Sterling Commerce

Platform BPML

Platform IFC 1.0

Sterling Commerce
An IBM Company

Contents

Application and BPML

A basic understanding of BPML fundamentals will enhance your understanding of Application and facilitate your monitoring, troubleshooting, business process modeling, and process analysis activities.

Business Process Modeling Language (BPML) is an XML-based language used to describe (model) and run business processes. Each business process is defined by a single, unique BPML document known as a business process model (.bpml or .bp file). Each business process model is the definition of the process as it will be run in Application.

This topic explains the basic components of BPML that make up your business process models so that you can identify the components in your day-to-day interaction with Application operations.

Where to Find Industry Information

For comprehensive information about XML and BPML, and related topic, XPath, visit the following Web sites:

- Sterling Commerce Support on Demand, at https://support.sterlingcommerce.com/user/login.aspx
- The Business Process Management Initiative, at www.bpmi.org
- The World Wide Web Consortium, at www.w3c.org
- The United Nations Centre for Trade Facilitation and Electronic Business (UN/UNECE and the Organization for the Advancement of Structured Information Standards), at www.oasis-open.org

Business Process BPML Components

BPML code includes *activities* and *elements*. An activity is a step in a business process, and may be comprised of multiple elements. Elements are defined components of code that provide structure and instructions regarding the activity they embody. BPML refers to entities outside of the business process as *participants*. An example of a participant is an inventory system.

- A simple activity is a single step in a business process. For examples, see *Simple BPML Activities* on page 22.
- A complex activity is an activity that comprises a set of steps in a business process. For examples, see *Complex BPML Activities* on page 25.

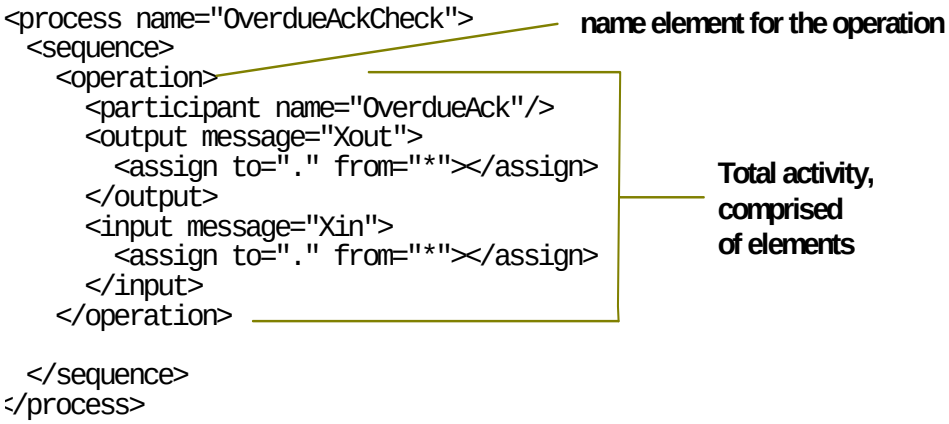
Activities within BPML code correspond to the icons you include in your business process models when you create them using the GPM. However, while an icon displays as a single, contained, unit, viewing the related BPML code shows the several elements that comprise the activity. You may refer to any service or other business process model component as an activity, but in the context of BPML, an activity may also be a BPML construct used to define the structure and progress of a business process flow.

The *operation activity* is a good example of this difference. The operation activity is the BPML component used to call a service within a business process.

Some BPML activities are represented by icons in the GPM (such as the Sequence icons and choice icons), while others are included within the service icons you can select in the GPM. For example, the operation activity is the BPML component used to call a service within a business process. If you use a text editor to write a business process model in BPML code, you include the operation activity, along with related

elements, to call a service. If you create a business process model using the GPM, simply including the appropriate icon for a service automatically builds the operation activity into the BPML source code for the process model.

For example, the following figure shows the BPML for a business process; within it, the operation activity is denoted by the <operation> name element:




```
<process name="OverdueAckCheck">
  <sequence>
    <operation>
      <participant name="OverdueAck"/>
      <output message="Xout">
        <assign to="." from="*"></assign>
      </output>
      <input message="Xin">
        <assign to="." from="*"></assign>
      </input>
    </operation>

  </sequence>
</process>
```

The elements between <operation> and </operation> define the particulars of the activity, including calling the EDI Overdue Acknowledgment Check service. The </operation> element indicates the conclusion of the operation activity.

If you create business process models by direct-coding, you are responsible for including all of the appropriate elements that make up activities. When you use the GPM to create your process models, the icons you include automatically create the required BPML element components (although you may also need to configure service parameters).

You can relate the example in the first figure to the GPM display in the following way:

- The Start icon provides the BPML code for the process name saved with the business process model.
- The Sequence Start icon provides the BPML for the sequence element.
- The participant name, output message, assign, and input message elements are provided by parameters associated with the EDI Overdue Acknowledgment Check service.

Process Element

The process element defines an activity and is the root element of a business process model.

- A process activity consists of exactly one simple or complex activity. The process completes after this activity completes.

- A process element begins and ends every business process model and can contain only one complex activity or one simple activity. The complex activity must contain all other activities required for the business process model in question. A process element uses the following syntax:

```
<process name>
<rule name/>*
( simpleActivity | complexActivity )
</process>
```

- A process element has a name attribute to indicate the name of the business process. The name attribute references any namespaces used in the business process.

```
<process name="ProcessCustomerOrder">
   .
   .
   .
</process>
```

What Is Process Data?

To configure business process models you need to understand process data and associated concepts.

Process data is data related to a business process that accumulates, according to configured instructions in the BPML, in an XML document during the life of the process. Business process writers use process data to enable manipulation of pieces of information that are crucial to completing the activities in the process, as follows:

- The BPML structure of business process models always includes a placeholder (root node) for process data.
- The creator of a business process model configures services and activities to put information in process data, and also configures services to access and use that data to complete the process activities. For example, a step in a process may pull a routing number from the primary document and add it to the process data, to be accessed by a subsequent step in the process that is configured to act using that information (the *primary document* is the core document in a business process, such as a purchase order).
- Exactly how these activities take place depends on the particulars of the configuration.
- The data that processes through a service in a step is always a combination of process data and the primary document.

The type of information a service places in process data is variable from service to service according to the service configuration and the task being completed. And some services operate solely on the primary document and never use process data at all. Both the process data and primary document may change during processing through a service.

Process data is likely to include:

- Information extracted from a document that is used for determining, from multiple choices, what the next step will be
- Information assigned in a process' BPML configuration to be used by a service in the business process, such as a map name or extract directory, which helps the service do its job but is not part of the primary document
- Information about the document or the processing of the document, placed by a service – such as a content type indicator or sender information, which helps a service do its job and is specific to the document

Because process data is XML, you can use XPath in your process models to access information within it. You can use an XPath statement to refer to primary document content even when the primary document is not in XML (see *XPath and Process Data* on page 15). Another benefit is that you can represent complex hierarchical data directly in the process data.

Ideally, services are configured to access the primary document, bypassing the need for process data in a step, however, complex processes generally require the use of process data to most efficiently obtain the desired results.

In your day-to-day monitoring activities, you can view process data for individual steps in a process, should you need to check the particulars of how the step uses process data. View the data through the Business Process Detail page (from the Administration menu, select **Business Processes** > **Current Processes** > *ID of the selected process* > **info** [in the Instance Data column for a step]).

If you need technical information about configuring business process models with regard to process data, you may find the rest of this topic useful. It assumes basic knowledge of BPML.

About Input and Output Messages and Process Data

Process data content is independent and is not directly available to services. Therefore, to configure a service to use information placed in process data, the process writer must configure the service to enable access to it.

The business process writer controls both the data that is sent *to* a service for a step and what data *from* a service is put into process data, by configuring input and output instructions in the process definition (BPML). The input and output data for a service is referred to as *input* and *output messages* (you can see these terms used in the Service Editor in the GPM and your service configurations in the Application interface).

The input and output instructions for the messages are BPML assign elements. Assign elements set a value in the business process data that is equal to a fixed value. Inside an input or output element, the assign element identifies a message it should receive from a participant or identifies the contents of a message it should send to a participant. Therefore, assign elements specify the message data, and messages are the mechanism by which services get data from and return data to a business process' process data. These messages are represented as XML (DOM) documents.

Note: In the GPM service editor, the output message – data being passed into the service – is referred to as the Message To Service, and the input message – data being passed out of the service – is named the Message From Service.

The syntax used to encode input and output messages with respect to the business process definition are as follows:

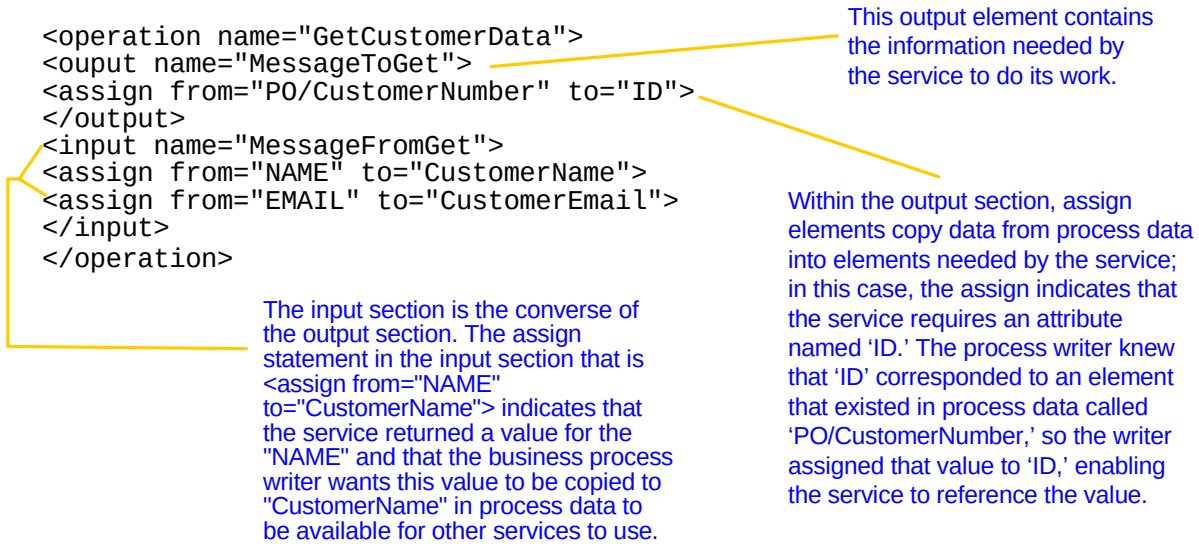
- The <output> tag specifies what data should be output from the business process to the service (that is, copied from the process data).
- The <input> tag specifies what data in the output of a service should be used as input to the business process (that is, copied to the process data).
- Not all services require input or produce output. Even if a service does produce output, the business process writer is not required to use any or all of the data returned by a service.
- Output assignments happen before the service executes. Input mappings happen after the service has completed.
- The 'from' attribute of an assign statement can be any XPath expression and can identify multiple nodes. The 'to' attribute of an assign statement must point to an individual node.

Note: Be sure to avoid the following possible problems when allowing a service to read and write the process data area directly:

- A service can read data that you may not have intended to be read.
- A service can write data, or overwrite data, that you did not intend to be written or overwritten.

Input and Output Examples

The following example illustrates output and input message configuration:



For this example, this is the process data:

```
<ProcessData>
  <PO>
     <CustomerNumber>12345</CustomerNumber>
  </PO>
</ProcessData>
```

Before the service runs, the output assigns are used to move data from process data to the input for the service. The message sent to GetCustomerData looks like:

```
<MessageToGet>
   <ID>12345</ID>
</MessageToGet>
```

The message produced by GetCustomerData looks like:

```
<MessageFromGet>
   <NAME>Bob Smith</NAME>
   <EMAIL>bob_smith@sterlcomm.com</EMAIL>
</MessageFromGet>
```

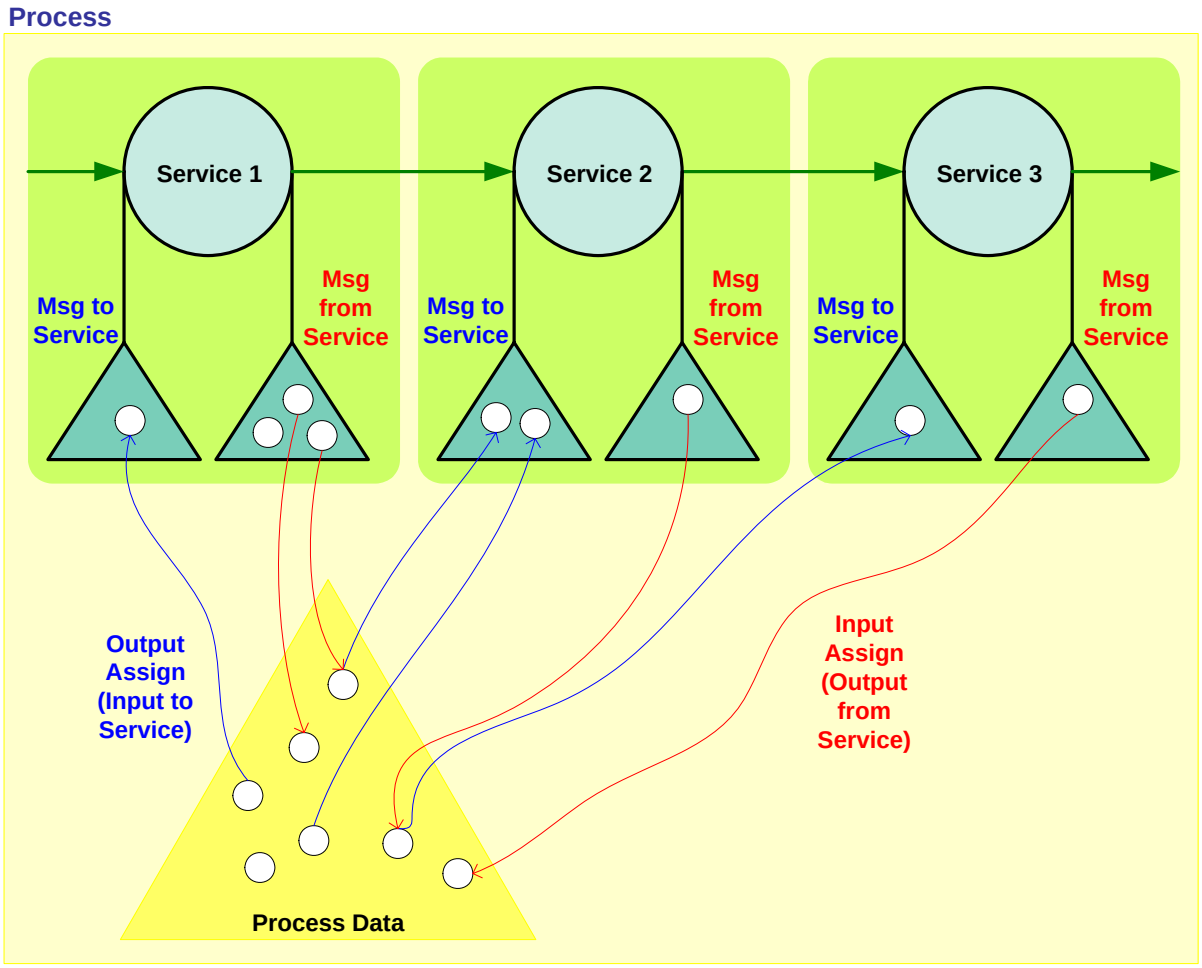
After the service runs, the input assigns are used to move data from the output produced by the service back into process data. After the input assignments are applied, the process data looks like:

```
<ProcessData>
   <PO>
      <CustomerNumber>12345</CustomerNumber>
   </PO>
   <CustomerName>Bob Smith</CustomerName>
   <CustomerEmail>bob_smith@sterlcomm.com</CustomerEmail>
```

```
</ProcessData>
```

The CustomerName and CustomerEmail data is now available for subsequent steps in the business process to use. Remember, the business process writer is not required to use any or all of the data returned by a service.

The following illustration shows a simple process that has three services. The triangles represent the XML documents (input and output messages) that contain the actual data values obtained from evaluating the assignments defined in each of the operations (services) in the process definition. The illustration provides a graphic example of how the messages and services interact with process data in a larger process.



Different Types of Assigns

The assign element is a BPML construct you can use to perform simple assignment operations within a service call, as illustrated in the first example (see *Input and Output Examples* on page 9). You can also use assign statements outside of any operation, input, or output tags. Sometimes called *top-level assignment*, this method enables you to manipulate the process data directly, outside the service call in a process rather than within the service step.

Within the Application application, the Assign service configuration is available for this purpose, and in the GPM, it is represented as the Assign *activity*, available as an icon from the BPML stencil. This topic discusses the assign element, not the service or activity, but it is important to note that top-level assignments follow the same conventions outlined here.

Note: If you are configuring a service in the GPM rather than by direct coding, and you want to include an assign element in the service configuration, add it to the parameters on the **Message To Service** or **Message From Service** tab, as appropriate. To add the element, click **Advanced**.

Using Assign Element To Copy All of Process Data Into the Message

In addition to the explicit assignments described thus far, you can also use assign elements within a service implicitly—that is, to copy *all* of process data into the Message To Service (output message) so that a service can access it. To use this method, code your assign element as follows:

```
assign from="*" to=".">
```

This instructs the service that all (*) of the process data document is available for use. You can use this type of assignment separately for both input and output messages. This is a catch-all construct used when a service may need access to all of process data. Recommended practice is to use *explicit* assignments in the messages. Explicit assigns, in which the process steps act on or pass on only the useful and necessary pieces of information in process data, yield more efficient overall processing that is less likely to negatively affect system performance.

In the following example, which is a fragment of a business process, the GetCustomerData service executes, processes, and produces the same information as in the first example (see *Input and Output Examples* on page 9). But, because there are no explicit assign elements, all of the data must be in the correct place in process data:

```
<operation name="GetCustomerData">
   <output name="MessageToGet">
      <assign from="*" to=".">
   </output>
   <input name="MessageFromGet">
      <assign from="*" to=".">
   </input>
</operation>
```

The following is process data:

```
<ProcessData>
   <ID>12345</ID>
</ProcessData>
```

As the service runs, it writes data directly to process data. After the service completes, the process data looks like the following:

```
<ProcessData>
   <ID>12345</ID>
   <NAME>Bob Smith</NAME>
   <EMAIL>bob_smith@sterlcomm.com</EMAIL>
   <ADDRESS>4600 Lakehurst Court, Dublin, OH, 43016</ADDRESS>
   <PHONE>614-793-7000</PHONE>
</ProcessData>
```

Consider the Order In Which You Configure Assigns

Configuring your processes is likely to involve situations in which a service needs to obtain parameters from both process data and messages. These kinds of scenarios involve multiple assigns, possibly both implicit and explicit, and the order in which the step processes the assigns is crucial to successful processing of the step. Your process definition may need to indicate whether services first request values from messages or process data.

Essentially, in cases where message data and process data are both involved in a step, you need to ensure that the assignments happen in the appropriate order, either by direct-coding them such that they process in the order you need, or by setting the order in the GPM Service Editor. You can specify different settings for input and output messages within the same service. In the GPM, you can select the following instructions for input and output messages:

Message To Service (Output Message)	Message From Service (Input Message)
◆ Messages Only – No process data is involved with this option.	◆ Allow Process Data write
◆ Obtain Message first, then Process Data – Use this to provide overrides for parameters. For example, if all but one parameter is set in process data, the one missing parameter could be put into the message for the service to pick up.	◆ Allow message write
◆ Obtain Process Data first, then Messages – In this scenario, when the service needs to look up data, it first checks the process data and then goes to the message. You can use this setting to provide defaults for parameters, such as when some parameters are not specified in process data.	

Specifying Constants (Literal Values) In Assign Elements

As well as obtaining runtime values from process data, you can also specify constants (or literals) as parameters. For example, if some parameters may not be specified in process data, defaults can be used.

Note: Because the BPML is represented as XML, literal values included in the ‘from’ attribute value must be enclosed in single quotes within the double quotes (“‘ ’”). For example `<assign to=”foo” from=”’bar’”/>`.

The following example illustrates the use of default parameters.

```
<operation name="GetCustomerData">
   <output name="MessageToGet">
      <assign to="ID">12345</assign>
      <assign from="*" to=".">
   </output>
   <input name="MessageFromGet">
      <assign from="NAME" to="CustomerName">
      <assign from="EMAIL" to="CustomerEmail">
   </input>
</operation>
```

In this example, the service configured with a setting of Obtain Message first, then Process Data and has a constant assignment to the ID parameter.

Tips For Accessing Content

The following methods are two ways to configure a a process to obtain the necessary data from the primary document and place it in process data:

- If the primary document is in XML format, use the DocTODOM XPath function.
- If the primary document is not in XML format, create a map for the XML Encoder service that copies only the required information from the primary document into process data.

XPath and Process Data

Use XPath syntax in an assign element to work with the process data. In the assign element, the two attributes *from* and *to* are XPath expressions. The from attribute indicates the location in the source context from which the business process should pull a value. The to attribute indicates the location in the target context in which the process should place a value.

The source and target contexts for an assign element depend on whether the assign element is in an output element, an input element, or an assign activity.

- In an output element, the target context is the message that the process sends to a participant. The source context is the process data.
- In an input element, the target context is the process data and the source context is the message the process receives from a participant.
- In an assign activity, the process data is both the source context and the target context.

The numbered examples following this section are based on the following sample process data:

```
<ProcessData>
   <PurchaseOrder>
   <LineItem>
   <Price>12.34</Price>
   </LineItem>

   <LineItem>
   <Price>95.61</Price>
   </LineItem>

   <LineItem>
   <Price>34.52</Price>
   </LineItem>
   </PurchaseOrder>
</ProcessData>
```

Example 1

Count the number of line items and assign the count to a variable called LineItem.

```
<assign to="LineItem" from="count()"/>
```

Example 2

Calculate the sum of the line items in the purchase order and assign it to a variable sum in the process data.

```
<assign to="sum" from="sum(PurchaseOrder/LineItem/Price)"/>
```

Example 3

Extract the first line item and save it to LineItem1 in the process data.

```
<assign to="LineItem1" from=" PurchaseOrder/LineItem[1]"/>
```

XPath and DocToDOM

The following function extracts the SCIObjectID attribute from the process data DOM and loads the corresponding document into process data:

```
DocToDOM (xpath expression [, validate, loadDTD])
```

The XPath expression is required to complete a query executes against the process data DOM tree. The node returned must have an SCIObjectID attribute. This enables the document to load by the ID and then parsed. You can add additional XPath criteria following the function call to run an XPath query against the document itself.

Additional functions shown in the previous example are:

- validate – Document validation value. Default value is true. Optional.
- loadDTD – Determines if the parser loads the DTD, as specified. Default value is true. Optional.

This document shows the XML file used in the following examples:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "examples.dtd">
<Testplan>
 <Test1>
  <Unit_Price>5.25</Unit_Price>
 </Test1>
 <Test2>
  <Customer_ID>3</Customer_ID>
  <Zip>43013</Zip>
  <Entry_Date>2006-10-10</Entry_Date>
 </Test2>
 <Test3>
  <Customer_ID>2</Customer_ID>
  <Quantity>80000</Quantity>
  <Entry_Date>11/2/2006</Entry_Date>
 </Test3>
 <Test4>
  <Customer_ID>5</Customer_ID>
  <SirName>Mr.</SirName>
  <FirstName>Joe</FirstName>
  <LastName>Smith</LastName>
  <Address>555 Main St.</Address>
  <City>Anywhere</City>
  <State>OH</State>
  <Zip>55555</Zip>
  <Phone_Num>555-5632</Phone_Num>
  <Orders_Placed>2</Orders_Placed>
 </Test4>
 <Test5>
  <Customer_ID>5</Customer_ID>
  <City>Columbus</City>
  <Entry_Date>12/5/2006</Entry_Date>
 </Test5>
 <Test6>
  <Customer_ID>5</Customer_ID>
 </Test6>
 <Test7>
  <Unit_Price>43.25</Unit_Price>
```

```
 </Test7>
 <Test8>
  <Customer_ID>3</Customer_ID>
  <Zip>43013</Zip>
  <Entry_Date>2006-10-10</Entry_Date>
 </Test8>
 <Test9>
  <Customer_ID>2</Customer_ID>
  <Quantity>80000</Quantity>
 </Test9>
 <Test10>
  <Customer_ID>1</Customer_ID>
  <City>Columbus</City>
 </Test10>
</Testplan>
```

DocToDOM function In an Assign Statement Examples

In the following example, the nodes returned are attached to process data at the root node:

```
<assign to="." from="DocToDOM(PrimaryDocument)"></assign>
```

In the following example, additional XPath (Test1) executes against the document to return only the nodes that exist under Test1. The result is then attached to process data, under the message_test node:

```
<assign to="message_test" from=" DocToDOM(PrimaryDocument)/Test1"></assign>
```

In following example, validation is turned off and enables the DTD to load:

```
<assign to="." from="DocToDOM(PrimaryDocument, 'false','false')"/>
```

The following BPML example shows the assign statement:

```
<process name="DocToDOM">
  <sequence>
   <assign to="." from="DocToDOM(PrimaryDocument)"></assign>
  </sequence>
</process>
```

This function runs an XPath query against the process data DOM and saves the node into a workflow document:

```
DOMToDoc(xpath expression [, document_name, standalone, root_name, encoding,
systemURL, dtdName, publicURL ])
```

The following parameters are shown in the previous example:

- XPath expression – XPath query to execute against the process data DOM tree. Required.
- documentName – Name of the workflow document. The default value is Document. Optional.
- standAlone – Value indicating whether this document is a single document (stands alone). Default value is no. Optional.

- rootName – Name of the document element that stores the result of the XPath query. This entire node is written to the workflow document. If a name is not provided, the document element stores the root node returning from the XPath query. Optional.
- encoding – String value that builds the encoding section of the XML header and specifies the encoding type of the workflow document. If an encoding is not provided, the default parameter is UTF-8. Optional.

The next set of parameters rebuilds the DOCTYPE element that was lost after executing the XPath query:

- systemURL – Value of the system URL. Optional.
- dtdName – Value of the DTD name. If a value for this parameter is specified, the publicURL parameter is ignored. Optional.
- publicURL – Value of the public URL. If a value for this parameter is specified, the DtdName parameter is ignored. Optional.

The following example shows the use of the DOMToDoc function in an assign statement:

```
<assign to="message_test" from="DOMToDoc(Testplan)"></assign>
```

The XPath specified in the following example identifies the nodes that exist below the root node. Therefore, do not specify the root element in your XPath expression:

```
<Root>
 <Testplan>
  <Test1>some data</Test1>
 </Testplan>
</Root>
```

If you want to return the text in the Test1 node then my XPath will be:

```
DOMToDoc(Testplan/Test1/text())  Correct.
DOMToDoc(//Root/Testplan/Test1/text())  Wrong.  Don't specify the root node in the
xpath query.
```

DocToDOM Function

The following example shows an XML document loading in to process data:

```
<process name = "DocToDOM_Example1">
  <sequence>
    <assign to="." from="DocToDOM(PrimaryDocument)" />
  </sequence>
</process>
```

Load a portion of an XML document into process data. The XPath query does not reference the document element following the DocToDOM call. The nodes returned are stored under the Testpoint tag:

```
<process name = "DocToDOM_Example2">
  <sequence>
    <assign to="Testpoint" from="DocToDOM(PrimaryDocument)/Test4" />
  </sequence>
</process>
```

DOMToDoc Function

The params in brackets “[ ]” are optional parameters.

DOMToDoc(XPath,[‘Document Name’],[‘stand alone?’],[‘root name’],[‘encoding’],[‘system url’],[‘dtd name’],[‘public url’],[‘attribute quoting’])

DOMToDoc Parameter Definitions

XPath

The xpath parameter is the only required parameter. It is an xpath to the xml node in the business process instance data (ProcessData) that you want extracted into the document.

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode)" />
```

Document Name

The document name value, if specified is the name the document in process data is given, with the data from the xpath node specified. If none is specified the default value of “Document” is used.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,’DOMToDoc_Document’)" />
```

Stand Alone

The values allowed for this option is “YES” or “NO”. It tells the function which value to use in the xml declaration line of the document. (<?xml version="1.0" encoding="UTF-8" standalone="YES or NO"?>) The default value is “no”.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,’’,’YES’)" />
```

Root Name

The root name sets the root xml element, to wrap the data specified by the XPath in the xml document created, and will specify it in the DOCTYPE declaration. The default value is null, which means the xml document will contain just the nodes specified in the XPath without any root element wrapping that data.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,’’,’’,’docroot’)" />
```

Default Behavior with XPath = /ProcessData/test

```
<ProcessData>
<test>
<data>123</data>
</test>
</ProcessData>
Becomes….
<?xml version=”1.0” encoding=”UTF-8” standalone=”no”?>
<test>
```

```
<data>123</data>
</test>
Behavior when specified
<ProcessData>
<test>
<data>123</data>
</test>
</ProcessData>
Becomes….
<?xml version=”1.0” encoding=”UTF-8” standalone=”no”?>
<docroot>
<test>
<data>123</data>
</test>
</docroot>
```

Encoding

The encoding parameter sets the documents character encoding. This is a string value, and will set the character encoding to any string passed, so it is important that you pass a valid encoding. No validation is done against this parameter. The default value is “UTF-8”.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,’’,’’,’’,’UTF-16’)" />
```

System URL

Specifying this parameter will create a document type declaration in the xml document and set it to SYSTEM with the url being the string specified. The default is null, meaning no declaration of system will be done.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,’’,’’,’’,’’,’http://ster-
comm.com/xml/example’)" />
```

This will result in the document having <!DOCTYPE SYSTEM "http://stercomm.com/xml/example">

DTD Name

Specifying this parameter will create a document type declaration in the xml document and set it to PUBLIC with the value being the dtd name specified in the value of this parameter. The default is null, meaning no declaration of PUBLIC will be done.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,’’,’’,’’,’’,’’,’testdtd’)"
/>
```

This will result in the document having <!DOCTYPE PUBLIC "testdtd">

Public URL

Specifying this parameterwill create a document type declaration in the xml document and set it to PUBLIC with the value being the dtd name specified in the value of this parameter. The default is null, meaning no declaration of PUBLIC will be done.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,'','','','','','','publi-
curl')" />
```

This will result in the document having <!DOCTYPE PUBLIC "publicurl">

Attribute Quoting

This parameter specifies whether to wrap xml attributes in single quotes or double quotes. Both are completely valid xml. The parameter takes a value of “single” or “double” with single quotes as the default behavior.

Usage Example:

```
<assign to="." from="DOMToDoc(/ProcessData/beginningxmlnode,'','','','','','','dou-
ble')" />
```

This will result in the document having <test name=”testattribute”> instead of <test name=’testattribute’>

DOMToDoc Examples

The following example shows the output of the process data tree in a document:

```
<process name = "DOMToDoc_Example1">
  <sequence>
    <assign to="." from="DocToDOM(PrimaryDocument)" />
    <assign to="." from="DOMToDoc(//)" />
  </sequence>
</process>
```

The following example shows the output of the document that loaded into process data and back into a document. I nodes returned from the DocToDOM call were placed into the Testpoint tag because the DOMToDoc call did not return the tag specified in the XPath. This prevents the loss of the document element.

```
<process name = "DOMToDoc_Example1">
  <sequence>
    <assign to="Testpoint" from="DocToDOM(PrimaryDocument)" />
    <assign to="." from="DOMToDoc(Testpoint)" />
  </sequence>
</process>
```

Simple BPML Activities

Simple BPML activities enable a business process to communicate with a participant. There are three types of simple activities:

- Produce – Sends messages to participants asynchronously
- Consume – Receives messages from participants asynchronously
- Operation – Exchanges messages with participants synchronously

Produce and Consume Activities

The produce and consume activities enable business processes to communicate. Produce and consume are used together.

Produce Activity

The produce activity sends a message to another business process.

In the following sample BPML code, a message called EmployeeDataMessage is sent to a business process instance created by the spawn activity. The assign elements write the name and social security number of an employee into the message. Note the significance of the name of the message; the spawned instance looks for a message with exactly that name.

```
<produce name='SendMessage'>
   <participant name='AnotherProcess'/>
   <output message='EmployeeDataMessage'>
   assign to='EmployeeName'>David</assign>
   assign to='SocialSecurityNumber'>123-45-6789</assign>
   </output>
</produce>
```

Consume Activity

The consume activity reads in a message sent to it by another business process. A consume element can contain a participant element. The participant element indicates which business process instances are relevant to the consume activity.

The following sample BPML code represents the simplest form of the consume activity. The activity reads in a message called EmployeeDataMessage, and then stores the message in process data.

```
<consume name='ReceiveMessage'>
   <input message='EmployeeDataMessage'>
   </input>
</consume>
```

The consume activity can also indicate that it will read a message only from a particular sending process, and store only parts of the received message in process data.

Operation Activity

The operation activity invokes an action against a participant. Using an operation activity is the only way that you can call a service.

The following specifications apply to the operation activity:

- An operation involves a synchronous request/response message exchange with a possible fault message.
- When an operation is invoked, it delivers a request message and waits for a response message. If a fault is communicated, the operation faults.
- How the business process communicates with a participant is left to the implementation. Use extension elements to provide implementation details.
- Use the operation element to invoke an operation on a service.
- The participant element must be used.
- The output element must precede the input element.

Participant Element

Use the participant element to define a participant in the business process, or to reference a participant within a simple activity. The participant element is used in the operation activity and with the produce and consume elements.

In the business process or a simple activity, the participant element defines a static participant that must be named using the name attribute. The definition can include annotations, metadata, and extension elements.

How the business process communicates with the participant and applies the proper security restrictions is left to the implementation. Use extension elements to provide implementation details.

The following sample operation refers to the online bookseller scenario example discussed in the previous sections, where the bookseller Process Customer Book Order process contacts the Inventory service to determine whether a book is in stock.

The participant, output, and input elements are required and must be in the order shown in the following example. Follow the specification for each line in the example exactly.

Line No.	BPML
1	<operation name='User Name'>
2	<participant name='InventoryService' />
3	<output message='checkStockRequest'>
4	<assign to='ISBN'>1-56592-488-6</assign>
5	</output>
6	<input message='checkStockResponse'>
7	<assign to='foundBook' from='InStock' />
8	</input>

Line No.	BPML
9	</operation>

The following table describes the elements and activities of each line in the BPML:

Line No.	Description
1	Type of activity is an operation. Supplying a name for the operation activity is optional. **Note:** This name has no impact when running the business process, but the GPM uses it as a label.
2	Operation activity communicates with a participant. The participant element in this line identifies the participant with which this operation is associated. For Application, the name attribute must match the name of an installed service, service instance, or service configuration. In this case, the name identifies the Inventory service.
3	Output element contains the information to send to the service. The message attribute is required in BPML, because a participant can perform different tasks, depending on the different messages it receives. For example, when the inventory service receives a checksStockRequest message, it determines an item is in stock. A restockItemRequest message prompts the inventory system to restock the item.
4	An assign element within the output element provides the required information for the output message. For example, if the business process and participant communicate through a message, the message attribute of the output element specifies the title, checkStockRequest. The assign element on line no. 4 specifies the first line of the message, ISBN = 1-56592-488-6. Each additional assign element in that output element adds another line to the message.
5	Ends the output element.
6	Within an output element, an input element identifies the response the process expects from the participant. When the participant receives the message written by the process, the participant responds with a return message that the input element reads. The message attribute of the input element identifies the name and type of the return message expected from the participant. Because a participant can send more than one type of message, a message attribute is required for the input element. The message attribute indicates which message is expected.
7	The assign attribute in an input element differs from the assign attribute in the output element. In the input element, the assign attribute reads information from the message and writes that information in the process data. For example, if the book for which the process performs a stock check is available, the InventoryService response message contains a line similar to InStock = true. The assign element within the input element writes the value from the InStock line to the process data. The InStock line includes an identifier written to the process data. In the bookseller example, the value from the InStock line is true, so the line written to the process data would be similar to foundBook = true.
8	Ends the input element.
9	Ends the operation activity.

Complex BPML Activities

A complex activity is a combination of simple activities that are child activities to the parent complex activity. The child activities can also be simple or complex.

A process element can contain only one parent complex activity. The parent complex activity contains all of the simple and complex activities necessary to complete the business process.

There are three kinds of complex activities:

- Sequence (serial)
- Choice (conditional)
- All (parallel)

Sequence Activity

A sequence activity runs a series of child activities in the order in which it lists them. When a process runs, all of the child activities are run. The sequence activity finishes only after the last child activity is finished.

The following example shows a sequence activity that contains two child activities: Check Inventory and Verify Credit Card. When the process runs, Application runs Check Inventory first and Verify Credit Card second because that is the order in which the activities are listed in the sequence. The name attribute in the sequence element is optional.

```
<process name="ProcessCustomerOrder">
   <sequence>
   <operation name='Check Inventory'>
   <participant name='InventoryService' />
   <output message='checkStockRequest'>
   <assign to='ISBN'>1-56592-488-6</assign>
   </output>
   <input message name='checkStockResponse'>
   <assign to='foundBook' from='InStock' />
   </input>
   </operation>
<operation name='Verify Credit Card'> </operation>
   </sequence>
</process>
```

All Activity

The all activity contains two or more complex child activities and runs all of them simultaneously. The all activity finishes only after the child activities are finished.

The following example contains three child activity sequences: Seq_1, Seq_2, and Seq_3. Application begins these sequences at the same time—that is, the first operations in the sequences (operations A, C, and E) start simultaneously. The sequences continue to run independently until the last operation in each is completed. The all activity finishes when all three child activity sequences finish.

```
<all>
   <sequence name='Seq_1'>
```

```
   <operation name='A'> … </operation>
   <operation name='B'> … </operation>
   </sequence>

   <sequence name='Seq_2'>
   <operation name='C'> … </operation>
   <operation name='D'> … </operation>
   </sequence>

   <sequence name='Seq_3'>
   <operation name='E'> … </operation>
   <operation name='F'> … </operation>
   <operation name='G'> … </operation>
   </sequence>
</all>
```

Choice Activity

The Choice activity makes decisions in the business process model and runs only one of the child activities it contains. Conditions determine which child activity runs.

You define the rules for the choice conditions. The only child activity to run is the one tied to the first case statement in the Choice activity that matches a rule. If none of the case statements in the Choice activity match a rule, the process continues to the next activity after the choice.

Branching

The choice activity makes it possible to model process branching. To model process branching, the process must evaluate one or more rules to reach a decision, and then specify which activity to run as a result of that decision.

- The choice activity must include the select element.
- The select element must include one or more case elements.
- Each case element links the outcome of a rule to a child activity. The activity attribute in the case element identifies the child activity to run.
- Each case element has a ref attribute, which contains the name of a rule that is defined at the top of the process. Case elements refer to rules instead of conditions because:
  - A rule can contain more than one condition. If a rule element contains more than one condition, all the conditions in the rule must be true in order for the rule to be true.
  - A condition is a single entity within a rule.
- If the rule is true, the activity runs.
- If the rule is false when the negative attribute is true, the activity runs.
- Multiple cases can reference the same activity. However, the activity runs only once.
- If an activity is not referenced, it does not run.
- If no activity runs, the choice activity completes immediately.
- Use a choice element with a simple activity to model a deferred activity.

- ✦ Priorities and defaults are modeled through rule dependencies, not through the order of the case elements.

When the business process runs the choice activity, Application checks the case statements in the order that they are listed in the select element. The first case statement in the select element determines whether the rule is true. If the rule is true, the activity named by the activity attribute is run immediately. When that activity completes, the choice activity is finished. If the second case statement in the select element refers to the same rule as the first but has a negative attribute set to true, the case statement runs the named activity if the rule is false.

In the online bookseller example discussed in previous sections, the Process Customer Book Order process verifies a customer's credit card only if the book the customer selects to buy is in stock. The BookInStock rule is true if the book is in stock and false if the book is out of stock.

If the inventory system determines that the book is in stock, the rule referred to in the case statement is true and the proceed activity is run.

```
<process name="ProcessCustomerOrder">
   <rule name="BookInStock">
   <condition>foundBook = true </condition>
   </rule>

   <sequence>
   <operation name='Check Inventory'>
   <participant name='InventoryService' />
   <output message='checkStockRequest'>
   <assign to='ISBN'>1-56592-488-6</assign>
   </output>
   <input message name='checkStockResponse'>
   <assign to='foundBook' from='InStock' />
   </input>
   </operation>

   <choice>
   <select>
   <case ref="BookInStock" activity="proceed"/>
   <case ref="BookInStock" negative="true" activity="stop"/>
   </select>

   <sequence name="proceed">
   <operation name='Verify Credit Card'> … </operation>
   …
   </sequence>

   <sequence name="stop">
   <operation name='Apologize to Customer'> … </operation>
   </sequence>
   </choice>

   <operation name='Update customer on status'> … </operation>
   </sequence>
</process>
```

Rule Element

The rule element defines a rule, the conditions by which the rule is met, and dependency on other rules.

- The condition element formulates an expression. The condition is met if the expression evaluates to true, or if the expression evaluates to false when the negative attribute is true.
- Multiple conditions inside a rule are listed in logical order.
- The rule element defines a rule that is referenced in a choice activity, used in an input element, or dependent on another rule. Dependencies are not affected by the order that rules are listed.

Condition Element

The contents of the condition element must correspond to an XPath expression. Application expects an XPath expression to evaluate a condition.

Process Activities and Elements

You can use several BPML activities and elements when creating business process models.

Input and Output Elements

The input element accepts a message delivered to the process. The input element has the following syntax:

```
<input message>
   <assign/>*
</input>
```

- Use the input element in the operation activity to accept a message delivered from a participant to the process (process input).
- Use the message attribute to reference the relevant message definition.
- Use the assign element to perform assignment (move data) from the message contents to the process data. Use multiple assignments for multi-part messages.

The output element constructs a message delivered by the process to a participant. The output element has the following syntax:

```
<output message>
   <assign/>
</output>
```

- The message attribute references the relevant message definition.
- Use the assign element to perform assignment (move data) from the process data to the message contents. Use multiple assignments for multi-part messages. No assignments are required for empty messages.

Assign Activity

The assign element performs assignment and is a process activity. When using assign elements in an operation, you must know the appropriate service parameters, such as the map name that the translator needs.

Some specifications for assign elements are:

- How the assign element is used determines what it uses as a data source destination.
- In the output element, the assign element performs assignment from the process data to the outgoing message. The to attribute corresponds to the path within the outgoing message and is used to construct the message contents.
- In the input element, the assign element performs assignment from the incoming message to the process data. The to attribute corresponds to a value within the process data.

The following example uses the assign element as an activity within an operation:

```
<operation name='name'>
   <participant name='name of specific service'/>
   <output message='output message from service'/>
   <input message='input message for service'>
```

```
   <assign to=’z’ from =‘x’/>
   </input>
</operation>
```

Using Assign as an Independent Activity

The assign elements works the same as the assign activity inside an input or output element, but it is not tied to a message. Used as an independent activity, the assign element inserts fixed numeric values or fixed strings into the process data. Application accepts the assign element only when it sets something in the process data equal to a fixed value.

When the assign element is used as an independent activity, it performs assignment from a constant to the process data. The to attribute corresponds to a path within the process data. The from attribute is used to extract information from a previously assigned value.

In the following example, the value 7 is assigned to X:

```
<assign to='X'>7</assign>
```

Assigns outside a service are visible to the whole process after the assign. Assigns within the service are only visible to the service.

Repeat Activity

The repeat activity repeats a complex activity without recursion. It directs Application to return to the beginning of a parent or child state, or to run an activity again. The repeat activity can be used only as a child activity of the complex activity that it references.

The repeat element accepts the ref attribute, which names the complex activity to be repeated. The activity to be repeated must be a complex activity that contains (directly or indirectly) the repeat activity.

In the following example, the repeat element is directly contained in the proceed sequence, which is contained in the choice element. This means that the choice sequence indirectly contains the repeat element.

```
<process name='repeat'>
   <rule name='checkX'>
   <condition>X true = </condition>
   </rule>

   <sequence>
   <assign to='X'>true</assign>

   <choice name='loop'>
   <select>
   <case ref="checkX" activity="proceed"/>
   </select>

   <sequence name="proceed">
   <operation name='…'> … </operation>
   …
   <repeat ref='loop' />
   </sequence>
   </choice>
```

```
   </sequence>
</process>
```

In this example, X is set to true. The choice element checks for the value X. Because X is true, the activities that are part of the proceed sequence begin to run. The last activity in the sequence is the repeat. The repeat activity refers to the choice named loop, so the process tries to run the choice activity again. If X is still true, the proceed sequence runs again. The sequence continues to repeat until X changes and becomes false. If X does not change to false, the process runs indefinitely, in an infinite loop.

Join and Spawn Activities

The join and spawn activities are used together. Use the join activity to wait for subprocesses to be merged. The join waits only for the subprocesses initiated from the same process in which the activity is run.

Use the spawn activity to start up a new process from within an existing process. The syntax is simple:

```
<spawn ref='another process'/>
```

The ref attribute must match the name of a process in Application. After this process is run, a new instance of a business process is created and run. The child process starts off with a copy of process data from its parent. The two processes are completely independent of one another—the original process will not wait for the spawned process to complete unless you explicitly configure it to.

The original and spawned processes can communicate using the produce and consume activities.

onFault Element

The onFault element is used to handle errors. You can include onFault elements in any complex activity for which it may be necessary to recover from faults so that the process can continue. The onFault element contains a fault handling activity. For information about using the onFault group in the GPM, see *Using the onFault Group in the GPM*.

The following specifications apply to the onFault element:

- If the complex activity containing the onFault activity faults before the complex activity is finished, the onFault activity runs.
- If the complex activity containing the onFault activity finishes successfully, the onFault activity finishes successfully, too.
- Application accepts multiple onFault elements within a single complex activity, making it possible to handle different fault codes.
  - Each set of onFault elements must use a unique fault code.
  - Omit the code attribute for only one element when using multiple onFault elements for a single complex activity. Here is the syntax:

    ```
    <onFault code?>
    activity+
    </onFault>
    ```

- In BPML, each fault can be associated with a unique code attribute. If the code attribute is provided, the associated onFault element is triggered only by a fault that matches that code attribute. For example, the onFault element has a code attribute set to SystemBusy. If the operation results in a fault associated with a different code attribute, the SystemBusy onFault element does not run.
- You can force the process to run the onFault activity for any fault encountered, by omitting a code attribute.
- If there are multiple layers of joins in BPML, each layer has onFault defined, an error occurs in the branch, and the final join of the inner join layer carries the error, the onFault of this layer is executed (even though the execution of the onFault does not have an error) and the error in the inner final join is propagated to the next layer of the join. This should not happen, because the inner layer of the join went to the inner onFault route and the on fault completes successfully. Avoid using complex multiple join layers in the BPML. Use Invoke Service Sync Mode for the branches; this mode merges all data from the subprocesses.

In the credit card example, the Verify Credit Card activity results in a SystemBusy fault, causing the process to run the onFault element. When the onFault element runs, the complex activity that it contains also runs. The onFault activity finishes when the complex activity it contains finishes. The complex activity that contains the onFault finishes at the same time, regardless of whether it was fully or partially run. As a result, the Verify Sufficient Credit activity does not fully run.

```
<process name="ProcessCustomerOrder">
   <rule name="BookInStock">
   <condition>foundBook true = </condition>
   </rule>

   <sequence>
   <operation name='Check Inventory'>
   <participant name='InventoryService' />
   <output message='checkStockRequest'>
   <assign to='ISBN'>1-56592-488-6</assign>
   </output>
   <input message name='checkStockResponse'>
   <assign to='foundBook' from='InStock' />
   </input>
   </operation>

   <choice>
   <select>
   <case ref="BookInStock" activity="proceed"/>
   <case ref="BookInStock" negative="true" activity="stop"/>
   </select>
   <sequence name="proceed">
   <sequence>
   <operation name='Verify Credit Card'> … </operation>
   <operation name='Verify Sufficient Credit'>…</operation>
   <onFault code='SystemBusy'>
   <sequence>
   <!-- Logic to wait and retry -->
   </sequence>
   </onFault>
   </sequence>
   …
   </sequence>
```

```
   <sequence name="stop">
   <operation name='Apologize to Customer'> … </operation>
   </sequence>
   </choice>

   <operation name='Update customer on status'> … </operation>

   <onFault>
   <sequence>
   <operation name='Inform Customer of Error'> … </operation>
   <operation name='Signal Operator'> … </operation>
   </sequence>
   </onFault>
   </sequence>
</process>
```

When Fault Code Does Not Match onFault Code

In a business process, a fault can occur in a complex activity that is not handled by an onFault element at that level of the process. When the fault code does not match the onFault element at a level, Application begins to search for the correct onFault element. When no code attribute exists and no match to a defined attribute exists for an onFault element at a level, Application moves to a general onFault (no code attribute) element. If Application does not find a fault code, Application halts the process.

In the credit card example, the Verify Credit Card activity results in a NetworkDown fault because Application first looked at the onFault elements associated with that activity. Application found only one onFault element in this activity and the code specified was not NetworkDown. Application then looked for the correct onFault element in the next activity up in the process. Eventually, Application found a general onFault element for which no code attribute exists at the highest level of the process. When the fault handling activities finish, Application runs the next activity in the process until no other activities exist in the process.

Index

P

R

S

X