the value should be less than 20 characters.
- ACTIVITY_IND must be either "N" (new) or "C"(change).
- NS_ID must exist in the database.
- STATUS must be "Pending" or "Inservice"
- HARD_SOFT_ASSIGN_CD must be "soft" or "hard" or "none".
- NS_TEMPLATE_NAME is mandatory and must exist in the database.
- DESC_100 must be less than 100 characters.
- LONG_NAME must be less than 50 characters.
- CUSTOMER_SYS_ID and PROVIDER_SYS_ID accepts a maximum of 20 characters.
- Customer attribute (CA) Name must exist in the database.

UI Invocation

UI invocation of the Select Network System execution point is not available. While the NETDESIGN task can be defined as a manual task and accessed from the Work Queue window within Work Management, if accessed in this manner, the execution point is not invoked.

XML API Invocation

The Select Network System execution point is not triggered by the XML API.

CORBA API Invocation

The Select Network System execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the NETDESIGN task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Select Customer Edge Component

MetaSolv Solution provides the ability to automatically design physical connections through the NETDESIGN task. This execution point enables you to extend logic that is triggered when the NETDESIGN task is executed automatically from the System Task Server. The extension logic enables you to select the customer edge component to use in the physical design of the connection. It executes prior to the existing NETDESIGN auto-provisioning logic. If a customer edge component is successfully selected by the extension logic, the existing NETDESIGN auto-provisioning logic is bypassed. If a customer edge component is not selected by the extension logic, the existing NETDESIGN auto-provisioning logic still executes.

Business Example

You enter a PSR order and assign a provisioning plan that defines the NETDESIGN task as a system task. The NETDESIGN task is used to automatically design physical connections. When the status of the NETDESIGN task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate customer edge component is selected for the design of the physical connection.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–40 when searching for an execution point to associate with the extension.

Table A–40 Select Customer Edge Component Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Network Systems (410)
Process Point	Network System Design (107)
Action Type	Select Customer Edge Component (61)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–41 shows the data that is passed to the extension Java class.

Table A–41 Select Customer Edge Component Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber

Table A–42 shows the data that is returned by the extension Java class.

Table A–42 Select Customer Edge Component Name/Value Pair Return Data

Data Description	Data Name
Activity code	activityCd
ArrayList of CaContainer objects, which define CA names and values	CaList
Customer Edge name	nsCompName
Customer edge location name	locationName
Customer edge type	nsCompType
Network component ID	nsCompIdThruExtension
Customer edge number, which is used along with nsCompName to uniquely differentiate each customer edge	ceNumberThruExt
Network system component network element ID	nsCompNetworkElementId

Returned data validation

The data returned by the NETDESIGN Maintenance - Select Customer Edge Component custom extension must adhere to certain rules. All components must pass the following validation logic:

- ACTIVITY_IND must be either "N" (new) or "C" (change).
- CE_NAME is mandatory must be unique.
- CE_LOCATION_NAME must exist in the database.
- CE_TYPE must exist in the database.
- Customer attribute (CA) Name must exist in the database.
- COMP_ID must exist in the database.

UI Invocation

UI invocation of the Select Customer Edge Component execution point is not available. While the NETDESIGN task can be defined as a manual task and accessed from the Work Queue window within Work Management, if accessed in this manner, the execution point is not invoked.

XML API Invocation

The Select Customer Edge Component execution point is not triggered by the XML API.

CORBA API Invocation

The Select Customer Edge Component execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the NETDESIGN task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Select End Component For Physical Connection

MetaSolv Solution provides the ability to automatically design physical connections through the NETDESIGN task. This execution point enables you to extend logic that is triggered when the NETDESIGN task is executed automatically from the System Task Server. The extension logic enables you to select an end component for the physical connection to use in the physical design of the connection. It executes prior to the existing NETDESIGN auto-provisioning logic. If an end component for the physical connection is successfully selected by the extension logic, the existing NETDESIGN auto-provisioning logic is bypassed. If an end component for the physical connection is not selected by the extension logic, the existing NETDESIGN auto-provisioning logic still executes.

Business Example

You enter a PSR order and assign a provisioning plan that defines the NETDESIGN task as a system task. The NETDESIGN task is used to automatically design physical connections. When the status of the NETDESIGN task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate end component for the physical connection is selected for the design of the physical connection.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–43 when searching for an execution point to associate with the extension.

Table A–43 Select End Component For Physical Connection Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Network Systems (410)
Process Point	Network System Design (107)
Action Type	Select End Component For Physical Connection (62)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–44 shows the data that is passed to the extension Java class.

Table A–44 Select End Component For Physical Connection Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber

Table A–45 shows the data that is returned by the extension Java class.

Table A–45 Select End Component For Physical Connection Name/Value Pair Return Data

Data Description	Data Name
String object that contains the connection identifier (ecckt)	HashMap key
End points for connection defines the following details:	HashMap value: EndPointsForConnection Object
Activity indicator	activityInd
One end of the connection	customEdgeName
Other end of the connection	providerEdgeName
Used along with customer edge name to uniquely differentiate CE	customerEdgeNumber
Used along with provider edge name to uniquely differentiate PE	providerEdgeNumber
Network system name in which PE is present	providerEdgeNetwork SystemName
Network system name which needs to be embedded as part of the VPN network	providerEdgeParent NetworkName
Connection identifier / name	conEcckt

Returned data validation

The data returned by the NETDESIGN Maintenance - Select End Component For Physical Connection custom extension must adhere to certain rules. End components of each connection must pass the following validation logic:

- ACTIVITY_IND must be "N" or "C".
- CE_NAME must be the same as what is returned from the Select Customer Edge Component extension.
- PE_NAME must exist in the database.
- PE_NETWORK_NAME must exist in the database and component with PE_NAME must be part of this network.
- CONNECTION_ECCKT must exist in the database and it must be part of the order given in the input parameter.

UI Invocation

UI invocation of the Select End Component For Physical Connection execution point is not available. While the NETDESIGN task can be defined as a manual task and accessed from the Work Queue window within Work Management, if accessed in this manner, the execution point is not invoked.

XML API Invocation

The Select End Component For Physical Connection execution point is not triggered by the XML API.

CORBA API Invocation

The Select End Component For Physical Connection execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the NETDESIGN task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Select Equipment For CE

MetaSolv Solution provides the ability to automatically design physical connections through the NETDESIGN task. This execution point enables you to extend logic that is triggered when the NETDESIGN task is executed automatically from the System Task Server. The extension logic enables you to select the equipment for the customer edge to use in the physical design of the connection. It executes prior to the existing NETDESIGN auto-provisioning logic. If equipment for the customer edge is successfully selected by the extension logic, the existing NETDESIGN auto-provisioning logic is bypassed. If equipment for the customer edge is not selected by the extension logic, the existing NETDESIGN auto-provisioning logic still executes.

Business Example

You enter a PSR order and assign a provisioning plan that defines the NETDESIGN task as a system task. The NETDESIGN task is used to automatically design physical connections. When the status of the NETDESIGN task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate equipment for the customer edge is selected for the design of the physical connection.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–46 when searching for an execution point to associate with the extension.

Table A–46 Select Equipment For CE Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Network Systems (410)
Process Point	Network System Design (107)
Action Type	Select Equipment For CE (63)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–47 shows the data that is passed to the extension Java class.

Table A–47 Select Equipment For CE Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber

Table A–48 shows the data that is returned by the extension Java class.

Table A–48 Select Equipment For CE Name/Value Pair Return Data

Data Description	Data Name
CE name that the equipment needs to be associated with	customerEdgeName
Used along with customer edge name to uniquely differentiate CE	customerEdgeNumber
Equipment ID which must be associated with the CE	equipIdThruExtension

Returned data validation

The data returned by the NETDESIGN Maintenance - Select Equipment For CE custom extension must adhere to certain rules. All components and equipment returned from the extension must pass the following validation logic:

- EQUIPMENT_ID must exist in the database.
- CUSTOMEREDGE_NAME must exist in the database, and must be same as that of CE returned from the Select Customer Edge Component extension.

UI Invocation

UI invocation of the Select Equipment For CE execution point is not available. While the NETDESIGN task can be defined as a manual task and accessed from the Work Queue window within Work Management, if accessed in this manner, the execution point is not invoked.

XML API Invocation

The Select Equipment For CE execution point is not triggered by the XML API.

CORBA API Invocation

The Select Equipment For CE execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the NETDESIGN task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

DS0/DS1 Automated Design

MetaSolv Solution provides the ability to automate the provisioning of Facility circuits with rate codes DS0 and DS1 through the AUTODSGN task. This execution point enables you to extend logic that is triggered when the AUTODSGN task is executed automatically from the System Task Server. The extension logic enables you to provide

the assignment information to use in the provisioning of the connection. If the assignment information is not provided in the extension, the default auto-provisioning logic executes. The default auto-provisioning logic makes an equipment port assignment at either end of the circuit and makes a "next-available" channel assignment to a parent circuit, which is coterminous with the circuit being auto-provisioned.

Business Example

You enter a PSR order and assign a provisioning plan that defines the AUTODSGN task as a system task. The AUTODSGN task is used to automatically provision the Facility circuits with rate codes DS0 and DS1.

When the status of the AUTODSGN task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the assignment information in the extension and the circuit information on the order, the appropriate DS0 and DS1 facility circuits are automatically provisioned. After the assignments are made, the extension logic would also create design issues using the appropriate information from the order. The status of the circuits will be changed to Record Issued.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–49 when searching for an execution point to associate with the extension.

Table A–49 DS0/DS1 Automated Design Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	Connection Design (108)
Action Type	DS0/DS1 Automated Design (70)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–50 shows the data that is passed to the extension Java class.

Table A–50 DS0/DS1 Automated Design Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Circuit Design ID	circuitDesignId

Table A–51 shows the data that is returned by the extension Java class.

Table A–51 DS0/DS1 Automated Design Name/Value Pair Return Data

Data Description	Data Name
A Java container object that holds the assignment information to use in the automated provisioning of the circuit.	provisioningInfoContainer

Returned data validation

The data returned by the DS0/DS1 Automated Design execution point must adhere to certain rules. See "DS0/DS1 Automated Design" in the "Running the Sample Code" section for detailed parameter-level validation information.

UI Invocation

UI invocation of the DS0/DS1 Automated Design execution point is not available. While the AUTODSGN task can be defined as a manual task to design the connections and accessed from the Work Queue window within Work Management, if accessed in this manner, the execution point is not invoked.

XML API Invocation

The DS0/DS1 Automated Design execution point is not triggered by the XML API.

CORBA API Invocation

The DS0/DS1 Automated Design execution point is not triggered by the CORBA API.

Additional Invocations

This execution point can also be triggered by the System Task Server for cases where the AUTODSGN task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Connection Id Automation

MetaSolv Solution provides the ability to automate the generation of Connection Id for the circuits created in PSR orders through the CKTID task. This execution point enables you to extend logic that is triggered when the CKTID task is executed automatically from the System Task Server. The extension logic enables you to provide the required information to be used in the Connection Id generation.

Business Example

You enter a PSR order and assign a provisioning plan that defines the CKTID task as a system task. The CKTID task is used to automatically generate the Connection Id for the appropriate products on the PSR order. When the status of the CKTID task becomes Ready, the System Task Server processes the task. The extension logic executes and based on the information in the extension and the information on the order, the appropriate Connection Ids are generated automatically.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–52 when searching for an execution point to associate with the extension.

Table A–52 Connection Id Automation Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	Connection Design (108)
Action Type	Connection Id Automation (71)

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–53 shows the data that is passed to the extension Java class.

Table A–53 Connection Id Automation Name/Value Pair Input Data

Data Description	Data Name
Document number	documentNumber
Service Item ID	circuitDesignId

Table A–54 shows the data that is returned by the extension Java class.

Table A–54 Connection Id Automation Name/Value Pair Return Data

Data Description	Data Name
A Java container object that contains information that is used in the automated generation of the connection ID.	ConnectionIdAutomationData

Returned Data Validation

The data returned by the Connection Id Automation execution point must adhere to certain rules. See "Connection Id Automation" for detailed parameter-level validation information.

UI Invocation

UI invocation of the Connection Id Automation execution point is not available. While the CKTID task can be defined as a manual task to design the connections and accessed from the Work Queue window within Work Management, if accessed in this manner, the execution point is not invoked.

XML API Invocation

The Connection Id Automation execution point is not triggered by the XML API.

CORBA API Invocation

The Connection Id Automation execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the CKTID task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Select Dedicated Plant

MetaSolv Solution provides the ability to implement your custom logic to determine the appropriate dedicated plant assignment for the service being provisioned through the Automated Design (AUTODSGN) or Physical Connection Design (PCONDES) tasks, in scenarios where the core logic does not meet your business requirements.

For example, consider a scenario where a dedicated plant, DP1, has a priority of **1** and another dedicated plant, DP2, has its priority set as **2** on the **Plant Administration** tab of the Product Catalog window. If you determine that DP2 is more suitable for the service being provisioned, you can create and implement your custom logic to change the priority of DP2 to **1**. As a result, the core logic processes the information returned by the extension and selects DP2 when provisioning the service.

During service provisioning, the core logic first queries for a dedicated plant reservation for the order. If a reserved dedicated plant is found, the reservation is redeemed and the assignment is made. Otherwise, the core logic queries for all the dedicated plants at the service address on the order.

In addition, the core logic filters the following:

- Dedicated plants that are already assigned.
- Dedicated plants that have blocking condition codes on the cable pair or port address.
- Dedicated plants that have non-owned reservations on the cable pair or port address.

The core logic calls the custom extension logic only if multiple dedicated plants (both supported and unsupported) are available.

The custom extension logic can use the Item Spec ID or Spec Name values to determine which dedicated plant must be selected. For items that require manual design, this logic does not provide any output dedicated plant and displays an error message. In this case, the AUTODSGN and PCONDES tasks fail and an error message is logged. You can view this error on the **Server Logs** tab in the Work Queue Manager window.

After calling the custom extension, the core logic goes through all of the dedicated plants in the same order as they are populated within the `OutputDedicatedPlantList` parameter. The core logic then validates whether each dedicated plant is valid for the service being provisioned and assigns the service to the supporting dedicated plant based on its priority. If no supporting dedicated plants are returned by the custom extension, the AUTODSGN and PCONDES tasks fail and an error message is logged. You can view this error on the **Server Logs** tab in the Work Queue Manager window.

Business Example

You enter a PSR order and assign a provisioning plan that defines the AUTODSGN or PCONDES tasks as a system task. When the status of the AUTODSGN or PCONDES tasks becomes Ready, the System Task Server processes the tasks. The extension logic executes and based on the information in the extension and the information on the order, the extension logic validates whether each dedicated plant is valid for the service being provisioned and assigns the service to the supporting dedicated plant based on its priority.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–55 when searching for an execution point to associate with the extension.

Table A–55 Select Dedicated Plant Execution Point

Field Name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)

Table A–55 (Cont.) Select Dedicated Plant Execution Point

Field Name	Option
Process Point	PCONDES Maintenance (103), Connection Design (108)
Action Type	Select Dedicated Plant

Data Passed / Data Returned

This is a recommended synchronous call, therefore data should be returned from the extension Java class.

Table A–56 shows the data that is passed to the extension Java class.

Table A–56 Select Dedicated Plant Name/Value Pair Input Data

Data Description	Data Name
Order being processed.	documentNumber
Circuit design ID of the service being designed.	circuitDesignId
Service item being designed.	ServItemId
List of available dedicated plants. You must set the following attributes for this parameter: ■ ded_cc_grp_id: Unique key of the available dedicated plants. ■ last_modified_date: Last modified date of the available dedicated plants. ■ supports_product: **True** or **False**. ■ equipment_id: Equipment ID of the card used to build the available dedicated plants. ■ item_spec_id: Specification ID of the service being designed. ■ item_spec_nm: Specification name of the service being designed. ■ priority_seq: Priority sequence of the equipment set within the **Plant Administration** tab.	InputDedicatedPlantList
This parameter is not populated.	numberOfBondedPairs

Table A–57 shows the data that is returned by the extension Java class.

Table A–57 Select Dedicated Plant Name/Value Pair Return Data

Data Description	Data Name
List of dedicated plants to assign. This parameter contains the following attributes: ■ ded_cc_grp_id: Unique key of the available dedicated plants. ■ last_modified_date: Last modified date of the available dedicated plants. ■ supports_product: **True** or **False**. ■ equipment_id: Equipment ID of the card used to build the available dedicated plants. ■ item_spec_id: Equipment specification ID of the service being designed. ■ item_spec_nm: Equipment specification name of the service being designed. ■ priority_seq: Priority sequence of the equipment set on the **Plant Administration** tab.	OutputDedicatedPlantList
Error message to be sent to the server log.	ErrorMessage

Returned data validation

The data returned by the Select Dedicated Plant execution point must adhere to certain rules. See "DedicatedPlantSelection" for detailed parameter-level validation information.

UI Invocation

UI invocation of the Select Dedicated Plant execution point is not available.

XML API Invocation

The Select Dedicated Plant execution point is not triggered by the XML API.

CORBA API Invocation

The Select Dedicated Plant execution point is not triggered by the CORBA API.

Additional Invocations

This execution point can also be triggered by the System Task Server for cases where the AUTODSGN/PCONDES tasks are defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. See "Invoking an Extension" for specific configuration information.

Create/Update End User Location

This execution point enables you to implement your custom logic that validates the end user location address information and returns the following return codes that determine the behavior of the application based on the custom logic:

- **Success**: Creates or updates the end user location.
- **Failure**: Does not create or update the end user location.

- **Warning**: Provides you with options that enable you to do one of the following:
  - Create or update end user location address information with the data returned by your custom logic
  - Create or update end user location address information with the original data you entered in the MSS application

This execution point is triggered when you do any of the following:

- Create or update end user location address information in the PSR Ordering Dialog
- Create or update service locations on a PSR order
- Create or update end user location address information on the **PRILOC/SECLOC Info** tab of the Product Service Request window
- Create or update end user location address information in the End User Location Maintenance window

Custom Extension Success Scenario Example

The following list includes examples of situations when the custom extension logic may return the Success return code based on your custom logic:

- No matching address already exists
- The address information you enter is correct

When the extension returns the Success return code, the MSS application creates or updates the end user location.

Custom Extension Failure Scenario Example

The following list includes examples of situations when the custom extension logic may return the Failure return code based on your custom logic:

- A matching address already exists
- The address information you enter is incorrect
- No new end user location should be created with the data you enter

When the extension returns the Failure return code, the return text that you specified in your custom logic is displayed, and the MSS application does not create or update the end user location address information.

Custom Extension Warning Scenario Example

The following list includes examples of situations when the custom extension logic may return the Warning return code based on your custom logic:

- A partially matching address already exists
- The address information you enter is partially correct

When the extension returns the Warning return code, one of the following occurs:

- In the End User Location window and on the **PRILOC/SECLOC Info** tab of the Product Service Request window, the Custom Address Validation window is displayed, which displays those values in red that are different than the values you specified in the input. Do one of the following:
  - Click **OK** to create or update the end user location address information with the data returned by your custom logic

  - Click **Override** to create or update the end user location address information with the original data you entered in the MSS application

- At the top of the PSR Ordering Dialog, only those values are displayed in red that are different than the values you specified in the input. Do one of the following:
  - Click **Next** to create or update the end user location address information with the data returned by your custom logic
  - Click **Override** to create or update the end user location address information with the original data you entered in the MSS application

Note: You can limit user access to the **Override** option in the Security Permissions window.

Business Example

You enter a PSR order and click **Add Service Location** to add a new end user location. In the End User Location Maintenance window, enter the required information in the fields and click **OK**. The execution point is triggered and it returns Success, Failure, or Warning return codes that determine the behavior of the application based on your custom logic. See Table A–61 for more information about the MSS UI windows from where you can trigger the execution point.

Execution Point Definition

When defining the extension in MetaSolv Solution, choose the options shown in Table A–58 when searching for an execution point to associate with the extension.

Table A–58 Create/Update Execution Points

Field Name	Option
Execution Mode	Synchronous
Building Block	All End User Locations
Process Point	EUL Maintenance (123), PSR (124)
Action Type	Update (91), Create (92)

Data Passed / Data Returned

This is a synchronous call, therefore data should be returned from the extension Java class.

Table A–59 shows the data that is passed to the extension Java class.

Table A–59 Create/Update Name/Value Pair Input Data

Data Name	Data Type	Data Description
addressId	String	Unique identifier for an address. This is 0 or null for a new end user location that you want to create.

Table A–59 (Cont.) Create/Update Name/Value Pair Input Data

Data Name	Data Type	Data Description
addressComponents	String	Address components, such as House Number, Street Name, City Name, and so on. Specify this information as String in the following format: <ADDRESS> <sfname></sfname> <structureFormatComponents> <id></id> <name></name> <componentType></componentType> <value></value> </structureFormatComponents> </ADDRESS>
eulName	String	Name of the end user location.
countryId	String	ID of a country.
locationId	String	ID of the location. This is 0 or null for a new end user location that you want to create.
addressFormat	String	Address structure format for the address.

Table A–60 shows the data that is returned by the extension Java class.

Table A–60 Create/Update Name/Value Pair Return Data

Data Name	Data Type	Mandatory/ Optional	Data Description
returnCode	String	Mandatory	Return code that you want the extension logic to return: ■ **Success** ■ **Failure** ■ **Warning**
returnText	String	Mandatory if you specify the returnCode as **Failure**; otherwise, it is optional.	Text that you want the extension logic to return for the Success, Failure, and Warning return codes.

Table A–60 (Cont.) Create/Update Name/Value Pair Return Data

Data Name	Data Type	Mandatory/ Optional	Data Description

returnAddrComponents	String	The returnAddrComponents or returnAddrId is mandatory if you specify the returnCode as **Warning**.
		
		The returnAddrComponents is ignored if you specify returnAddrId.
			Returns only those address components that are different than the address components you specified in the input.
			
			Address components, such as House Number, Street Name, City Name, and so on, that are returned as String by the extension logic in the following format:
			
			`<ADDRESS>`
			`<sfname></sfname>`
			`<structureFormatComponents>`
			`<id></id>`
			`<name></name>`
			`<componentType></componentType>`
			`<value></value>`
			`</structureFormatComponents>`
			`</ADDRESS>`
			
			where:
			
			■ `sfname`: Indicates the address format of the end user location. Refer to the SF_STRUCT_FORMAT_NM column in the SF_COMP table.
			
			■ `id`: Indicates the structure format component ID for an address component in the end user location. Refer to the SF_COMP_ID column in the SF_COMP table.
			
			■ `name`: Indicates the structure format component name for an address component in the end user location. Refer to the COMP_NM column in the SF_COMP table.
			
			■ `componentType`: Indicates the structure format component type for an address component in the end user location. Refer to the COMP_TYPE column in the SF_COMP table.
			
			■ `value`: Indicates the value of the structure format component based on the `componentType`.
			
			For example:
			
			The Street Name address component of type **N** has the actual value, for example, ABC Street.
			
			The State Code address component of type **G** has the value ID, for example, 123, which indicates the value ID of the state and not the actual name of the state.

returnAddrId	String	The returnAddrComponents or returnAddrId is mandatory if you specify the returnCode as **Warning**.	Unique identifier returned for an address.

Returned data validation

The data returned by the Create and Update execution points must adhere to certain rules. See "Create/Update End User Location" for detailed parameter-level validation information.

UI Invocation

Table A–61 lists the MSS UI windows that trigger the execution point when you create or update end user locations.

Table A–61 MSS UI Windows That Trigger the Execution Point

Building Block Type	Building Block Name	Process Point	Action Type	MSS UI Windows
Address (411)	All End User Locations	EUL Maintenance (123)	Create (92)	**End User Location Maintenance Window** The execution point is triggered when you do the following: ▪ When creating a new end user location, in the End User Location Maintenance window, enter the required information in the fields and click **OK**. See "Creating or Updating an End User Location from the End User Location Maintenance Window" for more information.
Address (411)	All End User Locations	EUL Maintenance (123)	Update (91)	**End User Location Maintenance Window** The execution point is triggered when you do the following: ▪ When updating an end user location, in the End User Location Maintenance window, update the existing information in the fields and click **OK**. See "Creating or Updating an End User Location from the End User Location Maintenance Window" for more information.

Table A–61 (Cont.) MSS UI Windows That Trigger the Execution Point

Building Block Type	Building Block Name	Process Point	Action Type	MSS UI Windows
Address (411)	All End User Locations	PSR (124)	Create (92)	**End User Location Maintenance Window** The execution point is triggered when you do the following: ■ When creating a service location, in the End User Location Maintenance window, enter the required information in the fields and click **OK**. See "Creating or Updating a Service Location on a PSR order" for more information. **PSR Ordering Dialog** The execution point is triggered when you do one the following: ■ In the PSR Ordering Dialog, click the **add a new customer location** link and enter the required information in the fields, and then either click **Add Another** or click **Next**. See "Creating or Updating an End User Location from the PSR Ordering Dialog" for more information. **PRILOC/SECLOC Assignment Window** The execution point is triggered when you do the following: ■ In the PRILOC/SECLOC Assignment window, click the address icon. In the Address Maintenance window, enter the required information in the fields and click **OK**. See "Creating or Updating an End User Location on the PRILOC/SECLOC Info Tab on a PSR Order" for more information.

Table A–61 (Cont.) MSS UI Windows That Trigger the Execution Point

Building Block Type	Building Block Name	Process Point	Action Type	MSS UI Windows
Address (411)	All End User Locations	PSR (124)	Update (91)	**End User Location Maintenance Window** The execution point is triggered when you do the following: ■ When updating a service location, in the End User Location Maintenance window, update the existing information in the fields and click **OK**. See "Creating or Updating a Service Location on a PSR order" for more information. **PSR Ordering Dialog** The execution point is triggered when you do one the following: ■ In the PSR Ordering Dialog, click an existing location and update the existing information in the fields and click **Next**. See "Creating or Updating an End User Location from the PSR Ordering Dialog" for more information. **PRILOC/SECLOC Assignment Window** The execution point is triggered when you do the following: ■ In the PRILOC/SECLOC Assignment window, click the address icon. In the Address Maintenance window, update the existing information in the fields and click **OK**. See "Creating or Updating an End User Location on the PRILOC/SECLOC Info Tab on a PSR Order" for more information.

Creating or Updating an End User Location from the End User Location Maintenance Window

To create or update an end user location from the End User Location Maintenance window:

1. On the navigation bar, select **Application Setup**, click **Location and Geography Setup**, and then click **End User Locations**.

   The End User Location Search window is displayed.

2. Do one of the following:

   - To create a new end user location, click **Add New**.
   - To update an existing end user location, specify your search criteria and click **Search**, and then double-click the end user location.

     The End User Location Maintenance window is displayed.

3. Enter the required information in the fields and click **OK**.

   The execution point is triggered. The end user location address information is sent to the custom logic (extension Java class) and one of the following occurs.

- The extension returns the Success return code. In this case, the end user location is created or updated.
- The extension returns the Failure return code and displays the return text specified in the custom logic. In this case, the end user location is not created or updated.
- The extension returns the Warning return code. In this case, the Custom Address Validation window is displayed, which displays those values in red that are different than the values you specified in the input. Do one of the following:
  - Click **OK** to create or update the end user location address information with the data returned by your custom logic.
  - Click **Override** to create or update the end user location address information with the original data you entered in the MSS application.

Creating or Updating a Service Location on a PSR order

To create or update a service location on a PSR order:

1. Open a PSR order.
2. Under **Order Maintenance**, click **Services**.
3. Do one of the following:
   - To add a new service location, click **Add Service Location**.

     The End User Location Search window is displayed.
     - Click **New Location**.
   - To update an existing service location, right-click a service location and select **Update Service Location**.

     The End User Location Maintenance window is displayed.
4. Enter the required information in the fields and click **OK**.

   The execution point is triggered. The end user location address information is sent to the custom logic (extension Java class) and one of the following occurs:
   - The extension returns the Success return code. In this case, the end user location is created or updated.
   - The extension returns the Failure return code and displays the return text specified in the custom logic. In this case, the end user location is not created or updated.
   - The extension returns the Warning return code. In this case, the Custom Address Validation window is displayed, which displays those values in red that are different than the values you specified in the input. Do one of the following:
     - Click **OK** to create or update the end user location address information with the data returned by your custom logic.
     - Click **Override** to create or update the end user location address information with the original data you entered in the MSS application.

Creating or Updating an End User Location from the PSR Ordering Dialog

To create or update an end user location from the PSR Ordering Dialog:

1. Open a PSR order.

2. Under **Order Maintenance**, click **Services**.
3. Select a product from the hierarchy.
4. Under **Service Item Actions**, click the **Configure Product** link.

   The PSR Ordering Dialog is displayed.
5. In the Do you want to include any of these existing locations? window, do one of the following:
   - To add a new location, click the **add a new customer location** link and enter the required information in the fields and do one of the following:
     - Click **Add Another**
     - Click **Next**
   - To update an existing location, click an existing location and update the fields as required and click **Next**.

     The execution point is triggered.
6. The end user location address information is sent to the custom logic (extension Java class) and one of the following occurs:
   - The extension returns the Success return code. In this case, the end user location is created or updated.
   - The extension returns the Failure return code and displays the return text specified in the custom logic. In this case, the end user location is not created or updated.
   - The extension returns the Warning return code. In this case, at the top of the PSR Ordering Dialog, only those values are displayed in red that are different than the values you specified in the input. Do one of the following:
     - Click **OK** to create or update the end user location address information with the data returned by your custom logic.
     - Click **Override** to create or update the end user location address information with the original data you entered in the MSS application.

Creating or Updating an End User Location on the PRILOC/SECLOC Info Tab on a PSR Order

To create or update an end user location on the PRILOC/SECLOC Info tab on a PSR order:

1. Open a PSR order.
2. Under **Order Maintenance**, click **Services**.
3. Expand the circuit product node and select a circuit.
4. Click the **PRILOC/SECLOC Info** tab.
5. Under the **PRILOC** section, do one of the following:
   - To assign a new primary/secondary location, click the **Assign** link.
   - To edit an existing primary/secondary location, click the **Edit** link.

     The PRILOC/SECLOC Assignment window is displayed.
6. On the **PRILOC** tab, select the **End User** option and complete the required fields.
7. On the **SECLOC** tab, select the **End User** option and complete the required fields.

8. Click the address icon.

   The Address Maintenance window is displayed.

9. Complete the required fields and click **OK**.

   The execution point is triggered. The end user location address information is sent to the custom logic (extension Java class) and one of the following occurs:

   - The extension returns the Success return code. In this case, the end user location is created or updated.
   - The extension returns the Failure return code and displays the return text specified in the custom logic. In this case, the end user location is not created or updated.
   - The extension returns the Warning return code. In this case, the Custom Address Validation window is displayed, which displays those values in red that are different than the values you specified in the input. Do one of the following:
     - Click **OK** to create or update the end user location address information with the data returned by your custom logic.
     - Click **Override** to create or update the end user location address information with the original data you entered in the MSS application.

XML API Invocation

The Create and Update execution points are not triggered by the XML API.

CORBA API Invocation

The Create and Update execution points are not triggered by the CORBA API.

B

Extensions Sample Code

This appendix provides information about the extensions sample code that comes with your installation.

Using Sample Code as a Reference for Best Practices

This section provides information regarding best practices for writing Java classes to extend the Oracle Communications MetaSolv Solution application logic. The best practices are explained by referencing the provided sample code. The sample code demonstrates how to throw an exception, send an email notification, and call a CORBA API method from an extension class.

Exception Handling

The **mss_ext_samples.jar** file contains the class SampleExtensionException.java. This class provides sample code that throws an exception from an extension class. The result of an extension class throwing an exception is an entry in the **appserverlog.xml** file that shows the error text provided by the extension class. No error is shown to the user.

Below is a sample of the message text logged to the **appserverlog.xml** file when this class executes:

```
PlugInReturn object returned from Extension contained errors:
Testing Extension Exception - Sample Error Message
processPoint 101 ActionType 46 BuildingBlock 1001 Caller USER.
```

E-mail Notification

The **mss_ext_samples.jar** file contains the class ExtensionFrameworkOneWayTest.java. This class provides sample code that sends an email notification from an extension class.

CORBA API Invocation

The **mss_ext_samples.jar** file contains the class InvokeCorbaAPIExtension.java. This class provides sample code that invokes a CORBA API method from an extension class. The sample code calls the CORBA API method getOrganization, which is defined in the TaskCompletionSubsession of the Work Management CORBA API.

Running the Sample Code

The extensions sample code provides concrete examples of how to code specific logic in the extension Java class such as error handling, sending an email notification, and making an API call. When executed, the sample code also provides concrete examples of the outcome of these actions. You can define any of the sample classes as an extension in the UI, associate an execution point with the extension, and then trigger the execution point to invoke the sample class extension and see the outcome.

The extension sample code provided with your installation of MetaSolv Solution is listed below, including the first release in which it is supported. All sample code related files are located in the **mss_ext_samples.jar** file. The installer copies the **mss_ext_samples.jar** file to your *MSLV_Home*/**appserver/samples** directory, where *MSLV_Home* is the directory in which MetaSolv Solution is installed.

- For a full installation, the contents of the **mss_ext_samples.jar** file are extracted into the appropriate path under your *MSLV_Home* directory. The appropriate path for each file is identified by the path specified in the **.jar** file.
- For an upgrade, you must manually extract the contents of the **mss_ext_samples.jar** file into the appropriate path under your *MSLV_Home* directory. The appropriate path for each file can be identified by the path specified in the **.jar** file.

Sample code options:

- AssignWorkQueues
- ProvPlanDefault
- ExtensionFrameworkOneWayTest
- SampleExtensionException
- InvokeCorbaAPIExtension

For each sample, the following file types exist in the **mss_ext_samples.jar** file. (The only exception is the InvokeCorbaAPIExtension sample, which does not have a supporting XML file because there is no input data needed for this sample.)

- .java: the extension Java source file
- .class: the corresponding compiled Java class file
- .xml: the supporting xml file that defines sample input data and sample configuration data that is passed to the extension logic

For example, the following three files that support the AssignWorkQueues sample exist in the **mss_ext_samples.jar** file:

- AssignWorkQueues.java
- AssignWorkQueues.class
- AssignWorkQueues.xml

AssignWorkQueues

The AssignWorkQueues sample is provided to show extension logic that assigns specific work queues, and uses a synchronous example. The sample logic shows how to return the specific data that the Assign Queues execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point is logged for your viewing.

To run the AssignWorkQueues sample code:

1. Through the UI, define a synchronous extension with the name AssignWorkQueues.
2. Through the UI, associate the Assign Queues execution point with the extension by searching for the following criteria:
   - Building Block: All Task Types
   - Process Point: Task Maintenance
   - Action Type: Assign Queues
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.

   where:

   MSLV_Home is the directory in which the MetaSolv Solution software is installed.

   server is the name of the WebLogic server.
5. Look at the **AssignWorkQueues.xml** file to understand what the expected results should be in step 7. Specifically, the **AssignWorkQueues.xml** file defines four tasks and the corresponding work queues to which the tasks are assigned. The work queues are returned by the AssignWorkQueues extension logic.
6. Through the UI, trigger the execution point by assigning work queues.
7. Verify the outcome by looking in the UI, and by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

ProvPlanDefault

The ProvPlanDefault sample is provided to show extension logic that defaults a provisioning plan, and uses a synchronous example. The sample logic shows how to return the specific data that the Provisioning Plan Default execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point is logged for your viewing.

To run the ProvPlanDefault sample code:

1. Through the UI, define a synchronous extension with the name ProvPlanDefault.
2. Through the UI, associate the Provisioning Plan Default execution point with the extension by searching for the following criteria:
   - Building Block: All Task Types
   - Process Point: Task Generation
   - Action Type: Provision Plan Default
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure that the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.

   where:

   MSLV_Home is the directory in which the MetaSolv Solution software is installed

   server is the name of the WebLogic server

5. Look at the **ProvPlanDefault.xml** file to understand what the expected results should be in step 7. Specifically, the **ProvPlanDefault.xml** file defines a specific provisioning plan ID that is returned by the ProvPlanDefault extension logic.
6. Through the UI, trigger the execution point by assigning a provisioning plan to an order.
7. Verify the outcome by looking in the UI, and by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

ExtensionFrameworkOneWayTest

The ExtensionFrameworkOneWayTest sample is provided to show extension logic that sends an email notification. When the sample code is executed, it shows the outcome of this action, and the notification is logged for your viewing. This sample also shows:

- How to read an XML file and determine what execution point invoked it.
- How to send an email notification.
- How to read the input name/value pair Array and put that data into an email.

To run the ExtensionFrameworkOneWayTest sample code:

1. Through the UI, define an extension with the name ExtensionFrameworkOneWayTest.
2. Through the UI, associate an execution point with the extension by searching for criteria such as:
   - Building Block: All Task Types
   - Process Point: Task Maintenance
   - Action Type: Assign Jeopardy
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure that the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.

   where:

   MSLV_Home is the directory in which the MetaSolv Solution software is installed

   server is the name of the WebLogic server
5. Look at the **ExtensionFrameworkOneWayTest.xml** file to understand what the expected results should be in step 7. Modify the data, such that the email recipient is a valid address that can check for the mail notification, and the SmtpServerKey value is valid for your location.
6. Through the UI, trigger the execution point that was selected in step 2.
7. Verify the outcome by looking in designated email inbox, and by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

SampleExtensionException

The SampleExtensionException sample is provided to show extension logic that sends an email notification and throws an exception. The code always throws an exception. When the sample code is executed, it shows the outcome of this action in the form of the email notification, and in the form of a logged error if the extension is defined as synchronous.

Note: If the extension is defined as asynchronous, the extension framework does not log an error, but it does send an email notification.

If the extension is defined as synchronous, the extension framework logs an error to the log file, in addition to sending the email notification.

Perform the following steps to run the SampleExtensionException sample code:

1. Through the UI, define a synchronous extension with the name SampleExtensionException.
2. Through the UI, associate an execution point with the extension by searching for criteria such as:
   - Building Block: All Task Types
   - Process Point: Task Maintenance
   - Action Type: Assign Jeopardy
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure that the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.

   where:

   MSLV_Home is the directory in which the MetaSolv Solution software is installed

   server is the name of the WebLogic server
5. Look at the **SampleExtensionException.xml** file to understand what the expected results should be in step 7. Modify the data, such that the email recipient is a valid address that can check for the exception notification, and the SmtpServerKey value is valid for your location.
6. Through the UI, trigger the execution point that was selected in step 2.
7. Verify the outcome by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

InvokeCorbaAPIExtension

The InvokeCorbaAPIExtension sample is provided to show how to code CORBA API calls in the extension logic. When the sample code is executed, it also shows the outcome of this action. Specifically, the sample calls the CORBA API method getOrganization(), so the organization is logged for your viewing.

To run the InvokeCorbaAPIExtension sample code:

1. Through the UI, define an extension with the name InvokeCorbaAPIExtension.
2. Through the UI, associate an execution point with the extension by searching for criteria such as:
   - Building Block: All Task Types
   - Process Point: Task Maintenance
   - Action Type: Assign Jeopardy

3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure that the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.

   where:

   MSLV_Home is the directory in which the MetaSolv Solution software is installed

   server is the name of the WebLogic server
5. Through the UI, trigger the execution point that was selected in step 2.
6. Verify the outcome by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

SelectComponent

The SelectComponent sample is provided to show extension logic that selects a component or element, and uses a synchronous example. The sample logic shows how to return the specific data that the Select Component or Element execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point is logged for your viewing.

This sample is very specific in its function. Other samples are open-ended and can apply to several execution points. This sample code calls specific methods to accomplish the component selection. Java documentation is provided in the sample code to give you additional information about the methods that the sample code calls.

To run the SelectComponent sample code:

1. Through the UI, define a synchronous extension with the name SelectComponent.
2. Through the UI, associate the Select Component or Element execution point with the extension by searching for the following criteria:
   - Building Block: All Connections
   - Process Point: PCONDES Maintenance
   - Action Type: Select Component or Element
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.

   where:

   MSLV_Home is the directory in which the MetaSolv Solution software is installed

   server is the name of the WebLogic server
5. Through the UI:
   - Set up a DSLAM network location.
   - Add a network element of type DSL Multiplexer to the DSLAM network location.
   - Add a DSL card with an available port matching the rate code of the ordered service to the DSL Multiplexer.

- Enter a PSR order with an end user location that has the same zip code as the DSLAM network location.
- On the PSR order, add a service to the end user location that can be auto provisioned.
- Assign a provisioning plan to the order that defines the PCONDES task.

6. Through the UI, trigger the execution point by completing the PCONDES task.
7. Verify the outcome by looking in the UI, and by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

SelectPort

The SelectPort sample is provided to show extension logic that selects a port address, and uses a synchronous example. The sample logic shows how to return the specific data that the Select Port Address execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point is logged for your viewing.

This sample is very specific in its function. Other samples are open-ended and can apply to several execution points. This sample code calls specific methods to accomplish the port selection. Java documentation is provided in the sample code to give you additional information about the methods that the sample code calls.

To run the SelectPort sample code:

1. Through the UI, define a synchronous extension with the name SelectPort.
2. Through the UI, associate the Select Port Address execution point with the extension by searching for the following criteria:
   - Building Block: All Connections
   - Process Point: PCONDES Maintenance
   - Action Type: Select Port Address
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.

   where:

   MSLV_Home is the directory in which the MetaSolv Solution software is installed

   server is the name of the WebLogic server
5. Through the UI:
   - Set up a DSLAM network location.
   - Add a network element of type DSL Multiplexer to the DSLAM network location.
   - Add a DSL card with an available port matching the rate code of the ordered service to the DSL Multiplexer.
   - Enter a PSR order with an end user location that has the same zip code as the DSLAM network location.
   - On the PSR order, add a service to the end user location that can be auto provisioned.

- Assign a provisioning plan to the order that defines the PCONDES task.

6. Through the UI, trigger the execution point by completing the PCONDES task.
7. Verify the outcome by looking in the UI, and by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

SelectComponentForVirtual

The SelectComponentForVirtual sample is provided to show extension logic that selects a component or element for a virtual connection using a synchronous call. The sample logic reads the values (NS_ID and NS_COMP_ID) from the corresponding XML file. Even though the sample logic uses values from an XML file instead of performing actual logic to retrieve those values, it does demonstrate how to format the return data as required by the calling method. When the sample code is executed, it shows the outcome of this action by logging the input parameters to the console.

To run the SelectComponentForVirtual sample code:

1. Through the UI, define a synchronous extension with the name SelectComponentForVirtual.
2. Through the UI, associate the Select Component or Element execution point with your newly created extension by searching for the following criteria:
   - Building Block-All Connections
   - Process Point-VCONDES Maintenance
   - Action Type-Select Component or Element
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Navigate to the **SelectComponentForVirtual.xml** file in the *MSLV_Home*/*server*/**appserver/samples/customExtension/xml** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

   The keys in this file represent the desired Network System (NS_ID) and Component (NS_COMP_ID) for the virtual connection to be provisioned to. This file is read by the custom extension in step 6, and therefore you must modify these key values to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.
6. Through the UI:
   - Enter a PSR order with one or more virtual connections.
   - Assign a provisioning plan to the order that defines the VCONDES task.
   - Open the Service Request Virtual Circuits window by opening the VCONDES task.
   - Select one or more connections and then select **Auto Provision** from the **Options** menu.
   - Verify the outcome by looking in the UI, and by looking in the *server*.**mss.xml** file located in the *MSLV_Home*/*server*/**appserver/logs** directory.

SelectNetworkSystemForNetDesign

The use of the Select Network System execution point is demonstrated through the SelectNetworkSystemForNetDesign sample code.

The SelectNetworkSystemForNetDesign sample is provided to show extension logic that selects a network system for a network design automation. The sample logic reads the expected values (which are listed below in sample XML file) from the corresponding XML file, but shows how to return the data that the Select Network System execution point is expecting. Even though the sample logic uses values from an XML file instead of performing actual logic to retrieve those values, it does demonstrate how to format the return data as required by the calling method.

To run the SelectNetworkSystemForNetDesign sample code:

1. Through the UI, define a synchronous extension with the name SelectNetworkSystemForNetDesign.
2. Through the UI, associate the SelectNetworkSystem execution point with your newly created extension by searching for the following criteria:
   - Building Block - Network System
   - Process Point - Network System Design
   - Action Type - Select Network System
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Navigate to the **SelectNetworkSystemForNetDesign.xml** file in the *MSLV_Home/server/* **appserver/samples/customExtension/xml** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

   The keys in this file are listed below in the provided sample data. The sample data represents the network system properties that would be designed as part of automation of NETDSGN task. This file is read by the custom extension in step 6, so you must modify the key values provided in the sample data to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home/server*/**appserver/config** directory.
6. Through the UI:
   - Enter a PSR order.
   - Assign a provisioning plan to the order that defines the NETDSGN task and assign this task to the SYSTEM work queue.

The following example shows the **SelectNetworkSystemForNetDesign.xml** file format when using the SelectNetworkSystem execution point:

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

<SAMPLEDATA>
<RETURNDATA KEY="NS_ID" VALUE="user input value"/>
<RETURNDATA KEY="ACTIVITY_IND" VALUE="user input value"/>
<RETURNDATA KEY="NS_TEMPLATE_NAME" VALUE="user input value"/>
<RETURNDATA KEY="SHORT_NAME" VALUE="user input value"/>
<RETURNDATA KEY="LONG_NAME" VALUE="user input value"/>
<RETURNDATA KEY="DESC_100" VALUE="user input value"/>
<RETURNDATA KEY="HARD_SOFT_ASSIGN_CD" VALUE="user input value"/>
<RETURNDATA KEY="CUSTOMER_SYS_ID" VALUE="user input value"/>
```

```
<RETURNDATA KEY="PROVIDER_SYS_ID" VALUE="user input value"/>
<CUSTOM_ATTRIBUTE NAME="user input value">
<VALUE>user input value</VALUE>
</CUSTOM_ATTRIBUTE>
<CUSTOM_ATTRIBUTE NAME="user input value">
<!-- to delete existing CA value below tag should be used(applicable
for multi-valued CA's)-- >
<VALUE DELETE="TRUE">CA Value</VALUE>
</CUSTOM_ATTRIBUTE>
<!-- add custom attribute tags as needed, depending on CA's that are
populated for the network system -->
</SAMPLEDATA>
```

Table B–1 describes the keys in the **SelectNetworkSystemForNetDesign.xml** file.

Table B–1 SelectNetworkSystemForNetDesign Keys

Key Name	Description	Mandatory/ Optional	Data Type	Sample/ Valid Values
ACTIVITY_IND	Activity indicator	Mandatory	Char	"N" or "C" for new and change activities
NS_TEMPLATE_ NAME	Network system template name	For New activity, this field is mandatory	String	N/A
SHORT_NAME	Network system name	Few New activity, this field is mandatory	String	N/A
LONG_NAME	Long name for network system	Optional	String	N/A
DESC_1OO	Network system description	Optional	String	N/A
NS_ID	Network system ID (can be populated in update activity)	Optional	Int	N/A
HARD_SOFT_ASSIGN_ CD	Hard soft value	Optional	String	HARD/SOFT/ NONE
CUSTOMER_SYS_ID	Customer system ID	N/A	String	N/A
PROVIDER_SYS_ID	Provider system ID	Optional	String	N/A
NAME (custom attribute name)	CA name	Mandatory if CA has to be populated	String	N/A
VALUE (custom attribute value)	CA value	Mandatory if CA has to be populated	String	N/A

SelectCustEdgeCompForNetDesign

The use of the Select Customer Edge Component execution point is demonstrated through the SelectCustEdgeCompForNetDesign sample code.

The SelectCustEdgeCompForNetDesign sample is provided to show extension logic that selects a customer edge components for a network design automation. The sample logic reads the expected values (which are listed below in sample XML file) from the corresponding XML file, but shows how to return the data that the Select Customer

Edge Component execution point is expecting. Even though the sample logic uses values from an XML file instead of performing actual logic to retrieve those values, it does demonstrate how to format the return data as required by the calling method.

To run the SelectCustEdgeCompForNetDesign sample code:

1. Through the UI, define a synchronous extension with the name SelectCustEdgeCompForNetDesign.
2. Through the UI, associate the Select Customer Edge Component execution point with your newly created extension by searching for the following criteria:
   - Building Block - Network System
   - Process Point - Network System Design
   - Action Type - Select Customer Edge Component
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Navigate to the **SelectCustEdgeCompForNetDesign.xml** file in the *MSLV_Home/server*/ **appserver/samples/customExtension/xml** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

   The keys in this file are listed below in the provided sample data. This data represents the customer edge component properties that would be designed as part of NETDSGN task automation. This file is read by the custom extension in step 6, so you must modify the key values provided in the sample data to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home/server*/**appserver/config** directory.
6. Through the UI:
   - Enter a PSR order.
   - Assign a provisioning plan to the order that defines the NETDSGN task and assign this task to SYSTEM work queue.

The following example shows the **SelectCustEdgeCompForNetDesign.xml** file format when using the Select Customer Edge Component execution point:

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

<SAMPLEDATA>
<CUSTOMEREDGE>
<RETURNDATA KEY="ACTIVITY_IND" VALUE="N"/>
<RETURNDATA KEY="CE_NAME" VALUE="user input value"/>
<RETURNDATA KEY="CE_LOCATION_NAME" VALUE="user input value"/>
<RETURNDATA KEY="CE_TYPE" VALUE="CUST_SITE"/>
<RETURNDATA KEY="CE_NUMBER" VALUE="12"/>
<CUSTOM_ATTRIBUTE NAME=" user input value ">
<VALUE>CA Value</VALUE>
<!-- to delete existing CA value below tag should be
used(applicable for multi-valued CA's)-- >
<VALUE DELETE="TRUE">CA Value</VALUE>
</CUSTOM_ATTRIBUTE>
<!-- add custom attribute tags as many as you want depending on CA's
that need to be populated for network systems -->
</CUSTOMEREDGE>
<!-- add custom attribute tags as needed, depending on CA's that are
```

```
populated for the network system -->
</SAMPLEDATA>
```

Table B–2 describes the keys in the **SelectCustEdgeCompForNetDesign.xml** file.

Table B–2 SelectCustEdgeCompForNetDesign Keys

Key Name	Description	Mandatory/ Optional	Data Type	Sample/ Valid Values
ACTIVITY_IND	Activity indicator	Mandatory	Char	"N" or "C" for new and change activities
CE_NAME	Customer edge component name	For New activity, this field is mandatory	String	N/A
CE_NUMBER	Integer number used with CE_ NAME to uniquely identify each CE	Optional	Int	N/A
CE_LOCATION_ NAME	CLLI code of customer edge location. It should be the same as location name on the order.	For New activity, this field is mandatory.	String	N/A
CE_TYPE	Type of customer edge component	For New activity, this field is mandatory.	String	Sample data: CUST_SITE, CE_RTR
NS_COMP_ID	Network component ID (can be populated in update activity)	Optional	Int	N/A
NETWORK_ELEMENT_ ID	Network element ID	Optional	String	N/A
NAME (custom attribute name)	CA name	Mandatory if CA has to be populated	String	N/A
NAME (custom attribute value)	CA value	Mandatory if CA has to be populated	String	N/A

SelectConnectionEndPoints

The use of the Select End Component For Physical Connection execution point is demonstrated through the SelectConnectionEndPoints sample code.

The SelectConnectionEndPoints sample is provided to show extension logic that selects a connection end point for each physical connection present on a PSR order. The sample logic reads the expected values (which are listed below in a sample XML file) from the corresponding XML file, but shows how to return the data that the Select End Component For Physical Connection execution point is expecting. Even though the sample logic uses values from an XML file instead of performing actual logic to retrieve those values, it does demonstrate how to format the return data as required by the calling method.

Perform the following steps to run the SelectConnectionEndPoints sample code:

1. Through the UI, define a synchronous extension with the name SelectConnectionEndPoints.

2. Through the UI, associate the Select End Component For Physical Connection execution point with your newly created extension by searching for the following criteria:
   - Building Block - Network System
   - Process Point - Network System Design
   - Action Type - SelectEndComponentForPhysicalConnection
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Navigate to the **SelectConnectionEndPoints.xml** file in the *MSLV_Home*/*server*/**appserver/samples/customExtension/xml** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

   The keys in this file represent the desired Network System (NS_ID) and Component (NS_COMP_ID) for the virtual connection to be provisioned to. This file is read by the custom extension in step6, and therefore you must modify these key values to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.
6. Through the UI:
   - Enter a PSR order and order for physical connections.
   - Assign a provisioning plan to the order that defines the NETDSGN task and assign this task to SYSTEM work queue.

The following example shows the **SelectConnectionEndPoints.xml** file format when using the Select End Component For Physical Connection execution point:

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

<SAMPLEDATA>
<CONNECTION>
<RETURNDATA KEY="ACTIVITY_IND" VALUE="(N)ew or (C)hange"/>
<RETURNDATA KEY="CE_NAME" VALUE="customer edge comp name"/>
<RETURNDATA KEY="CE_NUMBER" VALUE="customer edge number"/>
<RETURNDATA KEY="PE_NAME" VALUE="provider edge name"/>
<RETURNDATA KEY="PE_NUMBER" VALUE="provider edge number"/>
<RETURNDATA KEY="PE_NETWORK_NAME" VALUE="network system name of PE"/>
<RETURNDATA KEY="CONNECTION_ECCKT" VALUE="connection name"/>
<RETURNDATA KEY="PE_PARENT_NETWORK_NAME" VALUE="outer network of PE"/>
<!— CONNECTION tags can be added as needed, depending on the number of
physical connections on the order. -->
</SAMPLEDATA>
```

Table B–3 describes the keys in the **SelectConnectionEndPoints.xml** file.

Table B–3 SelectConnectionEndPoints Keys

Key Name	Description	Mandatory/ Optional	Data Type	Sample/ Valid Values
ACTIVITY_IND	Activity indicator	Mandatory	Char	"N" or "C"
CE_NAME	Customer edge component name. This should be the CE_NAME value returned from the SelectCusotmerEdgeComponent exeution point. This is one end of the connection.	For New activity, this field is mandatory.	String	N/A
CE_NUMBER	Integer number used with CE_NAME to uniquely identify each CE. This should be the CE_NUMBER value returned from the SelectCustomerEdgeComponent execution point.	Optional	Int	N/A
PE_NAME	Provider edge component name. This is the other end of the connection.	For New activity, this field is mandatory.	String	N/A
PE_NUMBER	Integer number used with PE_NAME to uniquely identify each PE.	Optional	Int	N/A
PE_NETWORK_NAME	Represents immediate parent network system of PE. This network should have a component with PE_NAME and PE_NUMBER values.	For New activity, this field is mandatory.	String	N/A
PE_PARENT_NETWORK_NAME	This field is useful when multiple layers of embedded networks exist for PE.	Optional	String	N/A
CONNECTION_ECCKT	Connection name	Mandatory	String	N/A

SelectCustEdgeEquipForNetDesign

The use of the Select Equipment For CE execution point is demonstrated through the SelectCustEdgeEquipForNetDesign sample code.

The SelectCustEdgeEquipForNetDesign sample is provided to show extension logic that selects an equipment and customer edge component so that the customer edge can be associated with equipment. The sample logic reads the expected values (which are listed below in a sample XML file) from the corresponding XML file, but shows how to return the data that the Select Equipment For CE execution point is expecting. Even though the sample logic uses values from an XML file instead of performing actual logic to retrieve those values, it does demonstrate how to format the return data as required by the calling method.

To run the SelectCustEdgeEquipForNetDesign sample code:

1. Through the UI, define a synchronous extension with the name SelectCustEdgeEquipForNetDesign.
2. Through the UI, associate the Select Equipment For CE execution point with your newly created extension by searching for the following criteria:
   - Building Block - Network System
   - Process Point - Network System Design
   - Action Type - Select Equipment For CE
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Navigate to the **SelectCustEdgeEquipForNetDesign.xml** file in the *MSLV_Home/server/* **appserver/samples/customExtension/xml** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

   The keys in this file represent the desired Network System (NS_ID) and Component (NS_COMP_ID) for the virtual connection to be provisioned to. This file is read by the custom extension in step6, and therefore you must modify these key values to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home/server*/**appserver/config** directory.
6. Through the UI:
   - Enter a PSR order and order for physical connections.
   - Assign a provisioning plan to the order that defines the NETDSGN task and assign this task to SYSTEM work queue.

The following example shows the **SelectCustEdgeEquipForNetDesign.xml** file format when using the Select Equipment For CE execution point:

```
<?xml version="1.0" encoding="UTF-8"?>
<SAMPLEDATA>
<CUSTOMEREDGE NAME="CE name"
CE_NUMBER="CE number"
EQUIPMENT_ID="equipment id" />
</CUSTOMEREDGE>
<!-- add CUSTOMEREDGE tags as needed, depending on number of equipments
associated with customer edge comp -->
</SAMPLEDATA>
```

Table B–4 describes the keys in the **SelectCustEdgeEquipForNetDesign.xml** file.

Table B–4 SelectCustEdgeEquipForNetDesign Keys

Key Name	Description	Mandatory/ Optional	Data Type	Sample/ Valid Values
CE_NAME	Customer edge component name. This should be the CE_NAME value returned from the SelectCusotmerEdgeComponent execution point.	Mandatory	String	N/A
CE_NUMBER	Integer number used with CE_NAME to uniquely identify each CE. This should be the CE_NUMBER value returned from the SelectCusotmerEdgeComponent execution point.	Optional	Int	N/A
EQUIPMENT_ ID	Equipment ID	Mandatory	Int	N/A

DS0/DS1 Automated Design

The use of the DS0/DS1 Automated Design execution point is demonstrated through the FacilityAutomatedDesign sample code.

The FacilityAutomatedDesign sample logic in the Java file shows the users how to provide the desired input assignment information (which is listed in the tables below). The sample logic demonstrates how to format and pass the data as required by the calling method.

To run the FacilityAutomatedDesign sample code:

1. Through the UI, define a synchronous extension with the name FacilityAutomatedDesign.
2. Through the UI, associate an execution point with the extension by searching for criteria such as:
   - Building Block: Connection
   - Process Point: Connection Design
   - Action Type: DS0/DS1 Automated Design
3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Navigate to the **FacilityAutomatedDesign** file in the *MSLV_ Home/server***/appserver/samples/customExtension/com/metasolv/custom/vendor/extension/FacilityAutomatedDesign** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

   The parameters in this file are listed below in the provided sample data. The sample data represents the assignments that would be designed as part of automation of the AUTODSGN task. This file is read by the custom extension in step 6, so you must modify the parameter values provided in the sample data to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home/server***/appserver/config** directory.
6. Through the UI:

- Enter a PSR order.
- Assign a provisioning plan to the order that defines the AUTODSGN task and assign this task to SYSTEM work queue.

Table B–5 describes the input parameters that need to be set in com.mslv.core.pi.internal.NetProv.ConnDesign.containerData.ProvisioningContainer.

Table B–5 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ProvisioningContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
networkBlocks	Collection of all the Network Assignment containers - NetworkAssignmentData.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no Optical Provisioning needs to be done.
faciltyBlocks	Collection of all the Facility Assignment containers - ChannelContainer.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no Facility Assignment needs to be done.
equipmentBlocks	Collection of all the Equipment Assignment containers - PortContainer.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no Equipment Assignment needs to be done.
cableBlocks	Collection of all the Cable Pair Assignment containers - CablePairBlockContainer.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no Cable Pair Assignment needs to be done.
crossReferenceBlocks	Collection of all the Cross Reference containers - CrossReferenceContainer.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no Cross Reference needs to be added.

Table B–5 (Cont.) Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ProvisioningContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
miscInfoBlocks	Collection of all the Miscellaneous Information. The miscellaneous information has to be passed in a String variable. Only 16 characters can be added in one design line and the rest will be wrapped in the subsequent lines. The first 160 characters will be taken for the assignment if the length exceeds 160 characters.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no Miscellaneous Information needs to be added.
foreignInfoBlocks	Collection of all the Foreign Info containers - ForeignInfoContainer.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no Foreign Info needs to be added.
notesBlocks	Collection of all the Notes containers - NotesContainer.	Optional	Vector	If not provided in the Custom Extension code, AUTODSGN assumes no notes are to be added.

Table B–6 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.NetworkAssignmentData.

Table B–6 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.NetworkAssignmentData

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
channelAssignments	Collection of all the Channel Assignment containers - one element for one channel position.	At least one element is mandatory	Vector of ChannelAssignmentData	If no element is present in the ChannelAssignments Vector, the AUTODSGN task will error out.
portAssignments	Collection of all the Port Assignment containers.	Optional	Vector of PortAssignmentData	N/A
commonNetworkNsId	Network ID of common Network System	Mandatory	Int	Should be a valid Network system ID, otherwise the AUTODSGN task will error out.
origCompId	Originating Component ID	Mandatory	Int	Should be a valid Component Id within the Network specified above, otherwise the AUTODSGN task will error out.
termCompId	Terminating Component ID	Mandatory	Int	Should be a valid Component ID within the Network specified above, otherwise the AUTODSGN task will error out.
additionalAssignmentSeqNbr	Additional assignment sequence number	Optional	Int	N/A
blockType	WP: Working Path PP: Protect Path (implies wp + pp)	Mandatory	String	WP and PP are the only valid values, otherwise the AUTODSGN task will error out.

Table B–7 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ChannelAssignmentData.

Table B–7 Input Parameters to Set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ChannelAssignmentData

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
parentCircuitDesignId	Circuit Design ID of the Parent Connection.	Mandatory	Int	Should be a valid Circuit Design ID within the Network specified above, otherwise the AUTODSGN task will error out.
circuitPositions	Circuit positions from the root parent circuit.	Mandatory	ArrayList	If given positions are not valid, the AUTODSGN task will error out.
protectPathIndicator	**Y:** Yes (for protect path segment) **N:** No (for working path segment)	Mandatory	Char	If not populated with one of the mentioned values, the AUTODSGN task will error out.
mainNetworkNsId	Network System ID.	Optional	Int	Should be a valid Network System ID.
sameChannelIndicator	Same channel Indicator.	Optional	Char	N/A

Table B–8 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.PortAssignmentData.

Table B–8 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.PortAssignmentData.

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
Root Equipment id	This represents base equipment (rack) on which required CARD is mounted.	Mandatory	long	If this is provided in input, but does not exist in the inventory, the AUTODSGN task will error out.
associatedToOrig	True: port associated at Originating Node. False: Port associated at Terminating Node	Mandatory	Boolean	N/A
Port Sequence	Port on the equipment to which the connection needs to be assigned.	Mandatory	long	If not provided, the AUTODSGN task will error out.
mountingPositions	List of mounting position numbers. This represents position of CARD with respect to RACK (root equipment) to which connection has to be assigned.	Optional	ArrayList	N/A

Table B–9 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ChannelContainer.

Table B–9 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ChannelContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
parentCircuitId	Circuit Design ID of the Parent Connection.	Mandatory	Int	Should be a valid Circuit Design ID, otherwise the AUTODSGN task will error out.
channelPositionNbr	Circuit position to which the child circuit should be assigned.	Mandatory	Int	If given position is not valid, the AUTODSGN task will error out.

Table B–10 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.PortContainer.

Table B–10 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.PortContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
equipmentId	The equipment on which the port assignment has to be made.	Mandatory	Int	If this is provided in input, but does not exist in the inventory, the AUTODSGN task will error out.
Port Sequence	Port on the equipment to which the connection needs to be assigned.	Mandatory	Int	If given port address is not valid, the AUTODSGN task will error out.
aZOtherCd	The code which identifies the side of the port assignment.	Mandatory	Char	The valid values are A, Z and O.

Table B–11 describes the input parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CablePairBlockContainer.

Table B–11 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CablePairBlockContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
sideOfCircuitInd	The indicator which determines the side of the circuit where the cable pair assignment has to be made. The valid values are A, Z and I.	Mandatory	Char	If the value is anything other than A, Z and I, the AUTODSGN task will error out.
sideOfCircuitSequenceNumber	The sequence number to identify the specific cable pair assignment within a Cable Pair Block. There can be multiple cable pair assignments under the sideOfCircuitindicator 'I'.	Conditional	Boolean	Mandatory if the sideOfCircuitInd is 'I'
cablePairSetVector	Collection of all the Cable Pair Set information.	Mandatory	Vector of CablePairSetContainer	N/A

Table B–12 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CablePairSelfContainer

Table B–12 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CablePairSetContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
additionaldBLoss	Allows the user to enter additional DB Loss for that Cable Pair Set.	Optional	Float	N/A
additionalResistance	Allows the user to enter additional Resistance for that Cable Pair Set.	Optional	Float	N/A
cableContainerVector	Collection of all the Cable pair information.	Mandatory	Vector of CableContainer	N/A

Table B–13 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CableContainer.

Table B–13 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CableContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
cableId	The cable ID of the desired Cable.	Mandatory	Int	If this does not exist in the inventory, the AUTODSGN task will error out.
cableComplementId	The Cable Complement ID of the desired Cable complement.	Mandatory	Int	If this does not exist in the inventory, the AUTODSGN task will error out. The Complement should have the above mentioned cable ID associated to it.
pairFibreChannelIdentifier	The identifier of the desired pair or fibre.	Mandatory	Int	The pair should exist for the specified Cable and Cable Complement and be in unassigned status. If not, the AUTODSGN task will error out.
functionCode	The Function code for the pair.	Conditional This code is optional if the user preference, Functional Code is Required, is set to **N**.	String	If the specified value is not one from the valid value list - T, R, S1, S2, X1, X2 - the AUTODSGN task will error out.
pendingDate	The pending date of the pair.	Optional	String	Should be a valid date, if not the AUTODSGN task will error out.
remarks	Remarks for the pair.	Optional	String	The terminal pair should be in unassigned status. If not, the AUTODSGN task will error out.
terminalPairsOriginating	The terminal pairs to be assigned on the Originating end if the Originating location is a Terminal Pair.	Optional	Int []	The terminal pair should be in unassigned status. If not, the AUTODSGN task will error out.
terminalPairsTerminating	The terminal pairs to be assigned on the Terminating end if the Terminating location is a Terminal Pair.	Optional	Int []	If this does not exist in the inventory, the AUTODSGN task will error out.
separationsRouteCode	The Identifier of the Separations route code to be assigned to the Pair assignment.	Optional	String	If this does not exist in the inventory, the AUTODSGN task will error out.

Table B–13 (Cont.) Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CableContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
srLocationdId	The originating location of the Separations Route.	Conditional	Int	It is mandatory if the separations route code is populated in the container. Also, if the combination of Separations Route code, Originating location ID and terminating location ID is invalid, the AUTODSGN task will error out.
srLocationdId2	The terminating location of the Separations Route.	Conditional	Int	It is mandatory if the separations route code is populated in the container. Also, if the combination of Separations Route code, Originating location ID and terminating location ID is invalid, the AUTODSGN task will error out.
numOfWires	The number of wires to be assigned.	Optional	Int	N/A.

Table B–14 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CrossReferenceContainer.

Table B–14 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CrossReferenceContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
crossReferenceType	The type of the Cross reference.	Optional	String	If not provided, the value will be taken from the preference. If provided, the value should be one from the valid value list - SYNONYM, ALIASCUSTOMER, ALIASPROVIDER, CHILD, CKR.
crossReferenceValue	The Cross Reference value to be assigned.	Mandatory	String	If not provided, the AUTODSGN task will error out.
status	The status of the Cross reference circuit.	Optional	Char	N/A

Table B–14 (Cont.) Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.CrossReferenceContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
accessProviderServCenter	The ICSC (Inter-Exchange Carrier Service Provider Code) value to be assigned.	Optional	String	The value should be a valid ICSC, if not, the AUTODSGN task will error out.
accessCust	The CCNA (Customer Carrier Naming Abbreviation) value to be assigned.	Optional	String	The value should be a valid CCNA, if not, the AUTODSGN task will error out.
associatedLocationId	The location ID corresponding to the Location to be assigned. For End User locations, the ID in the location_id_sr column from End_user_location table should be used.	Optional	Int	The value should be a valid Location ID, if not, the AUTODSGN task will error out.

Table B–15 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerDataForeignInfoContainer.

Table B–15 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ForeignInfoContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
AZTransmissionLevel	The value to be entered in the AZTransmissionLevel column of the Design line.	Optional	String	N/A
noteIndicator	The value to be entered in the note indicator column.	Optional	Char	If the value is one in the following list - I, O, D or 0 - the AUTODSGN task will error out.
equipTypeFacilityDesig	The value to be entered in the equipTypeFacilityDesig column of the Design line.	Optional	String	N/A
incrementalMileage	The value to be entered in the incrementalMileage column of the Design line.	Optional	String	N/A
location	The value to be entered in the location column of the Design line.	Optional	String	N/A
miscInfo	The value to be entered in the miscInfo column of the Design line.	Optional	String	N/A
relayRackFacilityType	The value to be entered in the relayRackFacilityType column of the Design line.	Optional	String	N/A

Table B–15 (Cont.) Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ForeignInfoContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
signalVoice	The value to be entered in the signalVoice column of the Design line.	Optional	String	N/A
unitChannel	The value to be entered in the unitChannel column of the Design line.	Optional	String	N/A
ZATransmissionLevel	The value to be entered in the ZATransmissionLevel column of the Design line.	Optional	String	N/A

Table B–16 describes the parameters to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.NoteContainer.

Table B–16 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.NotesContainer

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Error Conditions
noteIndicator	The value to be entered in the note indicator column.	Mandatory	Char	If this is not provided in input, the AUTODSGN task will error out. If the value is one in the following list - I, O, D or 0 - the AUTODSGN task will error out.
noteText	The value to be entered as a note to the circuit.	Mandatory	String	If this is not provided in input, the AUTODSGN task will error out.

ConnectionIdAutomation

The use of the Connection Id Automation execution point is demonstrated through the ConnectionIdAutomation sample code.

The ConnectionIdAutomation sample is provided to show extension logic that receives the information required for the Connection Id generation. The sample logic reads the expected values (which are listed below in a sample XML file) from the corresponding XML file, but shows how to return the data that the Connection Id Automation execution point is expecting. Even though the sample logic uses values from an XML file instead of performing actual logic to retrieve those values, it does demonstrate how to format the return data as required by the calling method.

Perform the following steps to run the ConnectionIdAutomation sample code:

1. Through the UI, define a synchronous extension with the name ConnectionIdAutomation.
2. Through the UI, associate the Connection Id Automation execution point with your newly created extension by searching for the following criteria:
   - Building Block: Connection
   - Process Point: Connection Design
   - Action Type: Connection Id Automation

3. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Navigate to the **ConnectionIdAutomation.xml** file in the *MSLV_Home*/*server*/**appserver/samples/customExtension/xml** directory, where *MSLV_Home* is the directory in which the MetaSolv Solution software is installed and *server* is the name of the WebLogic server.

   The keys in this file represent the desired information using which the Connection Id should be generated. This file is read by the custom extension in step 6, and therefore you must modify these key values to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver/config** directory.
6. Through the UI:
   - Enter a PSR order and order for physical connections.
   - Assign a provisioning plan to the order that defines the CKTID task and assign this task to SYSTEM work queue.

Notes:

- The CKTID Task Automation is applicable only to the circuits that are ordered through PSR.
- The out-of-the-box algorithm applies the automated circuit id generation to traditional circuits (CIRCUIT and INTRNCKT Item types) and template-based connections (CONNECTOR item types).
- The out-of-the-box functionality for traditional circuits generates either CLS or CLF type circuits.
- The out-of-the-box functionality for template-based connections uses the current process to generate OTS type circuits.
- The only valid connection formats for the custom extension and out-of-the-box functionality are CLS, CLF, OTF, and OTS.
- The only valid circuit types are C, F, and S.
- If the custom extension does not pass back an ID for CLS or CLF formats, the core code builds the correctly formatted ID.
- If the custom extension does not pass back an ID for OTS, the core code appends the **Constant for Freeformat Circuit ID** preference with "**/**" and a unique serial number to generate a unique ID.
- Network Location A and Z typically comes from the order information.
- If CE process point is not defined, then out-of-the-box functionality generates the Connection ID and other mandatory data that is required for circuit generation.
- The information that is populated on the order is used for the out-of-the-box functionality and for the extension logic. If the extension does not populate information that is populated on the order, the order information is used. If the extension populates the information differently than the order, the extension information is used. If the extension does populate the information differently than the order, the order information remains intact and the process does not update the original order information.
- Prior to 6.2.1, the current logic is used for template-based connections. If you have defined a stored procedure to define the Connection ID, it continues to still define

the Connection ID. If you have not defined a stored procedure, the current default logic is executed. You can completely customize all the circuit information on a template-based connection using the new extension.

- Currently, when you double-click the CKTID task and if there are CONNECTORS included in the task, the InvAutoIdProcess method is called to automatically generate the Connection ID. The code verifies if this extension has been implemented, if it has, then the InvAutoIdProcess method is not executed. An extension is considered implemented if an Extension Point has been associated to the Extension Summary.
- If the custom extension does not pass back an ID for OTS, the core code appends the **Constant for Freeformat Circuit ID** preference with "/" and a unique serial number to generate a unique ID.

The following fields are populated on the PSR Order:

- Document Number
- Serv Item Id
- Item Type Cd
- Item Alias
- Rate Code
- Framing
- Line Coding
- Framing ANSI indicator
- Jurisdiction Code
- Service Type Category
- Service Type Code
- Network Location A (Originating Location)
- Network Location Z (Terminating Location)

Note : There is currently an inconsistency between traditional and template-based connections and that is being addressed on another task.

The following example shows the **ConnectionIdAutomation.xml** file format when using the Connection Id Automation execution point:

```
<?xml version="1.0" encoding="UTF-8"?>
<SAMPLEDATA>
<RETURNDATA KEY="NETWORK_LOCATION_A" VALUE=""/>
<RETURNDATA KEY="NETWORK_LOCATION_Z" VALUE=""/>
  <RETURNDATA KEY="RATE_CODE" VALUE="DS1"/>
  <RETURNDATA KEY="SERVICE_TYPE_CATEGORY" VALUE="CLCI-SS LATA Access"/>
  <RETURNDATA KEY="SERVICE_TYPE_CODE" VALUE="DO"/>
  <RETURNDATA KEY="CONNECTION_TYPE" VALUE="S"/>
  <RETURNDATA KEY="CONNECTION_FORMAT" VALUE="CLS"/>
  <RETURNDATA KEY="FRAMING" VALUE="CBIT"/>
  <RETURNDATA KEY="LINE_CODING" VALUE="2B1Q"/>
  <RETURNDATA KEY="FRAMING_ANSI_INDICATOR" VALUE="Y"/>
  <RETURNDATA KEY="ALLOW_LOWER_RATES_INDICATOR" VALUE="Y"/>
  <RETURNDATA KEY="JURISDICTION_CODE" VALUE="0"/>
```

```
  <RETURNDATA KEY="PROTECTED_CIRCUIT_INDICATOR" VALUE="N"/>
  <RETURNDATA KEY="PARTITION_GROUP_ID" VALUE=""/>
  <RETURNDATA KEY="PREFIX" VALUE="EX"/>
  <RETURNDATA KEY="MODIFIER" VALUE="--"/>
  <RETURNDATA KEY="SERIAL_NUMBER" VALUE=""/>
  <RETURNDATA KEY="SUFFIX" VALUE=""/>
  <RETURNDATA KEY="TELCO_ID" VALUE="QFWU"/>
  <RETURNDATA KEY="SEGMENT" VALUE=""/>
  <RETURNDATA KEY="NETWORK_CHANNEL_SERVICE_CODE" VALUE=""/>
  <RETURNDATA KEY="NETWORK_CHANNEL_OPTION_CODE" VALUE=""/>
  <RETURNDATA KEY="FACILITY_DESIGNATION" VALUE=""/>
  <RETURNDATA KEY="FACILITY_TYPE" VALUE=""/>
  <RETURNDATA KEY="GENERATED_CONNECTION_ID" VALUE=""/>
</SAMPLEDATA>
```

Table B–17 describes the input parameters that need to be set in com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ConnectionIdAutomationData.

Table B–17 Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ConnectionIdAutomationData

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Sample/ Valid Values
NETWORK_ LOCATION_A	Location ID value of the primary location on the network location table. Network Location A typically comes from the order information.	Mandatory	Int	N/A
NETWORK_ LOCATION_Z	Location ID value of the secondary location on the network location table. Network Location Z typically comes from the order information.	Conditional. Mandatory for circuit type F or S. Optional for circuit type C.	Int	N/A
RATE_CODE	The rate code of the circuit.	Optional	String	N/A
SERVICE_ TYPE_ CATEGORY	The service type category of the circuit.	Mandatory	String	N/A
SERVICE_ TYPE_CODE	The service type code of the circuit.	Mandatory	String	N/A
CONNECTION _TYPE	The circuit type of the connection.	Mandatory	Char	Valid values: ■ C ■ F ■ S
CONNECTION _FORMAT	The format of the connection to be created.	Mandatory	String	Valid values: ■ CLF ■ CLS ■ OTS ■ OTF
LINE_CODING	The line coding value to be assigned to the circuit.	Optional	String	N/A

Table B–17 (Cont.) Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ConnectionIdAutomationData

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Sample/ Valid Values
FRAMING	The framing value to be assigned to the circuit.	Optional	String	N/A
FRAMING_ ANSI_ INDICATOR	The framing ANSI value to be assigned to the circuit.	Optional	Char	N/A
ALLOW_ LOWER_ RATES_ INDICATOR	The allow lower rate indicator to be stored for the circuit. Default is **N**.	Optional	Char	N/A
JURISDICTION _CODE	The Jurisdiction code of the circuit.	Mandatory	String	N/A
FACILITY_ DESIGNATION	The facility designation of the circuit to be created. If the custom extension does not pass back a designation the core algorithm is used to generate the designation. The core algorithm queries all designations between the two network locations, adds one, and compares the value to make sure it is within the CLF Designation Range if it exists. If no range has been set up, the range is 0-999999.	Conditional. Mandatory for only CLF.	String	N/A
FACILITY_ TYPE	The type of facility for the circuit. Defaults to service type code ID if not provided.	Conditional. Mandatory for only CLF.	String	N/A
MODIFIER	The modifier to be used for the connection ID generation in CLS format.	Conditional. Mandatory for only CLS.	String	N/A
TELCO_ID	The Telco ID to be used for the connection ID generation in CLS format. If this is not provided in the extension, the value in the preference, Service Request > Connection > Default Telco Id for Automation to default the Telco ID, is considered.	Conditional. Mandatory for only CLS.	String	N/A
SERIAL_ NUMBER	The serial number to be used for the connection ID generation in CLS format. The system wide sequence is used to generate Serial Number if not provided.	Conditional. Mandatory for only CLS.	String	N/A
PARTITION_ GROUP_ID	Enables you to assign partition group to the connection.	Conditional. The field is mandatory if the security preference, **Use Partition Level Security for Access to Inventory Data**, is set to **Y**. If the extension does not pass a value the system defaults the value to **101 – All Access**.	Int	N/A

Table B–17 (Cont.) Input Parameters to Set In com.mslv.core.api.internal.NetProv.ConnDesign.containerData.ConnectionIdAutomationData

Parameter Name	Description	Mandatory/Optional/ Conditional	Data Type	Sample/ Valid Values
PROTECTED_ CIRCUIT_ INDICATOR	The serial number to be used for the connection ID generation. Default is N.	Optional	Char	N/A
NETWORK_ CHANNEL_ SERVICE_ CODE	The network channel service code to be used for the connection ID generation.	Optional	String	N/A
NETWORK_ CHANNEL_ OPTION_ CODE	The network channel option code to be used for the connection ID generation.	Optional	String	N/A
PREFIX	The prefix value to be used for the connection ID generation in CLS format. If the value is not provided, the system uses the system preference value under Service Request > Connection > Default Prefix for CLS Circuit IDs. If preference also is not set, prefix will be 2 empty spaces.	Conditional. Mandatory for only CLS.	String	N/A
SUFFIX	The suffix value to be used for the connection ID generation in CLS format.	Conditional. Mandatory for only CLS.	String	N/A
SEGMENT	The segment value to be used for the Connection ID generation in CLS format.	Conditional. Mandatory for only CLS.	String	N/A
GENERATED_ CONNECTION _ID	This parameter holds the generated connection ID. If the custom extension does not pass back an ID for CLS or CLF formats, the core code builds the correctly formatted ID. If the custom extension does not pass back an ID for OTS, the core code appends the **Constant for Freeformat Circuit ID** preference with "/" and a unique serial number to generate a unique ID.	Conditional. Required for OTF. Optional for CLS, CLF, and OTF.	String	N/A

DedicatedPlantSelection

The DedicatedPlantSelection sample code demonstrates the use of the Select Dedicated Plant action type.

The DedicatedPlantSelection sample code provides information about how to provide the input information and return the data.

The sample code reverses the prioritized sort order of the input dedicated plant list and populates this reversed list in the output dedicated plant list.

To run the DedicatedPlantSelection sample code:

1. Create a synchronous extension with the name DedicatedPlantSelection.

2. Associate the Select Dedicated Plant execution point with the DedicatedPlantSelection extension by searching for the following criteria:

   For the PCONDES Task:

   - Building Block Type = Connection
   - Building Block Name = All Connections
   - Process Point = PCONDES Maintenance
   - Action Type = Select Dedicated Plant

   For the AUTODSGN task:

   - Building Block Type = Connection
   - Building Block Name = All Connections
   - Process Point = Connection Design
   - Action Type = Select Dedicated Plant

3. Ensure that the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
4. Ensure that the logging level is set correctly by verifying the **loggingconfig.xml** file located in the *MSLV_Home*/*server*/**appserver**/**config** directory.
5. Do one of the following:
   - To automate the PCONDES task:
     - Enter a PSR order for physical connections.
     - Assign a provisioning plan to the order that contains the PCONDES task and assign this task to the SYSTEM work queue.
   - To automate the AUTODSGN task:
     - Enter a PSR order for circuits that have either a DS0 or N/A rate code and have been ordered as either a Circuit or Line product.
     - Assign a provisioning plan to the order that contains the AUTODSGN task and assign this task to the SYSTEM work queue.

Table B–18 lists the input parameters that you must set in `com.mslv.core.api.internal.NetProv.containerData.design.DedicatedPlantExtensionData`.

Table B–18 Input parameters to be set in com.mslv.core.api.internal.NetProv.containerData.design.DedicatedPlantExtensionData

Input Parameters	Description
OutputDedicatedPlantList	List of dedicated plants to assign. This parameter contains the following attributes: ■ ded_cc_grp_id: Unique key of the available dedicated plants. ■ last_modified_date: Last modified date of the available dedicated plants. ■ supports_product: **True** or **False**. ■ equipment_id: Equipment ID of the card used to build the available dedicated plants. ■ item_spec_id: Equipment specification ID of the service being designed. ■ item_spec_nm: Equipment specification name of the service being designed. ■ priority_seq: Priority sequence of the equipment set on the **Plant Administration** tab.
ErrorMessage	Error message to be sent to the server log.

Create/Update End User Location

The use of the Create/Update End User Location execution point is demonstrated through the SampleAddressValidation sample code.

The SampleAddressValidation sample is provided to show extension logic that receives the information required for validating the end user locations when you do any of the following:

- Create or update end user location address information in the PSR Ordering Dialog
- Create or update service locations on a PSR order
- Create or update end user location information on the **PRILOC/SECLOC Info** tab of the Product Service Request window
- Create or update end user location address information in the End User Location Maintenance window

The sample logic reads the expected values from the corresponding XML file and shows how to return the data that the Create and Update execution points are expecting. In addition, the sample logic can also access any third party systems, run direct database queries, and so on.

To run the SampleAddressValidation sample code:

1. Define a synchronous extension and specify a name for the extension. For example, SampleAddressValidation.

   The extension name that you define must match the name and letter case of the Java class (for example, SampleAddressValidation.class) that contains your custom logic.

   The **SampleAddressValidation.class** and **SampleAddressValidation.java** files are located at:

   *MSLV_Home/server/***appserver/samples/customExtension/com/metasolv/custom/vendor/extension/SampleAddressValidation**

   where:

- *MSLV_Home* is the directory in which the MetaSolv Solution software is installed
- *server* is the name of the WebLogic server

2. Ensure the **gateway.ini** entry that defines the sample code path reflects the correct location of the sample files extracted from the **mss_ext_samples.jar** file.
3. Associate the Create/Update End User Location execution point with the SampleAddressValidation extension by searching using the following criteria:
   - Building Block Type = Address
   - Building Block Name = All End User Locations
   - Process Point = PSR, EUL Maintenance
   - Action Type = Create, Update
4. Navigate to the **SampleAddressValidation.xml** file in the *MSLV_Home*/*server*/**appserver**/**samples**/**customExtension**/**xml** directory.

   The **SampleAddressValidation.xml** file is read by the custom extension when it is triggered, and therefore you must modify the file and format the return data as required by the calling method. See "Sample Address Validation Return Data Format" for more information.
5. Trigger the execution point by doing one of the following:
   - Create or update end user location address information in the PSR Ordering Dialog
   - Create or update service locations on a PSR order
   - Create or update end user location address information on the **PRILOC/SECLOC Info** tab of the Product Service Request window
   - Create or update end user location address information in the End User Location Maintenance window
6. Verify the outcome by looking in the UI, and by looking in the **appserverlog.xml** file located in the *MSLV_Home*/*server*/**appserver**/**logs** directory.

Sample Address Validation Return Data Format

The following are examples of the return data format in the **SampleAddressValiation.xml** file for custom extension success, failure, and warning scenarios:

Success Scenario Sample Data

```
<SAMPLEDATA>
   <RETURNCODE>SUCCESS</RETURNCODE>
   <RETURNTEXT>This is a success message.</RETURNTEXT>
   <ADDRESS></ADDRESS>
   <ADDRESSID></ADDRESSID>
</SAMPLEDATA>
```

Failure Scenario Sample Data

```
<SAMPLEDATA>
   <RETURNCODE>FAILURE</RETURNCODE>
   <RETURNTEXT>This is a failure message.</RETURNTEXT>
   <ADDRESS></ADDRESS>
   <ADDRESSID></ADDRESSID>
```

```
</SAMPLEDATA>
```

Warning Scenario Sample Data 1

```
<SAMPLEDATA>
   <RETURNCODE>WARNING</RETURNCODE>
   <RETURNTEXT>This is a warning message.</RETURNTEXT>
   <ADDRESS>
         <sfname>MSAG</sfname>
         <structureFormatComponents>
                <id>33</id>
                <name>Street Name</name>
                <componentType>N</componentType>
                <value>Demo Street</value>
         </structureFormatComponents>
   </ADDRESS>
   <ADDRESSID></ADDRESSID>
</SAMPLEDATA>
```

In sample data 1 for the warning scenario, if you leave the <ADDRESSID> tag blank, the values within the <ADDRESS> tag are considered by the custom extension.

Warning Scenario Sample Data 2

```
<SAMPLEDATA>
   <RETURNCODE>WARNING</RETURNCODE>
   <RETURNTEXT>This is a warning message.</RETURNTEXT>
   <ADDRESS>
         <sfname>MSAG</sfname>
         <structureFormatComponents>
                <id>33</id>
                <name>Street Name</name>
                <componentType>N</componentType>
                <value>Demo Street</value>
         </structureFormatComponents>
   </ADDRESS>
   <ADDRESSID>153</ADDRESSID>
</SAMPLEDATA>
```

In sample data 2 for the warning scenario, if you specify a value within the <ADDRESSID> tag, the values within the <ADDRESS> tag are ignored by the custom extension.