HP OpenView Business Process Insight

For the Windows® Operating System

Software Version: 02.10

Integration Training Guide - BPEL API

This guide is not being updated for Business Process Insight, from versions 2.1x and later.

Document Release Date: January 2007
Software Release Date: January 2007

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notices

Trademark Notices

Java™ is a US trademark of Sun Microsystems, Inc.

Microsoft® is a US registered trademark of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Windows® and MS Windows® are US registered trademarks of Microsoft Corporation.

Documentation Updates

This manual's title page contains the following identifying information:

- Software version number, which indicates the software version
- Document release date, which changes each time the document is updated
- Software release date, which indicates the release date of this version of the software

To check for recent updates, or to verify that you are using the most recent edition of a document, go to:

http://ovweb.external.hp.com/lpe/doc_serv/

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

Support

Please visit the HP OpenView support Web site at:

http://www.hp.com/managementsoftware/support

This Web site provides contact information and details about the products, services, and support that HP OpenView offers.

HP OpenView online software support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valuable support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit enhancement requests online
- Download software patches
- Submit and track progress on support cases
- Manage a support contract
- Look up HP support contacts
- Review information about available services
- Enter discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and log in. Many also require a support contract.

To find more information about access levels, go to:

http://www.hp.com/managementsoftware/access_level

To register for an HP Passport ID, go to:

http://www.managementsoftware.hp.com/passport-registration.html

Contents

1 BPEL

The OVBPI Modeler is able to import a BPEL XML file and create a flow definition within the Model Repository. How to use the OVBPI Modeler to import BPEL is described in the *OVBPI Integration Training Guide - Importing BPEL*.

When importing a BPEL process, the Modeler maps the various BPEL elements to nodes within an OVBPI flow definition. Not all BPEL elements get mapped. Some BPEL elements are ignored as they are considered not necessary within an OVBPI flow definition. Some BPEL elements map directly to an activity node. Some BPEL elements reflect a structural aspect of the business process and so they may generate a block of nodes surrounded by a pair of opening and closing junction nodes.

The BPEL element to OVBPI node mapping is based on the BPEL 1.1 standard.

What if you wanted to alter this mapping? What if you had a particular style of BPEL that used XML elements and/or properties that are not currently within the BPEL 1.1 standard?

The OVBPI BPEL importer is written in such a way that you can extend the way that BPEL elements are mapped, thus you can control how the OVBPI flow definition is produced.

This manual describes how to extend the capabilities of the OVBPI Modeler's BPEL importer using the Repository API.

This chapter looks at the different ways to invoke the BPEL importer.

Prerequisites

This manual assumes that you have either read through the training guide *OVBPI Integration Training Guide - Repository API*, or you are familiar with the Repository API.

Invoking the BPEL Importer

There are two ways to invoke the OVBPI BPEL importer:

1. From within the OVBPI Modeler

   By selecting `File->Import Definitions...`

2. Using the Command Line Interface

Invoking the BPEL importer from within the OVBPI Modeler is described in the Modeler on-line help and discussed in the *OVBPI Integration Training Guide - Importing BPEL*.

Let's look at how to run the BPEL importer standalone using the command line...

BPEL Importer Command Line Interface

The following script enables you to run the BPEL importer outside the OVBPI modeler, as a standalone application. The script is:

`OVBPI-INSTALL-DIR\bin\ovbpibpelimport.bat`

If you run this script with no parameters, it outputs a usage message explaining the various command line parameters available. There are quite a few command line parameters, and some of them may well be familiar to you if you have imported BPEL processes from within the OVBPI Modeler.

Some of the main parameters are:

- `-bpel filename`

  You specify the name of the BPEL XML file to be imported.

- `-host hostname`

  You specify the host name of the Model Repository you wish to connect to.

- `-user username`
  `-password password`

  You specify the user name and password so that the importer can log on to the Model Repository at your specified host name.

- `-replace true|false`

  You can specify whether you want the import to replace any previous import of this BPEL process.

- `-newname` *`name`*

  You can specify whether you want to import the flow and have the flow's name set to the string provided.

- `-nameGeneratorFile` *`filename`*

  You can specify the file name that contains your name generator file.

For example:

```
ovbpibpelimport.bat -bpel c:\bpel\ex_InsuranceSelectionProcess.bpel
                    -nameGeneratorFile C:\bpel\NameOnly.properties
                    -host localhost
                    -user admin
                    -password ovbpi
                    -replace true
```

where the OVBPI Model Repository component is up and running on the host `localhost`.

You can also set parameters such as the page border and the gap between the nodes, as you can when importing from within the OVBPI Modeler.

For example:

```
ovbpibpelimport.bat -bpel c:\bpel\ex_InsuranceSelectionProcess.bpel
                    -nameGeneratorFile C:\bpel\NameOnly.properties
                    -host localhost
                    -user admin
                    -password ovbpi
                    -replace true
                    -horizontalPageBorder 100
                    -verticalPageBorder 100
                    -rowGap 100
                    -columnGap 120
```

Customizing the BPEL Scripts

Both the script which invokes the OVBPI Modeler (`biamodeler.bat`) and the BPEL importer command line script (`ovbpibpelimport.bat`) can be customized to run your own version of the BPEL importer.

Let's consider each script and explain which aspects can be customized.

biamodeler.bat

Before customizing the `biamodeler.bat` script, you should make a copy of this script (referred to from now on by the name `mybiamodeler.bat`) and make all modifications to this `mybiamodeler.bat` script.

Within your `mybiamodeler.bat` script you can modify the following classes:

- Name generator class (`set NAMEGENERATORCLASS=`)

  This allows you to provide your own name generator class to provide additional name transformations.

  The *OVBPI Integration Training Guide - Importing BPEL* includes an example that shows you how to do this.

- BPEL importer class (`set IMPORTERCLASS=`)

  This allows you to provide your own BPEL importer class and alter the way that BPEL elements are handled and mapped to OVBPI nodes. See Chapter 2, Custom Element Handlers for further details.

If you do alter any of these variables to refer to your own custom Java classes, then you also need to add these classes to the `CLASSPATH` of your script.

ovbpibpelimport.bat

Before customizing the `ovbpibpelimport.bat` script, you should make a copy of this script (referred to from now on by the name `myovbpibpelimport.bat`) and make all modifications to this `myovbpibpelimport.bat` script.

Within your `myovbpibpelimport.bat` script you can modify the following classes and settings:

- Name generator class (`set NAMEGENERATORCLASS=`)

  This allows you to provide your own name generator class to provide additional name transformations.

  The *OVBPI Integration Training Guide - Importing BPEL* includes an example that shows you how to do this.

- BPEL importer class (`set IMPORTERCLASS=`)

  This allows you to provide your own BPEL importer class and alter the way that BPEL elements are handled and mapped to OVBPI nodes. See Chapter 2, Custom Element Handlers for further details.

- Provide standard command line options (`set STANDARD_OPTIONS=`)

  This allows you to set one or more command line options so that whenever this script is invoked, these standard command line options are set for the user.

- Provide your own BPEL Command Line Class (`set MAINCLASS=`)

  This allows you to replace the main class of the command line interface itself. You could write your own BPEL importer main class that, for example, imports the BPEL and then carries out any automated clean up of the resultant flow definition, and then possibly configures any data definitions, event definitions, progression rules, etc.. See Chapter 3, Custom CLI Main Class for further details.

If you do alter any of these variables to refer to your own custom Java classes, then you also need to add these classes to the `CLASSPATH` of your script.

2 Custom Element Handlers

This chapter looks at how to write your own custom BPEL element handlers.

BpelImporter Class

The code that imports a BPEL file into the Model Repository is provided by the following class:

`com.hp.ov.bia.model.repository.api.utils.bpel.BpelImporter`

Within both the `biamodeler.bat` and the `ovbpibpelimporter.bat` script files, the variable `IMPORTERCLASS` is set to this class name.

By extending this `BpelImporter` class you are able to customize the way BPEL elements are imported. You are able to load in additional XML element handlers. You can also override any of the current element handlers with your own version of them.

The basic format of the code looks as follows:

```
package com.customer.bpel;

import com.hp.ov.bia.model.repository.api.utils.bpel.BpelImporter;

public class MyBpelImporter extends BpelImporter
{
    public MyBpelImporter()
    {
        // Load up the standard set of element handlers.
        super();

        // Add your custom element handler.
        setElementHandler(xml_namespace,
                          xml_element_name,
                          new MyElementHandler());

        // Add more custom element handlers...
    }
}
```

where:

- The class extends `BpelImporter`.
- Your constructor loads all the XML element handlers.

  By calling `super()` you load in all the pre-defined XML element handlers that come as standard with the base `BpelImporter` class.

- You then call `setElementHandler()` to load in your custom element handler.

  The `setElementHandler()` method takes three parameters:

  — The XML name space

    Declared as type `org.jdom.Namespace`.

  — The name of the XML element.

  — An instance of the element handler class to handle this XML element.

Before looking in more detail at writing your own element handler class, let's take a look at how to set up the name space for your element...

XML Name Space

The XML standard allows you to define XML name spaces and use these to qualify XML elements. For example, you can declare a name space called `gb`, and then embed custom XML, as shown in the following XML segment:

```
<process ....>
   <sequence>
      <gb:mytag>
           Some custom details...
      </gb:mytag>
   </sequence>
</process>
```

By using your own name space, you can embed your own custom BPEL elements within your BPEL XML. These custom elements are only processed by a BPEL importer that has been explicitly extended to interpret them.

Within BPEL XML, you define a name space within the `process` element as follows:

```
<process  ...  xmlns:namespace-prefix="namespaceURI">
```

where:

- *`namespace-prefix`* defines the string that prefixes all XML elements within this name space.
- *`namespaceURI`* is simply a unique identifier string.

  This *`namespaceURI`* can be any string, so long as it is unique among all the name spaces declared within the BPEL XML file.

  You often find that companies use the *`namespaceURI`* as a pointer to a real Web page containing information about the name space.

For example, consider the following `process` element:

```
<process abstractProcess="yes" name="NewOrder"
         targetNamespace="http://www.hp.com"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:gb="http://www.mycompany.com">
```

This `process` element declares that two XML name spaces are used within this process definition, as follows:

- The default name space if set by the `xmlns=` line.

  Any XML element that is simply of the form `<element>` is therefore within this default XML name space.

- The other name space is defined by the `xmlns:gb=` line.

  This says that any element relating to the `gb` name space is of the form `<gb:element>`.

Code Example

If you are extending the `BpelImporter` class and you want to recognize a custom name space, you need to use the `org.jdom.Namespace` class and then get the name space.

Suppose your BPEL `process` element is defined as follows:

```
<process abstractProcess="yes" name="NewOrder"
         targetNamespace="http://www.hp.com"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:gb="http://www.mycompany.com">
```

For you to be able to work with the `gb` name space, you need to get this name space by using the following code segment:

```
import org.jdom.Namespace;
...
Namespace gbNamespace = Namespace.getNamespace(
                                                    "gb",
                                                    "http://www.mycompany.com"
                        );
```

where:

- The first parameter of the `getNamespace()` method holds the string representation for the XML name space prefix.
- The second parameter of the `getNamespace()` method holds the URI for this XML name space.

  The URI must match the URI declared for this `gb` prefix, as declared within the `process` element.

- The variable `gbNamespace` is set to the `gb` name space.

Now, if you are extending the `BpelImporter` class, and you are loading in a custom element handler to handle the `gb:mytag` element, your call to the `setElementHandler()` method is as follows:

```
setElementHandler(gbNamespace,
                  "mytag",
                  new MyElementHandler());
```

Default Name Space

When writing a Java class that extends the `BpelImporter` class, the class inherits the variable `bpwsNamespace1_1` which is set to the default name space URI:

```
"http://schemas.xmlsoap.org/ws/2003/03/business-process/"
```

So, if you are adding a custom element called `mytag`, which is within the default BPEL name space, your call to the `setElementHandler()` method is as follows:

```
setElementHandler(bpwsNamespace1_1,
                  "mytag",
                  new MyElementHandler());
```

Refer to Handling Elements in the BPEL Name Space on page 32 for more discussion about working within the default BPEL name space.

Writing an Element Handler

When you write an XML element handler your class needs to extend the following class:

`com.hp.ov.bia.model.repository.api.utils.bpel.BaseElementHandler`

The `BaseElementHandler` class provides a number of methods, some of which you need to override and some of which you may call when creating nodes within the flow definition.

Let's first take a look at the overall code outline for an element handler, and then go through and discuss each method in more detail.

Code Outline

```
class MyElementHandler extends BaseElementHandler
{

      // The constructor
      MyElementHandler()
      {
            super(childElements, true_or_false);
      }



      public boolean isActivityElement()
      {
            // Does this element handler create any OVBPI nodes?
            return true_or_false;
      }
```

```
    public ExtraNodeInfo handleElement(Element element,
                                       ExtraNodeInfo precedingNode,
                                       ExtraNodeInfo parentNode)
    throws RepositoryException
    {
        // Say that you are processing this element
        journal.startedHandling(element,
                                precedingNode,
                                parentNode,
                                false);
        try
        {
             // Your code to handle this element

             return theLastNodeYouCreated_or_null;
        }
        finally
        {
          // Say that you have processed this element
          journal.completedHandling(element,
                                    precedingNode,
                                    parentNode,
                                    false);
        }
    }
}
```

The Constructor

Your constructor needs to call `super()` passing in two parameters.

- `childElements`

  This is either a `String[]` or a `null`.

  This parameter is only important if you are writing an element handler for a custom element that is within the default BPEL name space.

  If your element handler is for an element that is not within the default BPEL name space, you set this `childElements` parameter to `null`.

  If your element handler is for an element that is within the default BPEL name space, you need to pass an array of element names. This list specifies the nested elements that, when processed, can add further information to the current element. This array is required because of the way the BPEL importer handles elements within the BPEL name space.

- `true_or_False`

  This parameter indicates whether you expect any nested elements to create further OVBPI nodes. You pass either true or false.

  By passing the value `true` you are allowing nested elements to create nested OVBPI nodes.

The call to `super()` is straightforward when dealing with a custom element that is not in the default BPEL name space. You set the first parameter to `null`, and if you want to allow nested activities to create nested nodes then you set the second parameter to `true`.

When dealing with a custom element that is within the BPEL name space, things change slightly and the parameters passed in the `super()` method are more important. Refer to Handling Elements in the BPEL Name Space on page 32 for more details about how the settings of the parameters in the call to `super()` affect the BPEL import.

isActivityElement()

This method is called to see whether your custom element corresponds to any real-world activity. That is, does this element mean that some real work happens at this point in the business process. If this BPEL element does correspond to real-world activity, then it means that the `handleElement()` method is expected to create one, or more, OVBPI node within the flow definition.

The `isActivityElement()` method must return true if this BPEL element is expected to create one or more OVBPI nodes, otherwise, it returns false.

handleElement()

The `BpelImporter` class invokes your `handleElement()` method. Your `handleElement()` method needs to provide the code that handles your XML element. Your code might create OVBPI nodes as required, by calling `BaseElementHandler` methods such as `addNewNode()` and `addArc()`.

The general layout of the `handleElement()` method contains calls to the `journal` class. The idea of the `journal` is to provide a way to handle messages that occur during the import. See The Journal on page 69 for more details about the use of the journal.

If your `handleElement()` code is handling an element that can contain nested elements, then your code needs to handle all the nested XML elements by invoking appropriate element handlers as required.

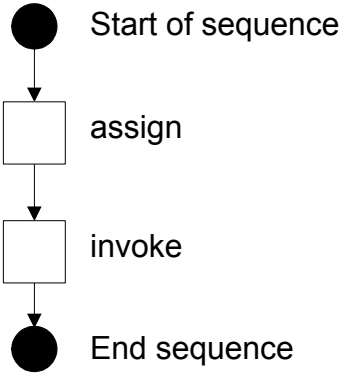
For example, suppose you are writing an element handler for a custom element that is similar to the BPEL `sequence` element. Your custom element is defined in your own `gb` name space, and the element is called `gb:myseq`. This `gb:myseq` element is a structural element in that it specifies that all elements within this sequence must be carried out in sequence, one after the other. So let's say you are processing the segment of XML as shown in Figure 1.

Figure 1 Sequence XML

```
<myseq>
    <assign>
    </assign>
    <invoke>
    </invoke>
</myseq>
```

Your `gb:myseq` element handler needs to produce the OVBPI junction node for the start of this sequence, then loop through all the nested elements and call their respective element handlers (which in this example add further nodes to the flow definition), and then add the OVBPI junction node for the close of this sequence. The resultant set of OVBPI nodes is shown in Figure 2 on page 23.

Figure 2 Resultant Structure Within the Flow Definition



When your element handler adds nodes to the OVBPI flow definition (as shown in Figure 2 on page 23) it does not need to specify any specific details about the actual node positions. This is because, once all the BPEL elements have been handled and the overall flow definition has been constructed, the BPEL importer goes through this flow definition and repositions all the nodes to produce a well formatted flow layout. So, to help you know where you are within the flow definition, your element handler is passed the following three parameters:

- `element`

  This is the actual XML element that is to be processed. This is an object of type `org.jdom.Element`.

- `precedingNode`

  This is the preceding OVBPI node in the current OVBPI flow definition. This is of type `com.hp.ov.bia.model.repository.api.utils.bpel.ExtraNodeInfo`.

  The preceding node is simply the node most recently added to the flow definition.

- `parentNode`

  This is the parent OVBPI node in the current OVBPI flow definition. This is of type `com.hp.ov.bia.model.repository.api.utils.bpel.ExtraNodeInfo`.

  The parent node is the node that defines the current hierarchy in which you are adding nodes. For example, when adding the `invoke` node as shown in Figure 2 on page 23, the parent node is the `gb:myseq` node, and

the preceding node is the `assign` node. That is, you add the `invoke` node specifying that it is within the parental structure of the `gb:myseq` node. You add nodes by calling the `addNewNode()` method (see addNewNode() on page 24).

addNewNode()

If your `handleElement()` method needs to create an OVBPI node, you can call the `BaseElementHandler` method `addNewNode()`.

The signature for the `addNewNode()` method is as follows:

```
protected ExtraNodeInfo addNewNode(
                                    NodeTypeEnum   nodeType,
                                    String         nodeName,
                                    ExtraNodeInfo  scopeNode,
                                    String         arrangement
                          )
throws RepositoryException
```

where:

- The method creates a new OVBPI node of the type specified, and returns this node within an object of type `com.hp.ov.bia.model.repository.api.utils.bpel.ExtraNodeInfo`.

  Among other things, this `ExtraNodeInfo` object contains the actual `Node` object.

- `nodeType`

  You pass in the type of node you are creating. You specify a constant from the `com.hp.ov.bia.model.repository.api.flow.NodeTypeEnum` class.

  For example:

  — `NodeTypeEnum.NODE_TYPE_JUNCTION` for a junction node.

  — `NodeTypeEnum.NODE_TYPE_ACTIVITY` for an activity node.

- `nodeName`

  You pass in the name of this node. See Setting the Node Name on page 31 for further details.

- `scopeNode`

  You pass in the parent node as this sets the scope of the node being added to the flow definition. That is, this dictates the hierarchical positioning of the node when the BPEL importer comes to reformatting the final flow definition.

- `arrangement`

  This parameter specifies the style of arrangement to use when laying out this node and its children. This style of arrangement is used when the BPEL importer is reformatting the final flow definition.

  The possible values for the arrangement parameter are defined as constants within the `ExtraNodeInfo` object. These values are:

  — `HORIZONTAL_COLLECTION`

  To start a new node collection where each node is laid out horizontally across the screen. For example, this arrangement is used when defining a `flow` junction node.

  — `VERTICAL_COLLECTION`

  To start a new node collection where each node is laid out vertically down the screen. For example, this arrangement is used when defining a `sequence` junction node.

  — `OFFSET_COLLECTION`

  You specify this arrangement when this node starts an offset vertical collection. For example, when you adding a node that starts a `fault` or `compensation` handler.

  — `END_OF_COLLECTION`

  You specify this arrangement value when adding the closing junction node of a collection.

  — `SINGLE_NODE`

  This arrangement is used when adding a single non-Junction node. For example, you would specify this arrangement when creating an activity node.

addArc()

If your `handleElement()` method needs to create an arc between two OVBPI nodes, you can use the `BaseElementHandler` method `addArc()`.

The signature for the `addArc()` method is as follows:

```
protected void addArc(
                        ExtraNodeInfo from,
                        ExtraNodeInfo to
                    )
throws RepositoryException
```

where this method draws an arc between the two OVBPI nodes that are passed in.

getChildHandler()

The `handleElement()` method needs to handle your element as well as any nested elements. That is, your element handler needs to decide whether it is simply going to ignore any nested elements, or iterate through the nested elements calling their respective element handlers.

By calling the `getChildren()` method on the `element` you are processing, you can then iterate through each of these nested elements. For each of these nested elements you can use the `BaseElementHandler` method `getChildHandler()` to get the appropriate element handler, and then invoke that element handle. The code outline is as follows:

```
List children = element.getChildren();
for (Iterator it = children.iterator(); it.hasNext();)
{
    Element child = (Element) it.next();
    ElementHandler handler = getChildHandler(child);

    ...invoke the element handler....
}
```

where:

- `element` is the element you are processing.
- Calling `getChildHandler()` returns the element handler for this nested element.

Ignoring All Nested Elements

If in your `handleElement()` method you decide to simply ignore all nested elements then you do not need to provide any code to iterate through them. It is up to your `handleElement()` method to handle your element and all nested elements. So if you do not handle the nested elements, then they are ignored.

Handling Nested Elements

The nested elements fall into two categories:

1. Activity elements

   These are elements where their `isActivityElement()` method returns `true`. This means that they either create a single node, or a hierarchy of nodes contained within an opening and closing junction node.

   For example, your BPEL may be as follows:

   ```
   <gb:myseq>
       <assign>
       </assign>
       <invoke>
       </invoke>
   </gb:myseq>
   ```

   where the nested elements `assign` and `invoke` each need to create further OVBPI nodes within the current sequence of nodes.

2. Non-Activity elements

   These are elements where their `isActivityElement()` method returns `false`. These elements do not correspond to further activity but may contain further details about the current element.

   For example, your BPEL may be as follows:

   ```
   <assign>
       <gb:description>
              Description about the assignment taking place.
       </gb:description>
   </assign>
   ```

   where the nested element `gb:description` contains the description of the current `assign` activity.

When iterating through the list of nested elements, your `handleElement()` code needs to check the return value of the `isActivityElement()` method. If the return value for the nested element is `true` then you need to invoke the element's handler, and then handle the new node that is created and returned to you. If the `isActivityElement()` method returns `false` then you can invoke the element's handler and ignore the return value.

The way that you handle nested elements that create nodes, depends on the type of element you are working with and the structure that it dictates.

Let's consider two different types of element, a sequential-style element and a parallel-style element, and consider their structure.

Sequential Style Element

If you are writing an element handler for an element that sets up a sequential set of activities, similar to the BPEL `sequence` element, your `handleElement()` method needs to follow these steps:

- Create the junction node that starts this sequence.

  Create the junction node that starts the sequence of steps, and link this junction node to any preceding node. The preceding node is passed to you as a parameter to this `handleElement()` method.

- Loop through all the nested elements.

  For each nested element, you call the appropriate element handler.

  You pass the node that started the sequence as the parent node. You pass the node that has just been created as the preceding node.

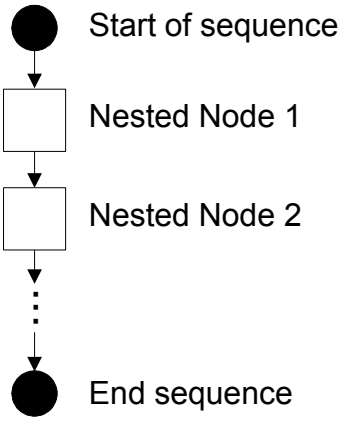
  Each nested element handler that corresponds to an activity, creates a new node and links this node to the preceding node.

- Create the junction node that ends this sequence.

  When all the nested elements have been handled, your element handler creates a closing junction node and links this to the previous node in the sequence.

This builds a flow structure as shown in Figure 3 on page 29.

Figure 3 A Sequence of Nodes



Refer to Handling a New Sequence Element on page 47 to see example code for a sequential element handler.

Parallel Style Element

If you are writing an element handler for an element that sets up a parallel set of activities, similar to the BPEL `flow` element, your `handleElement()` method needs to follow these steps:

- Create the junction node that starts this parallel set of activities.

  Create the junction node and link this junction node to any preceding node. The preceding node is passed as a parameter to this `handleElement()` method.

- Loop through all the nested elements.

  For each nested element, you call the appropriate element handler.

  You pass the node that started this parallel set of activities as both the parent node and the preceding node.

  Each nested element handler that corresponds to an activity, creates a new node and links this node to the preceding node.

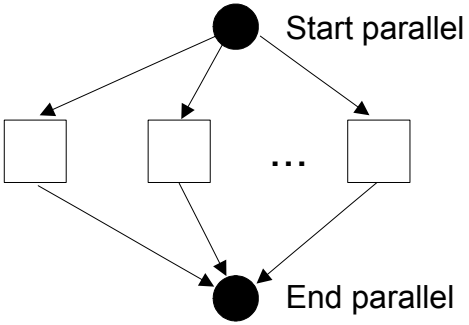
Your element handler needs to maintain a list of all the nested nodes that are created, as this allows you to create the necessary arcs when you come to create the closing junction node.

- Create the junction node that ends this set of parallel activities.

  When all the nested elements have been handled, your element handler creates a closing junction node. Your element handler then loops through the list of nested nodes that were created by the nested element handlers, and creates arcs between these and the closing junction node.

This builds a flow structure as shown in Figure 4 on page 30.

Figure 4 A Parallel Set of Nodes



Refer to Handling a New Parallel-Path Element on page 53 to see example code for a parallel-path element handler.

Setting the Node Name

If your element handler creates a node within the OVBPI flow definition, you need to assign this node a name. Your code can simply set the name by using a hard-coded string, or you can use the name generator that is configured for this import.

Your element handler inherits the object called `nameGenerator` from the `BaseElementHandler` class. This `nameGenerator` object gives you access to the name generator file that you specify when you run the BPEL importer.

Within your element handler you can set the node name with the following line of code:

```
String nodeName = nameGenerator.getName(element, "mytag.thing");
```

where:

- The `getName()` method looks in the name generator file and locates the block of entries that start with the text `mytag.thing`.
- You need to make sure that you have a corresponding block of entries within the selected name generator file.

For further explanation about the name generator file and its format, refer to the *OVBPI Integration Training Guide - Importing BPEL*.

Handling Elements in the BPEL Name Space

When you extend the BPEL importer and add a custom element handler, you expect the BPEL importer to call your element handler whenever it encounters your element within the XML. If your custom element is defined in a name space other than the default BPEL name space, then this is true. However, if your custom element is defined within the default BPEL name space, then the BPEL importer may not call your custom element handler.

When the BPEL importer encounters an element that is defined within the BPEL name space, the importer checks to ensure that the BPEL is valid before allowing this element to be processed. This validation checking is carried out within the `getChildHandler()` method of the `BaseElementHandler` class. That is, a call to the `getChildHandler()` method either returns the element handler for this element, or it returns a default element handler that simply ignores this element.

The validation that occurs within the `getChildHandler()` method is as follows:

- The `getChildHandler()` method checks to see if the BPEL 1.1 standard allows this element to be here within the XML.

  If the BPEL standard does allow this element to be here, then `getChildHandler()` returns the associated element handler, and the element can be processed.

- If the BPEL standard does not allow the element to be here, the `getChildHandler()` method then checks the current parent element to see if this element is specified in the list of valid child elements.

  The list of valid child elements is the first parameter of the `super()` call within the constructor of the parent element handler.

  If this element is specified as a valid child element, then `getChildHandler()` returns the associated element handler, and the element can be processed.

- If the element is not specified as a valid child element, the `getChildHandler()` method then checks to see if this element is an activity element, and whether or not the parent element has allowed nested activities.

To check that the parent element has allowed nested activities, the `getChildHandler()` method calls the `isActivityElement()` method on the parent element handler.

If the parent does allow nested elements to create activities, then the `getChildHandler()` method returns the associated element handler, and the element can be processed.

- If all the above tests fail, the element is ignored.

  If all the above tests have failed, the `getChildHandler()` method decides that this must be invalid BPEL, and therefore does not return the element's handler. Instead, the `getChildHandler()` method returns a default element handler, which simply ignores this element.

Compiling the Code

To compile your BPEL importer code and/or your element handler code, you need the following JAR files on your class path:

- *OVBPI-INSTALL-DIR*\java\bia-model-repository.jar
- *OVBPI-INSTALL-DIR*\nonOV\jdom\jdom.jar

An example compilation script might look as follows:

```
@echo off

set OVBPI_ROOT=C:/Program Files/HP OpenView/OVBPI

set CP=.
set CP=%CP%;%OVBPI_ROOT%/java/bia-model-repository.jar
set CP=%CP%;%OVBPI_ROOT%/nonOV/jdom/jdom.jar

javac -classpath "%CP%" com\customer\bpel\MyBpelImporter.java
```

Running your Importer

Once you have compiled your extension of the `BpelImporter` class, you can call this class from within the OVBPI Modeler, or from the standalone command line BPEL importer. Refer to Customizing the BPEL Scripts on page 11 for details about these options.

Let's focus here on using the command line to run the customized BPEL importer.

You need to edit your copy of the *OVBPI-INSTALL-DIR*`\bin\ovbpibpelimport.bat` script. Remember, you should make a copy of the `ovbpibpelimport.bat` script and call your copy `myovbpibpelimport.bat`. You then edit this copy of the script.

You edit your `myovbpibpelimport.bat` script as follows:

- Set the `IMPORTERCLASS` to point to your `MyBpelImporter` class.

  For example:

  ```
  set IMPORTERCLASS=com.customer.bpel.MyBpelImporter
  ```

- Set the `CLASSPATH` to include your `MyBpelImporter` class.

  Your `CLASSPATH` must be set to include both the path of your `MyBpelImporter` class and any name generator properties file you might specify when you invoke this script.

You then run your script, specifying your BPEL file, and any other options you require.

For example:

```
myovbpibpelimport -host localhost
                  -user admin
                  -password ovbpi
                  -replace true
                  -nameGeneratorFile MyNameGenerator.properties
                  -bpel ..\bpel\ExtraTokens.bpel
```

Code Examples

Let's walk through a number of code examples showing how to extend the BPEL importer.

Handling Node Description Text

Suppose your BPEL XML uses the custom element `gb:description` within standard BPEL elements, to add descriptive text about the action being performed.

For example, the `receive` element may contain descriptive text as shown in the following segment of XML.

```
<receive name="Initial Request"
         partnerLink="client"
         portType="com:insuranceSelectionPT"
         operation="SelectInsurance"
         variable="insuranceRequest"
         createInstance="yes" >
    <gb:description>
        This is the description for the Initial Request element.
    </gb:description>
</receive>
```

You decide to extend the OVBPI BPEL Importer to add a handler to handle this `gb:description` element. Your handler pulls out this descriptive text and sets it as the description text of the OVBPI node created by the `receive` element.

You need to extend the `BpelImporter` class to add your new element handler, and then write the actual element handler.

Extend the BPEL Importer Class

```
public class MyBpelImporter extends BpelImporter
{
    final protected Namespace gbNamespace
                = Namespace.getNamespace("gb", "http://www.mycompany.com");

    public MyBpelImporter()
    {
        super();

        // Handle <gb:description> element
        setElementHandler(gbNamespace,
                          "description",
                          new MyDescElementHandler());
    }
}
```

where:

- This code creates the class `MyBpelImporter` which is able to handle all the standard BPEL elements plus the `gb:description` element.
- The `gb` name space is set to match the URI `http://www.mycompany.com`. This must match the URI as specified within the `process` tag of the BPEL XML file.
- The `gb:description` element handler class is `MyDescElementHandler`.

The Element Handler Class

```
class MyDescElementHandler extends BaseElementHandler
{
    MyDescElementHandler()
    {
        super(null, false);
    }

    public boolean isActivityElement()
    {
        return false;
    }

    public ExtraNodeInfo handleElement(Element element,
                                       ExtraNodeInfo precedingNode,
                                       ExtraNodeInfo parentNode)
    throws RepositoryException
    {
        journal.startedHandling(element, precedingNode, parentNode, false);
        try
        {
            // Get the description text from the XML element
            String description = element.getText();
            if (description != null)
            {
                // Set this text as the description for the node.
                parentNode.flowNode.setDescription(description.trim());
            }

            return null;
        }
        finally
        {
           journal.completedHandling(element,precedingNode,parentNode,false);
        }
    }
}
```

where:

- The `MyDescElementHandler` class extends the `BaseElementHandler` class.

- The constructor contains the line of code:

  ```
  super(null, false);
  ```

  where:

  — You pass `null` for the first parameter as this element is not in the default BPEL name space.

  — `false` means that no nested elements create nodes.

- The `isActivityElement()` method returns false, because this `gb:description` element does not correspond to any real-world activity.

- The `handleElement()` method simply gets the text for this `gb:description` element, and then sets the description of the parent node to this text.

  The parent node is passed into the `handleElement()` method in the parameter `parentNode`. This `parentNode` is of type `com.hp.ov.bia.model.repository.api.utils.bpel.ExtraNodeInfo`.

  The `flowNode` property of the `ExtraNodeInfo` class for the `parentNode`, gives you access to the actual OVBPI node within the flow definition.

- The `handleElement()` method returns a `null` because it has not created a node.

Handling the Node Type

Suppose your BPEL XML uses the custom element `gb:type` within standard BPEL elements, to represent the type of work activity that is to be performed.

For example, you may have the following segment of XML:

```
<assign>
   <gb:type>123</gb:type>
</assign>
<reply name="Provide Quote"
       partnerLink="client"
       portType="com:insuranceSelectionPT"
       operation="SelectInsurance"
       variable="insuranceSelectionResponse">
       <gb:type>265</gb:type>
</reply>
```

Although this `gb:type` element can contain a range of values, the value `265` is used within your BPEL development environment to represent that this activity signals the end of the business process. You decide to extend your BPEL importer such that whenever an element contains a `gb:type` element of the value `265`, you map the parent element to an OVBPI end node.

You need to extend the `BpelImporter` class to add your new element handler, and then write the actual element handler.

Extend the BPEL Importer Class

```
public class MyBpelImporter extends BpelImporter
{
    final protected Namespace gbNamespace
                = Namespace.getNamespace("gb", "http://www.mycompany.com");

    public MyBpelImporter()
    {
        super();

        // Handle <gb:type> element
        setElementHandler(gbNamespace,
                          "type",
                          new MyNodeTypeElementHandler());
    }
}
```

where:

- This code creates the class `MyBpelImporter` which is able to handle all the standard BPEL elements plus the `gb:type` element.
- The `gb` name space is set to match the URI `http://www.mycompany.com`. This must match the URI as specified within the `process` tag of the BPEL XML file.
- The `gb:type` element handler class is `MyNodeTypeElementHandler`.

The Element Handler Class

```
class MyNodeTypeElementHandler extends BaseElementHandler
{
    MyNodeTypeElementHandler()
    {
        super(null, false);
    }

    public boolean isActivityElement()
    {
        return false;
    }

    public ExtraNodeInfo handleElement(Element element,
                                       ExtraNodeInfo precedingNode,
                                       ExtraNodeInfo parentNode)
    throws RepositoryException
    {
        journal.startedHandling(element, precedingNode, parentNode, false);
        try
        {
            String type = element.getText();
            if (type != null && type.trim().equals("265"))
            {
                parentNode.flowNode.setNodeType(NodeTypeEnum.NODE_TYPE_END);
            }
            return null;
        }
        finally
        {
           journal.completedHandling(element,precedingNode,parentNode,false);
        }
    }
}
```

where:

- The `MyNodeTypeElementHandler` class extends the `BaseElementHandler` class.
- The constructor contains the line of code:

  ```
  super(null, false);
  ```

  where:

  — You pass `null` for the first parameter as this element is not in the default BPEL name space.

  — `false` means that no child elements create nodes.
- The `isActivityElement()` method returns false, because this `gb:type` element does not correspond to any real-world activity.
- The `handleElement()` method gets the text for this `gb:type` element.

  If the text matches the string `265`, you set the node type of the parent node to be an OVBPI end node.

  The parent node is passed into the `handleElement()` method in the parameter `parentNode`. This `parentNode` is of type `com.hp.ov.bia.model.repository.api.utils.bpel.ExtraNodeInfo`.

  The `flowNode` property of the `ExtraNodeInfo` class for the `parentNode` gives you access to the actual OVBPI node within the flow definition.
- The `handleElement()` method returns `null` because it has not created a node.

Handling a New Element

Suppose your BPEL XML uses the custom element `gb:mytag` to represent some actual work that occurs within the business process.

For example, you may have the following segment of XML:

```
<assign>
   <copy>
        <from variable="insurance-A-Response" />
        <to variable="insuranceSelectionResponse" />
   </copy>
</assign>
<gb:mytag name="MyTag-1">
    This becomes an activity node.
</gb:mytag>
<reply name="Provide Quote"
       partnerLink="client"
       portType="com:insuranceSelectionPT"
       operation="SelectInsurance"
       variable="insuranceSelectionResponse">
</reply>
```

You want to be able to import this BPEL and produce an OVBPI activity node for the `gb:mytag` block of XML.

You need to extend the `BpelImporter` class to add your new element handler, and then write the actual element handler.

Extend the BPEL Importer Class

```
public class MyBpelImporter extends BpelImporter
{
    final protected Namespace gbNamespace
                = Namespace.getNamespace("gb", "http://www.mycompany.com");

    public MyBpelImporter()
    {
        super();

        // Handle <gb:mytag> element
        setElementHandler(gbNamespace,
                          "mytag",
                          new MyTagElementHandler());
    }
}
```

where:

- This code creates the class `MyBpelImporter` which is able to handle all the standard BPEL elements, plus the `gb:mytag` element.
- The `gb` name space is set to match the URI `http://www.mycompany.com`. This must match the URI as specified within the `process` tag of the BPEL XML file.
- The `gb:mytag` element handler class is `MyTagElementHandler`.

The Element Handler Class

```
class MyTagElementHandler extends BaseElementHandler
{
    MyTagElementHandler()
    {
        super(null, false);
    }

    public boolean isActivityElement()
    {
        return true;
    }

    public ExtraNodeInfo handleElement(Element element,
                                       ExtraNodeInfo precedingNode,
                                       ExtraNodeInfo parentNode)
    throws RepositoryException
    {
        journal.startedHandling(element, precedingNode, parentNode, false);
        try
        {
            // -1-
            // Get the node name using the name generator.
            String nodeName = nameGenerator.getName(element, "mytag.node");

            // -2-
            ExtraNodeInfo activityNode = addNewNode(
                                             NodeTypeEnum.NODE_TYPE_ACTIVITY,
                                             nodeName,
                                             parentNode,
                                             ExtraNodeInfo.SINGLE_NODE
                                         );

            // -3-
            addArc(precedingNode, activityNode);
```

```
            // Note the fact that you've created a new node...
            journal.createdNode(
                                    activityNode,
                                    element,
                                    Journal.QUALIFIER_NODE
            );

            // -4-
            List children = element.getChildren();
            for (Iterator it = children.iterator(); it.hasNext();)
            {
                 Element child = (Element) it.next();
                 ElementHandler handler = getChildHandler(child);
                 handler.handleElement(child, activityNode, activityNode);
            }

            // -5-
            // Return the OVBPI activity node you've just created
            return activityNode;
        }
        finally
        {
           journal.completedHandling(element,precedingNode,parentNode,false);
        }
    }
}
```

where:

- The `MyTagElementHandler` class extends the `BaseElementHandler` class.
- The constructor contains the line of code:

  `super(null, false);`

  where:

  — You pass `null` for the first parameter as this element is not in the default BPEL name space.

  — `false` means that no child elements create nodes.
- The `isActivityElement()` method returns true, because this `gb:mytag` element does correspond to real-world activity.
- Step **1**: Uses the `nameGenerator` to get the text for the name of this node.

  It looks in the name generator file (as specified when you run the BPEL importer) for the block of entries that start with the prefix `mytag.node`.

- Step **2**: Calls the `addNewNode()` method to add an activity node.

  The code passes in:

  — The node type (`NODE_TYPE_ACTIVITY`)

  — The node name as returned by the call to the `nameGenerator`

  — The parent node within which you are placing this new node (`parentNode`)

  — The type of structure the new node is part of.

  The constant `SINGLE_NODE` says that this is just a single OVBPI node and no child nodes are expected.

  The third parameter in the `addNewNode()` method defines the scope of the node to be added, and not the node it is to be linked to. By passing the parent node this allows the importer to build the correct layout of the nodes within the OVBPI flow definition.

- Step **3**: Calls `addArc()` to link in this newly created node.

  The `addNewNode()` method returns the node that it has just created. The `addArc()` method builds an arc from the preceding node to this new node.

- Step **4**: Loops through and handles all nested elements.

  The code assumes that all nested element handlers do not create further nodes. This is just an assumption within this code example. That is why the call to the nested element handler does not do anything with the return value. This allows nested elements to add further information to the current `gb:mytag` node, such as (for example) a description.

- Step **5**: Returns the activity node that is created by this element handler.

Handling a New Sequence Element

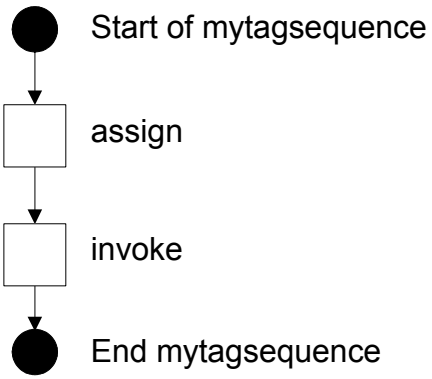
Suppose your BPEL XML uses the custom element `gb:mytagsequence` to correspond to a series of activities that occur in sequence within a business process.

For example, you may have the following segment of XML:

```
<gb:mytagsequence>
    <assign>
    </assign>
    <invoke>
    </invoke>
</gb:mytagsequence>
```

You want to be able to import this BPEL and produce a sequence of nodes as shown in Figure 5.

Figure 5 mytagsequence Set of Nodes



You need to extend the `BpelImporter` class to add your new element handler, and then write the actual element handler.

Extend the BPEL Importer Class

```
public class MyBpelImporter extends BpelImporter
{
    final protected Namespace gbNamespace
                = Namespace.getNamespace("gb", "http://www.mycompany.com");

    public MyBpelImporter()
    {
        super();

        // Handle <gb:mytagsequence> element
        setElementHandler(gbNamespace,
                          "mytagsequence",
                          new MyTagSequenceElementHandler());
    }
}
```

where:

- This code creates the class `MyBpelImporter` which is able to handle all the standard BPEL elements, plus the `gb:mytagsequence` element.
- The `gb` name space is set to match the URI `http://www.mycompany.com`. This must match the URI as specified within the `process` tag of the BPEL XML file.
- The `gb:mytagsequence` element handler class is `MyTagSequenceElementHandler`.

The Element Handler Class

```
class MyTagSequenceElementHandler extends BaseElementHandler
{
    MyTagSequenceElementHandler()
    {
        super(null, true);
    }

    public boolean isActivityElement()
    {
        return true;
    }
```

```
public ExtraNodeInfo handleElement(Element element,
                                   ExtraNodeInfo precedingNode,
                                   ExtraNodeInfo parentNode)
throws RepositoryException
{
    // Say you are handling this element...
    journal.startedHandling(element, precedingNode, parentNode, false);

    try
    {
        // -1-
        // Get the node name using the name generator.
        String nodeName = nameGenerator.getName(
                                                element,
                                                "mytagsequence.start"
                        );

        // -2-
        // Create the junction node, and declare how the
        // child nodes are to be organized.
        ExtraNodeInfo startNode = addNewNode(
                                    NodeTypeEnum.NODE_TYPE_JUNCTION,
                                    nodeName,
                                    parentNode,
                                    ExtraNodeInfo.VERTICAL_COLLECTION
                                );

        // -3-
        // Add an arc from the preceding node to this junction node.
        addArc(precedingNode, startNode);

        // Note the fact that you've created a new node...
        journal.createdNode(
                            startNode,
                            element,
                            Journal.QUALIFIER_START_NODE
        );

        // -4-
        // Keep track of the last node in this sequence of child nodes.
        // Start with the junction node you have just created.
        ExtraNodeInfo lastNode = startNode;

        // Handle each child...adding in nodes as necessary.
        List children = element.getChildren();
        for (Iterator it = children.iterator(); it.hasNext();)
        {
             Element child = (Element) it.next();
             ElementHandler handler = getChildHandler(child);
             boolean isActivity = handler.isActivityElement();
```

```
                if (isActivity)
                {
                    lastNode = handler.handleElement(
                                                        child,
                                                        lastNode,
                                                        startNode
                                );
                }
                else
                {
                    handler.handleElement(child, startNode, startNode);
                }
            }

            // -5-
            // Now create the final junction node.

            // Get the node name using the name generator.
            nodeName = nameGenerator.getName(element, "mytagsequence.end");

            // Create the closing juntion node.
            ExtraNodeInfo endJNode = addNewNode(
                                        NodeTypeEnum.NODE_TYPE_JUNCTION,
                                        nodeName,
                                        startNode,
                                        ExtraNodeInfo.END_OF_COLLECTION
                                    );
            addArc(lastNode, endJNode);

            journal.createdNode(
                                endJNode,
                                element,
                                Journal.QUALIFIER_END_NODE
            );

            // -6-
            // Return the closing junction node in the sequence.
            return endJNode;
        }
        finally
        {
           journal.completedHandling(element,precedingNode,parentNode,false);
        }
    }
}
```

where:

- The `MyTagSequenceElementHandler` class extends the `BaseElementHandler` class.

- The constructor contains the line of code:

  `super(null, true);`

  where:

  — You pass `null` for the first parameter as this element is not in the default BPEL name space.

  — `true` means that nested elements are able to create nodes.

- The `isActivityElement()` method returns true because this `gb:mytagsequence` element does correspond to real-world activity. Indeed, this node is going to create a series of nodes.

- Step **1**: Uses the `nameGenerator` to get the text for the name of this node.

  It looks in the name generator file (as specified when you run the BPEL importer) for the block of entries that start with the prefix `mytagsequence.start`.

- Step **2**: Calls the `addNewNode()` method to add a junction node.

  The node type is set to `NODE_TYPE_JUNCTION`. The constant `VERTICAL_COLLECTION` says that this junction node is the start of a set of nodes which are to be formatted vertically down the page.

  The third parameter (`parentNode`) defines the scope of the node to be added, and not the node it is to be linked to. By passing the parent node, this allows the importer to build the correct layout of the nodes within this sequence.

- Step **3**: Calls `addArc()` to link in this newly created junction node.

  The `addNewNode()` method returns the junction node that it has just created. The `addArc()` method builds an arc from the preceding node to this new junction node.

- Step **4**: Loops through and handles all nested elements.

  The code loops through the nested element handlers. It checks to see if the nested element handler corresponds to an activity element.

  — If the nested element handler does correspond to an activity element?

  Then this nested element handler creates one or more OVBPI nodes. Element handlers that create nodes also link them to the preceding node that is passed to them when they are invoked.

So when invoking this nested handler, you need to pass through the most recently created node and set this as the preceding node. You also pass through the junction node that you created at the start of this sequence and set this as the parent node. The start sequence junction node defines the scope of this set of nodes. That is, all nested nodes are nested within this sequence. You need to set the preceding and parent nodes correctly as this allows the BPEL importer to correctly format the layout of the final OVBPI flow definition.

— If the nested element handler does not correspond to an activity element?

Then this nested element handler may set further details about the parent node. The parent node is the junction node that you created at the start of this sequence. So you invoke the nested element handler passing this starting junction node as both the parent and the preceding node.

The variable `lastNode` is used to maintain a pointer to the last node in this sequence.

- Step **5**: Creates the final junction node.

  The code looks in the name generator file (as specified when you run the BPEL importer) for the block of entries that start with the prefix `mytagsequence.end`.

  The code then calls `addNewNode()` to create the closing junction node for this sequence. An arc is then created from the previous node in the sequence to this closing junction node.

- Step **6**: Returns the closing junction node which is the last node in this sequence.

Handling a New Parallel-Path Element

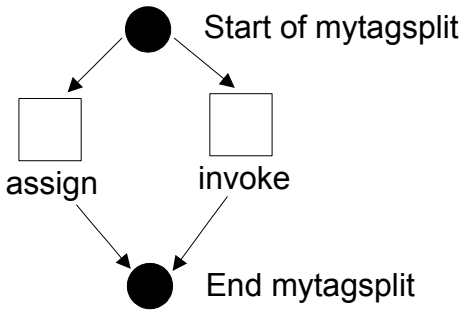
Suppose your BPEL XML uses the custom element `gb:mytagsplit` to represent a number of activities that occur in parallel within a business process.

For example, you may have the following segment of XML:

```
<gb:mytagsplit>
    <assign>
    </assign>
    <invoke>
    </invoke>
</gb:mytagsplit>
```

You want to be able to import this BPEL and produce a hierarchy of nodes as shown in Figure 6.

Figure 6 mytagsplit Set of Nodes



You need to extend the `BpelImporter` class to add your new element handler, and then write the actual element handler.

Extend the BPEL Importer Class

```
public class MyBpelImporter extends BpelImporter
{
    final protected Namespace gbNamespace
                = Namespace.getNamespace("gb", "http://www.mycompany.com");

    public MyBpelImporter()
    {
        super();

        // Handle <gb:mytagsplit> element
        setElementHandler(gbNamespace,
                          "mytagsplit",
                          new MyTagSplitElementHandler());
    }
}
```

where:

- This code creates the class `MyBpelImporter` which is able to handle all the standard BPEL elements, plus the `gb:mytagsplit` element.
- The `gb` name space is set to match the URI `http://www.mycompany.com`. This must match the URI as specified within the `process` tag of the BPEL XML file.
- The `mytagsequence` element handler class is `MyTagSplitElementHandler`.

The Element Handler Class

```
class MyTagSplitElementHandler extends BaseElementHandler
{
    MyTagSplitElementHandler()
    {
        super(null, true);
    }


    public boolean isActivityElement()
    {
        return true;
    }
```

```
public ExtraNodeInfo handleElement(Element element,
                                   ExtraNodeInfo precedingNode,
                                   ExtraNodeInfo parentNode)
throws RepositoryException
{
     // Say you are handling this element...
     journal.startedHandling(element, precedingNode, parentNode, false);
     try
     {
         // -1-
         // Get the node name using the name generator.
         String nodeName = nameGenerator.getName(
                                                  element,
                                                  "mytagsplit.start"
                           );

         // -2-
         // Create the junction node and declare
         // how the child nodes are to be organized.
         ExtraNodeInfo startNode = addNewNode(
                                      NodeTypeEnum.NODE_TYPE_JUNCTION,
                                      nodeName,
                                      parentNode,
                                      ExtraNodeInfo.HORIZONTAL_COLLECTION
                                   );

         // -3-
         // Draw the arc from the preceding node to the new junction node
         addArc(precedingNode, startNode);


         // Note the fact that you've created a new node...
         journal.createdNode(
                              startNode,
                              element,
                              Journal.QUALIFIER_START_NODE
         );
```

```
// -4-
// Keep track of all the child nodes that get created by
// the various child element handlers you are about to call.
ArrayList nodesToJoin = new ArrayList();

List children = element.getChildren();
for (Iterator it = children.iterator(); it.hasNext();)
{
    Element child = (Element) it.next();
    ElementHandler handler = getChildHandler(child);

    ExtraNodeInfo childNode = handler.handleElement(
                                                    child,
                                                    startNode,
                                                    startNode
                                );
    if (childNode != null)
    {
        nodesToJoin.add(childNode);
    }
}

// -5-
// Create the end junction node.
nodeName = nameGenerator.getName(element, "mytagsplit.end");

ExtraNodeInfo endJNode = addNewNode(
                                NodeTypeEnum.NODE_TYPE_JUNCTION,
                                nodeName,
                                startNode,
                                ExtraNodeInfo.END_OF_COLLECTION
                        );

// -6-
// Handle the situation where there are no child nodes.
if (nodesToJoin.isEmpty())
{
    // Note that a warning occurred.
    journal.message(
             Journal.LEVEL_WARNING_MISSING,
            "No sub-elements found within: " + getPath(element),
             element
    );

    // Join the start to the end.
    nodesToJoin.add(startNode);
}
```

```
            // -7-
            // Loop through all the nodes adding an arcs from
            // each node to the end junction node you just created.
            for (Iterator it = nodesToJoin.iterator(); it.hasNext();)
            {
                // Add an arc from the node to your end node
                ExtraNodeInfo n = (ExtraNodeInfo) it.next();
                addArc(n, endJNode);
            }

            // Note that you have created the final node for this element
            journal.createdNode(
                                endJNode,
                                element,
                                Journal.QUALIFIER_END_NODE
            );

            // -8-
            // Return the last node in the collection
            return endJNode;

        }
        finally
        {
           journal.completedHandling(element,precedingNode,parentNode,false);
        }
    }
}
```

where:

- The `MyTagSplitElementHandler` class extends the `BaseElementHandler` class.
- The constructor contains the line of code:

  `super(null, true);`

  where:

  — You pass `null` for the first parameter as this element is not in the default BPEL name space.

  — `true` means that nested elements are able to create nodes.
- The `isActivityElement()` method returns true because this `gb:mytagsplit` element does correspond to real-world activity. Indeed, this node is going to create a number of nodes.

- Step **1**: Uses the `nameGenerator` to get the text for the name of this node.

  It looks in the name generator file (as specified when you run the BPEL importer) for the block of entries that start with the prefix `mytagsplit.start`.

- Step **2**: Calls the `addNewNode()` method to add a junction node.

  The node type is set to `NODE_TYPE_JUNCTION`. The constant `HORIZONTAL_COLLECTION` says that this junction node is the start of a set of nodes which need to be formatted horizontally across the page.

  The third parameter (`parentNode`) defines the scope of the node to be added, and not the node it is to be linked to. By passing the parent node this allows the importer to build the correct layout of the nodes within this `gb:mytagsplit` set.

- Step **3**: Calls `addArc()` to link in this newly created junction node.

  The `addNewNode()` method returns the junction node that it has just created. The `addArc()` method builds an arc from the preceding node to this new junction node.

- Step **4**: Loops through and handles all nested elements.

  When invoking each nested element handler, the code passes through the newly created `gb:mytagsplit` junction node as both the preceding node and the parent node. This allows the nested element handler to create its node and link this to the preceding node (the starting `gb:mytagsplit` junction node), and this nested node is created within the scope of this `gb:mytagsplit` junction node.

  If the nested element handler does return a newly created node, this node is added to an array. The idea is that you loop through all the nested element handlers and keep a track of all the nodes that are created. Each of these nodes will have been linked to the `gb:mytagsplit` junction node by their respective element handlers. Once you have the full list of nodes created, you can then create the closing junction node and create links to this from all the nested nodes.

- Step **5**: Creates the final junction node.

  The code looks in the name generator file (as specified when you run the BPEL importer) for the block of entries that start with the prefix `mytagsplit.end`.

  The code then calls `addNewNode()` to create the closing junction node for this sequence.

- Step **6**: Handles the situation where no nested nodes were created.

  If after looping through all the nested element handlers, no nested nodes were created, then you issue a warning message and then add the start node (the junction node you created when you first handled this `gb:mytagsplit` element) to your node array.

- Step **7**: Link all the nested nodes.

  At this stage the starting junction node has been created and all nested nodes have been created with arcs joining from this starting junction node. You now need to loop through the list of nested nodes that have been created and create arcs from them to your closing junction node.

- Step **8**: Returns the closing junction node which is the last node in this set.

3 Custom CLI Main Class

When you run the `ovbpibpelimport.bat` script, it runs the standalone Java class `BpelImporterCli`. This standalone class reads certain options from the command line, and then issues the necessary Repository API calls to import the specified BPEL file. It then saves the resultant flow definition into the Model Repository.

This chapter explains how to write your own standalone class to import a BPEL process. This chapter also explains how you can modify the resultant OVBPI flow definition after the initial import from BPEL.

Importing BPEL

To write an application that imports a BPEL business process into the OVBPI Model Repository is very easy. The basic steps are as follows:

1. Instantiate a BPEL importer

   Create an instance of the `BpelImporter` class, or a subclass.

2. Select a name generator

   Create an instance of a name generator, and then tell your BPEL importer (created in step 1) to use it.

3. Set any import options

   Set any options within the BPEL importer (created in step 1). These options include things such as vertical page border size, row gap, etc.

4. Set up the Journal

   Create an instance of a journal and then tell your BPEL importer (created in step 1) to use it.

5. Import the BPEL

   Tell your BPEL importer to import the BPEL and create a flow definition.

6. Save the flow definition

   Now you have the flow definition you can save this into the OVBPI Model Repository.

Let's now go through each of these steps and consider them in more detail...

Instantiate a BPEL Importer

You need to create an instance of the `com.hp.ov.bia.model.repository.api.utils.bpel.BpelImporter` class, or a subclass.

In Chapter 2 you learnt how to extend this `BpelImporter` class to produce your own BPEL importer subclass. So you can create an instance of the default `BpelImporter` class, or your own subclass with its custom element handlers.

Here is an example code segment that creates an instance of a BPEL importer:

```
String bpelImporterClassName = "com.customer.bpel.MyBpelImporter";

final Class bpelImporterClass;
BpelImporter importerInstance = null;

try
{
    bpelImporterClass = Class.forName(bpelImporterClassName);
    importerInstance = (BpelImporter) bpelImporterClass.newInstance();
}
catch (ClassNotFoundException e)
{
    System.out.println("Unable to find class [" + bpelImporterClassName +"]");
    return;
}
catch (Throwable e)
{
    System.out.println("Could not create instance of class [" +
                        bpelImporterClassName + "]");
}
```

where you set the `bpelImporterClassName` variable to be either the default `BpelImporter` class, or your extension of this class.

Now that you have created an instance of a `BpelImporter` you can set various properties within it. Properties such as the name generator that it is to use (see Select a Name Generator on page 63), the various import options (see Set any Import Options on page 65), etc.. Once you have set up these properties you can tell the `BpelImporter` instance to actually perform the BPEL import (see Import the BPEL on page 67).

Select a Name Generator

Now that you have created an instance of a BPEL importer you can set the name generator class, and name generator properties file, that are to be used.

You create an instance of the `com.hp.ov.bia.model.repository.api.utils.bpel.NameGenerator` class, or a subclass. You then set the name generator properties file that this class is to use. Once you have instantiated your name generator class you can tell your BPEL importer to use it.

Configuring the name generator for your BPEL importer is optional. If you do not configure a name generator, then the BPEL importer uses the default `NameGenerator` class with name generator properties file `FullNames.properties` from the *`OVBPI-INSTALL-DIR`*`\msg\bia\BPELImporter_NameGenerator` directory.

If you want to see how to produce your own subclass of the `NameGenerator` class then refer to the *OVBPI Integration Training Guide - Importing BPEL* for details.

Here is an example code segment that creates an instance of a custom name generator and then tells the BPEL importer to use it:

```
// Create an instance of a name generator

String nameGeneratorClassName = "com.customer.bpel.MyNameGenerator";

final Class nameGeneratorClass;
final NameGenerator nameGeneratorInstance;

try
{
    nameGeneratorClass = Class.forName(nameGeneratorClassName);
    nameGeneratorInstance = (NameGenerator) nameGeneratorClass.newInstance();

    // Set the name generation file.
    //
    nameGeneratorInstance.loadInstructions("MyNameGenerator");
}
catch (ClassNotFoundException e)
{
    System.out.println("Unable to find class ["+nameGeneratorClassName+"]");
    return;
}
catch (Throwable e)
{
    System.out.println("Could not create instance of class [" +
                       nameGeneratorClassName + "]");
    return;
}

// Now set this within the BPEL importer
importerInstance.setNameGenerator(nameGeneratorInstance);
```

where:

- The name generator being instantiated is the custom class `com.customer.bpel.MyNameGenerator`.

- The name generator instance is configured to use the name generator properties file `MyNameGenerator.properties`.

  This name generator properties file is loaded as a Java resource bundle and so this file needs to be within your classpath. The `.properties` suffix is added when the Java resource bundle goes to locate the file.

- The BPEL importer instance is then configured to use this name generator instance.

  The `setNameGenerator()` method sets this name generator instance as the one to use.

Set any Import Options

The BPEL importer has a set of options that you can configure. These options affect the final layout of the flow definition that is produced by the importer.

The BPEL importer maintains its options within an object of class `com.hp.ov.bia.model.repository.api.utils.bpel.Options`.

The `getOptions()` method in the `BpelImporter` class allows you to get the options object. You can then set the options however you require.

Here is an example code segment that gets access to the BPEL importer's options object, and sets some of these options:

```
// Get the reference to the bpel import options.
Options bpelOptions = importerInstance.getOptions();

// Set some options for the import
bpelOptions.setColumnGap(10);
bpelOptions.setRowGap(30);
bpelOptions.setHorizontalPageBorder(30);
bpelOptions.setVerticalPageBorder(30);
bpelOptions.setLayoutGridSize(100, 60);
bpelOptions.setMinimumPageWidth(1122);
bpelOptions.setMinimumPageHeight(792);
bpelOptions.setIncludeCompensationHandlers(true);
bpelOptions.setIncludeFaultHandlers(true);
```

The options you can set should look familiar to you. You can set a subset of these when you import a BPEL file from within the OVBPI Modeler, and you can set all of these when you import a BPEL file using the `ovbpibpelimport` script.

Set up the Journal

When the BPEL importer is processing the BPEL elements, it logs information through the following interface:

```
com.hp.ov.bia.model.repository.api.utils.bpel.Journal
```

Their is a default implementation of the `Journal` interface called `com.hp.ov.bia.model.repository.api.utils.bpel.DefaultJournal`.

Setting the journal for your BPEL importer is optional. If you do not configure a journal, then the BPEL importer defaults to the `DefaultJournal` class.

The default journal provides a way for the BPEL importer to log warnings and errors that may occur during the import, and have these displayed to the user afterwards. You can extend the default journal and use it to further customize your BPEL importer. Refer to The Journal on page 69 for more information about configuring a journal.

Here is an example code segment that gets a default journal and then sets this to be the one used by your BPEL importer:

```
DefaultJournal journal = new DefaultJournal();
importerInstance.setJournal(journal);
```

To display the message after a BPEL import, you can call the `getText()` method of the `DefaultJournal`. For example:

```
System.out.print(journal.getText(false));
```

where the `getText()` method returns any messages as a single string. The parameter to the `getText()` method specifies whether or not you want the return string formatted as HTML. Setting the parameter to `false` returns a plain-text string.

Import the BPEL

To import a BPEL file you call the `importBpelFile()` method on your BPEL importer instance. This method takes a single parameter which is the import file, as a `File` object.

Here is an example code segment that imports a BPEL file:

```
File bpelFile = new File(filename);

try
{
    importerInstance.importBpelFile(bpelFile);
}
catch (RepositoryException e)
{
    System.out.println("***Failed to import file [" + bpelFile.getPath() +
                       "\nMessage: [" + e.getLocalizedMessage() +"]");
    return;
}

// After the import you can output the journal messages.
System.out.print(journal.getText(false));
```

where the variable *filename* holds the actual file name being imported.

Save the Flow Definition

After the BPEL has been imported, the flow definition is stored within your BPEL importer instance. You call the `getFlowDefinition()` method to get the flow definition.

Once you have the flow definition, you can connect to the Model Repository and save this flow definition. You may also check to see if a flow definition of this name already exists within the Model Repository, and if so, use the repository's `assumeIdentity()` method to supersede this flow definition. (Refer to the *OVBPI Integration Training Guide - Repository API* for further discussion about superseding flows and avoiding duplicate definitions.)

Here is an example code segment that gets the resultant flow definition from the BPEL importer, connects to the Model Repository, and then saves the flow definition into the repository:

```
Flow flow = importerInstance.getFlowDefinition();

Repository repository = null;
try
{
    repository = RepositoryFactory.accessRemoteRepository(hostName,
                                                          "44000",
                                  RepositoryFactory.DEFAULT_RMI_OBSERVER_PORT,
                                  new RepositoryCredentials(userName, password)
                 );
}
catch (Exception e)
{
    System.out.println("Cannot connect! [" + e.getLocalizedMessage() + "]");
    return;
}

repository.saveModel(flow);
```

The Journal

The BPEL API provides the interface `com.hp.ov.bia.model.repository.api.utils.bpel.Journal`.

The `Journal` interface defines a set of methods that are called by the BPEL importer during the import of a BPEL file. These methods are as follows:

- `startedImport()` – Called when the importer starts the import.
- `completedImport()` – Called when the importer has finished the import.
- `startedHandling()` – Called when the importer starts to handle a BPEL element.
- `completedHandling()` – Called when the importer has completed the handling of a BPEL element.
- `createdNode()` – Called when the importer has created a new node.
- `message()` – Called when the importer wants to log a message.

The idea is that the BPEL importer calls the journal while it is processing the BPEL input file. It is up to the implementation of the `Journal` interface as to how it behaves when called by the BPEL importer.

The BPEL API provides two implementations of the `Journal` interface, as follows:

- `DefaultJournal`

  This implementation records basic details about the import, such as any error or warning messages that have occurred. You can then output these as text after the import has completed.

- `VerboseJournal`

  This implementation records the same basic information as the `DefaultJournal` class, as well as maintaining a list of the nodes created. The class also maintains a summary of the number of nodes created, how many nodes of the different types were created, and how many BPEL elements were processed.

Using the Default Journal

The `DefaultJournal` class supports a number of methods. Let's consider the main methods available and show some code examples.

Element Handling

All the standard element handlers supplied with OVBPI make calls to announce when they have started and completed the handling of an element.

The format of these calls is as follows:

```
journal.startedHandling(element, precedingNode, parentNode, false);

journal.completedHandling(element, precedingNode, parentNode, false);
```

where:

- The `element`, `precedingNode` and `parentNode` are the parameters as passed into the element handler. Refer to Chapter 2 for more details about element handlers.
- The fourth parameter indicates whether or not this element handler has ignored this element.

  A value of `true` says that this element handler has ignored this element.

Log Messages

If an element handler encounters some form of error or warning, these are logged to the journal by using the `message()` method. The element handler may also log information messages using this method.

The parameters you pass to the `message()` method are as follows:

- The severity of the message to be logged.

  This is one of the defined constants:

  ```
  Journal.LEVEL_INFO
  Journal.LEVEL_ERROR
  Journal.LEVEL_WARNING_DISCARDING
  Journal.LEVEL_WARNING_EXTRA
  Journal.LEVEL_WARNING_MISSING
  Journal.LEVEL_INTERNAL_INFO
  ```

- The text of the message to be logged.
- The JDOM XML element for which this message is about.

For example,

```
journal.message(
                Journal.LEVEL_WARNING_MISSING,
                "No sub-elements found within: " + element.getName(),
                element
);
```

where this message is being logged at the level of `LEVEL_WARNING_MISSING`.

After the BPEL has been imported, you can get the messages from the journal in a number of ways. The `getText()` method returns all the messages, formatted into a single string. The `getMessages()` method allows you to get the set of messages for a specified level.

For example:

```
ArrayList al = journal.getMessages(Journal.LEVEL_ERROR)
```

where this call to `getMessages()` returns an array list containing all the error messages that have occurred.

Node Creation

Whenever an element handler creates an OVBPI node, it calls the `createdNode()` method on the journal.

The `createdNode()` methods takes three parameters, as follows:

- The node just created.

  This is of type `ExtraNodeInfo`.
- The JDOM XML element.
- A string qualifier.

  This is a string that represents the type of node being created. The defined string values are:

  — `Journal.QUALIFIER_NODE`

  This says that the node is a single activity node.

— `Journal.QUALIFIER_START_NODE`

  This says that the node is the start of a group of nodes.

— `Journal.QUALIFIER_END_NODE`

  This says that the node is the end of a group of nodes.

— `Journal.QUALIFIER_LOOPBACK_NODE`

  This is used when processing an element such as `while`.

  This says that the node is the one that carries the arc back up from the end of the loop to the start.

For example:

```
journal.createdNode(activityNode, element, Journal.QUALIFIER_NODE);
```

where this calls the `createdNode()` method to say that this `activityNode` has just been created.

Using a Custom Journal

The BPEL importer and the element handlers that are invoked all make calls to the journal as the BPEL is being processed. The default journal simply acts as a way of storing message text which can then be displayed after the import has completed. However, there is nothing stopping you from writing your own journal implementation.

You could implement your own journal class to store any information that might help you after the import. For example, you might keep a list of all the nodes as they are being created. You could then use this to update or remove certain nodes from the flow definition, before then saving the flow definition into the Model Repository.

To provide your own journal class you can simply extend the `DefaultJournal` class. It is then up to you to override whichever methods suit your needs.

Let's consider an example...

Removing Nodes from the Flow Definition

When you import a BPEL process, the resultant OVBPI flow definition always starts with two junction nodes. The first is a start process junction node, and this then connects to a start sequence junction node. You then have the rest of the actual flow. At the end of the flow definition you finish with two junctions nodes, the closing sequence junction node and the closing process junction node.

These four nodes are all created by the handling of the `process` element. The `process` element starts the process, hence you get the start process junction node. The `process` element then contains a `sequence` element as its only nested element. Hence you get the creation of the opening `sequence` node. The rest of the OVBPI flow is created by elements nested within this `sequence` element. When the BPEL importer reaches the end of the BPEL file it encounters the end of the outer `sequence` element followed by the end of the enclosing `process` element. Hence the BPEL importer creates the final two junction nodes of the flow.

Let's suppose you want to automatically remove these first two and last two junction nodes from the flow definition whenever you import a BPEL process.

By providing your own journal, you can keep track of the OVBPI nodes that are created for the `process` element. Your journal can hold these in an array. When the import has completed, you can then retrieve this array of nodes from your journal, and go about removing them from the flow definition.

The code for your custom journal might look as follows:

```
package com.customer.bpel.cli;

import com.hp.ov.bia.model.repository.api.utils.bpel.DefaultJournal;
import com.hp.ov.bia.model.repository.api.utils.bpel.ExtraNodeInfo;
import org.jdom.Element;

import java.util.ArrayList;


// -1-
public class MyJournal extends DefaultJournal
{
    // -2-
    protected ArrayList nodesToDelete = new ArrayList();
```

```
    // -3-
    public MyJournal()
    {
        super();
    }


    public void createdNode(ExtraNodeInfo nodeInfo,
                            Element element,
                            String qualifier)
    {
        // If this node has been created by the <process> element
        // then add it to the list.

        // -4-
        if ("process".equals(element.getName()))
        {
            nodesToDelete.add(nodeInfo.flowNode.getId());
        }
    }


    // -5-
    public ArrayList getNodesToDelete()
    {
        return nodesToDelete;
    }
}
```

where:

- Step **1**: This `MyJournal` class extends the `DefaultJournal` class.
- Step **2**: You declare a variable to hold the list of nodes.
- Step **3**: Your constructor just calls `super()`.
- Step **4**: If the current element being handled is the `process` element then store the node ID of the OVBPI node just created.
- Step **5**: This provides a method that your standalone importer can call to retrieve the array of node IDs.

Now that you have your journal class, you can use this within your standalone BPEL importer application.

The code outline for your standalone BPEL importer application is as follows:

```
public static void main(String[] args)
{
    // Load the BpelImporter class into the variable importerInstance.
    ...

    // Load the name generator class and properties file.
    ...

    // Set the name generator for your BPEL importer instance.
    importerInstance.setNameGenerator(nameGeneratorInstance);

    // Set any BPEL import layout options.
    ...

    // Set the journal to be your custom journal.

    MyJournal journal = new MyJournal();
    importerInstance.setJournal(journal);

    // Handle any command line options.
    ...

    // Do the actual import.
    importerInstance.importBpelFile(bpelFile);

    // Display any messages from the import.
    System.out.print(journal.getText(false));

    // Get the flow definition that you just imported.
    Flow flow = importerInstance.getFlowDefinition();

    // Mark the flow to be editable.
    flow.setEditable(true).

    // Retrieve the list of nodeIDs to remove.
    ArrayList nodesToDelete = journal.getNodesToDelete();

    ...Loop through these node IDs.
    ...For each node you must remove the arcs to/from the node
       as well as the node itself.

    // Connect to the Model Repository.
    ...

    // Save the final flow definition into the Model Repository.
    ...
}
```

Your code imports the BPEL and then uses the list of node IDs to remove these from the resultant flow definition. You just need to make sure that you remove the nodes and all the incoming/outgoing arcs for these nodes.