

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Chapter 8. BPEL Activity Highlights

This chapter highlights selected BPEL activities, as well as event handlers.

As noted earlier, BPEL activities are divided into two categories. A *basic activity* does a task such as assigning a value to a variable. A *structured activity* embeds other activities (basic or structured) and specifies an order or condition that affects the circumstance in which those activities run.

Any activity can include either or both of the following optional, standard attributes: `name` and `suppressJoinFailure`. The `name` attribute's value is used for documentation and possibly for display in a BPEL editor, but the attribute has an additional use in compensation. The `compensateScope` activity uses the `name` value to refer to a particular `scope` or `invoke` activity, as described later.

The `suppressJoinFailure` attribute affects what happens in a `flow` activity after a join condition fails. If the attribute value is `yes`, the BPEL engine issues the activity that logically follows the target activity. If the attribute value is `no`, the BPEL engine throws a fault.

Start Activities

In a BPEL process, the first activity (other than `flow`, `scope`, or `sequence`) must be a *start activity*, which is an activity that can create an instance of the BPEL process. A start activity must be able to receive a message and must have a `createInstance` attribute that is set to `yes`. Two kinds of start activities are available:

- a `receive` activity
- a `pick` activity that contains only `onMessage` events

A `flow` activity causes concurrent processing, so a start activity embedded in that activity isn't necessarily the first to run. Two rules apply. First, every activity that may run first in the process must be a start activity. The following outline isn't valid because in the second `receive` activity, the `createInstance` attribute is set to `no` by default, and that activity can be issued before the BPEL engine creates the process instance.

```
<!-- Not Valid! -->
<process>
  <flow>
    <receive createInstance="yes"/>
    <receive/>
  </flow>
</process>
```

Second, if multiple start activities are in a `flow` activity and any of them reference correlation sets, these activities must reference at least one correlation set in common; also, the `initiate` attribute for each of the common correlations must be set to `join`, as in the following outline.

```
<process>
  <correlationSets>
    <correlationSet name="invoiceSet"
      properties="invoiceNumber countryCode">
    </correlationSet>
  </correlationSets>
  <flow>
```

```
<receive createInstance="yes">
  <correlations>
    <correlation set="invoiceSet" initiate="join"/>
  </correlations>
</receive>
<receive createInstance="yes">
  <correlations>
    <correlation set="invoiceSet" initiate="join"/>
  </correlations>
</receive>
</flow>
</process>
```

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

assign Activity

The `assign` activity copies sources to targets.

A source can be a literal, an expression, a property, a partner link, a variable, or a part of a variable; and you may use a `query` element to retrieve data from a variable that is based on a message type. The same kinds of sources are available when you initialize a variable.

A target can be a property, a partner link, an expression, a variable, or a part of a variable; and you may use a `query` element or expression to identify a field in which to place a value.

Your BPEL engine also may provide an *extension assign operation*, which is an assignment that is not defined in the language itself.

The `assign` activity can copy multiple values, as shown in [Listing 8.1](#).

Listing 8.1. assign activity

```
<assign name="assignValues">
  <copy>
    <from variable="temporaryVariable"/>
    <to variable="quoteRequest"/>
  </copy>
  <copy>
    <from variable="quoteRequest"
      part="placeQuoteParameters"/>
    <query>
      /quoteInformation/applicantEmailAddr
    </query>
    </from>
    <to variable="emailAddress"/>
  </copy>
  <copy>
    <from variable="mvRecord" property="licenseNumber"/>
    <to variable="vehicleLicense"/>
  </copy>
  <copy>
    <from>$input.msgPart/descendant::Make[.="Ford"]</from>
    <to variable="output"/>
  </copy>
</assign>
<assign name="assignLink">
  <copy>
    <from partnerLink="dmv"
      endpointReference="partnerRole"/>
    <to variable="partnerDMVrole"/>
  </copy>
</assign>
```

The copies occur in sequential order, and the values assigned earlier are available for later operations in the same `assign` activity.

The `assign` activity includes the `validate` attribute, which defaults to *no*. If that attribute is set to *yes*, every variable changed in the `assign` activity is subject to being compared against the WSDL or XML Schema definition on which the variable is based. A validation failure causes the runtime *bpel:invalidVariables* fault, but some BPEL

engines turn off the validation for better performance.

Literals

The following examples demonstrate use of the `assign` activity to copy literals.

For a simple type:

```
<assign>
  <copy>
    <from>
      <literal>Main Street</literal>
    </from>
    <to variable="address" />
  </copy>
</assign>
```

For a string (such as `<You&Me>`) that must be hidden from the BPEL engine:

```
<assign>
  <copy>
    <from>
      <literal>
        <![CDATA[<You&Me>]]>
      </literal>
    </from>
    <to variable="printString" />
  </copy>
</assign>
```

For an empty string:

```
<assign>
  <copy>
    <from>
      <literal />
    </from>
    <to variable="emptyString" />
  </copy>
</assign>
```

For a complex type:

```
<assign>
  <copy>
    <from>
      <literal>
        <VehicleList>
          <OneVehicle/>
        </VehicleList>
      </literal>
    </from>
    <to variable="AllVehicles" />
  </copy>
```

```
</assign>
```

For a type that must be qualified by a namespace prefix:

```
<assign>
  <copy>
    <from>
      <literal>
        <highlight:VehicleList xmlns:highlight="abc">
          <highlight:OneVehicle/>
        </highlight:VehicleList>
      </literal>
    </from>
    <to variable="AllVehicles"/>
  </copy>
</assign>
```

Types and Namespaces

In general, a source and target in an `assign` activity must have compatible types. You cannot assign a string to a variable that is based on type `xsd:int`, for example, nor can you assign a variable based on a message type to a variable that is not based on a message type.

Let's look at a few other requirements:

- If the source is a variable of a given message type, the target must be a variable of the same message type.
- If the source is a variable of a given XSD element or type, the target must be a variable of the same XSD element or type.
- If the source is a combination of variable name and part, the definition of the target must be identical at the level of the XSD element or type definition. (Zzzzz. . .)

[Listing 8.2](#) shows an example of two message definitions, which may be in different WSDL files, and [Listing 8.3](#) shows the element on which each message part is based.

Listing 8.2. Sample message definitions

```
<wsdl:message name="receiveMsg">
  <wsdl:part element="highlightXSD:transferElement"
    name="receivePart"/>
</wsdl:message>

<wsdl:message name="sendMsg">
  <wsdl:part element="highlightXSD:transferElement"
    name="sendPart"/>
</wsdl:message>
```

Listing 8.3. Sample element definition

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://com.highlight/policy/xsd/"
  xmlns:highlightXSD=
    "http://com.highlight/policy/xsd/">

  <xsd:element name="transferElement">
    <xsd:complexType>
      <xsd:sequence>
```

```

        <xsd:element name="msgInfo"
                    type="highlightXSD:NoticeType" />
    </xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:complexType name="NoticeType">
    <xsd:sequence>
        <xsd:element name="noticeID"
                    type="xsd:string" />
        <xsd:element name="fields"
                    type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

Listing 8.4 shows an `assign` statement that works if you've created a variable for each message type. The statement works, however, only because the message parts are based on the same element, which means that the data types (including namespaces and local names) are identical.

Listing 8.4. Sample `assign` activity for message parts based on same XSD element

```

<assign name="copyNotice">
    <copy>
        <from variable="receiveVariable" part="receivePart" />
        <to variable="sendVariable" part="sendPart" />
    </copy>
</assign>

```

The name of any XSD type is a combination of namespace (which is often represented by a prefix) and local name, regardless of whether the type is `xsd:int` or a complex type that you created. If the source in your `assign` activity is a variable name and property, for example, the type of the target must be the same as the type of the property.

You may not be thinking about namespaces, however, when the source of an `assign` activity has one of the following formats:

- a query, literal, or other expression
- a combination of variable name and property

In this case, the source resolves to a value of a particular type, which is usually a simple type. The number 7 or the Customer ID *ABC* is what it is. If the target in the first case accepts an integer, and if the target in the second case accepts a string, all is well.

A target must have the appropriate type in all cases, but you're likely to be aware of namespace issues only in some.

Use of an Expression As the Target

In some cases, you can use an expression to identify the field that receives data. The situation applies when the `to` element includes either of the following:

- a variable based on a message type, followed by a query that identifies a field in that variable
- an expression that identifies a field in a variable

Here's an example of each variation.

```

<assign name="DefaultQuoteAssignment">
    <copy>
        <from><literal>false</literal></from>
        <to variable="highlightQuote" part="placeQuoteResult">
            <query>/quote/quoteProvided</query>
        </to>
    </copy>
</assign>

```

```

    <from><literal>false</literal></from>
    <to>$myVariable/quote/quoteProvided</to>
  </copy>
</assign>

```

In each case, the statement writes "false" to the `quoteProvided` field.

Use of Partner Links

Let's focus for a moment on assigning endpoint references. When the source is a partner link, you specify `myRole` or `partnerRole` to identify the endpoint reference being copied. When the target is a partner link, however, you don't specify `myRole` or `partnerRole` because you can assign the endpoint reference only for the partner service.

One source of endpoint references might be configuration settings in an SOA runtime product. Another source might be a directory of services.

One partner service might provide the endpoint reference that is necessary to access another. In that case, the BPEL process

- issues an input-message activity (such as a `receive` activity) to accept the endpoint reference from the first service into a variable
- includes the variable in an `assign` activity, to copy the endpoint reference to a partner link
- includes the partner link in an `invoke` activity, to access the other service

In another scenario, you can access a service that has the same interface as an often-accessed service but resides at a different location. In that case, the BPEL process

- declares a partner link and sets one of the partner-link attributes (`initializePartnerRole`) to require that the SOA runtime product initialize the partner link with an endpoint reference.
- includes the partner link in an `invoke` activity.
- also includes the partner link in a fault handler that runs only if the first `invoke` activity fails. The fault handler copies a second endpoint reference into the partner link and retries the service invocation.

Listing 8.5 shows a literal endpoint reference. The `sref:service-ref` element is provided with BPEL and conforms to whatever data is required to describe a particular kind of endpoint reference. We show subordinate elements for an example.

Listing 8.5. Literal endpoint reference

```

<assign>
  <copy>
    <from>
      <literal>
        <sref:service-ref>
          <addr:EndpointReference>
            <addr:Address>
              http://com.highlight/policy/ProcessPolicy/
            </addr:Address>
            <addr:ServiceName>
              RegistrationService
            </addr:ServiceName>
          </addr:EndpointReference>
        </sref:service-ref>
      </literal>
    </from>
    <to partnerLink="PolicyPL" />
  </copy>
</assign>

```

Attributes of Each Copy Element

Each `copy` element of the `assign` activity includes two attributes:

- If `keepSrcElementName` is set to *yes*, not only is content copied, but the root XML element name in the source replaces the root element name in the target.
- If `ignoreMissingFromData` is set to *yes*, missing data from the source of the copy has no effect; otherwise, missing data causes the runtime *bpel:selectionFailure* fault.

Both attributes default to *no*.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

fromParts and toParts

Consider the following scenario:

1. You issue a `receive` activity that accepts data into a variable.
2. You issue an `invoke` activity that submits the same data to another service.

In general, you cannot use the same variable to receive and submit data. Even if the message structure is the same (two strings and an integer, for example), messages in most cases are defined in different WSDL namespaces, as shown in a later example. If a variable based on one message definition is expected but a variable based on another message definition is used, a validation error occurs at design or run time.

One way around the problem is to use an `assign` activity between the `receive` and `invoke` activities. A second way, however, lets you avoid defining a variable that is specific to a message:

1. Define a variable that holds only the business data that you want to transfer into and out of your BPEL process. That variable is based on an XSD element or type.
2. Reference the variable in each activity that transfers data.

The `fromParts` and `toParts` syntax makes the second alternative possible.

Imagine, for example, a BPEL process that accepts a greeting for subsequent distribution by mail. An operation in that service might use the message definition shown in [Listing 8.6](#).

Listing 8.6. ReceiveNotice message definition

```
<wsdl:definitions
  name="ReceiveNotice"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://com.highlight/policy/"
  xmlns:highlightXSD="http://com.highlight/policy/xsd/"
  .
  .
  <wsdl:types>
    <xsd:schema>
      <xsd:import namespace="http://com.highlight/policy/xsd/"
        schemaLocation="PolicyDefinitions.xsd">
      </xsd:import>
    </xsd:schema>
  </wsdl:types>
  .
  .
  <wsdl:message name="receiveMsg">
    <wsdl:part element="highlightXSD:transferElement"
      name="receivePart"/>
  </wsdl:message>
</wsdl:definitions>
```

Also imagine the BPEL process invoking a mailing service that uses the message definition shown in [Listing 8.7](#).

Listing 8.7. SendNotice message definition

```

wsdl:definitions
  name="SendNotice"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://com.highlight/prINTER/"
  xmlns:highlightXSD="http://com.highlight/policy/xsd/"
  .
  .
  <wsdl:types>
    <xsd:schema>
      <xsd:import namespace="http://com.highlight/policy/xsd/"
        schemaLocation="PolicyDefinitions.xsd">
      </xsd:import>
    </xsd:schema>
  </wsdl:types>
  .
  .
  <wsdl:message name="sendMsg" >
    <wsdl:part element="highlightXSD:transferElement"
      name="sendPart"/>
  </wsdl:message>
</wsdl:definitions>

```

Our assumption is that the message part in each definition uses the same XSD element (in this case, `transferElement`), as defined in the same namespace (in this case, `http://com.highlight/policy/xsd/`). Use of the same namespace prefix (`highlightXSD`) is a convention.

The purpose of the process outlined in Listing 8.8 is to transfer data from the inbound message (the greeting) to the outbound message (the mail content). You may be unfamiliar with some of the syntax, which demonstrates use of a `receive`, `assign`, and `invoke` activity. Be aware that in the `invoke` activity, `inputVariable` provides data for transmission to the service being invoked.

Listing 8.8. BPEL process that accepts and distributes a greeting

```

<process
  .
  .
  xmlns:tns1="http://com.highlight/policy/"
  xmlns:tns2="http://com.highlight/prINTER/">

  <!-- import of a third WSDL
    that holds partner link type is not shown -->
  <import namespace="http://com.highlight/policy/"
    location="../highlight/policy/ReceiveNotice.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/" />
  <import namespace="http://com.highlight/prINTER/"
    location="../highlight/prINTER/SendNotice.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/" />

  <variables>
    <variable name="receiveVariable"
      messageType="tns1:receiveMsg" />
    <variable name="sendVariable"
      messageType="tns2:sendMsg" />
  </variables>

  <sequence>
    <receive name="receiveNotice"
      partnerLink="receiveNoticePL"
      portType="tns1:noticePT"
      operation="receiveNoticeOp"
      variable="receiveVariable">
    </receive>

    <assign name="copyNotice">
      <copy>
        <from variable="receiveVariable" part="receivePart" />

```

```

        <to variable="sendVariable" part="sendPart" />
    </copy>
</assign>

    <invoke name="sendLetter"
        partnerLink="sendLetterPL"
        portType="tns2:sendLetterPT"
        operation="sendLetterOp"
        inputVariable="sendVariable" />
</sequence>
</process>

```

An alternative is to use a variable that is based on an XSD element and to use `fromParts` and `toParts` syntax. Listing 8.9 demonstrates this approach.

Listing 8.9. BPEL process that uses `toParts` and `fromParts` instead

```

<process
.
.
xmlns:tns1="http://com.highlight/policy/"
xmlns:tns2="http://com.highlight/printer/"
xmlns:tns3="http://com.highlight/policy/xsd/">

    <!-- XSD import is required. WSDL imports are not shown. -->
    <import namespace=" http://com.highlight/policy/xsd"
        location=" ../highlight/policy/xsd/PolicyDefinitions.xsd"
        importType="http://www.w3.org/2001/XMLSchema" />

    <variables>
        <variable name="transferVariable"
            element="tns3:transferElement" />
    </variables>

    <sequence>
        <receive name="receiveNotice"
            partnerLink="receiveNoticePL"
            portType="tns1:noticePT"
            operation="receiveNoticeOp"

            <fromParts>
                <fromPart part="receivePart"
                    toVariable="transferVariable" />
            </fromPart>
            </fromParts>
        </receive>
        <invoke name="sendLetter"
            partnerLink="sendLetterPL"
            portType="tns2:sendLetterPT"
            operation="sendLetterOp"

            <toParts>
                <toPart part="sendPart"
                    fromVariable="transferVariable" />
            </toPart>
            </toParts>
        </invoke>
    </sequence>
</process>

```

The `toParts` and `fromParts` syntax supports multi-part messages. As noted earlier, however, the best practice is to define single-part messages.

In summary, when identifying the target of an inbound message (for example, in a `receive` activity), you have a choice: either specify a variable to receive all the data or use the `fromParts` syntax. In the latter case, you specify a set of variables, each based on an XSD element or type, along with a message part for each. You can exclude some message parts to avoid accepting data that the process does not use. Each message part is described in the WSDL

`message` element for the operation.

Similarly, when identifying the target of an outbound message in an `invoke` or `reply` activity, you have a choice: either specify a variable that contains all the data or use the `toParts` syntax. In the latter case, you specify a set of variables, each based on an XSD element or type, along with a message part for each. You must ensure that each variable has content. The message part is described in the WSDL `message` element for the operation.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

invoke Activity

The `invoke` activity invokes a service synchronously or asynchronously.

```
<invoke partnerLink="agent" portType="policyPT"
        operation="requestQuote"
        inputVariable="quoteRequest"
        outputVariable="requestInfo">
</invoke>
```

In relation to this activity, the word "input" and the elements `inputVariable` and `toParts` refer to the data that the BPEL process sends to the partner service.

Among the attributes of the `invoke` activity:

- `partnerLink` identifies the partner link used to connect to the service.
- `portType` identifies the port type but is optional because the port type is implied by the partner link and by the role identifier `partnerRole` in that partner link. Many developers repeat the port type in the `invoke` activity because they are familiar with the port-type name, and the repetition adds clarity at design time. However, the repetition creates a small dependency: if the partner link type (in the WSDL) changes to refer to a port type that has a different name, the BPEL process is no longer valid.
- `operation` identifies the service operation to invoke.
- `inputVariable` identifies a variable that contains the business data being *sent* to the service. Specify this attribute only if you don't specify the `toParts` element.
- `outputVariable` identifies a variable that receives the business data being *returned* from the service. Specify this attribute for synchronous processing, but only if you don't specify the `fromParts` element.

Among the elements in the `invoke` activity:

- `correlations` references one or more correlation sets.
- `toParts` identifies a set of message parts and related variables for data sent to the partner service. Specify this element only if you don't specify the `inputVariable` attribute.
- `fromParts` identifies a set of message parts and related variables for data returned to the BPEL process. Specify this element for synchronous processing, but only if you don't specify the `outputVariable` attribute.

A special form of the `invoke` activity embeds fault handlers, a compensation handler, or both. This form provides a shortcut to those who are working directly with the XML. [Listing 8.10](#) shows an example.

Listing 8.10. Sample invoke activity with embedded compensation handler

```
<invoke name="AgentQuoteRequest"
        partnerLink="agent"
        portType="policyPT"
        operation="requestQuote"
        inputVariable="quoteRequest"
        outputVariable="requestInfo">
```

```

<compensationHandler>
  <invoke partnerLink="agent "
    portType="policyPT"
    operation="cancelQuoteRequest "
    inputVariable="requestInfo"
    outputVariable="confirmationInfo" />
</compensationHandler>
</invoke>

```

Listing 8.11 shows an example of the equivalent statements when the scope is declared explicitly.

Listing 8.11. Sample invoke activity with an explicitly declared scope

```

<scope name="AgentQuoteRequest">
  <compensationHandler>
    <invoke partnerLink="agent "
      portType="policyPT"
      operation="cancelQuoteRequest "
      inputVariable="requestInfo"
      outputVariable="confirmationInfo" />
  </compensationHandler>
  <invoke name="AgentQuoteRequest "
    partnerLink="agent "
    portType="policyPT"
    operation="requestQuote"
    inputVariable="quoteRequest "
    outputVariable="requestInfo">
  </invoke>
</scope>

```

If the special form of the `invoke` activity has a name, the BPEL engine accepts that name as the name of an implicit scope, and a `compensateScope` activity in the immediately higher-level scope can reference the name.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

receive Activity

The `receive` activity waits for a message that matches the detail in the related WSDL operation.

```
<receive name="getQuote"
  partnerLink="agent"
  operation="requestQuote"
  variable="quoteDetails"
  createInstance="yes" />
```

Among the attributes of the `receive` activity:

- `partnerLink` identifies the partner link used to connect to the service.
- `portType` identifies the port type but is optional because the port type is implied by the partner link and by the role identifier `myRole` in that partner link.
- `operation` identifies the service operation.
- `variable` identifies a variable that receives the business data from the service. Specify this attribute only if you don't specify the `fromParts` element.
- `createInstance` indicates whether the activity creates a BPEL instance. Valid values are *yes* and *no*. The default is *no*.
- `messageExchange` references a message exchange.

Among the elements in the `receive` activity:

- `correlations` references one or more correlation sets.
- `fromParts` identifies a set of message parts and related variables to receive business data. Specify the `fromParts` element only if you don't specify the `variable` attribute.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

reply Activity

The `reply` activity responds to a message received by one of the following kinds of *inbound message activities* (IMAs): a `receive` activity, an `onMessage` event (within the `pick` activity), or an `onEvent` handler.

```
<reply name="confirmQuote"
  partnerLink="agent"
  operation="requestQuote"
  variable="quoteResponse" />
```

The partner link, port type, and operation used in the `reply` activity are the same as the partner link, port type, and operation used in the IMA.

Consider the following port type.

```
<wsdl:portType name="agentPT">
  <wsdl:operation name="requestQuote">
    <wsdl:input message="requestMessage" />
    <wsdl:output message="responseMessage" />
  </wsdl:operation>
</wsdl:portType>
```

When the BPEL process is using a combination of IMA and `reply` activity:

- the message sent by the partner service is described by the input message
- the message returned by the BPEL process is described by the output message

The port type implies synchronous processing, which means that the requester suspends processing while waiting for a response. The requester may treat the invocation as synchronous or asynchronous, however, depending on the runtime software that handles the interaction between the two services.

Among the attributes of the `reply` activity:

- `partnerLink` identifies the partner link used to connect to the service.
- `portType` identifies the port type but is optional because the port type is implied by the partner link and by the role identifier `myRole` in that partner link.
- `operation` identifies the service operation.
- `variable` identifies a variable that contains the business data being sent to the service. Specify this variable only if you don't specify the `toParts` element.
- `faultName` identifies a WSDL fault name, as used when sending a fault message to the partner service.
- `messageExchange` references a message exchange.

Among the elements in the `reply` activity:

- `correlations` references one or more correlation sets.
- `toParts` identifies a set of message parts and related variables for data sent to the partner service. Specify this element only if you don't specify the `variable` attribute.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

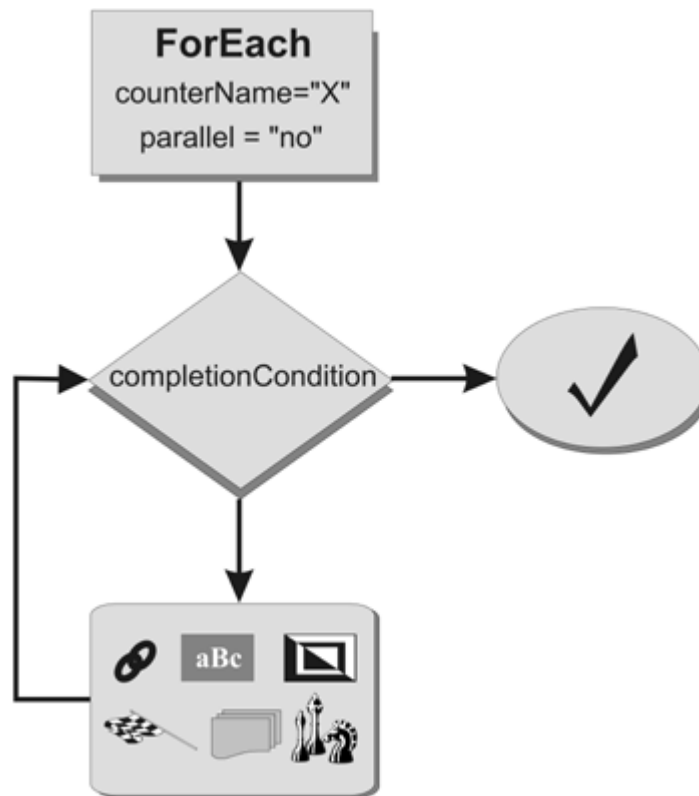
No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

forEach Activity

The `forEach` activity issues its embedded scope repeatedly, either in sequence or concurrently.

Figure 8.1 depicts the sequential `forEach` activity. As shown, the sequential `forEach` is similar (but not identical) to a "for loop" in other programming languages, where the logic is expressed by the phrase "for X equals start to finish."

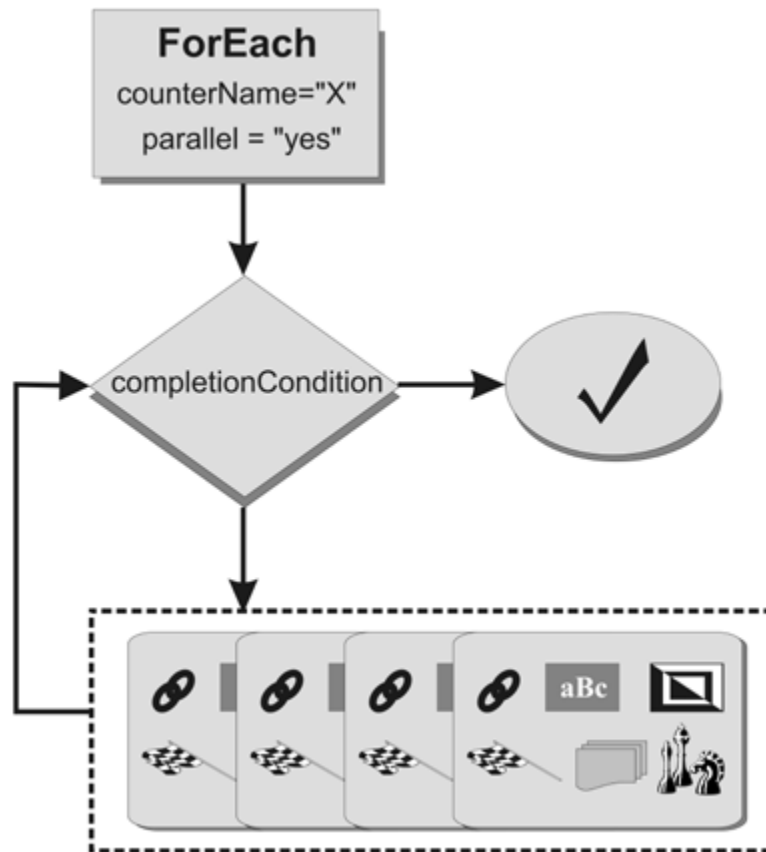
Figure 8.1. Sequential forEach



The maximum number of iterations is specified when the sequential `forEach` begins, and a counter indicates which iteration is running at a given time.

Figure 8.2 illustrates the concurrent (or *parallel*) `forEach` activity. As shown, the concurrent `forEach` also runs the embedded scope repeatedly, but each scope (really, each scope instance) starts at the same time.

Figure 8.2. Concurrent forEach



The starting number of scope instances is specified when the concurrent `forEach` begins, and a counter is available to distinguish one scope instance from the next.

Depending on specified options, the `forEach` activity ends normally

- when all scope instances complete;
- when a preset number of scope instances complete; or
- when a preset number of scope instances complete successfully.

Here's an example of a `forEach` activity.

```
<forEach counterName="myVar" parallel="yes"
  name="forLoop">
  <startCounterValue>1</startCounterValue>
  <finalCounterValue>5</finalCounterValue>
  <completionCondition>
    <branches successfulBranchesOnly="yes">
      3
    </branches>
  </completionCondition>
  <scope name="forEachScope">
    .
  </scope>
</forEach>
```

Among the attributes of the `forEach` activity:

- `counterName` is an unsigned integer variable that is implicitly declared in the `forEach` scope, which is the scope in the activity. The variable value is available at run time. Any changes have no effect on the number of scope

instances because the number is preset when the `forEach` activity starts. Moreover, changes are lost after any particular scope instance ends.

- `parallel` accepts the value *yes* (for the parallel version of the activity) or *no* (for the sequential).

Among the elements in the `forEach` activity:

- `startCounterValue` has the starting counter value, which does not change after the `forEach` activity begins. If the value is greater than that of `finalCounterValue`, the `forEach` activity is equivalent to an `empty` activity.
- `finalCounterValue` has the maximum counter value, which does not change after the `forEach` activity begins.
- `branches` (within the `completionCondition` element) indicates the number of scope instances whose completion ends the `forEach` activity. If the number is meant to refer to the number of successfully completed scope instances, set the `successfulBranchesOnly` attribute to *yes*.

The specified number must be less than or equal to the maximum number of scope instances that can run in the `forEach` activity. (The maximum is one more than the difference between `startCounterValue` and `finalCounterValue`.)

When running a parallel `forEach` activity that has a value in the `branches` element, the BPEL engine may end the activity only after more completions occur than are specified in that element.

Any of those elements can accept an expression that resolves to a number.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Other Loops

BPEL provides two other loops: `while` and `repeatUntil`.

while Activity

The `while` activity defines a loop that iterates while a Boolean expression at the top of the loop evaluates to *true*. If the expression initially evaluates to *false*, the loop does not run.

```
<while>
  <condition>$numberOfDrivers < 5</condition>
  <sequence>
    .
    .
  </sequence>
</while>
```

repeatUntil Activity

The `repeatUntil` activity defines a loop that always runs at least once. The loop iterates until the Boolean expression at the bottom of the loop evaluates to *true*.

```
<repeatUntil>
  <sequence>
    .
    .
  </sequence>
  <condition>$numberOfDrivers = 5</condition>
</repeatUntil>
```


User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

pick Activity

The `pick` activity waits for one of potentially several events to occur, often including a timeout and always including receipt of an inbound message. The activity completes when any one of its immediately embedded activities completes:

- The `onMessage` event processes an inbound message and is similar to a `receive` activity.
- The `onAlarm` event waits for a passage of time or for a specific date and time.

Both those events can embed other activities.

A variation of the `pick` activity creates an instance of a BPEL process, but in that case, no timeout is possible and any inbound message can create an instance.

In most cases, the `pick` activity is not concurrent with other activities.

Listing 8.12 shows an example. A further explanation of each embedded activity follows.

Listing 8.12. pick activity

```
<pick name="CustomerResponseToQuote">
  <onMessage operation="requestMore"
    partnerLink="ProcessQuotePL"
    portType="ProcessQuotePT"
    variable="furtherConsiderationReq">
    <correlations>
      <correlation initiate="no" set="applicantEMailAddr" />
    </correlations>
    <sequence name="customerWantsUnderwriterToContactPostReview">
      <assign name="copyQuoteForReview">
        <copy>
          .
          .
        </copy>
      </assign>
      <invoke name="underwriterFollowupAndCall"
        partnerLink="UnderwriterPL"
        portType="UnderwriterPT"
        operation="retrieveAndFollowUp"
        inputVariable="underwriterReview"/>
    </sequence>
  </onMessage>
  <onAlarm>
    <for>'PT5M'</for>
    <invoke .../>
  </onAlarm>
</pick>
```

onMessage Event

Among the attributes of the `onMessage` event:

- `createInstance` indicates whether the activity creates a BPEL instance. Valid values are *yes* and *no*. The default is *no*.

- `partnerLink` identifies the partner link used to connect to the service.
- `portType` identifies the port type but is optional because the port type is implied by the partner link and by the role identifier `myRole` in that partner link.
- `operation` identifies the service operation.
- `variable` identifies a variable that receives the business data from the service. Specify the `variable` attribute only if you don't specify the `fromParts` element.
- `messageExchange` references a message exchange.

Among the elements in the `onMessage` event:

- `correlations` references one or more correlation sets.
- `fromParts` identifies a set of message parts and related variables to receive business data. Specify the `fromParts` element only if you don't specify the `variable` attribute.

onAlarm Event

The `onAlarm` event requires the presence of at least one of the following elements:

- `for` identifies the duration that passes before the alarm is fired. The start time for that duration is when the `pick` activity begins. Specify this element only if you don't specify the `until` element.
- `until` identifies the calendar date and clock time when the alarm is triggered. Specify this element only if you don't specify the `for` element.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

wait Activity

Like the `onAlarm` event of the `pick` activity, the `wait` activity causes a delay for a specified period or until a specified date and time.

```
<sequence>
  <wait>
    <until>'2010-12-25T18:00+01:00'</until>
  </wait>
  <invoke ... />
</sequence>
```

```
<sequence>
  <documentation>
    wait for 1 Year 2 Months and 1 Hour
  </documentation>
  <wait>
    <for>'P1Y2MT1H'</for>
  </wait>
  <invoke ... />
</sequence>
```

Elements nested in the `wait` activity include the following:

- `for` identifies the duration that passes before the activity ends. The start time is when the activity starts. Specify this element only if you don't specify the `until` element.
- `until` identifies the calendar date and clock time when the activity ends. Specify this element only if you don't specify the `for` element.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Event Handlers

An event handler is a set of activities that run concurrently with either the primary activity of the same scope or, outside a scope, with the primary activity of the process. The handler also may run concurrently with other event handlers and with other instances of the same handler. Unlike the other handlers (compensation, fault, and termination), the event handler runs during normal processing.

Two kinds of event handlers are available: `onEvent` and `onAlarm`.

An `onEvent` handler processes an inbound message and is similar to a `receive` activity. You can use this kind of handler for an interaction that occurs only in some process instances, as when a customer cancels a purchase before a product is shipped.

An `onAlarm` handler waits for a passage of time or for a specific date and time. After firing an initial alarm, the handler can fire an alarm repeatedly at a specified interval. You can use an `onAlarm` handler to send a renewal notice.

An event handler is enabled when the BPEL process receives its initial message or when the scope that encloses the event handler begins. An event handler is disabled when the process or scope ends. If an event handler is running, though, the BPEL engine allows the activities to complete.

The `onEvent` handler accepts messages for as long as the handler is enabled, whereas the `onAlarm` handler does not fire again unless you requested a repetition.

onEvent Handler

The BPEL process implicitly declares a variable that receives data and is local to the scope of the handler. You can receive data only into that implicitly declared variable.

The implicit declaration occurs whether you've specified the `variable` attribute, along with a message type or XSD element, or the `fromParts` element. (We've ignored the possibility of several variable declarations, one for each part in the inbound message.) All local variables in the scope of the handler are local to each running instance of the handler.

Among the attributes of the `onEvent` handler:

- `partnerLink` identifies the partner link used to connect to the service.
- `portType` identifies the port type but is optional because the port type is implied by the partner link and by the role identifier `myRole` in that partner link.
- `operation` identifies the service operation.
- `variable` identifies a variable that receives the business data from the service. Specify the `variable` attribute only if you don't specify the `fromParts` element.
- `messageExchange` references a message exchange in a scope that encloses the `onEvent` handler.

Among the elements in the `onEvent` handler:

- `correlations` references one or more correlation sets.
- `fromParts` identifies a set of message parts and related variables to receive business data. Specify the `fromParts` element only if you don't specify the `variable` attribute.

onAlarm Handler

The `onAlarm` handler requires the presence of at least one of the following elements:

- `for` identifies the duration that passes before the alarm is fired. The start time for that duration is when the BPEL process receives its initial message or when the scope that encloses the event handler begins. You can specify this element only if you don't specify the `until` element.
- `until` identifies the calendar date and clock time when the alarm is triggered. You can specify this element only if you don't specify the `for` element.
- `repeatEvery` causes the alarm to be triggered repeatedly and specifies the interval between triggers. The interval begins when the initial trigger fires or, if neither of the other elements is specified, when the BPEL process receives its initial message or when the scope that encloses the event handler begins.

Listing 8.13 shows several `onAlarm` handler examples.

Listing 8.13. `onAlarm` handlers

```
<process>
  <eventHandlers>

    <!-- the event fires after 5 minutes -->
    <onAlarm>
      <for>'PT5M'</for>
      <invoke .../>
    </onAlarm>

    <!-- the event fires at 9am on Dec 24 2010 -->
    <onAlarm>
      <until>'2010-12-24T09:00+01:00'</until>
      <invoke .../>
    </onAlarm>

    <!-- the event fires every 3 hours -->
    <onAlarm>
      <repeatEvery>'PT3H'</repeatEvery>
      <invoke .../>
    </onAlarm>

    <!-- the event fires after 5 minutes
         and every day thereafter -->
    <onAlarm>
      <for>'PT5M'</for>
      <repeatEvery>'P1D'</repeatEvery>
      <invoke .../>
    </onAlarm>
  </eventHandlers>
</process>
```

You can assign some or all of the element values from data that was received in an inbound message. Also, if you wish to fire the alarm repeatedly but only a specified number of times, you can create a sequential `forEach` activity and embed an `onAlarm` handler that doesn't have a `repeatEvery` element.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Message Exchanges, Revisited

As noted earlier, the BPEL engine can pair a [reply](#) activity with an inbound message activity, but only if the paired activities reference the same partner link, operation, and message exchange. The pairing allows the BPEL engine to direct the reply appropriately.

In most cases, the process creates a default message exchange for each activity, and you need to specify only the partner link and operation. An explicit message exchange is necessary at times, as in the following example.

In response to most car-insurance policy requests, the BPEL process CheckCredit requires financial data from the applicant and co-applicant. The process includes a [receive](#) activity for each applicant in sequence.

```
<sequence>
  <receive partnerLink="CreditCheck"
    operation="checkFamilyCredit"
    variable="applicantInfo"
    createInstance="yes"
    messageExchange="applicant" />
  <receive partnerLink="CreditCheck"
    operation="checkFamilyCredit"
    variable="spouseInfo"
    createInstance="no"
    messageExchange="spouse" />
</sequence>
```

As shown, the two activities are based on the same partner link and operation and are distinguished only by the message-exchange value. The message-exchange values were defined earlier, as follows.

```
<messageExchanges>
  <messageExchange name="applicant" />
  <messageExchange name="spouse" />
</messageExchanges>
```

Assume that in the subsequent [flow](#) activity, shown in [Listing 8.14](#), a calculation occurs at the same time for each applicant (within the [sequence](#) activity that we show for simplicity). We cannot know which calculation will end first, but in any case, a reply is necessary to the appropriate applicant, and the message-exchange value is required so that the BPEL engine can match a [receive](#) activity and the subsequent reply.

Listing 8.14. Use of message exchanges

```
<flow>
  <links>
    <link name="ApplicantLink" />
    <link name="SpouseLink" />
  </links>
```

```

<!-- assume that activities in each sequence activity
      perform a calculation. The reply statements that follow
      the calculation run concurrently, but must respond to a
      specific request. ->

<sequence name="Calculate credit report for spouse">
  .
  .
  <sources>
    <source linkName="SpouseLink" />
  </sources>
</sequence>

<reply partnerLink="CreditCheck"
      operation="checkFamilyCredit"
      variable="resultSpouse"
      messageExchange="spouse">

  <targets>
    <target linkName="SpouseLink" />
  </targets>
</reply>

<sequence name="Calculate credit report for applicant">
  .
  .
  <sources>
    <source linkName="ApplicantLink" />
  </sources>
</sequence>

<reply partnerLink="CreditCheck"
      operation="checkFamilyCredit"
      variable="resultApplicant"
      messageExchange="applicant">
  <targets>
    <target linkName="ApplicantLink" />
  </targets>
</reply>
</flow>

```

In general, you must do as follows if multiple IMA-and-reply pairs reference the same partner link and operation and if those activity pairs may run at the same time:

- Declare a message exchange for each IMA-and-reply pair.
- Reference the appropriate message exchange when you define each activity, to make the pairings explicit.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

doXSLTransform

Extensible Stylesheet Transformations (XSLT) 1.0 is a language for copying data from an XML source into a second, differently structured text format (called a *result document*), which in most cases is also in XML. Here's the situation in BPEL:

- The primary purpose of XSLT is to reorganize the data received from one service so that the process can transmit the data to another.
- The XML source must contain a single XPath element node, which may contain descendants.

To reorganize data from an XML source, do as follows:

1. Write an extensible stylesheet (XSL), which is an XML file that includes
 - XPath 1.0 expressions that select the data of interest
 - optionally, parameters that accept values from outside the XSL and that help specify what data to review from the XML source, as well as what data to place in the result document
 - XSLT statements that specify the result document, which may include data from the XML source as well as from the XSL
2. In the BPEL process, invoke the function `doXSLTransform` and specify
 - a path to the XSL
 - a variable that holds the XML source, or an XPath expression that resolves to a single element node
 - optionally, pairs of parameter names and values for use by the XSL

The function `doXSLTransform` returns a result document. If the function is a source in a `copy` element of the `assign` activity, the result document is placed in the appropriate target field.

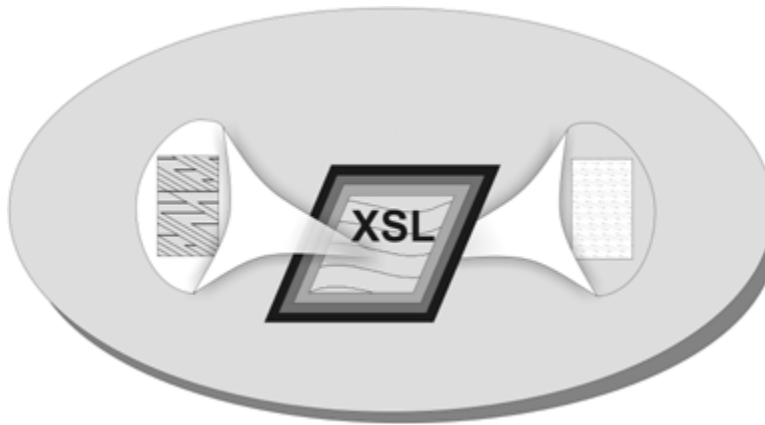
Our next examples reflect two ways to use `doXSLTransform`:

- *single transformation*, which is the conversion of an XML source to a result document by one invocation of the function
- *iterative construction*, which is the building of an increasingly large result document by repeated invocations of the function

Single Transformation

Figure 8.3 illustrates the use of `doXSLTransform` for a single transformation.

Figure 8.3. doXSLTransform, single transformation



As shown in the figure, a BPEL process can receive data from a partner service, use that data to invoke `doXSLTransform` in an `assign` activity, and use the result document when invoking a second partner service.

Listing 8.15 shows the XML source for a single transformation.

Listing 8.15. XML source for a single transformation

```
<Insured CustomerID="5">
  <CarPolicy PolicyType="Auto">
    <Vehicle Category="Sedan">
      <Make>Honda</Make>
      <Model>Accord</Model>
    </Vehicle>
    <Vehicle Category="Sport" Domestic="True">
      <Make>Ford</Make>
      <Model>Mustang</Model>
    </Vehicle>
  </CarPolicy>
  <CarPolicy PolicyType="Antique">
    <Vehicle Category="Sport">
      <Make>Triumph</Make>
      <Model>Spitfire</Model>
    </Vehicle>
    <Vehicle Category="Coupe" Domestic="True">
      <Make>Buick</Make>
      <Model>Skylark</Model>
    </Vehicle>
    <Vehicle Category="Sport">
      <Make>Porsche</Make>
      <Model>Speedster</Model>
    </Vehicle>
  </CarPolicy>
</Insured>
```

Listing 8.16 shows the result document.

Listing 8.16. Result document for a single transformation

```
<VehicleList>
  <OneVehicle>
    <Make>Honda</Make>
    <Model>Accord</Model>
  </OneVehicle>
  <OneVehicle>
    <Make>Ford</Make>
    <Model>Mustang</Model>
  </OneVehicle>
  <OneVehicle>
```

```

<Make>Triumph</Make>
<Model>Spitfire</Model>
</OneVehicle>
<OneVehicle>
<Make>Buick</Make>
<Model>Skylark</Model>
</OneVehicle>
<OneVehicle>
<Make>Porsche</Make>
<Model>Speedster</Model>
</OneVehicle>
</VehicleList>

```

Listing 8.17 shows an outline of the BPEL process.

Listing 8.17. BPEL process outline for single transformation

```

<process>
.
.
  <variables>
    <variable name="CarPolicies" element="InsuredElement" />
    <variable name="Vehicles" element="VehiclesElement" />
  </variables>
  <sequence>
    <invoke ... outputVariable="CarPolicies" />
    <assign>
      <copy>
        <from>
          bpel:doXslTransform
            ("urn:stylesheets:Insured2Vehicles.xsl",
             $CarPolicies)
        </from>
        <to variable="Vehicles" />
      </copy>
    </assign>
    <invoke ... inputVariable="Vehicles" />
  </sequence>
.
.
</process>

```

Last (and though a review of XSLT is out of scope), Listing 8.18 shows the XSL, which selects the `CarPolicy` nodes and, for each, writes a `Make` and `Model` node to the result document.

Listing 8.18. XSL to process the `CarPolicy` nodes

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml"
    version="1.0"
    encoding="ISO-8859-1"
    omit-xml-declaration="yes"
    indent="yes" />

  <xsl:template match="/">
    <VehicleList>
      <xsl:apply-templates select="Insured/CarPolicy"/>
    </VehicleList>
  </xsl:template>

  <xsl:template match="CarPolicy">

```

```

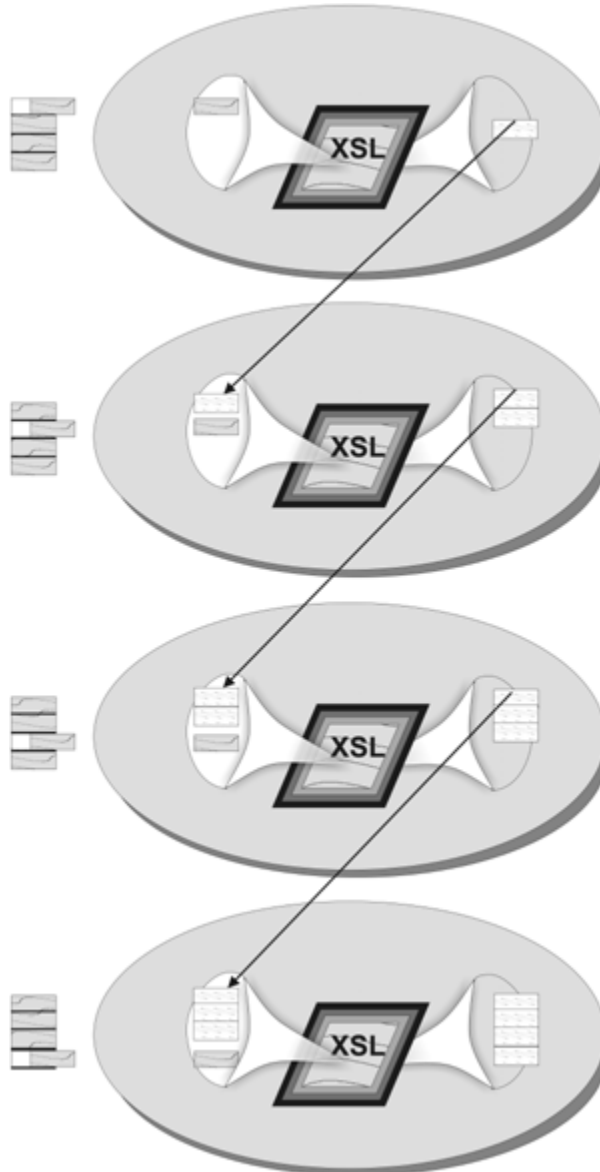
<xsl:for-each select="Vehicle">
  <OneVehicle>
    <xsl:copy-of select="Make">
    <xsl:copy-of select="Model">
  </OneVehicle>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>

```

Iterative Construction

Figure 8.4 illustrates the use of `doXSLTransform` for iterative construction.

Figure 8.4. `doXSLTransform`, iterative construction



As shown in the figure, a BPEL process loops through the following sequence:

1. Receive data from a partner service.

2. Invoke the function `doXSLTransform` in an `assign` activity to add new data to the data received in previous iterations.

In this case:

- The source document (in variable `AllVehicles`) is different at each iteration, and the invocation of `doXSLTransform` is as follows.

```
bpel:doXslTransform
  ("urn:stylesheets:AddNewVehicle.xsl", $AllVehicles,
   "NewVehicle", $OneVehicle)
```

- As shown, `doXSLTransform` submits a parameter (called `NewVehicle`) to the XSL. The content of that parameter is the data most recently provided by the partner service.

On completing the loop, the BPEL process invokes a second partner service with the content that was collected during the loop.

Outline of the BPEL Process

Listing 8.19 shows an outline of the BPEL process.

Listing 8.19. BPEL process outline for iterative construction

```
<process>
.
.
<variables>
  <variable name="OneEntry"      element="OneVehicleElement" />
  <variable name="AllVehicles"  element="VehicleListElement"
    <from>
      <literal>
        <VehicleList>
          <OneVehicle/>
        </VehicleList>
      </literal>
    </from>
  </variable>
</variables>

...<while>
  <condition> ... </condition>
  <sequence>
    <invoke ... outputVariable="OneEntry" />
    <assign>
      <copy>
        <from>
          bpel:doXslTransform
            ("urn:stylesheets:AddNewVehicle.xsl",
             $AllVehicles, "NewVehicle", $OneEntry)
          </from>
        <to variable="AllVehicles" />
      </copy>
    </assign>
  </sequence>
</while>
<invoke ... inputVariable="AllVehicles" />
.
.
</process>
```

Effect of the BPEL Process

The declaration of `AllVehicles` initializes the variable as follows.

```
<VehicleList>
<OneVehicle/>
</VehicleList>
```

We'll assume that each iteration provides details on only one vehicle and that successive iterations result in the following content.

```
<VehicleList>
<OneVehicle/>
<OneVehicle>
<Make>Honda</Make>
<Model>Accord</Model>
</OneVehicle>
<OneVehicle>
<Make>Ford</Make>
<Model>Mustang</Model>
</OneVehicle>
<OneVehicle>
<Make>Triumph</Make>
<Model>Spitfire</Model>
</OneVehicle>
</VehicleList>
```

Listing 8.20 shows the XSL, which selects the `OneVehicle` nodes, copies them to the (growing) result document, and adds the `OneVehicle` node that was provided in the current iteration of the while loop.

Listing 8.20. XSL to select the `OneVehicle` nodes

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:param name="NewVehicle"/>

  <xsl:output method="xml"
    version="1.0"
    encoding="ISO-8859-1"
    omit-xml-declaration="yes"
    indent="yes" />

  <xsl:template match="/">
    <VehicleList>
      <xsl:apply-templates select="descendant::OneVehicle"/>
    </VehicleList>
  </xsl:template>

  <xsl:template match="OneVehicle">
    <xsl:copy-of select="." />
    <xsl:if test="position()=last()">
      <xsl:copy-of select="$NewVehicle">
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```