HP OpenView Business Process Insight

For the Windows® Operating System

Software Version: 02.10

Integration Training Guide - Importing BPEL

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 Introduction to BPEL

This chapter provides an introduction to Business Process Execution Language (BPEL) and leads you through an example business process.

BPEL Basics

Business Process Execution Language (BPEL) defines a notation for specifying business process behavior, typically based on Web Services. Sometimes BPEL is also known by the acronyms WS-BPEL (Web Services BPEL) or BPEL4WS (BPEL for Web Services).

BPEL was first developed back in 2002 by BEA, IBM, and Microsoft. Since then the majority of vendors have become involved which has resulted in several modifications and improvements, and adoption of version 1.1 in March 2003. In April 2003, BPEL was submitted to OASIS (Organization for the Advancement of Structured Information Standards) for standardization purposes. This has led to even broader acceptance across the industry. Having said that, BPEL is still in its early days and you find that actual implementations of BPEL can vary between the different manufacturers.

BPEL has been designed specifically as a language for the definition of business processes. BPEL supports two different types of business processes:

- Executable processes

  This is where you specify the exact details of your business process. This business process definition can then be executed by a BPEL Process Engine.

- Abstract processes

  This is where you specify a high level overview of your business process. An abstract process definition does not include the internal details of each step and decision within the business process, and an abstract process definition can not be executed by a BPEL Process Engine.

A BPEL process specifies the order in which participating services should be invoked. Service can be invoked sequentially or in parallel. With BPEL, you can express conditional behavior, for example, a Web service invocation can depend on the value of a previous invocation. You can also construct loops, declare variables, copy and assign values, define fault and exception handlers, and so on. By combining all these constructs, you can define complex business processes in an algorithmic manner. So BPEL is indeed a form of programming language. Hence a BPEL Process Engine can be given an executable BPEL process definition, and actually run the business process.

BPEL process definitions are written in XML.

A BPEL Business Process

A BPEL business process is initiated by receiving a request. To fulfill this request, the process then invokes the specified Web service(s) and finally responds to the original caller. To be able to communicate with the involved Web services, the BPEL process requires the WSDL (Web Service Definition Language) definitions for each Web service.

A BPEL process consists of steps. Each step is called an activity.

XML Elements

Within BPEL there are XML elements for specifying actual work activities, such as the following:

- `<invoke>` - to invoke a Web service .
- `<receive>` - to wait for the client to invoke the business process through sending a message.
- `<reply>` - to generate a response for synchronous operations.
- `<assign>` - to manipulate data variables.
- `<throw>` - to indicate faults and exceptions.
- `<wait>` - to wait for some time.
- `<terminate>` - to terminate the entire process.

There are also XML elements within BPEL for defining code structures, such as the following:

- `<sequence>` - to define a set of activities that are to be invoked in an ordered sequence.
- `<flow>` - to define a set of activities that are to be invoked in parallel.
- `<switch>` and `<case>` - to implement branches.
- `<while>` - to define a while loop.
- `<pick>` - to select one of a number of alternative paths.

Each BPEL process also declares variables, using the `<variable>` element.

Partner Links

All external services that a BPEL process interacts with, are called partner links. Partner links can be links to Web services that are invoked by the BPEL process. Partner links can also be links to clients which invoke the BPEL process. Each BPEL process has at least one client partner link, because there has to be a client that invokes the BPEL process itself.

It is usual for a BPEL process to have at least one invoked partner link, because the process most likely invokes at least one Web service :-) Invoked partner links may, however, become client partner links - this is usually the case with asynchronous services, where the process invokes an operation. Later the service (or partner) invokes the call-back operation on the process to return the requested data.

All partner links are accessed through WSDL port types. A port type is a WSDL definition that describes the call interface (methods available) for a Web service.

The Insurance Selection Process

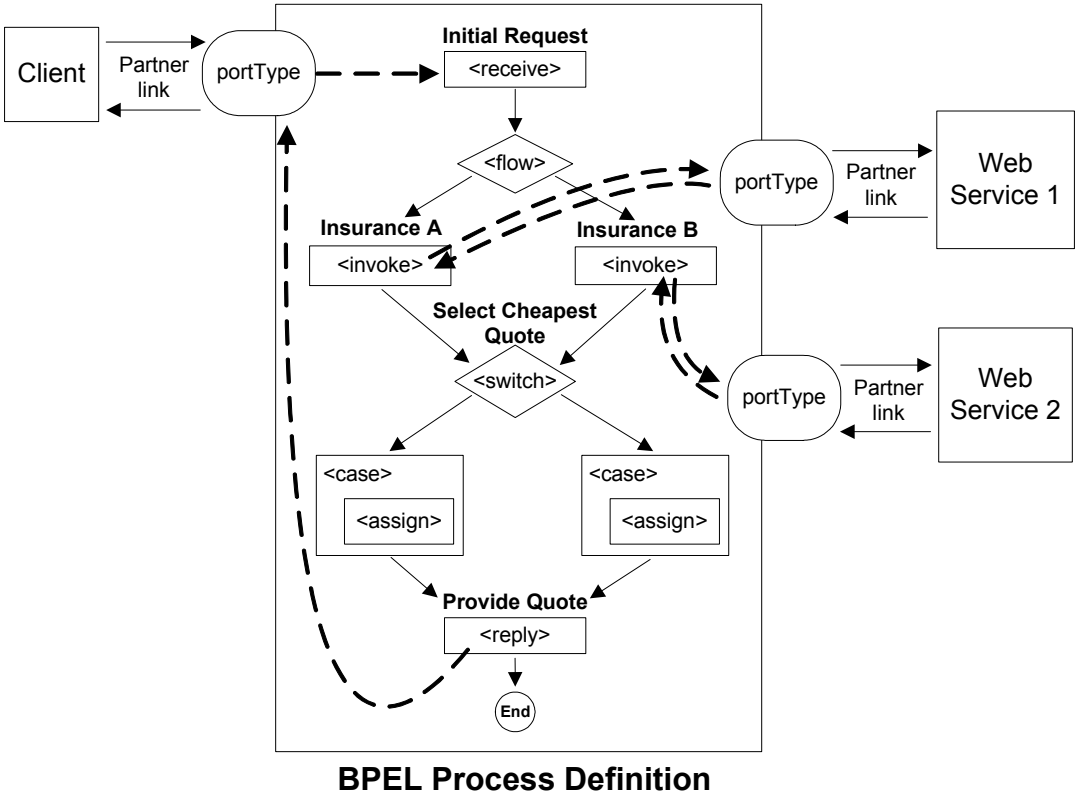
Let's consider an example business process and see how it might be represented in BPEL.

Suppose you have a simple process that looks for the cheapest insurance quote from two insurance brokers. The basic steps of this Insurance Selection process are as follows:

1. A client initiates a request to find the cheapest insurance quote.
2. The process then issues requests to both the insurance brokers, asking them to provide an insurance quote.
3. The process then compares the two quotes, and chooses the quote that is for the cheapest amount of money.
4. The cheapest quote is then passed back to the initiator.

Figure 1 on page 14 shows how this process might execute within a BPEL/ Web Services environment.

Figure 1 Insurance Selection BPEL Process



Let's consider the steps shown above in Figure 1:

- Someone initiates the process.

  An external client (a partner link) invokes this BPEL process as a Web service.

- The first step in the BPEL process receives the request for an insurance quote.

- The process then starts up two parallel paths.

  Each path of the process invokes the appropriate Web service to get an insurance quote.

- The process then chooses the cheapest quote and assigns the return details to be this cheapest quote.

- The cheapest quote details are then returned to the initiating client.

Let's now look in more detail at the actual BPEL XML for defining this example Insurance Selection process.

The first thing you do is declare the process and all XML name spaces that are used within this process, as follows:

```
<process name="Insurance Selection Process"
         targetNamespace="http://somewhere.com/bpel/example/"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:ins="http://somewhere.com/bpel/insurance/"
         xmlns:com="http://somewhere.com/bpel/company/" >
```

The process is called `Insurance Selection Process`.

You then declare the partner links to the BPEL process client (called `client`) and the two insurance Web services (called `insuranceA` and `insuranceB`). The XML looks as follows:

```
   <partnerLinks>
      <partnerLink name="client"
                   partnerLinkType="com:selectionLT"
                   myRole="insuranceSelectionService"/>

      <partnerLink name="insuranceA"
                   partnerLinkType="ins:insuranceLT"
                   myRole="insuranceRequester"
                   partnerRole="insuranceService"/>

      <partnerLink name="insuranceB"
                   partnerLinkType="ins:insuranceLT"
                   myRole="insuranceRequester"
                   partnerRole="insuranceService"/>

   </partnerLinks>
```

The actual syntax for these `<partnerLink>` elements is not important for your basic understanding of BPEL. Essentially, each partner link is defining the link to each Web service used by this BPEL process definition.

Next, you declare the variables for the insurance request, insurance A and B responses, and for final selection. The XML looks as follows:

```
<variables>
   <!-- Input for BPEL process -->
   <variable name="insuranceRequest"
             messageType="ins:insuranceRequestMessage"/>
   <!-- Output from insurance A -->
   <variable name="insurance-A-Response"
             messageType="ins:insuranceResponseMessage"/>
   <!-- Output from insurance B -->
   <variable name="insurance-B-Response"
             messageType="ins:insuranceResponseMessage"/>
   <!-- Output from BPEL process -->
   <variable name="insuranceSelectionResponse"
             messageType="ins:insuranceResponseMessage"/>
</variables>
```

Finally, you specify the actual steps of the process. First you wait for the initial request message from the client (`<receive>`). Then you invoke both insurance Web services (`<invoke>`) in parallel using the `<flow>` activity. The insurance Web services return the insurance premiums. Then you select the lower amount (`<switch>`/`<case>`/`<assign>`) and return the result to the client (the caller of the BPEL process) using the `<reply>` activity. The XML look as follows:

```
<sequence>

   <!-- Receive the initial request from client -->
   <receive name="Initial Request"
            partnerLink="client"
            portType="com:insuranceSelectionPT"
            operation="SelectInsurance"
            variable="insuranceRequest"
            createInstance="yes" />

   <!-- Make concurrent invocations to Insurance A and B -->
   <flow name="Get Quotes">

      <!-- Invoke Insurance A Web service -->
      <invoke name="Insurance A"
              partnerLink="insuranceA"
              portType="ins:computeInsurancePremiumPT"
              operation="ComputeInsurancePremium"
              inputVariable="insuranceRequest"
              outputVariable="insurance-A-Response" />
```

```
         <!-- Invoke Insurance B Web service -->
         <invoke name="Insurance B"
                 partnerLink="insuranceB"
                 portType="ins:computeInsurancePremiumPT"
                 operation="ComputeInsurancePremium"
                 inputVariable="insuranceRequest"
                 outputVariable="insurance-B-Response" />

      </flow>

      <!-- Select the best offer and construct the response -->
      <switch name="Select Cheapest Quote">

        <case condition="bpws:getVariableData('insurance-A-Response',
                         'resultData','/resultData/Amount')
                         &lt;= bpws:getVariableData('insurance-B-Response',
                         'resultData','/resultData/Amount')">

           <!-- Select Insurance A -->
           <assign>
             <copy>
               <from variable="insurance-A-Response" />
               <to variable="insuranceSelectionResponse" />
             </copy>
           </assign>
        </case>

        <otherwise>
           <!-- Select Insurance B -->
           <assign>
             <copy>
               <from variable="insurance-B-Response" />
               <to variable="insuranceSelectionResponse" />
             </copy>
           </assign>
        </otherwise>
      </switch>

      <!-- Send a response to the client -->
      <reply name="Provide Quote"
             partnerLink="client"
             portType="com:insuranceSelectionPT"
             operation="SelectInsurance"
             variable="insuranceSelectionResponse"/>

   </sequence>

</process>
```

Notice that you close off the process with a closing `</process>` element.

As each BPEL process is itself a Web service, each BPEL process also needs a WSDL document. However, understanding WSDL documents is beyond the scope of this document.

From this example BPEL process, you might think that this process could alternatively have been written in a programming language such as Java. Indeed, it could have been. However, it is important to realize that a BPEL process is portable, even outside the Java platform. BPEL processes can be executed on BPEL process engines that are based on either a Java platform, or on any other software platform (for example .NET). This is particularly important in business-to-business interactions where different partners may be using different platforms.

BPEL Process Engines/Development Tools

There are BPEL process engines for both the J2EE and .NET platforms. Some examples are as follows:

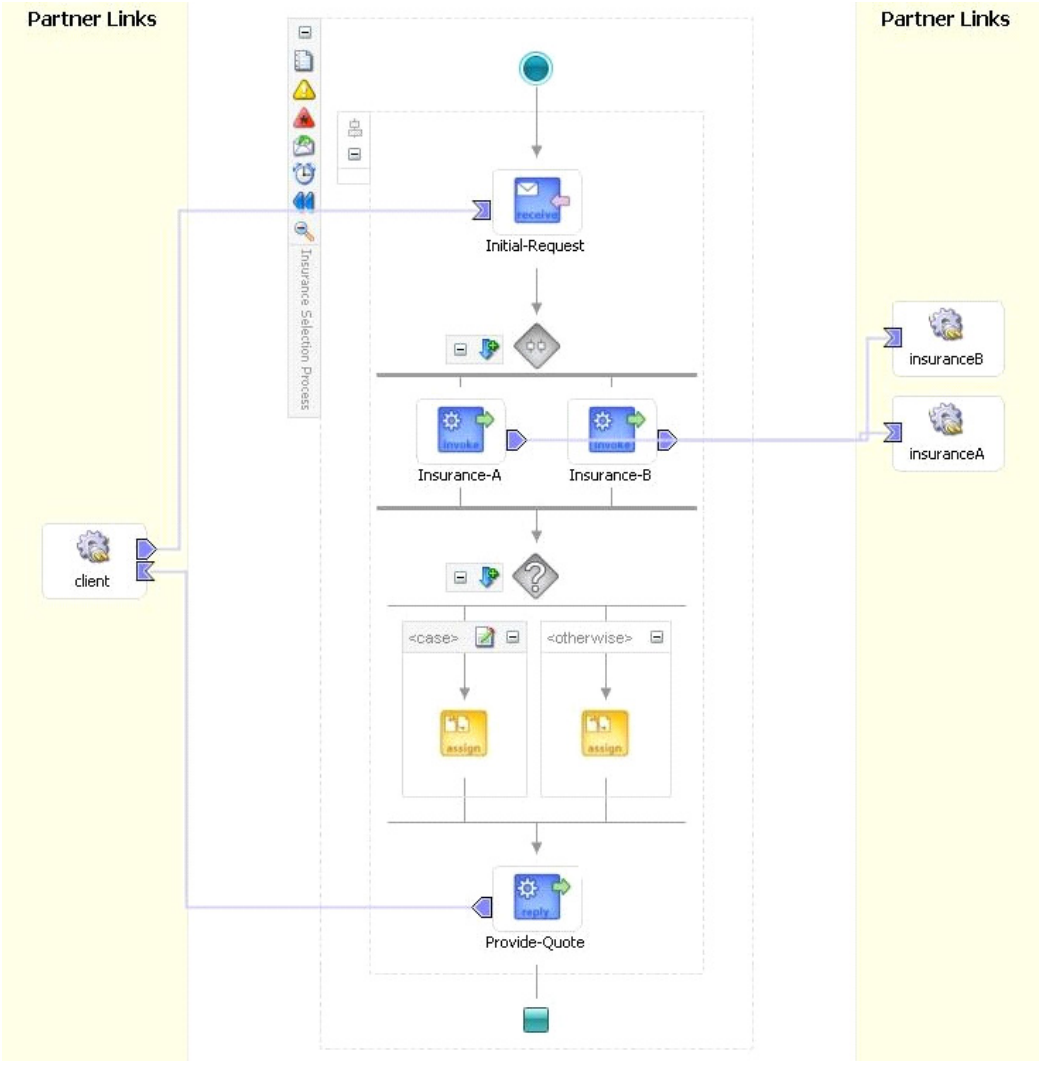
- Oracle BPEL Process Manager (J2EE)
- IBM WebSphere Business Integration Server Foundation (J2EE)
- ActiveBPEL engine (J2EE)
- OpenStorm Service Orchestrator (J2EE or .NET)
- Microsoft BizTalk (.NET)

In addition to BPEL process engines, there are also BPEL design tools available. These tools enable graphical development of BPEL processes and are often shipped with the BPEL process engine. Some example BPEL development tools are as follows:

- Oracle BPEL Designer
- ProVision - from Proforma
- IBM WebSphere Studio Application Developer, Integration Edition
- IBM BPWS4J Editor
- Active Endpoints Active Webflow Designer

To show you an example of how a BPEL design tool might represent a BPEL business process, let's consider the example Insurance Selection process as shown in Figure 1 on page 14. If you were to look at this business process from within the Oracle BPEL Designer, the business process might be represented as shown in Figure 2.

Figure 2 Oracle BPEL Designer



Notice that this diagram shows the partner links and the connections to/from these. The Oracle BPEL Designer also lets you collapse and expand the various sections of the process diagram.

Exporting BPEL

The BPEL design and development tools allow you to design your BPEL processes using a user friendly GUI front end. These tools also offer the ability to export the process as a BPEL XML file.

It is worth noting that the BPEL standard is still in its infancy and (believe it or not) not all of the BPEL design and development tools necessarily export BPEL that matches the business process being exported.

The fact that a BPEL development tool says it has “exported the process as BPEL” does not necessarily means that it is correct. Indeed, with some of the BPEL tools at the moment, if you export the process as BPEL. and then re-import that BPEL, you can end up with a business process that bares little resemblance to the one you started with :-(This can become an issue when you are given a BPEL file and asked to import it into the OVBPI Modeler. The OVBPI Modeler correctly interprets the BPEL, but if the BPEL does not match the actual business process, then the OVBPI flow definition will not match the business process either.

2 The OVBPI Modeler

This chapter looks at how to import a BPEL process into the OVBPI Modeler, and looks at the resultant OVBPI flow definition.

Importing BPEL

The OVBPI Modeler is able to import a business process from a BPEL XML file.

Let's consider the Insurance Selection process from Chapter 1. Take a look at Figure 1 on page 14 to remind yourself of the business process. If you take the BPEL for this flow (as listed in Chapter 1) and import this into the OVBPI Modeler (`File->Import Definitions...`) it creates an OVBPI flow diagram as shown in the two figures Figure 3 and Figure 4:

Figure 3 Insurance Selection Process - Part 1

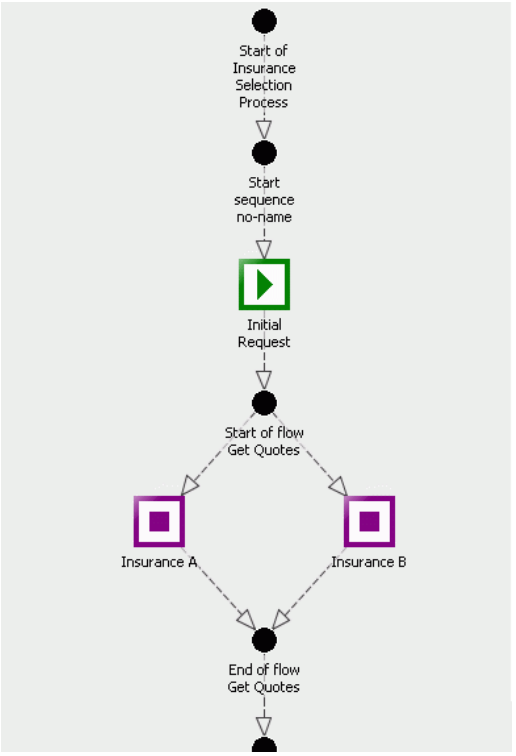
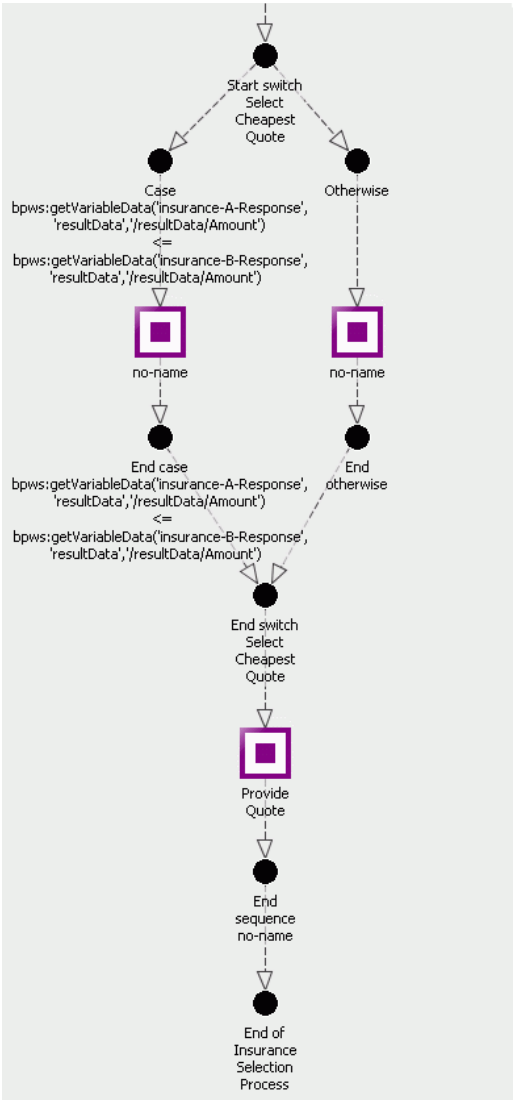


Figure 4 Insurance Selection Process - Part 2



Let's talk in more detail about figures Figure 3 and Figure 4, and gain an understanding of how the OVBPI BPEL importer works.

Junction Nodes

The first thing to notice is that the flow contains quite a lot of junction nodes. This is mainly because the OVBPI BPEL importer has been written to preserve the BPEL flow structure. The idea is that you can see the original BPEL structure and this might help you to better understand the resulting flow diagram.

Closing Junction Nodes

For every junction node that starts a BPEL structure, there is a corresponding junction node for the closing of this structure. For example, in the BPEL definition, the `<switch>` element has a closing `</switch>` element, hence the OVBPI flow diagram has a `Start switch Select Cheapest Quote` junction node, and an `End switch Select Cheapest Quote` junction node.

This is true for all BPEL structural elements. This why you also see junction nodes for the start and end of the process itself, the start and end of the enclosing sequence that contains the flow, the start and end of the switch, and the two cases within this switch structure. etc..

Indeed, within BPEL there is an assumption that, unless an element explicitly terminates the process, then the flow of control continues down to the next logical element in the process. Thus, when you import BPEL into the OVBPI Modeler, the default behavior is to draw arcs such that the nodes are linked to the logically next node in the flow. For example, notice the `Start switch Select Cheapest Quote` junction node in Figure 4 on page 25. You see that two arcs come out of this junction node, and the flow of control continues until these two paths are merged back at the closing junction node called `End switch Select Cheapest Quote`.

Start and End Nodes

The OVBPI BPEL importer tries to identify any BPEL elements that may start, or end, the BPEL process. Not all BPEL definitions necessarily contain this information within their elements.

In this example, the node `Initial Request` is shown as a start node. This is because the actual BPEL tag for the corresponding node contained the `createInstance` option, as follows:

```
<receive name="Initial Request"
         partnerLink="client"
         portType="com:insuranceSelectionPT"
         operation="SelectInsurance"
         variable="insuranceRequest"
         createInstance="yes" />
```

As there was no BPEL element marked as terminating the process, no end node has been shown in the OVBPI flow diagram.

It is not uncommon to import a BPEL process and not see any start node or end node. Even if some nodes are shown as start or end nodes, it is up to you, as the OVBPI developer, to consider the resultant flow and decide which node, or nodes, should become start or end nodes.

No Partner Links

The resulting OVBPI flow does not show the partner links within the BPEL process. Partner link information does not have any affect on the resultant flow diagram, so it is ignored.

Node Names

Notice that in Figure 4 on page 25, some of the nodes have been created with a name of `no-name`. This is simply because the corresponding BPEL element did not contain the `name` attribute. If you have more than one node with the name `no-name` then the OVBPI Modeler marks these as `ToDo` errors and you need to assign appropriate names before you can deploy the flow.

You can also see that in Figure 4 on page 25, junction nodes have been created that represent the test conditions within the `<case>` elements (within the `<switch>` element). When processing a `<case>` element, the default behavior is to set the node name to be the test condition. This can help you to understand the logic of the original BPEL process. In this example scenario, some of the resultant node names are too long and the OVBPI Modeler `ToDo` list highlights these errors. You need to shorten these node names before the flow can be deployed.

When you use the OVBPI BPEL importer you can specify a name generator file in which you can alter the way that node names are constructed. For example, you can configure which properties of the BPEL element are used to construct the resultant node name. See Name Generation on page 36 for more details.

Cleaning up the Flow Diagram

The resultant flow diagram as shown in Figure 3 on page 24 and Figure 4 on page 25 is a direct interpretation of the BPEL being imported. The Insurance Selection process was defined as executable BPEL, hence the detailed information of the actual test criteria in the `<case>` element. An abstract BPEL flow might contain less detailed process information.

So how useful is the resultant flow diagram?

Once you have cleaned up details such as any duplicate node names, and/or node names that are too long, you then need to consider the actual flow itself.

Let's look at some of the things you need to consider about the resultant flow diagram...

Start and End Nodes

You need to consider where the start and end node(s) should be on your flow diagram. Remember, the OVBPI BPEL importer may not be able to correctly identify the start or end node(s).

Junction Nodes

Do you want all the junction nodes left in the diagram?

The Junction nodes help you to understand the structure of the underlying business process, however you may not need all this detail within your OVBPI flow diagram.

Activity Nodes

Are the activity nodes all correct for what you need to monitor?

The fact that the OVBPI BPEL importer has created an activity node does not necessarily mean that you need to keep this node in your flow diagram. You may decide that some activity nodes are not needed. On the other hand, you might decide that more activity nodes need to be added. It all depends on how you are wanting to monitor the business process.

Is the Flow Correct?

Not all BPEL development tools produce correct BPEL when they export a business process. You may find that the resultant OVBPI flow diagram does not correctly represent your business process.

It is important to go through the OVBPI flow and check that it matches your understanding of the corresponding business process. Do not simply assume that the BPEL you have imported is correct.

If there is any doubt about the accuracy of the BPEL, try importing the BPEL file back into the original BPEL development tool that produced the BPEL export. You may find that the BPEL development tool is unable to correctly rebuild the original process from its own exported BPEL.

A Final Flow Diagram

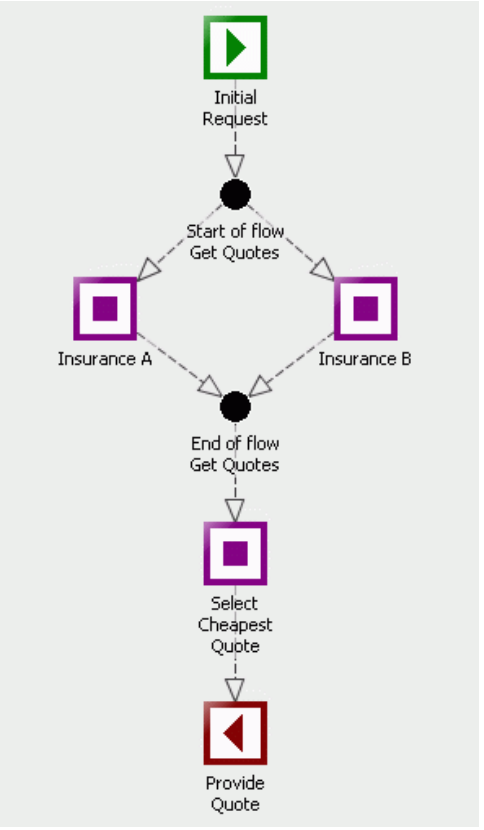
Let's consider the imported flow as shown in Figure 3 on page 24 and Figure 4 on page 25, and look at one possible way to simplify this flow definition.

You could simplify this flow definition as follows:

1. Remove the first two `Start process` and `Start sequence` nodes.
2. Remove the corresponding `End process` and `End sequence` nodes.
3. Collapse the switch/case/assign segment of the diagram to be a simple activity node that assigns the cheapest insurance quote.
4. Change the `Provide Quote` node to be an End node.

The resultant flow diagram is shown in Figure 5 on page 31.

Figure 5 Final Insurance Selection Process



How you choose to alter your flow definition is up to you. For example, you may decide to leave all the junction nodes in the diagram. It also depends on the business events that you configure and the progression rules you set up for the nodes. The flow diagram shown in Figure 5 on page 31 is just one possible alternative.

What Next?

So how helpful is it that you can import the BPEL into the OVBPI Modeler?

Some people might make the observation that, given a diagram of the original business process, the flow diagram as shown in Figure 5 on page 31 would not take very long to define yourself using the OVBPI Modeler - and for some processes that is going to be the case.

The main benefit that BPEL importing gives you is a starting point. That is, you are not just staring at a blank OVBPI Modeler screen and wondering how to start defining your flow. Mind you, the flow definition that is created from a BPEL import is likely to contain almost every step in you business process. Don't forget that in OVBPI you are trying to produce a high-level flow definition that is useful for monitoring this business process, not run it.

The other thing to realize is that the BPEL import helps you define the flow definition. You still need to then define and configure the data definition, the event definitions, the event subscriptions and node progression rules.

The BPEL importer can help you get started with the flow definition, but you still have work to do before you can deploy this flow and have it monitor your business process.

The BPEL Import Wizard

To actually import a BPEL process into the OVBPI Modeler, it is as simple as selecting the menu option: `File->Import Definitions...`

The various menu options within the BPEL import wizard are fully described in the OVBPI Modeler on-line help subsystem, so refer to the Modeler help if you need any details explained.

There are just a few points that are worth reiterating in this manual.

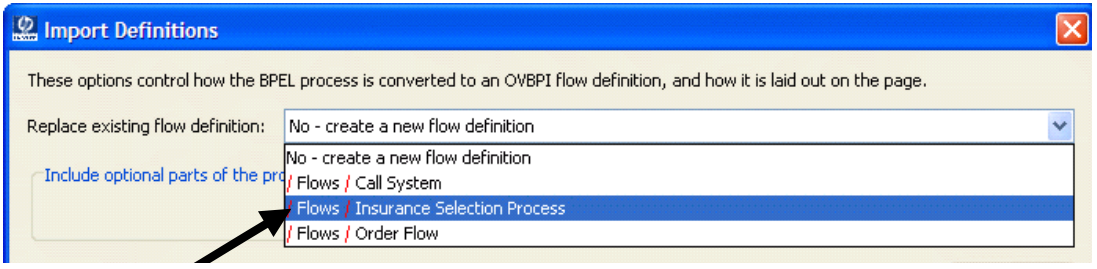
Replace Existing Flow Definition

When importing a BPEL process into the OVBPI Modeler, the default behavior is to import the process and create it as a brand new flow within the Model Repository. So the default is to **not** replace an existing flow definition.

If you have previously imported the same BPEL process, the default behavior is to create a new flow. This can cause some confusion at first, as you can end up with two (or more) flows in the Modeler, both listed in the Navigator pane with the same name. If you click on each flow name within the Navigator pane, the Summary pane (below the Navigator pane) shows you the date and time at which that selected flow was created.

If you want to replace an existing flow, then you need to select an existing flow from the pull-down, as shown in Figure 6 on page 33. Once you have selected a flow name from the pull-down, the BPEL flow is imported and then renamed to the flow name you selected. The imported flow then replaces that selected flow within the Model Repository. You need to make sure that you select the correct flow name, otherwise you can overwrite a completely unrelated flow definition.

Figure 6 Replacing an Existing Flow



Overall Size of the Flow Canvas

When you import a BPEL process into the OVBPI Modeler, you are presented with various options such as `Grid cell size`, `Gaps between cells`, `Page border`, and `Minimum page size`. These options all contribute to the resultant layout of the imported flow. Again, please refer to the OVBPI Modeler on-line help for specific details about each option.

But there is one overall detail that is worth discussing here. When you import a BPEL process, you are probably only thinking about the final layout of the resultant flow. You might also want to think about whether you intend adding more nodes to this flow and whether you will have enough room to do this.

Whenever a flow is created within the OVBPI Modeler, the flow is created within a flow canvas. That is, each flow has a limit to the overall size that it can grow to.

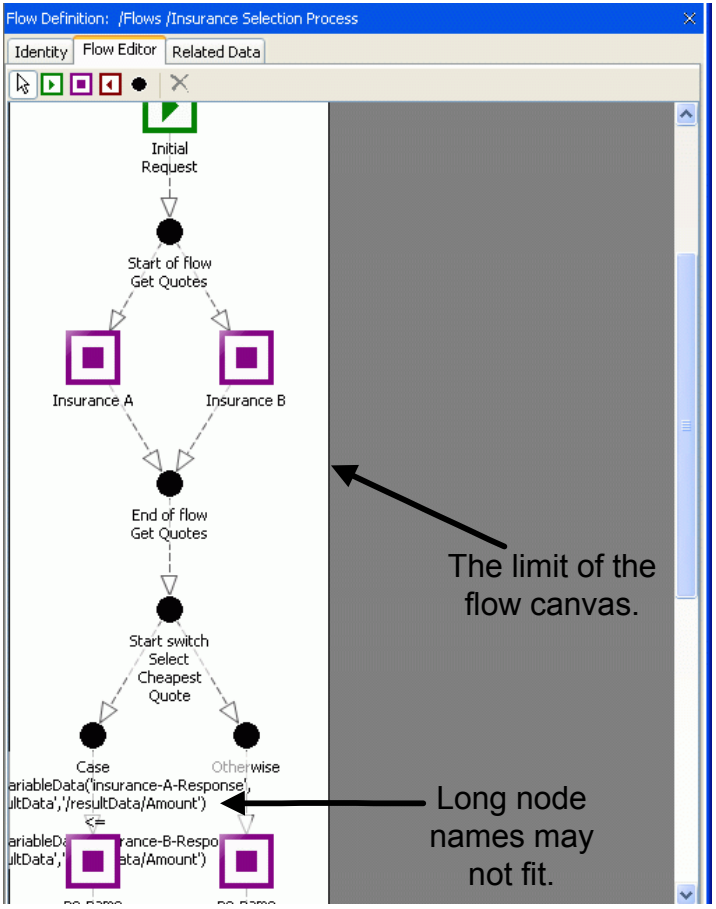
When you import a BPEL process, the resultant flow might be quite large. So the OVBPI BPEL importer starts by allocating a default flow canvas, as defined by the import setting `Mimimum page size`. The importer then goes about creating the OVBPI flow within this canvas. If the importer needs to grow the flow canvas to fit a node on to the flow, it does. The canvas is increased accordingly until all the nodes and arcs have been added, and the flow is complete.

Once a flow has been created, the flow canvas is set and cannot grow any further. So when you import a BPEL process, take a look at the boundaries of the resultant flow canvas. If you do not have enough space to add or reposition nodes, then you might want to re-import the BPEL process with different values for settings such as the `Page Border`.

For example:

> Suppose you import the Insurance Selection BPEL process where you explicitly set the import option `Mimimum page size` to be zero(0)/zero(0). This imports the process, and creates a flow with a flow canvas just large enough to fit the newly imported flow, plus any `Page border` size specified.
>
> The flow might look in the OVBPI Modeler as shown in Figure 7 on page 35. The flow canvas is just wide enough to fit the nodes of the flow. Notice also that the flow canvas calculation does not take into account any extra-long node names that may be created.

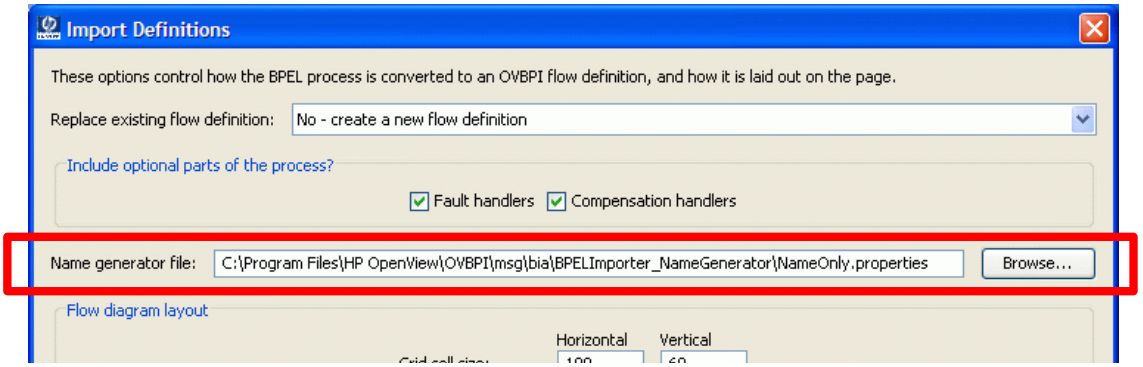
Figure 7 Minimum Flow Canvas



Name Generation

When you import a BPEL file into the OVBPI Modeler you need to specify a `Name Generator file`. The option appears on the import screen as shown in Figure 8.

Figure 8 BPEL Import - Name Generator File



The name generator file determines how the OVBPI flow name and node names are constructed from the BPEL elements being imported.

File Location

When the import dialog appears, as shown in Figure 8, you can browse to any location on your network, and choose a name generator file.

OVBPI provides a directory that contains some default name generator files that are set up ready for you to use. These name generator files are located in the directory:

OVBPI-install-dir`\msg\bia\BPELImporter_NameGenerator`

The name generator files provided in this `BPELImporter_NameGenerator` directory are as follows:

- `NameOnly.properties`

  This file specifies that the flow name and node names are to be generated by using only the `name` attribute from the corresponding BPEL element.

- `EscapedNameOnly.properties`

  This file specifies that the flow name and node names are to be generated by using only the `name` attribute from the corresponding BPEL element, in the same way that the file `NameOnly.properties` does. However, if the `name` attribute contains any hexidecimal encoding of the form `_xN_` where `N` is a two, three or four digit number, then this is transformed (decoded) into the actual character it represents. (See Built-in Transformations (_hex_) on page 42 for further details.)

- `FullNames.properties`

  This file specifies that the flow name and node names are to be generated by using more than just the `name` attribute from the corresponding BPEL element. The main feature of this name generator file is that the resultant node names contain the name of their corresponding BPEL element. This allows you to see which BPEL elements map to which OVBPI nodes.

  Be aware that selecting this property file can produce very long node names that you may need to manually edit, and shorten, within the OVBPI Modeler.

- `EscapedFullNames.properties`

  This file is similar to the `FullNames.properties` file, and it also specifies that any hexidecimal encoding of the form `_xN_` where `N` is a two, three or four digit number, is transformed (decoded) into the actual character it represents. (See Built-in Transformations (_hex_) on page 42 for further details.)

  Be aware that selecting this property file can produce very long node names that you may need to manually edit, and shorten, within the OVBPI Modeler.

File Format

Each name generator file contains a set of entries that specifies how to construct a name from each BPEL element.

Syntax

The syntax is as follows:

```
BPEL-element.identifier.pattern=
BPEL-element.identifier.number.source=
BPEL-element.identifier.number.default=
BPEL-element.identifier.number.transform=
```

where:

- *BPEL-element*

  This is set to the name of the BPEL element.

- *identifier*

  For each BPEL element there are pre-defined identifiers. For example, the code that handles the BPEL element `case`, looks for the two identifiers `start` and `end`. Hence in the name generator file you see entries prefixed with `case.start` and `case.end`.

  The name generator files shipped with OVBPI define the mappings for all elements handled by the OVBPI BPEL importer. So you just have to look into any of the name generator files to see all the identifiers for each BPEL element.

- `pattern`

  You specify the pattern for the resultant name.

  To substitute values from the corresponding BPEL element you can specify parameter substitution. For example, the pattern `Node {0}` forms a name starting with the text `Node` , followed by the contents of parameter zero. You then need to define parameter zero by setting up the `source`, `default` and `transform` properties.

- *number*.source
  number.default
  number.transform

  This is how you set up parameter values that can then be substituted into the pattern string.

  You must set all three properties for each numbered parameter. You can specify more than one parameter. Parameter numbers must be consecutive, starting from zero.

  You set source to be the value of an attribute within the current BPEL element. You simply specify the name of the BPEL attribute, prefixed by the @ sign. For example, assign.node.0.source=@name sets parameter zero to be the contents of the name attribute from within the assign BPEL element.

  You set default to be the string you wish to use if the specified source attribute does not exist.

  You set transform to be either none or the name of a predefined transformation. OVBPI provides one predefined transformation, which is called _hex_. (See Built-in Transformations (_hex_) on page 42 for further details.)

Example 1

The following properties configure how the BPEL element assign is used to create a name for the node that is created in the OVBPI flow:

```
assign.node.pattern={0}
assign.node.0.source=@name
assign.node.0.default=unnamed assign
assign.node.0.transform=none
```

where:

- The resulting name is simply the contents of the name attribute from the BPEL assign element.

  The string {0} is replaced by the contents of parameter zero.

- The source defines that parameter zero is the contents of the name attribute from the BPEL assign element.

- If the name attribute does not exist then parameter zero is set to the string unnamed assign.

- The `transform=none` line simply says that no transformation is to be applied to this parameter.

So if the BPEL element is defined as follows:

```
<assign name="My Assign">
```

the corresponding node name within the OVBPI flow becomes `My Assign`.

Example 2

The following properties configure how the BPEL element `case` is used to create OVBPI node names:

```
case.start.pattern=Case {0}
case.start.0.default=no-condition
case.start.0.source=@condition
case.start.0.transform=none

case.end.pattern=End case {0}
case.end.0.default=no-condition
case.end.0.source=@condition
case.end.0.transform=none
```

where:

- The OVBPI importer code that handles the `case` element is able to handle the creation of both a `start` case and an `end` case node.
- When creating the case start node, the node name is going to be the string `Case` , followed by the value of the `condition` attribute within the BPEL element.
- When creating the case end node, the node name is going to be the string `End case` , followed by the value of the `condition` attribute within the BPEL element.
- If there is no `condition` attribute within the BPEL element, the condition parameter is set to the string `no-condition`.

If the BPEL element is defined as follows:

```
<case condition="ExpediteOrder=No">
```

the corresponding node name within the OVBPI flow becomes:

```
Case ExpediteOrder=No
```

Example 3

The following properties configure how the BPEL element `receive` is used to create a name for the node that is created in the OVBPI flow:

```
receive.node.pattern="{0}" - {1} - {2} (receive)
receive.node.0.default=no-name
receive.node.0.source=@name
receive.node.0.transform=_hex_
receive.node.1.default=no-partnerLink
receive.node.1.source=@partnerLink
receive.node.1.transform=none
receive.node.2.default=no-operation
receive.node.2.source=@operation
receive.node.2.transform=_hex_
```

where:

- The pattern substitutes three parameters.
- Parameter zero is set to the `name` attribute and this is passed through the `_hex_` transformation. (See Built-in Transformations (_hex_) on page 42 for further details.)
- Parameter one is set to the `partnerlink` attribute and this is not passed through any transformation.
- Parameter two is set to the `operation` attribute and this is passed through the `_hex_` transformation. (See Built-in Transformations (_hex_) on page 42 for further details.)

If the BPEL element is defined as follows:

```
<receive name="Initial Request"
         partnerLink="client"
         portType="com:insuranceSelectionPT"
         operation="SelectInsurance"
         variable="insuranceRequest"
         createInstance="yes" />
```

the corresponding node name within the OVBPI flow becomes:

```
"Initial Request" - client - SelectInsurance (receive)
```

Built-in Transformations (_hex_)

OVBPI provides one predefined transformation that you can use when configuring your name mapping. This transformation is called `_hex_`.

The `_hex_` transformation converts any hexidecimal encoding of the form `_xN_`, where `N` is a two, three or four digit number, into the actual character it represents.

For example, suppose you have the following scenario.

Your BPEL element is defined as:

```
<receive name="Initial_x20_Request"
         partnerLink="client"
         portType="com:insuranceSelectionPT"
         operation="SelectInsurance"
         variable="insuranceRequest"
         createInstance="yes" />
```

Your name generator file has the entry:

```
receive.node.pattern="{0}" (receive)
receive.node.0.default=no-name
receive.node.0.source=@name
receive.node.0.transform=_hex_
```

The resultant node name becomes: `"Initial Request" (receive)`. This is because the string `_x20_` is transformed into the space character.

Custom Transformations

OVBPI provides the built-in transformation called `_hex_`. But suppose you are importing BPEL files that use a different type of encoding within the XML. Maybe you have a combination of `_xN_` style encoding and some other more involved encoding.

If you need to be able to carry out additional name transformations, then you simply need to write some Java code to extend the supplied name generator class, and provide your own transformations.

Transformer Code

The default name generator class that you need to extend is: `com.hp.ov.bia.model.repository.api.utils.bpel.NameGenerator`.

Your class then needs to do the following main things:

1. Initialize itself by calling `super()`.
2. Define a new `NameGenerator.Transform`, and implement the `transform()` method.
3. Call `addTransform()` to add this new transformation and assign it a name.

The code might look as follows:

```
package com.customer.bpel;

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

public class MyNameGenerator extends NameGenerator
{
    public MyNameGenerator()
    {
        super();

        /**
         * Now define an additional transform called "MyXform".
         */
        addTransform("MyXform", new NameGenerator.Transform()
        {
            public String transform(String text)
            {
                if (text == null)
                {
                    return null;
                }

                // Now do the transformation
                return text.replaceAll("-mysub-", "***");
            }
        });
    }
}
```

where:

- The `MyNameGenerator` class extends `NameGenerator`.
- The `addTransform()` method adds a new `NameGenerator.Transform` class, and calls this transform `MyXform`.
- The `NameGenerator.Transform` class just needs to implement the method `transform()`.

  The `transform()` method is passed the current name as a string. You then code whatever transformations you require and return the resultant string.

  This example shows a substitution where any occurrence of the string `-mysub-` is replaced with the string `***`.

Compiling the Code

To compile your transformer code, you need the following JAR files on your classpath:

- *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\MyNameGenerator.java
```

The Name Generator Properties File

You can now edit your name generator file to make use of your newly created transformation (`MyXform`).

You set the `transform` property for all BPEL elements where you would like this `MyXform` transform to be performed.

Note that you can only call one transformation for a BPEL attribute. Hence if you need this single transformation to cope with both the `-mysub-` replacement and the built-in `_xN_` replacement, then you need to implement both transformations within your own transform code.

Each element in your name generator properties file can use a different transform, but each element can only invoke one transform.

An extract from your name generator properties file might look as follows:

```
assign.node.pattern={0}
assign.node.0.default=unnamed assign
assign.node.0.source=@name
assign.node.0.transform=MyXform

process.flow.pattern={0}
process.flow.0.default=no-name
process.flow.0.source=@name
process.flow.0.transform=_hex_
```

where:

- The `assign` element has its name passed through the `MyXform` transformation.
- The `process` element has its name passed through the `_hex_` transformation.

Update the Modeler Script

To be able to use your new transform, you need to create a new script that can invoke the OVBPI Modeler and include your name generator class on the classpath.

Rather than edit the current script that invokes the OVBPI Modeler you probably should create a copy and make your changes in the copy.

These are the steps you need to carry out:

1. Change to the directory: *OVBPI-install-dir*\bin
2. Make a copy of the file: `biamodeler.bat`

   Call this copy: `mybiamodeler.bat`
3. In the `mybiamodeler.bat` script, locate the line:

```
set NAMEGENERATORCLASS=
                com.hp.ov.bia.model.repository.api.utils.bpel.NameGenerator
```

   (all on one line)
4. Edit this line to set `NAMEGENERATORCLASS` to point to your newly compiled name generator class (`MyNameGenerator.class`).

   For example:

```
set NAMEGENERATORCLASS=com.customer.bpel.MyNameGenerator
```

5. Now add this class (`MyNameGenerator`) to the classpath.

   Locate the lines that set the `CLASSPATH` variable, and then add your class to the classpath.

   For example:

```
set CLASSPATH=%CLASSPATH%;C:\ovbpi\manuals\bpel-tg\sols\src
```

6. Now run the `mybiamodeler.bat` script.

   The OVBPI modeler starts up.

7. Import a BPEL file

   — `File->Import Definition...`

   — Select the name generator file that makes use of your new `MyXform` transform.

   — Import your BPEL process

   Your new transformations are applied to the incoming BPEL.

3 BPEL Element Mappings

This chapter shows how each BPEL XML element is mapped to a corresponding OVBPI node, or nodes.

This chapter looks at each BPEL element and provides the following information:

- The general format of the XML for the BPEL element.
- An indication of which attributes within the BPEL element XML are handled by the OVBPI BPEL Importer.
- The resultant OVBPI node structure.
- Details of how the OVBPI node names are derived.

Formatting Conventions

Here are the convention used throughout this chapter.

Within the BPEL XML listings:

- Attributes shown in **bold** type are handled by the BPEL Importer.
- All non-bold attributes are ignored.
- The token `activity` is used to represent a nested element that may correspond to more OVBPI nodes. This nested element can be any one of the elements <assign>, <compensate>, <empty>, <flow>, <invoke>, <pick>, <receive>, <reply>, <scope>, <sequence>, <switch>, <terminate>, <throw>, <wait>, <while>.
- To save space, wherever you see the token `source` this refers to the element definition:

  ```
  <source linkName="ncname" transitionCondition="bool-expr"?/>
  ```

  and wherever you see the token `target` this refers to the element definition:

  ```
  <target linkName="ncname" />
  ```

- Some attributes are optional within BPEL XML and these are shown with a trailing `?` character.
- Some attributes can appear more than once. This is indicated by a trailing `+` or `*` character, where:
  - `+` indicates one or more occurrences.
  - `*` indicates zero or more occurrences.

Within the diagrams that show the resultant OVBPI node structure:

- The “cloud” image is used to indicate any nested activities.

process

The BPEL XML

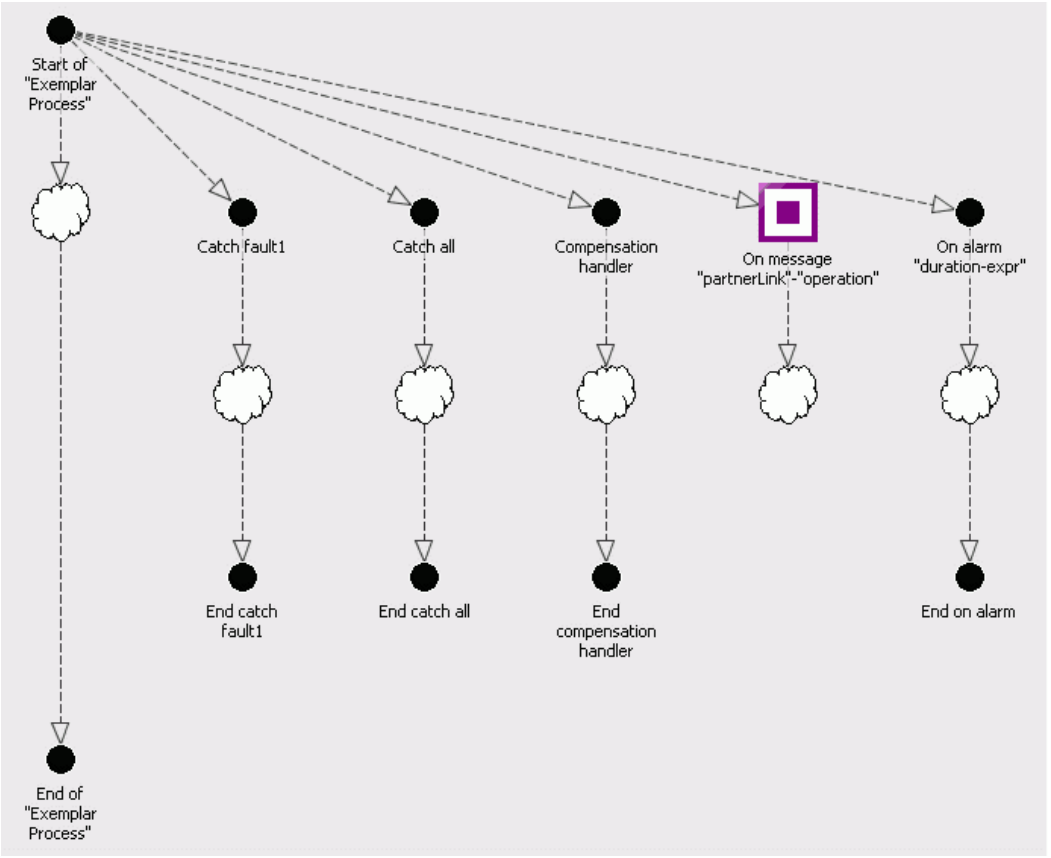
```
<process
  name="ncname"
  targetNamespace="uri"
  abstractProcessProfile="anyURI"?
  queryLanguage="anyURI"?
  expressionLanguage="anyURI"?
  suppressJoinFailure="yes|no"?
  enableInstanceCompensation="yes|no"
  abstractProcess="yes|no"?
  xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
  <extensions>?
    <extension
      namespace="anyURI"
      mustUnderstand="yes|no"/>*
  </extensions>
  <import namespace="uri" location="uri" importType="uri"/>*
  <partnerLinks>?
    <partnerLink
      name="ncname"
      partnerLinkType="qname"
      myRole="ncname"?
      partnerRole="ncname"?>+
    </partnerLink>
  </partnerLinks>
  <partners>?
    <partner name="ncname">+
      <partnerLink name="ncname"/>+
    </partner>+
  </partners>
  <variables>?
    <variable
      name="ncname"
      messageType="qname"?
      type="qname"?
      element="qname"?/>+
  </variables>
  <correlationSets>?
    <correlationSet
      name="ncname"
      properties="qname-list"/>+
  </correlationSets>
  <faultHandlers>?
```

```
  <catch
    faultName="qname"?
    faultVariable="ncname"?
    faultMessageType="qname"?
    faultElement="qname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
<compensationHandler>?
  activity
</compensationHandler>
<eventHandlers>?
  <onMessage
    partnerLink="ncname"
    portType="qname"?
    operation="ncname"
    variable="ncname">*

    <correlations>?
      <correlation
        set="ncname"
        initiate="yes|no"?/>+
    </correlations>
    activity
  </onMessage>
  <onAlarm
    for="duration-expr"?
    until="deadline-expr"?>*
    activity
  </onAlarm>
  <onEvent
    partnerLink="ncname"
    portType="qname"?
    operation="ncname"
    messageType="qname"
    variable="ncname"
    messageExchange="ncname"? >*
    <correlationSets>?
      <correlationSet
        name="ncname"
        properties="qname-list"/>+
    </correlationSets>
    <correlations>?
      <correlation
        set="ncname"
        initiate="yes|join|no"?/>+
    </correlations>
```

```
      scope-activity
    </onEvent>
    <onAlarm>*
      (
        (  <for expressionLanguage="anyURI"?>duration-expr</for> |
          <until expressionLanguage="anyURI"?>deadline-expr</until> )
        <repeatEvery expressionLanguage="anyURI"?>
          duration-expr
        </repeatEvery>?
      ) |
      <repeatEvery expressionLanguage="anyURI"?>
        duration-expr
      </repeatEvery>
      scope-activity
    </onAlarm>
  </eventHandlers>
  activity
</process>
```

Figure 9 Resultant process Node Structure



If there are no fault or compensation handlers defined within the BPEL then these branches are simply not created in the resultant OVBPI flow definition.

If there is more than one activity where only one is expected, the OVBPI BPEL Importer issues a warning and continues to import the process anyway. All activities are included in the resultant flow diagram and they are included in parallel, but only one of them is then properly linked to following nodes.

If no activities are found when one or more are expected the OVBPI BPEL Importer issues a warning, and proceeds to attach any following nodes to the start junction node.

The OVBPI BPEL Importer issues a message to say whether the process is executable or abstract. This is derived from the `abstractProcess` attribute.

The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	Default Name	Occurrence
process.start	Start of "{@name}"	1
process.end	End of "{@name}"	1
catch.start	Catch {@faultName}	0 or more
catch.end	End catch {@faultName}	0 or more
catchall.start	Catch all	0 or more
catchAll.end	End catch all	0 or 1
compensationHandler.start	Compensation handler	0 or 1
compensationHandler.end	End compensation handler	0 or 1
onMessage.node	On message "{@partnerLink}"-"{@operation}"	0 or more
onAlarm.for.start	On alarm "{@for}"	0 or more
onAlarm.until.start	On alarm "{@until}"	
onAlarm.neither.start	On alarm "no-condition"	
onAlarm.end	End on alarm	0 or more

The above table assumes that you are using the name generator file `FullNames.properties`.

assign

The BPEL XML

```
<assign
  validate="yes|no"?
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  (  <copy>
       from-spec
       to-spec
     </copy> |
     <extensibleAssign>
       ...assign-element-of-other-namespace...
     </extensibleAssign> ) +
</assign>
```

OVBPI Node Structure

Figure 10 Resultant assign Node Structure

The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
assign.node	{@name} (assign)	1

The above table assumes that you are using the name generator file `FullNames.properties`.

compensate

The BPEL XML

```
<compensate
  scope="ncname"?
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
</compensate>
```

OVBPI Node Structure

Figure 11 Resultant compensate Node Structure

The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
compensate.node	"{@name}" compensate scope {@scope}	1

The above table assumes that you are using the name generator file `FullNames.properties`.

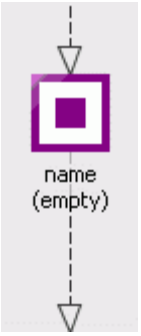
empty

The BPEL XML

```
<empty
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
</empty>
```

OVBPI Node Structure

Figure 12 Resultant empty Node Structure



The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
empty.node	{@name} (empty)	1

The above table assumes that you are using the name generator file `FullNames.properties`.

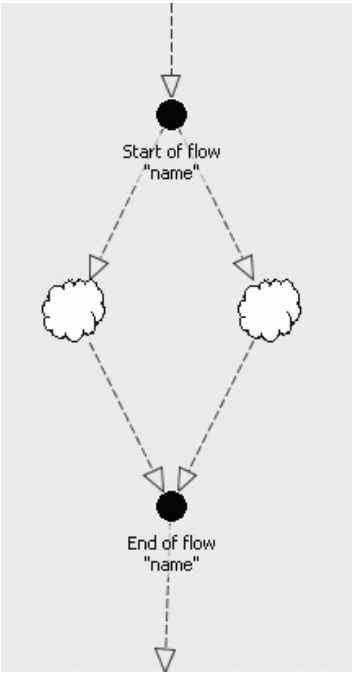
flow

The BPEL XML

```
<flow
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <links>?
    <link name="ncname"/>+
  </links>
  activity+
</flow>
```

OVBPI Node Structure

Figure 13 Resultant flow Node Structure



The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	Default Name	Occurrence
flow.start	Start of flow "{@name}"	1
flow.end	End of flow "{@name}"	1

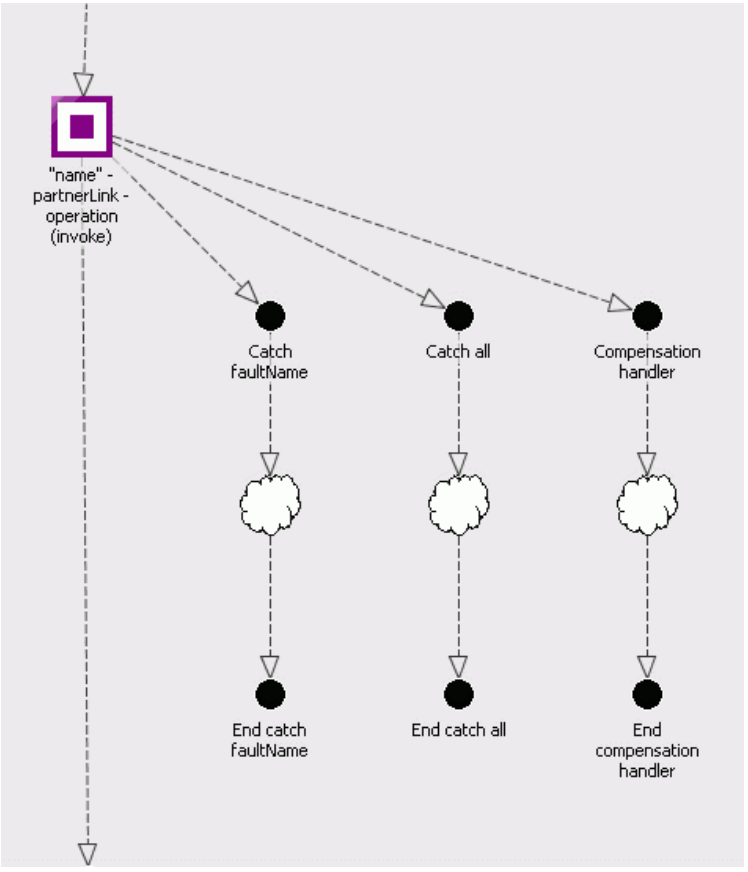
The above table assumes that you are using the name generator file `FullNames.properties`.

invoke

The BPEL XML

```
<invoke
  partnerLink="ncname"
  portType="qname"?
  operation="ncname"
  inputVariable="ncname"?
  outputVariable="ncname"?
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <correlations>?
    <correlation
      set="ncname"
      initiate="yes|join|no"?
      pattern="in|out|out-in"/>+
  </correlations>
  <catch
    faultName="qname"?
    faultVariable="ncname"?
    faultMessageType="qname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
  <toPart part="ncname" fromVariable="ncname"/>*
  <fromPart part="ncname" toVariable="ncname"/>*
</invoke>
```

Figure 14 Resultant invoke Node Structure



The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	**Default Name**	**Occurrence**
invoke.node	"{@name}" - {@partnerLink} - {@operation} (invoke)	1
catch.start	Catch {@faultName}	0 or more
catch.end	End catch {@faultName}	0 or more
catchall.start	Catch all	0 or 1
catchAll.end	End catch all	0 or 1
compensationHandler.start	Compensation handler	0 or 1
compensationHandler.end	End compensation handler	0 or 1

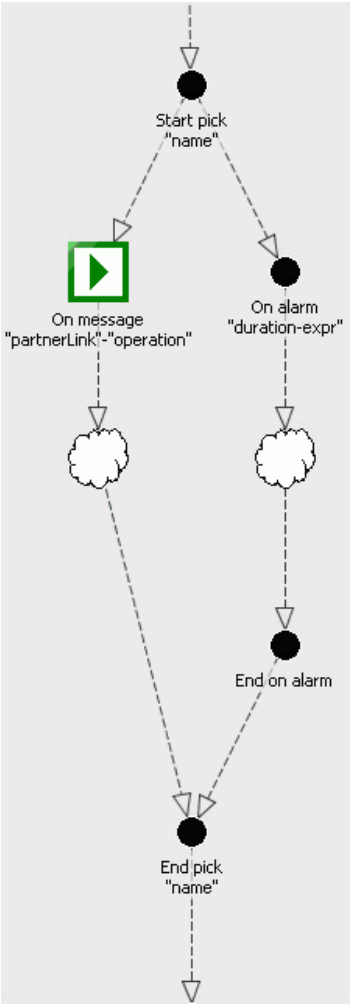
The above table assumes that you are using the name generator file `FullNames.properties`.

pick

The BPEL XML

```
<pick
  createInstance="yes|no"?
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <onMessage
    partnerLink="ncname"
    portType="qname"?
    operation="ncname"
    variable="ncname"?
    messageExchange="ncname"? >+
    <correlations>?
      <correlation set="ncname" initiate="yes|join|no"?/>+
    </correlations>
    <fromPart part="ncname" toVariable="ncname"/>*
    activity
  </onMessage>
  <onAlarm ( for="duration-expr" | until="deadline-expr" ) >*
     activity
  </onAlarm>
  <onAlarm>*
      ( <for expressionLanguage="anyURI"?>duration-expr</for> |
      <until expressionLanguage="anyURI"?>deadline-expr</until> )
     activity
  </onAlarm>
</pick>
```

Figure 15 Resultant pick Node Structure



If the `createInstance` attribute in the pick element is set to `yes` then the `onMessage` node(s) are start nodes, otherwise they are activity nodes.

The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	Default Name	Occurrence
pick.start	Start pick "{@name}"	1
pick.end	End pick "{@name}"	1
onMessage.node	On message "{@partnerLink}"-"{@operation}"	0 or more
onAlarm.for.start	On alarm "{@for}"	0 or more
onAlarm.until.start	On alarm "{@until}"	
onAlarm.neither.start	On alarm "no-condition"	
onAlarm.end	End on alarm	0 or more

The above table assumes that you are using the name generator file `FullNames.properties`.

receive

The BPEL XML

```
<receive
  partnerLink="ncname"
  portType="qname"?
  operation="ncname"
  variable="ncname"?
  createInstance="yes|no"?
  messageExchange="ncname"?
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <correlations>?
    <correlation
      set="ncname"
      initiate="yes|join|no"?/>+
  </correlations>
  <fromPart
    part="ncname"
    toVariable="ncname"/>*
</receive>
```

OVBPI Node Structure

Figure 16 Resultant receive Node Structure

This results in a single activity node being created, unless the `createInstance` attribute is set to `yes`, in which case a start node is created.

The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
receive.node	"{@name}" - {@partnerLink} - {@operation} (receive)	1

The above table assumes that you are using the name generator file `FullNames.properties`.

reply

The BPEL XML

```
<reply
  partnerLink="ncname"
  portType="qname"?
  operation="ncname"
  variable="ncname"?
  faultName="qname"?
  messageExchange="ncname"?
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <correlations>?
    <correlation
      set="ncname"
      initiate="yes|join|no"?/>+
  </correlations>
  <toPart
    part="ncname"
    fromVariable="ncname"/>*
</reply>
```

OVBPI Node Structure

Figure 17 Resultant reply Node Structure

The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
reply.node	"{@name}" - {@partnerLink} - {@operation} (reply)	1

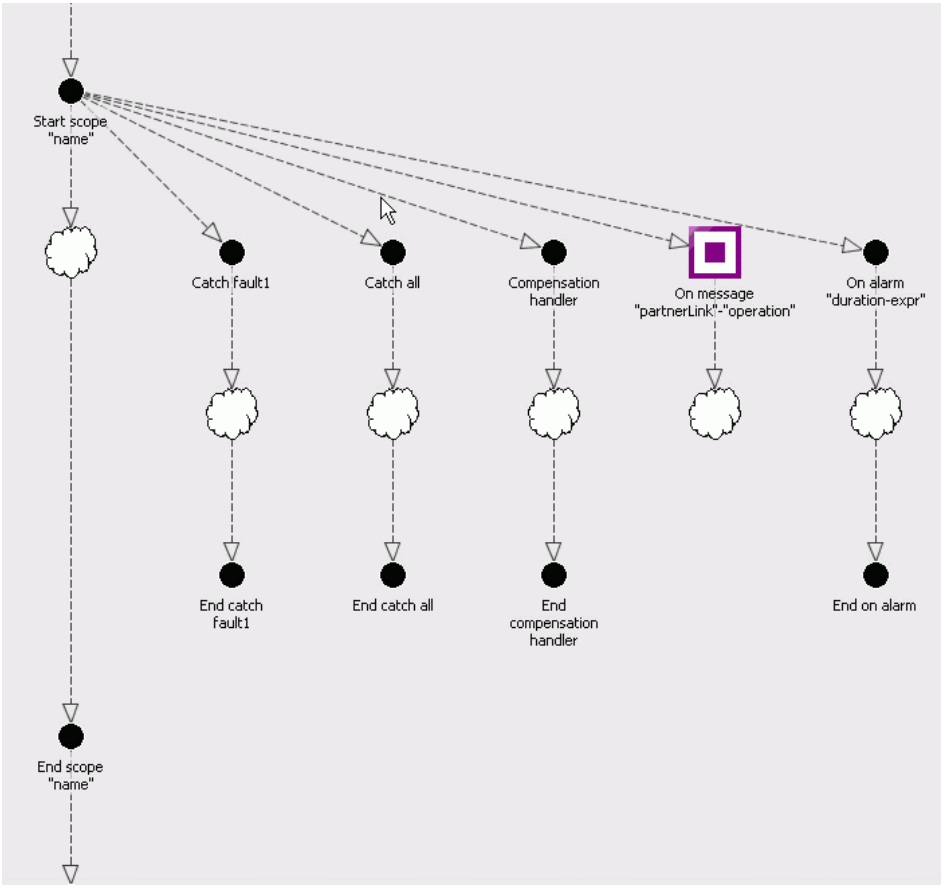
The above table assumes that you are using the name generator file `FullNames.properties`.

scope

The BPEL XML

```
<scope
  variableAccessSerializable="yes|no"
  isolated="yes|no"
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <partnerLinks>?
    ... see process on page 51 for syntax ...
  </partnerLinks>
  <variables>?
    ... see process on page 51 for syntax ...
  </variables>
  <correlationSets>?
    ... see process on page 51 for syntax ...
  </correlationSets>
  <faultHandlers>?
    ... see process on page 51 for syntax ...
  </faultHandlers>
  <compensationHandler>?
    ... see process on page 51 for syntax ...
  </compensationHandler>
  <terminationHandler>?
    ...
  </terminationHandler>
  <eventHandlers>?
    ... see process on page 51 for syntax ...
  </eventHandlers>
  activity
</scope>
```

Figure 18 Resultant scope Node Structure



The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	Default Name	Occurrence
scope.start	Start scope "{@name}"	1
scope.end	End scope "{@name}"	1
catch.start	Catch {@faultName}	0 or more

Name Identifier	Default Name	Occurrence
catch.end	End catch {@faultName}	0 or more
catchall.start	Catch all	0 or 1
catchAll.end	End catch all	0 or 1
compensationHandler.start	Compensation handler	0 or 1
compensationHandler.end	End compensation handler	0 or 1
onMessage.node	On message "{@partnerLink}"-"{@operation}"	0 or more
onAlarm.for.start	On alarm "{@for}"	0 or more
onAlarm.until.start	On alarm "{@until}"	
onAlarm.neither.start	On alarm "no-condition"	
onAlarm.end	End on alarm	0 or more

The above table assumes that you are using the name generator file `FullNames.properties`.

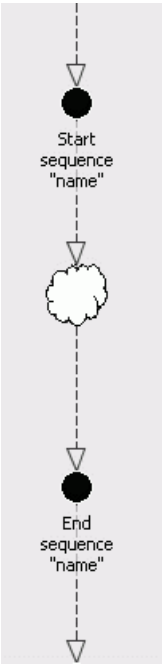
sequence

The BPEL XML

```
<sequence
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  activity+
</sequence>
```

OVBPI Node Structure

Figure 19 Resultant sequence Node Structure



The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	Default Name	Occurrence
sequence.start	Start sequence "{@name}"	1
sequence.end	End sequence "{@name}"	1

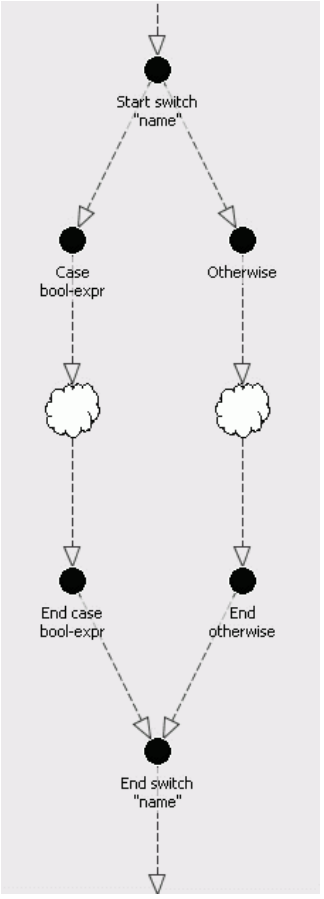
The above table assumes that you are using the name generator file `FullNames.properties`.

switch

The BPEL XML

```
<switch
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <case
    condition="bool-expr">+
    activity
  </case>
  <otherwise>?
    activity
  </otherwise>
</switch>
```

Figure 20 Resultant switch Node Structure



The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	Default Name	Occurrence
switch.start	Start switch "{@name}"	1
switch.end	End switch "{@name}"	1
case.start	Case {@condition}	1 or more

Name Identifier	Default Name	Occurrence
case.end	End case {@condition}	1 or more
otherwise.start	Otherwise	0 or 1
otherwise.end	End otherwise	0 or 1

The above table assumes that you are using the name generator file `FullNames.properties`.

terminate

The BPEL XML

```
<terminate
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
</terminate>
```

OVBPI Node Structure

Figure 21 Resultant terminate Node Structure

The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
terminate.node	"{@name}" (terminate)	1

The above table assumes that you are using the name generator file `FullNames.properties`.

throw

The BPEL XML

```
<throw
   faultName="qname"
   faultVariable="ncname"?
   name="ncname"?
   suppressJoinFailure="yes|no"?>
   source+
   target+
</throw>
```

OVBPI Node Structure

Figure 22 Resultant throw Node Structure

The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
throw.node	"{@name}" - {@faultName} (throw)	1

The above table assumes that you are using the name generator file `FullNames.properties`.

wait

The BPEL XML

```
<wait
  for="duration-expr"
  until="deadline-expr"
  name="ncname"?
  suppressJoinFailure="yes|no"?>
  source+
  target+
  (  <for expressionLanguage="anyURI"?>duration-expr</for> |
    <until expressionLanguage="anyURI"?>deadline-expr</until> )
</wait>
```

OVBPI Node Structure

Figure 23 Resultant wait Node Structure

The following table shows the identifier used by the name generator to name the node, the format of the default name assigned to the node, and the number of occurrences of the node that may be created:

Name Identifier	Default Name	Occurrence
wait.for.node	"{@name}" wait for {@for}	1
wait.until.node	"{@name}" wait until {@until}	
wait.neither.node	"{@name}" (wait)	

The above table assumes that you are using the name generator file `FullNames.properties`.

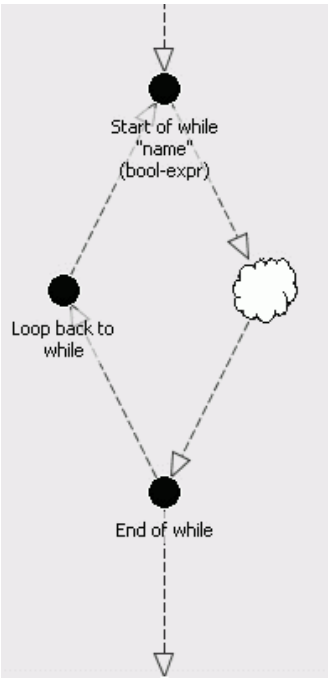
while

The BPEL XML

```
<while
  name="ncname"?
  condition="bool-expr"
  suppressJoinFailure="yes|no"?>
  source+
  target+
  <condition expressionLanguage="anyURI"?>
    ... bool-expr ...
  </condition>
  activity
</while>
```

OVBPI Node Structure

Figure 24 Resultant while Node Structure



The following table shows the identifier used by the name generator to name each node, the format of the default name assigned to each node, and the number of occurrences of each node that may be created:

Name Identifier	Default Name	Occurrence
while.start	Start of while "{@name}" ({@condition})	1
while.end	End of while	1
while.loop	Loop back to while	1

The above table assumes that you are using the name generator file `FullNames.properties`.