User name:
Book: Business Process Execution Language for Web Services

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 4. Advanced BPEL

In the previous chapter, we covered the basics of BPEL and provided an introduction to the structure of business processes. We are now familiar with defining business processes, invoking web service operations sequentially and in parallel, defining partner links, defining variables, and assigning values. However, using BPEL for complex real-world business processes requires additional functionality. Sometimes the activities of a business process need to be performed in loops. Often activities might have links that would affect the execution order. This is usually the case with concurrent flows. Sometimes we will have to wait either for a message event or an alarm event to occur.

One very important aspect of business process modeling is fault handling. Particularly in business processes that span multiple enterprises and use web services over the Internet, we can assume that faults will occur quite often due to various reasons, including broken connections, unreachable web services, unavailability of services, and so on. If business processes do not finish successfully, we might need a way to undo the partial work. This is called compensation and is one of the features of BPEL.

In this chapter, we will look at these and other advanced BPEL features including:

- BPEL activities not covered in the previous chapter, such as loops, delays, and process termination
- Fault handling
- Scopes and serialization
- Compensation
- Events and event handlers
- Concurrent activities and links
- The business process lifecycle
- Correlations and message properties
- Dynamic partner links
- Abstract business processes
- A model-driven approach for generating BPEL processes from UML activity diagrams

Advanced Activities

In the previous chapter, we familiarized ourselves with important BPEL activities, including invoking web service operations (`<invoke>`), receiving messages from partners (`<receive>`), returning results to process clients (`<reply>`), declaring variables (`<variable>`), updating variable contents (`<assign>`), sequential and concurrent structured activities (`<sequence>` and `<flow>`), and conditional behavior (`<switch>`).

However, these activities are not sufficient for complex real-world business processes. Therefore, in the first part of this chapter we will become familiar with the other important activities offered by BPEL, particularly activity names, loops, delays, empty activities, and process termination. We will not discuss concrete use cases where these activities can be used, because they are well known to developers. We will, however, use these activities later in the chapter, where we will present some examples. Let us first look at activity names.

Activity Names

For each BPEL activity, we can specify a name by using the `name` attribute. This attribute is optional and can be used with all basic and structured activities. For instance, the Employee Travel Status web service invocation activity from the example in Chapter 3 could be named `EmployeeTravelStatusSyncInv`; this is shown in the code excerpt below. We will see that naming activities is useful on several occasions; for example, when invoking inline compensation handlers or when synchronizing activities:

```
...
<invoke name="EmployeeTravelStatusSyncInv"
partnerLink="employeeTravelStatus"
portType="emp:EmployeeTravelStatusPT"
operation="EmployeeTravelStatus"
inputVariable="EmployeeTravelStatusRequest"
outputVariable="EmployeeTravelStatusResponse" />
        ...
```

Activity names also improve the readability of BPEL processes.

Loops

When defining business processes we will sometimes want to perform a certain activity or a set of activities in a loop; for example, perform a calculation or invoke a partner web service operation several times, and so on.

BPEL supports loops through the `<while>` activity. It repeats the enclosed activities until the Boolean condition no longer holds true. The Boolean condition is expressed through the `condition` attribute, using the selected expression language (the default is XPath 1.0). The syntax of the `<while>` activity is shown in the following code excerpt:

```
<while condition="boolean-expression" >
<!-- Perform an activity or a set of activities enclosed by <sequence>,
<flow>, or other structured activity -->
  </while>
```

Let us consider a scenario where we need to check flight availability for more than one person. Let us also assume that we need to invoke a web service operation for each person, similar to the example in Chapter 3. In addition to the variables already present in the example, we would need two more: `NoOfPassengers` to hold the number of passengers, and `Counter` to use in the loop. The code excerpt with variable declarations is shown below:

```
<variables>
...
<variable name="NoOfPassengers"
type="xs:int"/>
<variable name="Counter"
type="xs:int"/>
...
   </variables>
```

We also need to assign values to the variables. The `NoOfPassengers` can be obtained from the Employee Travel web service. In the following code, we initialize both variables with static values:

```
<assign>
<copy>
<from expression="number(5)"/>
<to variable="NoOfPassengers"/>
</copy>
```

```
<copy>
<from expression="number(0)"/>
<to variable="Counter"/>
</copy>
      </assign>
```

The loop to perform the web service invocation is shown in the code excerpt below. Please remember that this excerpt is not complete:

```
<while condition=
"bpws:getVariableData('Counter') &lt;
bpws:getVariableData('NoOfPassengers')">
<sequence>
<!-- Construct the FlightDetails variable with passenger data -->
...
<!-- Invoke the web service -->
<invoke partnerLink="AmericanAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" />
<receive partnerLink="AmericanAirlines"
portType="trv:FlightCallbackPT"
operation="FlightTicketCallback"
variable="FlightResponseAA" />
...
<!-- Process the results ... -->
...
<!-- Increment the counter -->
<assign>
<copy>
<from expression="bpws:getVariableData('Counter') + 1"/>
<to variable="Counter"/>
</copy>
</assign>
</sequence>
    </while>
```

Loops are helpful when dealing with arrays. In BPEL, arrays can be simulated using XML complex types where one or more elements can occur more than once (using the `maxOccurs` attribute in the XML Schema definition). To iterate through multiple occurrences of the same element, we can use XPath expressions.

Delays

Sometimes a business process may need to specify a certain delay. In BPEL, we can specify delays either for a specified period of time or until a certain deadline is reached, by using the <wait> activity. Typically, we could specify delays to invoke an operation at a specific time; or wait for some time and then invoke an operation; for example, we could choose to wait before we pool the results of a previously initiated operation or wait between iterations of a loop.

The <wait> activity supports two attributes:

- for: We can specify duration using this attribute; we specify a period of time.

```
<wait for="duration-expression"/>
```

- until: We can use this attribute to specify a deadline; we specify a certain date and time.

```
<wait until="deadline-expression"/>
```

Deadline and Duration Expressions

To specify deadline and duration expressions, BPEL uses lexical representations of corresponding XML Schema data types. For deadlines, these data types are either dateTime or date. For duration, we use the duration data type. The lexical representation of expressions should conform to the XPath 1.0 (or the selected query language) expressions. The evaluation of such expressions should result in values that are of corresponding XML Schema types: dateTime and date for deadline and duration for duration expressions.

All three data types use lexical representation inspired by the ISO 8601 standard, which can be obtained from the ISO web page http://www.iso.ch. ISO 8601 lexical format uses characters within the date and time information. Characters are appended to the numbers and have the following meaning:

- C represents centuries
- Y represents years
- M represents months
- D represents days
- h represents hours
- m represents minutes
- s represents seconds. Seconds can be represented in the format ss.sss to increase precision.
- Z is used to designate Coordinated Universal Time (UTC). It should immediately follow the time of day element.

For the dateTime expressions there is another designator:

- T is used as time designator to indicate the start of the representation of the time.

Examples of deadline expressions are shown in the code excerpts below:

```
<wait until="'2004-03-18T21:00:00+01:00'"/>
<wait until="'18:05:30Z'"/>
```

For duration expressions the following characters can also be used:

- P is used as the time duration designator. Duration expressions always start with P.
- Y follows the number of years.
- M follows the number of months or minutes.
- D follows the number of days.
- H follows the number of hours.
- S follows the number of seconds.

To specify a duration of 4 hours and 10 minutes, we use the following expression:

```
<wait for="'PT4H10M'"/>
```

To specify the duration of 1 month, 3 days, 4 hours, and 10 minutes, we need to use the following expression:

```
<wait for="'P1M3DT4H10M'"/>
```

The following expression specifies the duration of 1 year, 11 months, 14 days, 4 hours, 10 minutes, and 30 seconds:

```
<wait for="'P1Y11M14DT4H10M30S'"/>
```

Empty Activities

When developing BPEL processes, you may come across instances where you need to specify an activity as per rules, but you do not really want to perform the activity. For example, in `<switch>` activities, we need to specify an activity for each `case`. However, if we do not want to perform any activity for a particular case, we can specify an `<empty>` activity. Not specifying any activity in this case would result in an error, because the BPEL process would not correspond to the BPEL schema. Empty activities are also useful in fault handling, when we need to suppress a fault.

The syntax for the `<empty>` element is rather straightforward:

```
<empty/>
```

Process Termination

BPEL provides the `<terminate>` activity to terminate a business process before it has finished. We can use it to immediately terminate processes that are in execution. Often we use `<terminate>` in switches, where we need to terminate a process when certain conditions are not met.

The `<terminate>` activity terminates the current business process instance and no fault and compensation handling is performed. Process instances, faults, and compensations are discussed later in this chapter.

The syntax is very simple and is shown below:

```
<terminate/>
```

Now that we have become familiar with loops, delays, empty activities, and process termination (which we will use in examples in the rest of this chapter) we will go on to fault handling.

User name:
Book: Business Process Execution Language for Web Services

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.

Fault Handling and Signaling

Business processes specified using BPEL will interact with their partners through operation invocations of web services. Web services are based on loosely coupled Service Oriented Architecture (SOA). The communication between web services is done over Internet connections that may or may not be highly reliable. Web services could also raise faults due to logical errors and execution errors arising from defects in the infrastructure. Therefore BPEL business processes will need to handle faults appropriately. BPEL processes may also need to signal faults themselves. Fault handling and signaling is an important aspect of business processes designed using BPEL.

Faults in BPEL can arise in various situations:

- When a BPEL process invokes a synchronous web service operation, the operation might return a WSDL fault message, which results in a BPEL fault.
- A BPEL process can explicitly signal (throw) a fault.
- A fault can be thrown automatically, for example, when a join failure has occurred. We will discuss join failures later in this chapter.
- The BPEL server might encounter error conditions in the run-time environment, network communications, or any other such reason. BPEL defines several standard faults; these are listed in Appendix A.

WSDL Faults

WSDL faults occur due to synchronous operation invocations on partner web services. In WSDL, such faults are denoted with the `<fault>` element within the `<operation>` declaration. In BPEL, WSDL faults are identified by the qualified name of the fault and the target namespace of the corresponding port type used in the operation declaration.

In the Synchronous Business Travel Process example in the previous chapter, we have used the `TravelApproval` operation on the `TravelApprovalPT` port type with input and output messages. This is shown in the WSDL excerpt below:

```
...
<portType name="TravelApprovalPT">
<operation name="TravelApproval">
<input message="tns:TravelRequestMessage" />
<output message="aln:TravelResponseMessage" />
</operation>
</portType>
...
```

To add fault information to the operation, we first need to define a corresponding message. For simplicity, this message will be of the `xs:string` type:

```
...
<message name="TravelFaultMessage">
<part name="error" type="xs:string" />
```

```
</message>
...
```

Now we will add the fault declaration to the operation signature shown above:

```
...
<portType name="TravelApprovalPT">
<operation name="TravelApproval">
<input message="tns:TravelRequestMessage" />
<output message="aln:TravelResponseMessage" />
    <fault name="fault" message="tns:TravelFaultMessage" />

</operation>
</portType>
...
```

WSDL does not require that we use unique fault names within the namespace used to define the operation. This implies that faults that have the same name and are defined within the same namespace will be considered as the same fault in BPEL. Keep this in mind when designing web services that can potentially become partners of BPEL business processes because this can lead to conflicts in fault handling during execution. This is a shortcoming of the current WSDL version 1.1 fault model, and should be removed in future versions.

Signaling Faults

A business process may sometimes need to explicitly signal a fault. For such a situation, BPEL provides the `<throw>` activity. It has the following syntax:

```
<throw faultName="name" />
```

BPEL does not require that we define fault names in advance, prior to their use in the `<throw>` activity. This flexible approach can also be error-prone because there is no compile-time checking of fault names. Therefore, a typo could result in a situation where a misspelled fault might not be handled by the designated fault handler.

Faults can also have an associated variable that usually contains data related to the fault. If such a variable is associated with the fault, we need to specify it when throwing the fault. This is done by using the optional `faultVariable` attribute as shown here:

```
<throw faultName="name" faultVariable="variable-name" />
```

The following example shows the most straightforward use of the `<throw>` activity, where a `WrongEmployeeName` fault is thrown—no variable is needed. Remember that fault names are not declared in advance:

```
<throw faultName="WrongEmployeeName" />
```

The faults raised with the `<throw>` activity have to be handled in the BPEL process. Fault handling is covered later in this chapter. Faults that are not handled will *not* be automatically propagated to the client as is the case in modern programming languages (Java for example). Rather, the BPEL process will terminate abnormally. Sometimes, however, we may want to signal faults to clients.

Signaling Faults to Clients in Synchronous Replies

A BPEL process offers operations to its clients through the `<receive>` activity. If the process wants to provide a synchronous request/response operation, it sends a `<reply>` activity in response to the initial `<receive>`. Remember that the type of the operation is defined in the WSDL document of the BPEL process. A synchronous request/response

operation is defined as an operation that has an input and an output message and an optional fault message.

If such an operation has the fault part specified, we can use the <reply> activity to return a fault instead of the output message. The syntax of the <reply> activity in this case is:

<reply partnerLink="*partner-link-name*"

portType="*port-type-name*"

operation="*operation-name*"

variable="*variable-name*" <!-- optional -->

faultName="*fault-name***" >** </reply>

When we specify a fault name to be returned through the <reply> activity, the variable name is optional. If we specify a variable name, then the variable has to be of the fault message type as defined in WSDL.

Example

Let's modify the BPEL process definition in the synchronous travel example and signal the fault (TravelFaultMessage) to the client by using the <reply> activity.

First, we need to declare an additional variable that will hold the fault description to return to the client. The variable is of the TravelFaultMessage type:

...

<variables>

...

<!-- fault to the BPEL client -->

```
<variable name="TravelFault" messageType="trv:TravelFaultMessage"/>
```

</variables>

```
...
```

Then we return the fault to the BPEL process client. We will need to check if something went wrong in the travel process. For the purpose of this example, we will check whether the selected flight ticket has been approved. This information is stored in the confirmationData part of the TravelResponse variable in the Approved element (see previous chapter for the complete schema definition). Note that this is an oversimplification but it demonstrates how to return faults. We can use a <switch> activity to determine whether the ticket is approved; then we construct the fault variable and use the <reply> activity to return it to the client. This is shown in the following code:

...

```
<!-- Check if the ticket is approved -->
<switch>

  <case condition="bpws:getVariableData(
                       'TravelResponse',
                       'confirmationData',
                       '/confirmationData/aln:Approved')='true' ">
              <!-- Send a response to the client -->


              <reply partnerLink="client"
                     portType="trv:TravelApprovalPT"
                     operation="TravelApproval"
                     variable="TravelResponse"/>       </case>

                    <otherwise>
    <sequence>
```

```
            <!-- Create the TravelFault variable with fault description -->
                <assign>
                   <copy>
                     <from expression="string('Ticket not approved')" />
                     <to variable="TravelFault" part="error" />
                   </copy>
                </assign>

                    <!-- Send a fault to the client -->

                    <reply partnerLink="client"
                           portType="trv:TravelApprovalPT"
                           operation="TravelApproval"
                           variable="TravelFault"
                           faultName="fault" />          </sequence>
      </otherwise>
    </switch>


</sequence>
</process>
```

If the ticket is not approved, the following fault is signaled to the client:

```
<TravelFault>
<part name="error">
<error xmlns="http://packtpub.com/bpel/travel/">
Ticket not approved
</error>
</part>
</TravelFault>
```

We have seen that signaling faults in synchronous replies is easy. Let us now discuss signaling faults in asynchronous scenarios.

Signaling Faults to Clients in Asynchronous Scenarios

If an asynchronous BPEL process needs to notify the client about a fault, it cannot use the `<reply>` activity. Remember that in asynchronous scenarios the client does not wait for the reply—rather the process uses a callback. To return a fault in callback scenarios, we usually define additional callback operations on the same port type. Through these callback operations, we can signal that an exceptional situation has prevented normal completion of the process.

To demonstrate how faults can be propagated to the client using a callback operation, we will use the asynchronous travel process example. First, we need to modify the travel BPEL process WSDL and introduce another operation called `ClientCallbackFault`. This operation consists of an input message called `tns:TravelFaultMessage`. The message is of the `string` type (similar to

the synchronous example). The declaration of the operation and the message is shown in the following code excerpt:

```
...
<message name="TravelFaultMessage">
  <part name="error" type="xs:string" />
</message>


<portType name="ClientCallbackPT">
<operation name="ClientCallback">
```

```
<input message="aln:TravelResponseMessage" />

</operation>

  <operation name="ClientCallbackFault">
    <input message="tns:TravelFaultMessage" />
  </operation>


</portType>

...
```

We can use the `<switch>` activity to determine whether the ticket has been approved, as in the synchronous example. If the ticket is not approved, however, we `<invoke>` the `ClientCallbackFault` operation instead of using the `<reply>` activity to signal the fault to the client. This is shown in the code excerpt below:

```
...

<!-- Check if the ticket is approved -->

<switch>

<case condition="bpws:getVariableData('TravelResponse',

'confirmationData',

'/confirmationData/aln:Approved')='true' ">

          <!-- Make a callback to the client -->
          <invoke partnerLink="client"
                  portType="trv:ClientCallbackPT"
                  operation="ClientCallback"
                  inputVariable="TravelResponse" />


</case>

<otherwise>

<sequence>

<!-- Create the TravelFault variable with fault description -->

<assign>

<copy>

<from expression="string('Ticket not approved')" />

<to variable="TravelFault" part="error" />

</copy>

</assign>

            <!-- Send a fault to the client -->
            <invoke partnerLink="client"
                    portType="trv:ClientCallbackPT"
                    operation="ClientCallbackFault"
                    inputVariable="TravelFault" />


</sequence>

</otherwise>

</switch>

</sequence>
```

```
</process>
```

In the next section, we will look at how to handle faults thrown in BPEL processes.

Handling Faults

Now that we are familiar with how faults are signaled, let us consider how the business process handles faults. When a fault occurs within a business process (this can be a WSDL fault, a fault thrown by the BPEL process, or any other type of fault), it means that the process may not complete successfully. The process can complete successfully only if the fault is handled within a scope. Scopes are discussed in the next section.

> Business processes handle faults through fault handlers.

A business process can handle a fault through one or more fault handlers. Within a fault handler, the business process defines custom activities that are used to recover from the fault and recover the partial (unsuccessful) work of the activity in which the fault has occurred.

The fault handlers are specified before the first activity of the BPEL process, after the partner links and variables. The overall structure is shown in the following code excerpt:

```
<process ...>

<partnerLinks>

...

</partnerLinks>

<variables>

...

</variables>

   <faultHandlers>

      <catch ... >
         <!-- Perform an activity -->
      </catch>
      <catch ... >
         <!-- Perform an activity -->
      </catch>
      ...
      <catchAll>                        <!-- catchAll is optional -->
         <!-- Perform an activity -->
      </catchAll>
   </faultHandlers>


<sequence>

...

</sequence>

</process>
```

We can see that within the fault handlers we specify several `<catch>` activities where we indicate the fault that we would like to catch and handle. Within a fault handler, we have to specify at least one `<catch>` or a `<catchAll>` activity. Of course, the `<catchAll>` activity can be specified only once within a fault handler.

Usually we will specify several `<catch>` activities to handle specific faults and use the `<catchAll>` to handle all other faults. The `<catch>` activity has two attributes, of which we have to specify at least one:

- `faultName`: Specifies the name of the fault to be handled

- `faultVariable`: Specifies the variable type used for fault data

The flexibility of `<catch>` activities is high and all the following variations are permissible:

```
<faultHandlers>
<catch faultName="trv:TicketNotApproved" >
<!-- First fault handler -->
<!-- Perform an activity -->
</catch>
<catch faultName="trv:TicketNotApproved" faultVariable="TravelFault" >
<!-- Second fault handler -->
<!-- Perform an activity -->
</catch>
<catch faultVariable="TravelFault" >
<!-- Third fault handler -->
<!-- Perform an activity -->
</catch>
<catchAll>
<!-- Perform an activity -->
</catchAll>
   </faultHandlers>
```

We can see that fault handlers in BPEL are very similar to try/catch clauses found in modern programming languages.

Selection of a Fault Handler

Let us consider the fault handlers listed above and discuss the scenarios for which the `<catch>` activities will be selected:

- The first `<catch>` will be selected if the `trv:TicketNotApproved` fault has been thrown and the fault carries no fault data.
- The second `<catch>` will be selected if the `trv:TicketNotApproved` fault has been thrown and carries data of type matching that of the `TravelFault` variable.
- The third `<catch>` will be selected if a fault has been thrown whose fault variable type matches the `TravelFault` variable type and whose name is not `trv:TicketNotApproved`.
- In all other cases the `<catchAll>` will be selected.

We can see that the selection of the `<catch>` activity within fault handlers is quite complicated. It may even happen that a fault matches several `<catch>` activities. Therefore, BPEL specifies exact rules to select the fault handler that will process a fault:

- For faults without associated fault data, the fault name will be matched. The `<catch>` activity with a matching `faultName` will be selected if present; otherwise the default `<catchAll>` handler will be used if present.
- For faults with associated fault data, a `<catch>` activity specifying a matching `faultName` value and `faultVariable` value will be selected, if present. Otherwise, a

`<catch>` activity with no specified `faultName` and a matching `faultVariable` will be selected, if present. Otherwise, the default `<catchAll>` handler will be used, if present.

The <catchAll> activity will execute only if no other <catch> activity has been selected.

If no <catch> is selected and <catchAll> is not present, the fault will be re-thrown to the immediately enclosing scope, if present. Otherwise, the process will terminate abnormally. This situation is similar to explicitly terminating the process using the <terminate> activity.

Synchronous Example

Let's go back to the synchronous BPEL travel process example to add a fault handlers section. We need to define a fault handler that will simply signal the fault to the client. In real-world scenarios, a fault handler can perform additional work to try to recover the work done by an activity or retry the activity itself.

To signal the fault to the client, we use the same TravelFaultMessage message that we defined in the previous section. Here is an excerpt from the WSDL:

```
...
<message name="TravelFaultMessage">
<part name="error" type="xs:string" />
</message>
<portType name="TravelApprovalPT">
<operation name="TravelApproval">
<input message="tns:TravelRequestMessage" />
<output message="aln:TravelResponseMessage" />
<fault name="fault" message="tns:TravelFaultMessage" />
</operation>
</portType>
...
```

We define a fault handler and add a <faultHandlers> section immediately after the <variables> definition and before the <sequence> activity, as shown below. The fault handler for the trv:TicketNotApproved fault is defined with the associated TravelFault variable. This handler will use the <reply> activity to signal the fault to the BPEL client. We will also provide a default <catchAll> handler, which will first create a variable and then use the <reply> activity to signal the fault to the client:

```
   ...
   <faultHandlers>
      <catch faultName="trv:TicketNotApproved" faultVariable="TravelFault">
            <reply partnerLink="client"
               portType="trv:TravelApprovalPT"
               operation="TravelApproval"
               variable="TravelFault"
               faultName="fault" />


</catch>
<catchAll>
<sequence>
<!-- Create the TravelFault variable -->
<assign>
<copy>
```

```
<from expression="string('Other fault')" />
<to variable="TravelFault" part="error" />
</copy>
</assign>
<reply partnerLink="client"
portType="trv:TravelApprovalPT"
operation="TravelApproval"
variable="TravelFault"
faultName="fault" />
</sequence>
</catchAll>
</faultHandlers>
    ...
```

We also have to modify the process itself. Instead of replying to the client (`<reply>`) in the `<switch>` activity if the ticket has not been approved, we will simply throw a fault, which will be caught by the corresponding fault handler. The fault handler will also catch other possible faults:

```
...
<!-- Check if the ticket is approved -->
<switch>
<case condition="bpws:getVariableData('TravelResponse',
'confirmationData',
'/confirmationData/aln:Approved')='true' ">
<!-- Send a response to the client -->
<reply partnerLink="client"
portType="trv:TravelApprovalPT"
operation="TravelApproval"
variable="TravelResponse"/>
</case>
<otherwise>
<sequence>
<!-- Create the TravelFault variable with fault description -->
<assign>
<copy>
<from expression="string('Ticket not approved')" />
<to variable="TravelFault" part="error" />
</copy>
</assign>
                <!-- Throw fault -->
                <throw faultName="trv:TicketNotApproved"
```

```
                    faultVariable="TravelFault" />


</sequence>

</otherwise>

</switch>

      ...
```

Faults that are not handled by the BPEL process result in abnormal termination of the process and are not propagated to the client. In other words, unhandled faults do not cross service boundaries unless explicitly specified using a <reply> activity as we did in our example. This differentiates BPEL from Java and other languages where unhandled exceptions are propagated to the client.

Asynchronous Example

In asynchronous BPEL processes, faults are handled in the same way as in synchronous processes by using <faultHandlers>. We need to define a fault handler that, in our example, will simply forward the fault to the client. We cannot, however, use the <reply> activity to signal the fault to the client. Instead, we need to define an additional callback operation and use the <invoke> activity, as we did in our previous example. In this example we will use the same fault callback operation as in the previous asynchronous example:

...

```
<message name="TravelFaultMessage">
  <part name="error" type="xs:string" />
</message>


<portType name="ClientCallbackPT">

<operation name="ClientCallback">

<input message="aln:TravelResponseMessage" />

</operation>

  <operation name="ClientCallbackFault">
    <input message="tns:TravelFaultMessage" />
  </operation>


</portType>

...
```

Now we will define the <faultHandlers> section. The difference to the synchronous example will be that we will use the <invoke> activity to invoke the newly defined operation instead of the <reply> activity to propagate the fault to the client:

...

```
<faultHandlers>

<catch faultName="trv:TicketNotApproved" faultVariable="TravelFault">

        <!-- Make a callback to the client -->
        <invoke partnerLink="client"
                portType="trv:ClientCallbackPT"
                operation="ClientCallbackFault"
                inputVariable="TravelFault" />


</catch>
```

```
<catchAll>
<sequence>
<!-- Create the TravelFault variable -->
<assign>
<copy>
<from expression="string('Other fault')" />
<to variable="TravelFault" part="error" />
</copy>
</assign>
                <invoke partnerLink="client"
                        portType="trv:ClientCallbackPT"
                        operation="ClientCallbackFault"
                        inputVariable="TravelFault" />


</sequence>
</catchAll>
</faultHandlers>
   ...
```

Another important question related to fault handling is how the BPEL process can be notified of faults that occurred in asynchronously invoked partner web service operations. A typical example is the invocation of the American and Delta Airlines web services in our example. To invoke the operation, we used the `<invoke>` activity and then a `<receive>` activity to wait for the callback.

BPEL provides a way to wait for more than just one message (operation call) using the `<pick>` activity, which is described later in this chapter in the *Managing Events* section. By using `<pick>` instead of `<receive>`, our BPEL process can wait for several incoming messages. One of these can be a message for regular callback; others can be messages that signal fault conditions. With `<pick>`, we can even specify a timeout for receiving a callback. For further information on these issues, please see the *Managing Events* section.

Inline Fault Handling

The loosely coupled model of web services and the use of Internet connections for accessing them make the invocation of operations on web services particularly error prone. Numerous situations can prevent a BPEL process from successfully invoking a partner web service operation, such as broken connections, unavailability of web services, changes in the WSDL, and so on.

Such faults can be handled in the general `<faultHandlers>` sections. However, a more efficient way is to handle faults related to the `<invoke>` activity directly and not rely on the general fault handlers. The `<invoke>` activity provides a shortcut to achieve this—inline fault handlers.

> Inline fault handlers can catch WSDL faults for synchronous operations, and also other faults related to the run-time environment, communications, and so on.

The syntax for inline fault handlers in the `<invoke>` activity is similar to the syntax of the `<faultHandlers>` section. As shown in the code excerpt below we can specify zero or more `<catch>` activities and we can also specify a `<catchAll>` handler. The only difference is that in inline `<catch>` activities, we have to specify a fault name. Optionally, we may specify the fault variable:

```
<invoke ... >
<catch faultName="fault-name" >
<!-- Perform an activity -->
```

```
</catch>
...
<catch faultName="fault-name" faultVariable="fault-variable" >
<!-- Perform an activity -->
</catch>
...
<catchAll>
<!-- Perform an activity -->
</catchAll>
   </invoke>
```

The following code excerpt shows an inline fault handler for invoking the Employee Travel Status web service from our BPEL travel process example. Please notice that this also requires modifying the Employee Travel Status WSDL and declaring an additional fault message for the operation. Because this code is similar to what we did in previous examples, it is not repeated here again. The following code excerpt demonstrates inline fault handling:

```
<invoke partnerLink="employeeTravelStatus"
portType="emp:EmployeeTravelStatusPT"
operation="EmployeeTravelStatus"
input Variable="EmployeeTravelStatusRequest"
outputVariable="EmployeeTravelStatusResponse" >
<catch faultName="emp:WrongEmployeeName" >
<!-- Perform an activity -->
</catch>
<catch faultName="emp:TravelNotAllowed" faultVariable="FaultDesc" >
<!-- Perform an activity -->
</catch>
<catchAll>
<!-- Perform an activity -->
</catchAll>
   </invoke>
```

This brings us to the thought that it would be useful if we could specify more than one `<faultHandlers>` section in a BPEL process. It would be great if we could specify different fault handlers sections for different parts of the process, particularly for complex processes. This is possible if we use scopes, described in the next section. We will see that inline fault handling of the `<invoke>` activity is equal to enclosing the `<invoke>` activity in a local scope.

User name:
Book: Business Process Execution Language for Web Services

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.

Scopes

Scopes provide a way to divide a complex business process into hierarchically organized parts—*scopes.* Scopes provide behavioral contexts for activities. In other words scopes address the problem that we identified in the previous section and allow us to define different fault handlers for different activities (or sets of activities gathered under a common structured activity, such as `<sequence>` or `<flow>`). In addition to fault handlers, scopes also provide a way to declare variables that are visible only within the scope. Scopes also allow us to define local correlation sets, compensation handlers, and event handlers. We will discuss these topics later in this chapter.

The code excerpt below shows how scopes are defined in BPEL. We can specify `<variables>`, `<correlationSets>`, `<faultHandlers>`, `<compensationHandlers>`, and `<eventHandlers>` locally for the scope:

```
<scope>
<variables>
<!-- Variables definitions local to scope -->
</variables>
<correlationSets>
<!-- Correlation sets will be discussed later in this chapter -->
</correlationSets>
<faultHandlers>
<!-- Fault handlers local to scope -->
</faultHandlers>
<compensationHandler>
<!-- Compensation handlers will be discussed later in this chapter -->
</compensationHandler>
<eventHandlers>
<!-- Event handlers will be discussed later in this chapter -->
</eventHandlers>
activity
</scope>
```

Each scope has a primary activity. This is similar to the overall process structure, where we have said that a BPEL process also has a primary activity. The primary activity, often a `<sequence>` or `<flow>`, defines the behavior of a scope for normal execution. Fault handlers and other handlers define the behavior for abnormal execution scenarios.

The primary activity of a scope can be a basic activity such as `<invoke>` or it can be a structured activity such as `<sequence>` or `<flow>`. Enclosing the `<invoke>` activity with a scope and defining the fault handlers is equivalent to using inline fault handlers. The inline fault handler shown in the previous section is equal to the following scope:

```
<scope>
```

```
<faultHandlers>
<catch faultName="emp:WrongEmployeeName" >
<!-- Perform an activity -->
</catch>
<catch faultName="emp:TravelNotAllowed" faultVariable="Description" >
<!-- Perform an activity -->
</catch>
<catchAll>
<!-- Perform an activity -->
</catchAll>
</faultHandlers>
<invoke partnerLink="employeeTravelStatus"
portType="emp:EmployeeTravelStatusPT"
operation="EmployeeTravelStatus"
inputVariable="EmployeeTravelStatusRequest"
outputVariable="EmployeeTravelStatusResponse" >
</invoke>
 </scope>
```

If the primary activity of a scope is a structured activity, it can have many nested activities where the nesting depth is arbitrary. The scope is shared by all nested activities. A scope can also have nested scopes with arbitrary depth.

The variables defined within a scope are only visible within that scope. Fault handlers attached to a scope handle faults of all nested activities of a scope. Faults not caught in a scope are re-thrown to the enclosing scope. Scopes in which faults have occurred are considered to have ended abnormally even if a fault handler has caught the fault and not re-thrown it.

Example

To demonstrate how scopes can be used in BPEL processes, we will rewrite our asynchronous travel process example and introduce three scopes:

- In the first scope we will retrieve the employee travel status (`RetrieveEmployeeTravelStatus`).
- In the second scope we will check the flight availability with both airlines (`CheckFlightAvailability`).
- In the third scope we will call back to the client (`CallbackClient`).

We will also declare those variables that are limited to a scope locally within the scope. This will reduce the number of global variables and make the business process easier to understand. The major benefit of scopes is the capability to define custom fault handlers, which we will also implement. The high-level structure of our travel process will be as follows:

```
<process ...>
<partnerLinks/>...</partnerLinks>
<variables>...</variables>
<faultHandlers>
<catchAll>...</catchAll>
</faultHandlers>
<sequence>
```

```
<!-- Receive the initial request for business travel from client -->
<receive .../>
      <scope name="RetrieveEmployeeTravelStatus">

<variables>...</variables>
<faultHandlers>
<catchAll>...</catchAll>
</faultHandlers>
<sequence>
<!-- Prepare the input for Employee Travel Status Web Service -->
<!-- Synchronously invoke the Employee Travel Status Web Service -->
<!-- Prepare the input for AA and DA -->
</sequence>
      </scope>

      <scope name="CheckFlightAvailability">

<variables>...</variables>
<faultHandlers>
<catchAll>...</catchAll>
</faultHandlers>
<sequence>
<!-- Make a concurrent invocation to AA and DA -->
<flow>
<!-- Async invoke the AA web service and wait for the callback -->
<!-- Async invoke the DA web service and wait for the callback -->
</flow>
<!-- Select the best offer and construct the TravelResponse -->
</sequence>
      </scope>

<scope name="CallbackClient">
<faultHandlers>...</faultHandlers>
<!-- Check if the ticket is approved -->
      </scope>

</sequence>
</process>
```

To signal faults to the BPEL process client, we will use the `ClientCallbackFault` operation on the client partner link, which we defined in the previous section. This operation has a string message, which we will use to describe the fault. In real-world scenarios the fault message is more complex and includes a fault code and other relevant information.

Let us start with the example. The process declaration and the partner links have not changed:

```
<process name="Travel"

targetNamespace="http://packtpub.com/bpel/travel/"

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

xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/" xmlns:trv="http://packtpub.com
/bpel/travel/"

xmlns:emp="http://packtpub.com/service/employee/"

xmlns:aln="http://packtpub.com/service/airline/" >

<partnerLinks>

<partnerLink name="client"

partnerLinkType="trv:travelLT"

myRole="travelService"

partnerRole="travelServiceCustomer"/>

<partnerLink name="employeeTravelStatus"

partnerLinkType="emp:employeeLT"

partnerRole="employeeTravelStatusService"/>

<partnerLink name="AmericanAirlines"

partnerLinkType="aln:flightLT"

myRole="airlineCustomer"

partnerRole="airlineService"/>

<partnerLink name="DeltaAirlines"

partnerLinkType="aln:flightLT"

myRole="airlineCustomer"

partnerRole="airlineService"/>

</partnerLinks>

    ...
```

The variables section will now define only global variables. These are `TravelRequest`, `FlightDetails`, `TravelResponse`, and `TravelFault`. We have reduced the number of global variables, but we will have to declare other variables within scopes:

```
...

<variables>

<!-- input for this process -->

<variable name="TravelRequest"

messageType="trv:TravelRequestMessage"/>

<!-- input for the Employee Travel Status web service -->

<variable name="FlightDetails"
```

```
messageType="aln:FlightTicketRequestMessage"/>
<!-- output from BPEL process -->
<variable name="TravelResponse"
messageType="aln:TravelResponseMessage"/>
<!-- fault to the BPEL client -->
<variable name="TravelFault"
messageType="trv:TravelFaultMessage"/>
</variables>
    ...
```

Next we define the global fault handlers section. Here we use the `<catchAll>` activity, through which we handle all faults not handled within scopes. We will signal the fault to the BPEL client:

```
...
<faultHandlers>
<catchAll>
<sequence>
<!-- Create the TravelFault variable -->
<assign>
<copy>
<from expression="string('Other fault')" />
<to variable="TravelFault" part="error" />
</copy>
</assign>
<invoke partnerLink="client"
portType="trv:ClientCallbackPT"
operation="ClientCallbackFault"
inputVariable="TravelFault" />
</sequence>
</catchAll>
</faultHandlers>
    ...
```

The main activity of the BPEL process will still be `<sequence>`, and we will also specify the `<receive>` activity to wait for the incoming message from the client:

```
...
<sequence>
<!-- Receive the initial request for business travel from client -->
<receive partnerLink="client"
portType="trv:TravelApprovalPT"
```

```
operation="TravelApproval"
variable="TravelRequest"
createInstance="yes" />
    ...
```

First Scope

Now let's define the first scope for retrieving the employee travel status. Here we will first declare two variables needed for the input and output messages for web service operation invocation:

```
...
<scope name="RetrieveEmployeeTravelStatus">
<variables>
<!-- input for the Employee Travel Status web service -->
<variable name="EmployeeTravelStatusRequest"
messageType="emp:EmployeeTravelStatusRequestMessage" />
<!-- output from the Employee Travel Status web service -->
<variable name="EmployeeTravelStatusResponse"
messageType="emp:EmployeeTravelStatusResponseMessage" />
</variables>
    ...
```

Next we will define the fault handlers section for this scope. We will use the `<catchAll>` activity to handle all faults, including Employee web service WSDL faults, communication faults, and other run-time faults. We will signal all faults to the client, although in real-world scenarios we could invoke another web service or perform other recovery operations:

```
...
<faultHandlers>
<catchAll>
<sequence>
<!-- Create the TravelFault variable -->
<assign>
<copy>
<from expression= "string('Unable to retrieve employee travel status')" />
<to variable="TravelFault" part="error" />
</copy>
</assign>
<invoke partnerLink="client"
portType="trv:ClientCallbackPT"
operation="ClientCallbackFault"
inputVariable="TravelFault" />
<terminate/>
</sequence>
```

```
</catchAll>
</faultHandlers>
    ...
```

Next we will start a sequence (which is the main activity of the scope) and prepare the input variable, invoke the Employee web service, and prepare the input for both airlines' web services:

```
...
<sequence>
<!-- Prepare the input for the Employee Travel Status Web Service -->
<assign>
<copy>
<from variable="TravelRequest" part="employee"/>
<to variable="EmployeeTravelStatusRequest" part="employee"/>
</copy>
</assign>
<!-- Synchronously invoke the Employee Travel Status Web Service -->
<invoke partnerLink="employeeTravelStatus"
portType="emp:EmployeeTravelStatusPT"
operation="EmployeeTravelStatus"
inputVariable="EmployeeTravelStatusRequest"
outputVariable="EmployeeTravelStatusResponse" />
<!-- Prepare the input for AA and DA -->
<assign>
<copy>
<from variable="TravelRequest" part="flightData"/>
<to variable="FlightDetails" part="flightData"/>
</copy>
<copy>
<from variable="EmployeeTravelStatusResponse" part="travelClass"/>
<to variable="FlightDetails" part="travelClass"/>
</copy>
</assign>
</sequence>
</scope>
    ...
```

Second Scope

In the second scope we check the flight availability with both airlines' web services. First we declare two variables for storing output from both web service operations:

```
...
```

```
<scope name="CheckFlightAvailability">
<variables>
<!-- output from American Airlines -->
<variable name="FlightResponseAA"
messageType="aln:TravelResponseMessage"/>
<!-- output from Delta Airlines -->
<variable name="FlightResponseDA"
messageType="aln:TravelResponseMessage"/>
</variables>
    ...
```

Next we define the fault handlers section, where we use the `<catchAll>` activity similarly to in the first scope:

```
...
<faultHandlers>
<catchAll>
<sequence>
<!-- Create the TravelFault variable -->
<assign>
<copy>
<from expression= "string('Unable to invoke airline web service')" />
<to variable="TravelFault" part="error" />
</copy>
</assign>
<invoke partnerLink="client"
portType="trv:ClientCallbackPT"
operation="ClientCallbackFault"
inputVariable="TravelFault" />
<terminate/>
</sequence>
</catchAll>
</faultHandlers>
    ...
```

The main activity of the second scope will be a `<sequence>` in which we will first concurrently invoke both airlines' web services using a `<flow>` activity and then select the best offer using a `<switch>` activity:

```
...
<sequence>
<!-- Make a concurrent invocation to AA and DA -->
<flow>
```

```
<sequence>
<!-- Async invoke of the AA web service
                        and wait for the callback -->

<invoke partnerLink="AmericanAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" />
<receive partnerLink="AmericanAirlines"
portType="aln:FlightCallbackPT"
operation="FlightTicketCallback"
variable="FlightResponseAA" />
</sequence>
<sequence>
<!-- Async invoke of the DA web service
                        and wait for the callback -->

<invoke partnerLink="DeltaAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" />
<receive partnerLink="DeltaAirlines"
portType="aln:FlightCallbackPT"
operation="FlightTicketCallback"
variable="FlightResponseDA" />
</sequence>
</flow>
<!-- Select the best offer and construct the TravelResponse -->
<switch>
<case condition="bpws:getVariableData('FlightResponseAA', 'confirmationData','/confirmationData/aln:Price')
&lt;= bpws:getVariableData('FlightResponseDA', 'confirmationData','/confirmationData/aln:Price')">
<!-- Select American Airlines -->
<assign>
<copy>
<from variable="FlightResponseAA" />
<to variable="TravelResponse" />
</copy>
</assign>
</case>
```

```
<otherwise>
<!-- Select Delta Airlines -->
<assign>
<copy>
<from variable="FlightResponseDA" />
<to variable="TravelResponse" />
</copy>
</assign>
</otherwise>
</switch>
</sequence>
</scope>
    ...
```

Third Scope

In the third scope we call back to the BPEL client. For this scope we do not need additional variables. However, we define a fault handler to handle the `TicketNotApproved` fault. Therefore we explicitly specify the fault name and the fault variable. Note that we do not use the `<catchAll>` activity in this fault handlers section, so all unhandled faults will be re-thrown to the main process fault handler:

```
...
<scope name="CallbackClient">
<faultHandlers>
<catch faultName="trv:TicketNotApproved"
faultVariable="TravelFault">
<!-- Make a callback to the client -->
<invoke partnerLink="client"
portType="trv:ClientCallbackPT"
operation="ClientCallbackFault"
inputVariable="TravelFault" />
</catch>
</faultHandlers>
    ...
```

The main activity of this scope is the `<switch>` activity, where we check if the flight ticket has been approved:

```
...
<!-- Check if the ticket is approved -->
<switch>
<case condition="bpws:getVariableData('TravelResponse',
'confirmationData',
'/confirmationData/aln:Approved')='true' ">
```

```
<!-- Make a callback to the client -->
<invoke partnerLink="client"
portType="trv:ClientCallbackPT"
operation="ClientCallback"
inputVariable="TravelResponse" />
</case>
<otherwise>
<sequence>
<!-- Create the TravelFault variable with fault description -->
<assign>
<copy>
<from expression="string('Ticket not approved')" />
<to variable="TravelFault" part="error" />
</copy>
</assign>
<!-- Throw fault -->
<throw faultName="trv:TicketNotApproved"
faultVariable="TravelFault" />
</sequence>
</otherwise>
</switch>
</scope>
</sequence>
</process>
```

Serializable Scopes

For each scope we can specify whether we require concurrency control over shared variables. We will need such control if, in our scenario, more than one instance uses shared variables concurrently. This can occur, for example, if we use an event handler through which we react to an event while the main process is executing. This is discussed later in this chapter.

Scopes that require concurrency control are called **serializable scopes**. In serializable scopes, access to all shared variables is serialized; in other words, concurrency is prohibited. This guarantees that there will be no conflicting situations if several concurrent scopes access the same set of shared variables. Conflicting operations are in this case all read/write and write-only activities, such as assignments, incoming messages stored in variables, etc. The semantics of serializable scopes are similar to the *serializable* transaction isolation level.

We denote a scope as serializable using the optional attribute `variableAccessSerializable` and setting it to `yes`. The default value of this attribute is `no`. Serializable scopes must not contain other serializable scopes (but may contain scopes that are not marked as serializable). The fault handlers associated with the scope also share the serializability. The code excerpt below shows how to declare a scope as serializable:

```
<scope variableAccessSerializable="yes" >
...
</scope>
```

At the time of writing this book, not all BPEL severs have supported this feature, so it is wise to check for support before relying on the serializable behavior.

User name:
Book: Business Process Execution Language for Web Services

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.

Compensation

Compensation, or undoing steps in the business process that have already completed successfully, is one of the most important concepts in business processes. Let us discuss the compensation on our travel process and suppose that in addition to checking the flight availability, our business process would also have to confirm the flight tickets, make the payments, reserve a hotel room, and make the payment for the hotel room. If the business travel is canceled (for various reasons) the reservation and payment activities would have to be undone—compensated. In business processes, the compensation behavior must be explicitly defined. Therefore, when defining the BPEL process, we would have to explicitly define how to compensate the flight ticket confirmation, how to compensate the flight ticket payment, etc.

> The goal of compensation is to reverse the effects of previous activities that have been carried out as part of a business process that is being abandoned.

Compensation is related to the nature of most business processes, which are long running and use asynchronous communication with loosely coupled partner web services. Business processes are often sensitive in terms of successful completion because the data they manipulate is sensitive. Because they usually span multiple partners (often multiple enterprises) special care has to be taken that business processes either fully complete their work or that the partial (not fully completed) results are undone – compensated.

In enterprise information systems, processes that have not been able to finish all their activities and need to undo the partial work are usually handled with transactions, more exactly with the ACID distributed transaction model, such as X/Open DTP (Distributed Transaction Processing). ACID stands for Atomicity, Consistency, Isolation, and Durability and defines a transaction model that uses data locking and isolation. Such a model works perfectly well in trusted domains within enterprises under the prerequisite that the duration of transactions can be relatively short.

The problem with business processes is that they usually last a long time, sometimes several hours, sometimes even a few days. This is much too long for the ACID model, because we cannot afford to lock certain data for such a long time and to isolate the access to these data.

In business processes compensation is used instead of ACID to reverse the effects of an unfinished process. Compensation requires that an activity specifies a reverse activity, which can be invoked if it is necessary to undo the effect of that activity. BPEL supports the concept of compensation with the ability to define compensation handlers, which are specific to scopes, and calls this feature *Long-Running Transactions* (LRT).

The concept of compensation and LRTs as defined by BPEL is independent of any transaction protocol and can be used with various business transaction protocols. Because BPEL is bound to web services it is, however, reasonable to expect that in most cases the LRTs will be used with the WS-BusinessActivity (WS-Transaction) specification. It has been described in Chapter 2. The BPEL specification even defines a detailed model of BPEL LRTs based on WS-BusinessActivity concepts.

It is very important to understand that compensation differs from fault handling. In fault handling a business process tries to recover from an activity that could not finish normally because an exceptional situation has occurred. The objective of compensation on the other hand is to reverse the effects of a previous activity or a set of activities that have been carried out successfully as part of a business process that is being abandoned. Note that the order in which compensation activities are run is often important. BPEL addresses this aspect with scopes.

Compensation Handlers

To define the compensation activities, BPEL provides compensation handlers. Compensation handlers gather all activities that have to be carried out to compensate another activity. Compensation handlers can be defined:

- For the whole process
- For the scope

- Inline for the `<invoke>` activity

The compensation handler for the whole BPEL process is defined immediately after the fault handlers section and before the main activity of the process, as shown in the next code excerpt:

```
<process ...>

<partnerLinks>

...

</partnerLinks>

<variables>

...

</variables>

<faultHandlers>

...

</faultHandlers>

   <compensationHandler>
     <!-- Compensation activity
          (or several activities within a <sequence>, <flow>,
          or other structured activity) -->
   </compensationHandler>


main activity

</process>
```

The compensation handler for a scope is also defined after the fault handlers section:

```
<scope>

<variables>

...

</variables>

<correlationSets>

<!-- Correlation sets will be discussed later in this chapter -->

</correlationSets>

<faultHandlers>

...

</faultHandlers>

    <compensationHandler>
     <!-- Compensation activity
          (or several activities within a <sequence>, <flow>,
          or other structured activity) -->
    </compensationHandler>


<eventHandlers>

<!-- Event handlers will be discussed later in this chapter -->

</eventHandlers>

activity
```

```
</scope>
```

Sometimes it is reasonable to define a compensation handler for each <invoke> activity. We could define a scope for each <invoke>. However, BPEL provides a shortcut where we can inline the compensation handler rather than explicitly using an immediately enclosing scope. This is similar to the inline capability of fault handlers. The syntax is shown below:

```
<invoke ... >
     <compensationHandler>

       <!-- Compensation activity
            (or several activities within a <sequence>, <flow>,
            or other structured activity) -->
     </compensationHandler>


   </invoke>
```

The syntax of the compensation handler is the same for all three cases: we specify the activity that has to be performed for compensation. This can be a basic activity such as <invoke> or a structured activity such as <sequence> or <flow>.

Example

Let us suppose that within a business process we will invoke a web service operation through which we will confirm the flight ticket. The compensation activity would be to cancel the flight ticket. The most obvious way to do this is to define the inline compensation handler for the <invoke> activity as shown in the following example:

```
<invoke name="TicketConfirmation"

partnerLink="AmericanAirlines"

portType="aln:TicketConfirmationPT"

operation="ConfirmTicket"

inputVariable="FlightDetails"

outputVariable="Confirmation" >

<compensationHandler>

<invoke partnerLink="AmericanAirlines"

portType="aln:TicketConfirmationPT"

operation="CancelTicket"

inputVariable="FlightDetails"

outputVariable="Cancellation" />

</compensationHandler>

     </invoke>
```

Let us now suppose that the business process performs two operations in a sequence. First it confirms the ticket and then makes the payment. To compensate these two activities we could define an inline compensation handler for both <invoke> activities. Alternatively, we could also define a scope with a dedicated compensation handler, as shown in the example that follows:

```
<scope name="TicketConfirmationPayment" >

<compensationHandler>

<invoke partnerLink="AmericanAirlines"
```

```
portType="aln:TicketConfirmationPT"
operation="CancelTicket"
inputVariable="FlightDetails"
outputVariable="Cancellation" />
<invoke partnerLink="AmericanAirlines"
portType="aln:TicketPaymentPT"
operation="CancelPayment"
inputVariable="PaymentDetails"
outputVariable="PaymentCancellation" />
</compensationHandler>
<invoke partnerLink="AmericanAirlines"
portType="aln:TicketConfirmationPT"
operation="ConfirmTicket"
inputVariable="FlightDetails"
outputVariable="Confirmation" />
<invoke partnerLink="AmericanAirlines"
portType="aln:TicketPaymentPT"
operation="PayTicket"
inputVariable="PaymentDetails"
outputVariable="PaymentConfirmation" />
  </scope>
```

Which approach is better depends on the nature of the business process. In most cases we will define inline compensation handlers or compensation handlers within scopes. In the global BPEL process compensation handler, we will usually invoke compensation handlers for specific scopes and thus define the order in which the compensation should perform. Let's have a look at how to invoke a compensation handler.

Invoking Compensation Handlers

Compensation handlers can be invoked only after the activity that is to be compensated has completed normally. If we try to compensate an activity that has completed abnormally, nothing will happen because an `<empty>` activity will be invoked. This is useful because it is not necessary to track the state of activities to know which can be compensated and which cannot.

BPEL provides the `<compensate>` activity to invoke a compensation handler. The syntax is simple and is shown below. The `<compensate>` activity has an optional `scope` attribute through which we can specify which compensation handler should be invoked. We have to specify the name of the scope. To invoke the inline compensation handler, we specify the name of the `<invoke>` activity:

```
<compensate scope="name" />
```

To invoke the compensation handler for the `TicketConfirmationPayment` scope (shown in the previous section) we could simply write:

```
<compensate scope="TicketConfirmationPayment" />
```

To invoke the inline compensation handler for the `TicketConfirmation` `<invoke>` activity (also shown in the previous section) we write:

```
<compensate scope="TicketConfirmation" />
```

If we invoke a compensation handler for a scope that has no compensation handler defined, the default handler invokes the compensation handlers for the immediately enclosed scopes in the reverse order of completion (remember that the order in which compensations are performed is often important). This behavior is also performed if we use the `<compensate>` activity without specifying the scope name.

Compensation handlers can be explicitly invoked only from:

- The fault handler of the scope that immediately encloses the scope for which compensation should be performed
- The compensation handler of the scope that immediately encloses the scope for which compensation should be performed

The compensation handler defined for the whole BPEL process can be invoked only after the process has completed normally. Invoking it is specific to the run-time environment (BPEL server). Usually the environment will provide a command through which the compensation can be invoked. We can control this behavior using the `enableInstanceCompensation` attribute, which can be `yes` or `no`. The default value of this attribute is `no`, which means that the compensation is not allowed.

> When a compensation handler is invoked, it sees a frozen snapshot of all variables as they were when the scope being compensated was completed.

In the compensation we can use the same variables as in regular activities and these variables will have the same values as when the activity being compensated finished. This means that the compensation handler cannot update live data in the variables the BPEL process is using. The compensation handler *cannot* affect the global state of the business process.

In future versions, BPEL will provide two-way communication between the business process and the compensation handler. We expect that compensation handlers will be supplemented with input and output parameters.

User name:
Book: Business Process Execution Language for Web Services

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.

Managing Events

A business process may have to react on certain events. We already know that a business process specified in BPEL usually waits for an incoming message using the `<receive>` activity. This incoming message is the event that activates the whole process. A business process also often invokes web service operations asynchronously. For such operations, results are returned using callbacks. The BPEL process often waits for callback messages, which are also events.

Using the `<receive>` activity, we can wait for an exactly specified message on a certain port type. Often, however, it is more useful to wait for more than one message, of which only one will occur. Let us go back to our example, where we invoked the `FlightAvailability` operation and waited for the `FlightTicketCallback` callback. In a real-world scenario, it would be very useful to wait for several messages, `FlightTicketCallback` being one of them. The other messages could include `FlightNotAvaliable`, `TicketNotAvaliable`, etc.

Even more useful would be to specify that we will wait for the callback for a certain period of time (for example, 5 minutes). If no callback is received, we continue the process flow. This is particularly useful in loosely coupled service-oriented architectures, where we cannot rely on web services being available all the time. This way, we could proceed with the process flow even if American Airlines' web service does not return an offer— we would then invoke another airline web service operation.

In most business processes, we will need to react on two types of events:

- **Message events**: These are triggered by incoming messages through operation invocation on port types
- **Alarm events:** These are time related and are triggered either after a certain duration or at a specific time.

Pick Activity

BPEL provides the `<pick>` activity through which we can specify that the business process awaits the occurrence of one of a set of events. Events can be message events handled using the `<onMessage>` activity and alarm events handled using the `<onAlarm>` activity. For each event we then specify an activity or a set of activities that should be performed.

The syntax of the `<pick>` activity is shown below:

```
<pick>

<onMessage ...>

<!-- Perform an activity -->

</onMessage>

<onMessage ...>

<!-- Perform an activity -->

</onMessage>

...

<onAlarm ...>

<!-- Perform an activity -->

</onAlarm>

...
```

```
</pick>
```

Within <pick> we can specify several <onMessage> elements and several <onAlarm> elements. The <onAlarm> elements are optional (we can specify zero or more), but we have to specify at least one <onMessage> element.

Message Events

Both elements take additional attributes. The <onMessage> element is identical to the <receive> activity, and has the same set of attributes. We have to specify the following attributes:

- partnerLink: Specifies which partner link will be used for the invoke, receive, or reply, respectively
- portType: Specifies the used port type
- operation: Specifies the name of the operation to wait for being invoked
- variable: Specifies the name of the variable used to store the incoming message

The syntax is shown in the code excerpt below:

```
<pick>

      <onMessage partnerLink="name"
                 portType="name"
                 operation="name"
                 variable="name">


         <!-- Perform an activity or a set of activities enclosed by
              <sequence>, <flow>, etc. or throw a fault -->


      </onMessage>


...
   </pick>
```

Alarm Events

The <onAlarm> element is similar to the <wait> element. We can specify:

- A duration expression using a for attribute
- A deadline expression using an until attribute

For both expressions we use the same literal format as for the <wait> activity described earlier in this chapter.

Most often we will use the <onAlarm> event to specify duration. A typical example is for a business process to wait for the callback a certain amount of time, for example 15 minutes. If no callback is received the business process invokes another operation or throws a fault. The deadline approach is useful for example if the business process should wait for a callback until an exactly specified time and then throw a fault or perform a backup activity.

The code excerpt below shows examples of both with hard-coded times/dates:

```
<pick>

<onMessage ...>

<!-- Perform an activity -->

</onMessage>

...

<onAlarm for="'PT15M'">
```

```
        <!-- Perform an activity or a set of activities enclosed by
             <sequence>, <flow>, etc. or throw a fault -->
      </onAlarm>


   </pick>
```

```
<pick>

<onMessage ...>

<!-- Perform an activity -->

</onMessage>

...

<onAlarm until="'2004-03-18T21:00:00+01:00'">

        <!-- Perform an activity or a set of activities enclosed by
             <sequence>, <flow>, etc. or throw a fault -->
      </onAlarm>


   </pick>
```

Instead of hard-coding the exact date and time or the duration we can use a variable and access the information using the `getVariableData()` function.

Example

Going back to our travel example we could replace the `<receive>` activity, where the business process waited for the `FlightTicketCallback`, with the `<pick>` activity, where the business process will also wait for the `FlightNotAvaliable` and `TicketNotAvaliable` operations and throw corresponding faults. The business process will wait no more than 30 minutes, when it will throw a `CallbackTimeout` fault. The code excerpt is shown overleaf:

```
<pick>

<onMessage partnerLink="AmericanAirlines"

portType="aln:FlightCallbackPT"

operation="FlightTicketCallback"

variable="FlightResponseAA">

<empty/>

<!-- Continue with the rest of the process -->

</onMessage>

<onMessage partnerLink="AmericanAirlines"

portType="aln:FlightCallbackPT"

operation="FlightNotAvaliable"

variable="FlightFaultAA">

<throw faultName="trv:FlightNotAvaliable" faultVariable="FlightFaultAA"/>

</onMessage>

<onMessage partnerLink="AmericanAirlines"

portType="aln:FlightCallbackPT"

operation="TicketNotAvaliable"
```

variable="FlightFaultAA">

<throw faultName="trv:TicketNotAvaliable" faultVariable="FlightFaultAA"/>

</onMessage>

<onAlarm for="'PT30M'">

<throw faultName="trv:CallbackTimeout" />

</onAlarm>

```
            </pick>
```

For this example to work, we also need to declare the `FlightFaultAA` variable and to modify the Airline web service WSDL to add the `FlightNotAvaliable` and `TicketNotAvaliable` callback operations. This is not shown here but can be seen from the example, which can be downloaded from Packt's website.

Event Handlers

The `<pick>` activity is very useful when we have to specify that the business process should wait for events. Sometimes, however, we would like to react on events that occur while the business process executes. In other words, we do not want the business process to wait for the event (and do nothing else but wait). Instead the process should execute, and still listen to events and handle them whenever they occur.

For this purpose BPEL provides event handlers. If the corresponding events occur, event handlers are invoked concurrently with the business process. Typical usage of event handlers is to handle a cancellation message from the client. For example, in our travel process we could define an event handler that would allow the BPEL process client to cancel the travel at any time.

We can specify event handlers for the whole BPEL process as well as for each scope. Event handlers for the whole process are specified immediately after the compensation handlers and before the main process activity as shown below:

<process ...>

<partnerLinks>

...

</partnerLinks>

<variables>

...

</variables>

<faultHandlers>

...

</faultHandlers>

<compensationHandler>

...

</compensationHandler>

```
   <eventHandlers>
      <onMessage ...>
         <!-- Perform an activity -->
      </onMessage>
      ...
      <onAlarm ...>
         <!-- Perform an activity -->
      </onAlarm>
      ...
   </eventHandlers>
```

activity

</process>

Event handlers for the scope are also specified after compensation handlers, as shown in the excerpt below:

<scope>

<variables>

...

</variables>

<correlationSets>

<!-- Correlation sets will be discussed later in this chapter -->

</correlationSets>

<faultHandlers>

...

</faultHandlers>

<compensationHandler>

...

</compensationHandler>

```
<eventHandlers>
  <onMessage ...>
     <!-- Perform an activity -->
  </onMessage>
  ...
  <onAlarm ...>
     <!-- Perform an activity -->
  </onAlarm>
  ...
</eventHandlers>
```

activity

</scope>

The syntax of the event handler section is similar to the syntax of the <pick> activity. The only difference is that within the event handler, we can specify zero or more <onMessage> events and/or zero or more <onAlarm> events.

> Message events in event handlers can occur multiple times, even concurrently, while the corresponding scope is active. We have to take care of concurrency and use serializable scopes if necessary.

Example

Let us go back to the example and define the event handler that will allow the BPEL process client to cancel the travel at any time. The difficult part here is to define the appropriate activities to be performed when the client does the cancellation. The simplest solution is to terminate the process, as shown in the example below:

<process name="Travel"

enableInstanceCompensation="yes" ... >

...

<eventHandlers>

```
<onMessage partnerLink="client"
portType="trv:TravelApprovalPT"
operation="CancelTravelApproval"
variable="TravelRequest" >
<terminate/>
</onMessage>
</eventHandlers>
...
</process>
```

In the real world, we would want to undo some work when a cancellation actually occurs. Since we cannot invoke a compensation handler from an event handler, a better approach is to terminate the process and invoke the compensation handler for the whole process. To enable this, we have to set the `enableInstanceCompensation` attribute of the `<process>` tag to `yes`.

Another possibility would be to specify the alarm event, which would prevent a business process from executing for too long. The following example shows an alarm using a duration expression of 12 hours. We could use variable data to specify the duration instead of hard-coding it.

```
<process name="Travel"
enableInstanceCompensation="yes" ... >
...
<eventHandlers>
<onAlarm for="'PT12H'">
<terminate/>
</onAlarm>
</eventHandlers>
...
</process>
```

Other usage scenarios depend on the actual business process. Note that the examples shown for the process could also be defined within scopes. Because the code differences are minimal these examples are not shown.

The event handlers associated with the scopes are enabled when the associated scope starts. The event handlers associated with the global BPEL process are enabled as soon as the process instance is created. This brings us to the process lifecycle, which we will discuss in the next section.

User name:
Book: Business Process Execution Language for Web Services

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.

Business Process Lifecycle

A business process specified in BPEL has a well-defined structure. It usually waits for the client to invoke the process. This is done using the `<receive>` activity, as we have seen in the previous chapter. A business process can also use the `<pick>` activity to wait for the initial incoming message. Then the business process typically invokes several operations on partner web services and waits for partners to invoke callback operations. The business process also performs some logic, such as comparison and calculation of certain values. The business process terminates after all activities have been performed.

We can see that each BPEL process has a well-defined lifecycle. To communicate with partners BPEL uses web services. Web services provide a stateless model for operation invocation. This means that a web service does not provide a common approach to store client-dependent information between operation invocations. For example, consider a shopping cart where a client uses an `add` operation to add items to the cart. Of course there could be several simultaneous clients using the shopping cart through the web service. We would like each client to have its own cart. To achieve this using web services each client would have to pass its identity for each invocation of the `add` operation. This is because the web services model is a stateless model—a web service does not distinguish between different clients.

For business processes a stateless model is inappropriate. Let us consider the business travel scenario where a client sends a travel order, through which it initiates the business process. The process then communicates with several web services and first sends a ticket approval to the client. Later it sends a hotel approval and an invoice. There are usually several concurrent clients using the business travel process. Also, a single client can start more than one interaction with the business process. The business process has to remember each interaction in order to know to whom to return the results.

> In contrast to stateless web services, BPEL business processes are stateful long-running interactions.

BPEL business processes are stateful and support long-running interactions with a well-defined lifecycle. For each interaction with the process, a **process instance** is created. Therefore we can think of the BPEL process definition as a template for creating process instances. This is similar to the class-object relation where classes represent templates for creating objects at run time.

In BPEL, we do not create instances explicitly as we would in programming languages (there is no `new` command for example). Rather, the creation is implicit and occurs when the process receives the initial message that starts the process. This can happen within the `<receive>` or `<pick>`

Activities, so both activities provide an attribute called `createInstance`. Setting this attribute to `yes` indicates that the occurrence of that activity causes a new instance of the business process to be created.

We usually annotate the initial `<receive>` or `<pick>` of each business process with the `createInstance` attribute. Going back to our business travel example, this is shown in the excerpt below:

```
...

<sequence>

<!-- Receive the initial request for business travel from client -->

<receive partnerLink="client"

portType="trv:TravelApprovalPT"

operation="TravelApproval"

variable="TravelRequest"
```

```
createInstance="yes" />    ...
```

If, however, we would like to specify more than one operation we can use a special form of the <pick> activity. Using <pick> we can specify several operations and receiving any one of these messages will result in business process instance creation. We specify the createInstance attribute for the <pick> activity. However, we can only specify <onMessage> events; <onAlarm> events are not permitted in this specific form.

The following example shows the initial business process activity, which waits for the TravelApproval or TravelCancellation operations. Receiving one of these messages results in business process instance creation:

```
...
<pick createInstance="yes">
<onMessage partnerLink="client"
portType="trv:TravelApprovalPT"
operation="TravelApproval"
variable="TravelRequest" >
<!-- Perform activities -->
</onMessage>
<onMessage partnerLink="client"
portType="trv:TravelCancellationPT"
operation="TravelCancellation"
variable="TravelCancel" >
<!-- Perform activities -->
</onMessage>
   </pick>
```

A business process can be terminated normally or abnormally. **Normal termination** occurs when all business process activities complete. **Abnormal termination** occurs either when a fault occurs within the process scope, or a process instance is terminated explicitly using the <terminate> activity.

In more complex business processes more than one start activity could be enabled concurrently. Such start activities are required to use correlation sets.

User name:
Book: Business Process Execution Language for Web Services

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.

Correlation and Message Properties

Business processes use a stateful model. When a client starts a business process, a new instance is created. This instance lives for the duration of the business process. Messages sent to the business process (using operations on port types and ports) need to be delivered to the correct instance of the business process. We would expect this to be provided by the run-time environment, such as a BPEL server. This is the case if an appropriate transport mechanism can be used, such as WS-Addressing. However, in some cases where several partners are involved (for example if the BPEL process calls service A, which calls service B, and service B makes a direct callback to the BPEL process), or a lightweight transport mechanism is used that does not provide enough information to explicitly identify instances (such as JMS), manual correlation is required. In such cases we will have to use specific business data, such as flight numbers, social security numbers, chassis number, etc.

BPEL provides a mechanism to use such specific business data to maintain references to specific business process instances and calls this feature **correlation**. Business data used for correlation is contained in the messages exchanged between partners. The exact location usually differs from message to message—for example the flight number in the message from the passenger to the airline might be in a different location than in the confirmation message from the airline to the passenger etc. To specify which data is used for correlation, message properties are used.

Message Properties

Messages exchanged between partner web services in a business process usually contain application-specific data and protocol-specific data. Application-specific data is the data related to the business process. In our example, such data includes the employee name, employee travel status, travel destination, and dates, etc. To actually transfer this data (as SOAP messages, for example) additional protocol-specific data has to be added, such as security context, transaction context, etc. In SOAP, protocol-specific data is usually gathered in the `Header` section and application-specific data in the `Body` section of a SOAP message. However, not all protocols differentiate application- and protocol-specific data.

In business processes we will always need to manipulate application-specific data, and sometimes even protocol-specific data. BPEL provides a notion of **message properties**, which allow us to associate relevant data with names that have greater significance than just the data types used for such data.

For example, a chassis number can be used to identify a motor vehicle in a business process. The chassis number will probably appear in several messages and it will always identify the vehicle. Let us suppose that the chassis number is of type `string`, because a chassis number consists of numbers and characters. Naming it with a global property name `chassisNo` gives this string a greater significance than just the data type `string`.

Examples of such globally significant data are numerous and include social security numbers, tax payer numbers, flight numbers, license plate numbers, etc. These data can be denoted as properties whose significance goes beyond a single business process and can therefore be used for correlation. Other properties will be data significant for a single business process only, such as uniform identifiers, employee numbers, etc.

> Message properties have global significance in business processes and are mapped to multiple messages. So, it makes sense to name them with global property names.

Message properties are defined in WSDL through the WSDL extensibility mechanism, similarly to partner link types. However, in contrast to partner link types, the standard BPEL namespace is used: http://schemas.xmlsoap.org/ws/2003/03/business-process/. The syntax is simple and shown below. We have to define a property name and its type:

```
<wsdl:definitions

xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"

... >
```

```
...
   <bpws:property name="name" type="type-name" />

...
</wsdl:definitions>
```

Let's go back to our travel process example. The flight number is such a significant data element that it makes sense to define it as a property in the Airline web service WSDL:

```
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:emp="http://packtpub.com/service/employee/"
xmlns:tns="http://packtpub.com/service/airline/"
targetNamespace="http://packtpub.com/service/airline/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/" >
...
   <bpws:property name="FlightNo" type="xs:string" />

...
</definitions>
```

Mapping Properties to Messages

Properties are parts of messages, usually embedded in the application-specific part of messages. To map a property to a specific element (or even attribute) of the message, BPEL provides **property aliases**. With property aliases, we map a property to a specific element or attribute of the selected message part. We can then use the property name as an alias for the message part and the location. This is particularly useful in abstract business processes where we focus on message exchange description.

Property aliases are defined in WSDL. The syntax is shown below. We have to specify the property name, the message type, message part, and the query expression to point to the specific element or attribute. The query expression is written in the selected query language; the default is XPath 1.0:

```
<wsdl:definitions ...
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
... >
...
   <bpws:propertyAlias propertyName="property-name"
                       messageType="message-type-name"
                       part="message-part-name"
                       query="query-string"/>

...
```

```
</wsdl:definitions>
```

We now define the property alias for the flight number property defined in the previous section. In our travel process example we have defined the `TravelResponseMessage` in the airline WSDL:

```
...
<message name="TravelResponseMessage">
<part name="confirmationData" type="tns:FlightConfirmationType" />
</message>
...
```

The `FlightConfirmationType` has been defined as a complex type with the `FlightNo` element of type `xs:string` being one of the elements. For the complete WSDL with the type definition please look at Chapter 3. To define the alias we write the following code:

```
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:emp="http://packtpub.com/service/employee/"
xmlns:tns="http://packtpub.com/service/airline/"
targetNamespace="http://packtpub.com/service/airline/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/" >
...
<bpws:property name="FlightNo" type="xs:string" />
...
   <bpws:propertyAlias propertyName="tns:FlightNo"
                       messageType="tns:TravelResponseMessage"
                       part="confirmationData"
                       query="/confirmationData/FlightNo"/>

...
</definitions>
```

With this, we have defined a global property `FlightNo` as an alias for the `confirmationData` part of the `FlightConfirmationType` message type on the location specified by the `query`.

Extracting Properties

To extract property values from variables, BPEL defines an extension function called `getVariableProperty`, which is defined in the standard BPEL namespace. The function takes two parameters, the variable name and the property name, and returns the node that represents the property. The syntax is shown below:

```
bpws:getVariableProperty ('variableName', 'propertyName')
```

To extract the FlightNo property from the TravelResponse variable we write the following:

```
bpws:getVariableProperty ('TravelResponse', 'FlightNo')
```

The use of properties increases flexibility in extracting relevant data from the message compared to the getVariableData function. Using properties, we do not have to specify the exact location of the data (such as flight number), but rather use the property name. If the location changes, we only have to modify the property definition.

Properties and Assignments

Properties can also be used in assignments, which is particularly useful in abstract processes. We can copy a property from one variable to another using the <assign> activity, as shown in the code excerpt overleaf:

```
<assign>
<copy>
<from variable="variable-name" property="property-name"/>
<to variable="variable-name" property="property-name"/>
</copy>
</assign>
```

To copy the FlightNo property from the FlightResponseAA variable to the TravelResponse variable we write the following:

```
<assign>
<copy>
<from variable="FlightResponseAA" property="FlightNo"/>
<to variable="TravelResponse" property="FlightNo"/>
</copy>
</assign>
```

Correlation Sets

Now that we are familiar with properties, let's go back to the problem of correlation of messages. Correlation in BPEL uses the notion of properties to assign global names to relevant data used for correlation messages (such as flight number) and to define aliases through which we specify the location of such data in messages.

> A set of properties shared by messages and used for correlation is called a **correlation set**.

When correlated messages are exchanged between business partners, two roles can be defined. The partner that sends the first message in an operation invocation is the **initiator** and defines the values of the properties in the correlation set. Other partners are **followers** and get the property values for their correlation sets from incoming messages. Both initiator and followers must mark the first activity that binds the correlation sets.

A correlation set is used to associate messages with business process instances. Each correlation set has a name. A message can be related to one or more correlation sets. The initial message is used to initialize the values of a correlation set. The subsequent messages related to this correlation set must have property values identical to the initial correlation set. Correlation sets in BPEL can be declared globally for the whole process or within scopes. The syntax is shown below:

```
<correlationSets>
<correlationSet name="correlation-set-name"
properties="list-of-properties"/>
<correlationSet name="correlation-set-name"
```

properties="*list-of-properties*"/>

...

</correlationSets>

An example of a correlation set definition named `VehicleOrder` that includes two properties `chassisNo` and `engineNo` is shown below:

<correlationSets>

<correlationSet name="VehicleOrder"

properties="tns:chassisNo tns:engineNo"/>

</correlationSets>

Going back to our example, let's define a correlation set named `TicketOrder` with a single property, `FlightNo`:

```
<process ... >


<partnerLinks>...</partnerLinks>

<variables>...</variables>

   <correlationSets>
      <correlationSet name="TicketOrder"
                      properties="aln:FlightNo"/>
   </correlationSets>    ...
```

Using Correlation Sets

We can use correlation sets in `<invoke>`, `<receive>`, `<reply>`, and the `<onMessage>` parts of `<pick>` activities or event handlers. To specify which correlation sets should be used, we use the `<correlation>` activity nested within any of the above-mentioned activities. The syntax is shown below:

<correlations>

<correlation set="*name*"

initiate="yes|no" <!-- Optional -->

pattern="in|out|out-in" /> <!-- Used in invoke -->

</correlations>

We must specify the name of the correlation set used and indicate whether the correlation set should be initiated. The default value of the `initiate` attribute is `no`. When we use the correlation with the `<invoke>` activity, we must also specify the `pattern` attribute. The `in` value specifies that the correlation applies to inbound messages, `out` to outbound, and `out-in` to both messages.

The following example shows how to use correlation sets in a scenario where the BPEL process first checks the flight availability using an asynchronous `<invoke>` and then waits for the callback. The callback message contains the flight number (`FlightNo`), and is used to initiate the correlation set. Next, the ticket is confirmed using a synchronous `<invoke>`. Here the correlation set is used with the `out-in` pattern. Finally, the result is sent to the BPEL process client using a callback `<invoke>` activity. Here the correlation set is used with the `out` pattern:

...

<sequence>

...

<!-- Check the flight avaliablity -->

```
<invoke partnerLink="AmericanAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" />
<!-- Wait for the callback -->
<receive partnerLink="AmericanAirlines"
portType="aln:FlightCallbackPT"
operation="FlightTicketCallback"
variable="TravelResponse" >
          <!-- The callback includes flight no
               therefore initiate correlation set -->
          <correlations>
             <correlation set="TicketOrder"
                          initiate="yes" />
          </correlations>

</receive>
...
<!-- Synchrnousy confirm the ticket -->
<invoke partnerLink="AmericanAirlines"
portType="aln:TicketConfirmationPT"
operation="ConfirmTicket"
inputVariable="FlightRespnseAA"
outputVariable="Confirmation" >
          <!-- Use the correlation set to confirm the ticket -->
          <correlations>
             <correlation set="TicketOrder"
                          pattern="out-in" />
          </correlations>

</invoke>
...
<!-- Make a callback to the client -->
<invoke partnerLink="client"
portType="trv:ClientCallbackPT"
operation="ClientCallback"
inputVariable="TravelResponse" >
          <!-- Use the correlation set to callback the client -->
          <correlations>
             <correlation set="TicketOrder"
                          pattern="out" />
          </correlations>

</invoke>
</sequence>
```

```
</process>
```

User name:
Book: Business Process Execution Language for Web Services

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.

Concurrent Activities and Links

In business processes activities often occur concurrently. In BPEL, such concurrent activities are modeled using the `<flow>` activity. Activities within `<flow>` start concurrently as soon as the `<flow>` is started. The `<flow>` completes when all nested activities complete. Gathering nested activities within `<flow>` is straightforward and very useful for expressing concurrency scenarios that are not too complicated. We have used it in the examples in this and the previous chapter.

To express more complex concurrency scenarios, `<flow>` provides the ability to express synchronization dependencies between activities. In other words, we can specify which activities can start and when (depending on other activities) and define dependencies that are more complex than those expressed with a combination of `<flow>` and `<sequence>` activities. For example, we will often specify that a certain activity or several activities cannot start before another activity or several activities have finished.

We express synchronization dependencies using the `<link>` construct. For each link we specify a name. Links have to be defined within the `<flow>` activity. Link definitions are gathered within a `<links>` element. This is shown in the code excerpt below:

```
<flow>

<links>

<link name="TravelStatusToTicketRequest" />

<link name="TicketRequestToTicketConfirmation" />

</links>

...

</flow>
```

These links can now be used to link activities together. For actual linking we use standard elements that can be used with any BPEL activity.

Sources and Targets

For each BPEL activity, whether basic or structured, we can specify two **standard elements** for linking activities and expressing synchronization dependencies. These two standard elements are nested within the activity:

- `<source>` is used to annotate an activity as being a source of one or more links.
- `<target>` is used to annotate an activity as being a target of one or more links.

Every link declared within `<flow>` must have exactly one activity within the flow as its `<source>`. It must also have exactly one activity within the flow as its `<target>`.

> A link's target activity can be performed only after the source activity has been finished.

The syntax of the `<source>` element is shown below. We have to specify the link name, which has to be defined within the `<flow>` activity. Optionally we can specify the transition condition. We will say more on transition conditions later in this section. If the transition condition is not specified, the default value is `true`.

```
<source linkName="name"
        transitionCondition="boolean-expression" />
```

The syntax of the <target> element is even simpler. We only have to specify the link name:

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

Example

Let's now consider the business travel example. There the process had to invoke the Employee Travel Status web service first (synchronous invocation) to get the employee travel class information. Then it asynchronously invoked the American and Delta Airlines' web services to get flight ticket information. Finally, the process selected the best offer and sent the callback to the BPEL client.

In Chapter 3 we used a combination of <sequence> and <flow> activities to control the execution order. These two activities allowed us to perform basic synchronization, but they are not appropriate for expressing complex synchronization scenarios. In such scenarios, we should use links.

To demonstrate how to use links let's use the business travel example, but keep in mind that the scenario of our example is simple enough to be expressed using a combination of <flow> and <sequence> activities without the need for links. We will use the example for simplicity reasons. In the real world, we use links only where the scenario is so complex that it cannot be expressed using a combination of <flow> and <sequence> activities.

We have modified the asynchronous travel example and gathered all activities except the initial <receive> and the final <invoke> within a single <flow> activity. We have also added the name attribute to each activity. Although this attribute is optional, we have added it because it simplifies understanding which activities have to be linked:

```
<process name="Travel"
... >
<partnerLinks>
...
</partnerLinks>
<variables>
...
</variables>
<sequence>
<!-- Receive the initial request for business travel from client -->
<receive name="InitialRequestReceive"
partnerLink="client"
portType="trv:TravelApprovalPT"
operation="TravelApproval"
variable="TravelRequest"
createInstance="yes" />
      <flow>

<!-- Prepare the input for the Employee Travel Status Web Service -->
<assign name="EmployeeInput">
<copy>
<from variable="TravelRequest" part="employee"/>
```

```
<to variable="EmployeeTravelStatusRequest" part="employee"/>
</copy>
</assign>
<!-- Synchronously invoke the Employee Travel Status Web Service -->
<invoke name="EmployeeTravelStatusSyncInv"
partnerLink="employeeTravelStatus"
portType="emp:EmployeeTravelStatusPT"
operation="EmployeeTravelStatus"
inputVariable="EmployeeTravelStatusRequest"
outputVariable="EmployeeTravelStatusResponse" />
<!-- Prepare the input for AA and DA -->
<assign name="AirlinesInput">
<copy>
<from variable="TravelRequest" part="flightData"/>
<to variable="FlightDetails" part="flightData"/>
</copy>
<copy>
<from variable="EmployeeTravelStatusResponse" part="travelClass"/>
<to variable="FlightDetails" part="travelClass"/>
</copy>
</assign>
<!-- Async invoke of the AA web service and wait for the callback -->
<invoke name="AmericanAirlinesAsyncInv"
partnerLink="AmericanAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" />
<receive name="AmericanAirlinesCallback"
partnerLink="AmericanAirlines"
portType="aln:FlightCallbackPT"
operation="FlightTicketCallback"
variable="FlightResponseAA" />
<!-- Async invoke of the DA web service and wait for the callback -->
<invoke name="DeltaAirlinesAsyncInv"
partnerLink="DeltaAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" />
<receive name="DeltaAirlinesCallback"
```

```
partnerLink="DeltaAirlines"
portType="aln:FlightCallbackPT"
operation="FlightTicketCallback"
variable="FlightResponseDA" />
<!-- Select the best offer and construct the TravelResponse -->
<switch name="BestOfferSelect">
<case condition="bpws:getVariableData('FlightResponseAA', 'confirmationData','/confirmationData/aln:Price')
&lt;= bpws:getVariableData('FlightResponseDA', 'confirmationData','/confirmationData/aln:Price')">
<!-- Select American Airlines -->
<assign>
<copy>
<from variable="FlightResponseAA" />
<to variable="TravelResponse" />
</copy>
</assign>
</case>
<otherwise>
<!-- Select Delta Airlines -->
<assign>
<copy>
<from variable="FlightResponseDA" />
<to variable="TravelResponse" />
</copy>
</assign>
</otherwise>
</switch>
      </flow>

<!-- Make a callback to the client -->
<invoke name="ClientCallback"
partnerLink="client"
portType="trv:ClientCallbackPT"
operation="ClientCallback"
inputVariable="TravelResponse" />
</sequence>
</process>
```

Note that all activities gathered within <flow> will start concurrently, which is not what we want. We therefore use links to express dependencies. First we identify the dependencies:

- The input for the Employee web service (`EmployeeInput`) has to be prepared before the Employee web service can be invoked (`EmployeeTravelStatusSyncInv`).
- The invocation (`EmployeeTravelStatusSyncInv`) of the Employee web service has to be finished before the input for both airlines' web services can be prepared (`AirlinesInput`).
- The input for both airlines' web services has to be prepared (`AirlinesInput`) before the process can invoke the web services of both airlines (`AmericanAirlinesAsyncInv` and `DeltaAirlinesAsyncInv`).
- The invocation of the American Airlines web service (`AmericanAirlinesAsyncInv`) has to be finished before the callback can be received (`AmericanAirlinesCallback`).
- The invocation of the Delta Airlines web service (`DeltaAirlinesAsyncInv`) has to be finished before the callback can be received (`DeltaAirlinesCallback`).
- Both callbacks (from American and Delta Airlines: `AmericanAirlinesCallback` and `DeltaAirlinesCallback`) have to be received before the best offer can be selected (`BestOfferSelect`).

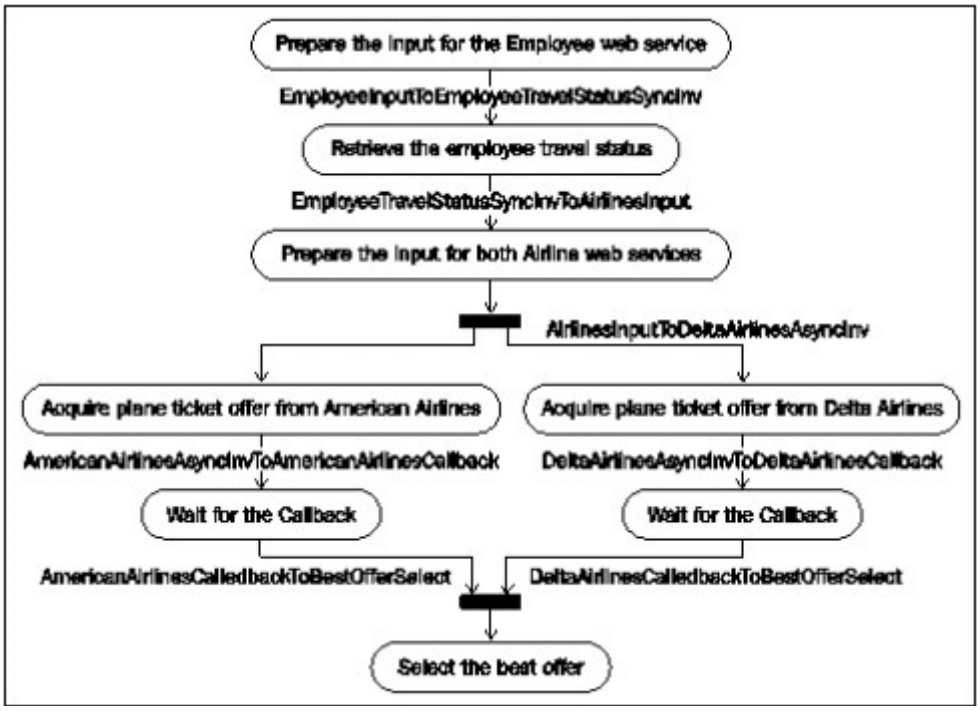
Let us now name the links. We will need the following eight links:

- The link from the `EmployeeInput` to `EmployeeTravelStatusSyncInv`
- The link from the `EmployeeTravelStatusSyncInv` to the `AirlinesInput` preparation
- Two links form the `AirlinesInput` preparation to `AmericanAirlinesAsyncInv` and `DeltaAirlinesAsyncInv`
- The link from `AmericanAirlinesAsyncInv` to the receive callback `AmericanAirlinesCallback`
- The link from `DeltaAirlinesAsyncInv` to the receive callback `DeltaAirlinesCallback`
- The link from `AmericanAirlinesCallback` to `BestOfferSelect`
- The link from `DeltaAirlinesCallback` to `BestOfferSelect`

We have to define the links within the `<flow>` activity, as shown in the code excerpt below:

```
<flow>
<links>
<link name="EmployeeInputToEmployeeTravelStatusSyncInv" />
<link name="EmployeeTravelStatusSyncInvToAirlinesInput" />
<link name="AirlinesInputToAmericanAirlinesAsyncInv" />
<link name="AirlinesInputToDeltaAirlinesAsyncInv" />
<link name="AmericanAirlinesAsyncInvToAmericanAirlinesCallback" />
<link name="DeltaAirlinesAsyncInvToDeltaAirlinesCallback" />
<link name="AmericanAirlinesCallbackToBestOfferSelect" />
<link name="DeltaAirlinesCallbackToBestOfferSelect" />
</links>
      ...
```

The dependency of links and activities is shown in the following activity diagram:



Let us now add the `<source>` and `<target>` elements to the BPEL process activities:

```
...
<!-- Prepare the input for the Employee Travel Status Web Service -->
<assign name="EmployeeInput">
          <source linkName="EmployeeInputToEmployeeTravelStatusSyncInv" />

<copy>
<from variable="TravelRequest" part="employee"/>
<to variable="EmployeeTravelStatusRequest" part="employee"/>
</copy>
</assign>
<!-- Synchronously invoke the Employee Travel Status Web Service -->
<invoke name="EmployeeTravelStatusSyncInv"
partnerLink="employeeTravelStatus"
portType="emp:EmployeeTravelStatusPT"
operation="EmployeeTravelStatus"
inputVariable="EmployeeTravelStatusRequest"
outputVariable="EmployeeTravelStatusResponse" >
          <target linkName="EmployeeInputToEmployeeTravelStatusSyncInv" />
          <source linkName="EmployeeTravelStatusSyncInvToAirlinesInput" />

</invoke>
<!-- Prepare the input for AA and DA -->
```

```
<assign name="AirlinesInput">
          <target linkName="EmployeeTravelStatusSyncInvToAirlinesInput" />
          <source linkName="AirlinesInputToAmericanAirlinesAsyncInv" />
          <source linkName="AirlinesInputToDeltaAirlinesAsyncInv" />


<copy>
<from variable="TravelRequest" part="flightData"/>
<to variable="FlightDetails" part="flightData"/>
</copy>
<copy>
<from variable="EmployeeTravelStatusResponse" part="travelClass"/>
<to variable="FlightDetails" part="travelClass"/>
</copy>
</assign>
<!-- Async invoke of the AA web service and wait for the callback -->
<invoke name="AmericanAirlinesAsyncInv"
partnerLink="AmericanAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" >
          <target linkName="AirlinesInputToAmericanAirlinesAsyncInv" />
          <source
              linkName="AmericanAirlinesAsyncInvToAmericanAirlinesCallback" />


</invoke>
<receive name="AmericanAirlinesCallback"
partnerLink="AmericanAirlines"
portType="aln:FlightCallbackPT"
operation="FlightTicketCallback"
variable="FlightResponseAA" >
          <target
              linkName="AmericanAirlinesAsyncInvToAmericanAirlinesCallback" />
          <source linkName="AmericanAirlinesCallbackToBestOfferSelect" />


</receive>
<!-- Async invoke of the DA web service and wait for the callback -->
<invoke name="DeltaAirlinesAsyncInv"
partnerLink="DeltaAirlines"
portType="aln:FlightAvailabilityPT"
operation="FlightAvailability"
inputVariable="FlightDetails" >
```

```
          <target linkName="AirlinesInputToDeltaAirlinesAsyncInv" />
          <source linkName="DeltaAirlinesAsyncInvToDeltaAirlinesCallback" />


</invoke>

<receive name="DeltaAirlinesCallback"

partnerLink="DeltaAirlines"

portType="aln:FlightCallbackPT"

operation="FlightTicketCallback"

variable="FlightResponseDA" >

          <target linkName="DeltaAirlinesAsyncInvToDeltaAirlinesCallback" />
          <source linkName="DeltaAirlinesCallbackToBestOfferSelect" />


</receive>

<!-- Select the best offer and construct the TravelResponse -->

<switch name="BestOfferSelect">

          <target linkName="AmericanAirlinesCallbackToBestOfferSelect" />
          <target linkName="DeltaAirlinesCallbackToBestOfferSelect" />


<case condition="bpws:getVariableData('FlightResponseAA', 'confirmationData','/confirmationData/Price')

&lt;= bpws:getVariableData('FlightResponseDA', 'confirmationData','/confirmationData/Price')">

<!-- Select American Airlines -->

<assign>

<copy>

<from variable="FlightResponseAA" />

<to variable="TravelResponse" />

</copy>

</assign>

</case>

<otherwise>

<!-- Select Delta Airlines -->

<assign>

<copy>

<from variable="FlightResponseDA" />

<to variable="TravelResponse" />

</copy>

</assign>

</otherwise>

</switch>

      </flow>
```

With this we have defined synchronization dependencies between activities. Note that according to the BPEL specification, every link within the <flow> activity must have exactly one activity within the flow as its source and exactly one activity within the flow as its target. This prevents us from using the same link as the source or target of two activities.

Transition Conditions

A <source> element specifies that a certain activity defines an outgoing link. When BPEL processes are executed, outgoing links are evaluated after the activity has finished. Each outgoing link can have a positive or negative status. This status is important when the decision is made to start the linked activity (denoted with <target>).

In our example, the AmericanAirlinesCallback <receive> activity defines an outgoing link AmericanAirlinesCallbackToBestOfferSelect. This link is the incoming link of the BestOfferSelect <switch> activity. The BestOfferSelect <switch> activity has another incoming link, DeltaAirlinesCallbackToBestOfferSelect, which is the outgoing link of the DeltaAirlinesCallback <receive> activity.

After the AmericanAirlinesCallback <receive> activity has finished, the outgoing AmericanAirlinesCallbackToBestOfferSelect link is evaluated. More precisely, the transitionCondition attribute of the outgoing link is evaluated. If the transitionCondition is evaluated to true, the link status is positive. Otherwise it is negative.

We have already mentioned that the <source> element has an optional attribute called transitionCondition. We have also mentioned that if the attribute is omitted, a default value of true is used. In our previous example, therefore, the outgoing link status was always true.

Let's now modify the example and explicitly add the transition condition. The outgoing link will be positive only if the flight ticket is approved. This is signaled using the Approved element of the FlightConfirmationType complex type, which is the confirmationData part of the TravelResponseMessage message, used for the FlightResponseAA and FlightResponseDA variables (see the previous chapter for corresponding WSDL definitions).

We will use the getVariableData function and extract the Approved element from the confirmationData part of the message stored in the FlightResponseAA variable. The code is shown below:

...

<!-- Receive the callback -->

```
        <receive name="AmericanAirlinesCallback"
                 partnerLink="AmericanAirlines"
                 portType="aln:FlightCallbackPT"
                 operation="FlightTicketCallback"
                 variable="FlightResponseAA" >

          <target
              linkName="AmericanAirlinesAsyncInvToAmericanAirlinesCallback" />
          <source linkName="AmericanAirlinesCallbackToBestOfferSelect"
                  transitionCondition="bpws:getVariableData(
                                'FlightResponseAA',
                                'confirmationData',
                                '/confirmationData/aln:Approved')='true'" />


</receive>

        ...
```

We will do the same for the DeltaAirlinesCallback <receive> activity:

...

<!-- Receive the callback -->

```
        <receive name="DeltaAirlinesCallback"
                 partnerLink="DeltaAirlines"
                 portType="aln:FlightCallbackPT"
                 operation="FlightTicketCallback"
                 variable="FlightResponseDA" >
```

```
        <target linkName="DeltaAirlinesAsyncInvToDeltaAirlinesCallback" />
        <source linkName="DeltaAirlinesCallbackToBestOfferSelect"
                transitionCondition="bpws:getVariableData(
                                'FlightResponseDA',
                                'confirmationData',
                                '/confirmationData/aln:Approved')='true'" />


</receive>

        ...
```

Both outgoing links are now evaluated using the transition conditions and statuses can be determined.

Join Conditions and Link Status

The `AmericanAirlinesCallbackToBestOfferSelect` and the `DeltaAirlinesCallbackToBestOfferSelect` are the incoming links for the `BestOfferSelect <switch>` activity. In order to start the `BestOfferSelect` activity:

- The status of both incoming links has to be determined. As we already know, the status is determined using the `transitionCondition` expression.
- The join condition for the `BestOfferSelect` activity has to be evaluated.

The join condition is specified using the standard attribute called `joinCondition`. This attribute may be specified for each activity that is the target of a link (has at least one incoming link). If no `joinCondition` is specified, the default (for the default expression language XPath 1.0) is the logical disjunction (logical `or`) of the link status of all incoming links of this activity. In other words, if the `joinCondition` is not explicitly defined, all incoming link statuses are evaluated and the status of at least one incoming link has to be positive. The consequence of evaluating all incoming link statuses is the synchronization of all incoming activities.

In our example, the default (implicit) join condition for the `BestOfferSelect` is therefore a disjunction of both incoming link statuses, the `AmericanAirlinesCallbackToBestOfferSelect` and the `DeltaAirlinesCallbackToBestOfferSelect`. The join condition will be evaluated to `true` if at least one of the airlines has approved the flight tickets. Please notice that the incoming link statuses of both links will be evaluated prior the decision.

Sometimes the default disjunction will not fit our needs and we will want to define our own join condition. To do this we will use the `joinCondition` attribute. We have to specify this attribute for the target link activity. In our example we would define the `joinCondition` for the `BestOfferSelect <switch>` activity.

For the `joinCondition` we can specify any valid Boolean expression using the selected expression language (the default is XPath 1.0). Often we will also want to check the status of the incoming links. For these purposes BPEL provides a special function called `getLinkStatus`. This function is defined in the standard BPEL namespace `http://schemas.xmlsoap.org/ws/2003/03/business-process/`. The syntax is straightforward as we only have to provide the name of the incoming link as the parameter. The function returns `true` if the status of the link is positive and `false` if the status of the link is negative. This function can be used only in join conditions:

```
getLinkStatus ( 'link-name' )
```

Suppose instead of the disjunction of link statuses we would rather use a conjunction. Then we would define the following `joinCondition`:

```
...

<!-- Select the best offer and construct the TravelResponse -->

<switch name="BestOfferSelect"

                joinCondition="bpws:getLinkStatus(
                                'AmericanAirlinesCallbackToBestOfferSelect')
                                and
                                bpws:getLinkStatus(
                                'DeltaAirlinesCallbackToBestOfferSelect')" >        ...
```

Join Failures

Join conditions are evaluated before the activity is started. In our example the join condition would be evaluated to `true` only if both link statuses (`AmericanAirlinesCallbackToBestOfferSelect` and `DeltaAirlinesCallbackToBestOfferSelect`) are positive. Positive join condition is required for starting the activity.

If a join condition evaluates to `false`, a standard `joinFailure` fault is thrown. A `joinFailure` can be thrown even if a join condition is not explicitly specified. In our previous example (before explicitly specifying the join condition) the default join condition would be used and would be

evaluated to `false` if both link statuses were negative. This would be the case if neither American nor Delta Airlines would approve the flight ticket.

Suppressing Join Failures

Sometimes it would be more useful if instead of throwing a `joinFailure` fault the activity would simply not be performed without any fault thrown. BPEL provides an attribute through which we can express this behavior. The attribute is called `suppressJoinFailure` and is a *standard attribute* that can be associated with each activity (basic or structured). The value of the attribute can be either `yes` or `no`. The default is `no`.

In our example we could suppress join failure for the `BestOfferSelect` `<switch>` activity as shown below:

...

<!-- Select the best offer and construct the TravelResponse -->

<switch name="BestOfferSelect"

joinCondition="bpws:getLinkStatus('AmericanAirlinesCallbackToBestOfferSelect')

and

bpws:getLinkStatus('DeltaAirlinesCallbackToBestOfferSelect')"

suppressJoinFailure="yes" > ...

This means that if even one link status is negative, the activity will not be performed and no fault will be thrown—in other words the activity would be silently skipped. Skipping the activity is equivalent to catching the fault locally with an `<empty>` fault handler.

The consequence of skipping an activity is that outgoing links become negative. This way the next activity figures out that the previous activity has been skipped. In our example, the `BestOfferSelect` activity does not have outgoing links.

The default value of the `suppressJoinFailure` attribute is `no`. This is because in simple scenarios without complex graphs such behavior is preferred. In simple scenarios, links without transition conditions are often used. Here the developers often do not think about join conditions. Suppressing join failures would lead to unexpected behavior where activities would be skipped.

In complex scenarios with networks of links, the suppression of join failures can be desirable. If such behavior is desirable for the whole BPEL process, we can set the `suppressJoinFailure` attribute to `yes` in the first process element (often a `<sequence>`). Skipping activities with join conditions evaluated to false and setting the outgoing link statuses to negative is called **dead-path-elimination**. The reason is that in complex networks of links with transition conditions such behavior results in propagating the negative link status along entire paths until a join condition is reached that evaluates to `true`.

With this we have concluded our discussion on concurrent activities, links, and transition conditions. In the next section, we discuss dynamic partner links.

User name:
Book: Business Process Execution Language for Web Services

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.

Dynamic Partner Links

So far we have discussed BPEL processes where all partner links have been defined at the design time and related to actual web services. We have used a single partner link for each web service we have communicated with.

In an advanced BPEL process we might want to define the partner link endpoint references at run time. This means that the BPEL process will dynamically determine which actual web service it will use for a certain invocation, based on the variable content. This is particularly useful in scenarios where the BPEL process communicates with several web services that have the same WSDL interface. This has been the case for our travel process example where American and Delta Airline web services shared the same interface.

To understand how we can define partner link endpoint references dynamically at run time, let us look at how endpoint references are represented in BPEL. BPEL uses endpoint references as defined by the WS-Addressing. For each BPEL process instance and for each partner role in a partner link a unique endpoint reference is assigned. We already know that this assignment can take place at deployment or at run time. To make such an assignment at run time we use the `<assign>` activity. There are several ways in which we can use this. We can copy from one partner link to another using the following syntax:

```
<assign>

<copy>

<from partnerLink="name" endpointReference="myRole|partnerRole"/>

<to partnerLink="name"/>

</copy>

</assign>
```

In the `<from>` activity we have to specify the endpoint role `myRole` or `partnerRole`, while in the `<to>` activity we always copy to the `partnerRole`. We can also copy a partner link to a variable:

```
<assign>

<copy>

<from partnerLink="name" endpointReference="myRole|partnerRole"/>

<to variable="varName"/>

</copy>

</assign>
```

The most interesting, however, is to copy a variable, expression, or XML literal to a partner link. This way we can store the partner link endpoint reference in a variable and copy it to the partner link at run time, thus selecting the service, which will be invoked dynamically. The syntax for copying a variable to partner link is shown below:

```
<assign>

<copy>

<from variable="varNname"/>

<to partnerLink="name"/>
```

```
</copy>
</assign>
```

The partner link endpoint reference in BPEL is represented as the `wsa:EndpointReference` XML element defined by the WS-Addressing. The `wsa` namespace URL is `http://schemas.xmlsoap.org/ws/2003/03/addressing`. The `wsa:EndpointReference` element is of type `wsa:EndpointReferenceType` and has the following structure:

```
<EndpointReference xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
<Address>ServiceURL</Address>
<ReferenceProperties>...</ReferenceProperties> <!-- optional -->
<PortType>PortTypeName</PortType> <!-- optional -->
<ServiceName PortName="...">ServiceName</ServiceName> <!-- optional -->
</EndpointReference>
```

We can see that the endpoint reference `<Address>` is the only required element. The `<Address>` should include a valid URL of the partner link service.

To dynamically assign an endpoint reference to a partner link we have to declare a variable of element type `wsa:EndpointReference` and copy it to the partner link. Alternatively we can hard-code the address into the BPEL process and copy the XML literal to the partner link. This is shown in the following example. It is assumed that a service is available on the specified URL:

```
<assign>
<copy>
<from>
<EndpointReference xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
<Address>
http://localhost:9700/orabpel/default/AmericanAirline
</Address>
</EndpointReference>
</from>
<to partnerLink="Airline"/>
</copy>
</assign>
```

With this we have concluded the discussion on dynamic partner links. Please refer to Chapter 6 for a working demo. In the next section, we discuss abstract business processes.

User name:
Book: Business Process Execution Language for Web Services

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.

Abstract Business Processes

Although the BPEL name suggests that this is a language for specifying executable business processes, BPEL supports both executable business processes and abstract business processes. Abstract business processes specify public message exchange between parties only.

The objective of abstract business processes in BPEL is to specify only the externally observable aspects of process behavior (often also called public process behavior) without the exact details of how the process executes. Abstract processes are not executable. An abstract business process should provide a complete description of external behavior relevant to a partner or several partners it interacts with.

The description of the externally observable behavior of a business process may be related to a single web service or a set of web services. It might also describe the behavior of a participant in a business process. In the later case the abstract processes of all partners must be coupled together, usually using a separate global protocol structure description.

We will define abstract processes mainly for two scenarios. First, with an abstract process we can describe the behavior of a web service even though we do not know exactly in which business process it will take part. In this scenario we will use partner links with `myRole` attributes only. With such an abstract process we can provide a web service behavioral description that does not place any requirements on the partners except that they respect the behavior of the web service.

Second, we can use an abstract process to define collaboration protocols among multiple parties and precisely describe the external behavior of each party. Such abstract processes will usually be defined by large enterprises to define protocols for their partners, or by vertical standards organizations such as RosettaNet, to define business protocols for their domains.

Because abstract processes are not executable, the question is: What are they useful for? The most common scenario is to use abstract processes as a template to define executable processes. Abstract processes can be used to replace sets of rules usually expressed in natural language, which is often ambiguous. This reduces misunderstandings and errors. We also expect tools to generate abstract processes for partner web services based on underlying executable processes.

Abstract processes must specify the `abstractProcess` attribute of the `<process>` tag. This attribute should have the value `yes`:

```
<process name="AbstractBusinessTravelProcess"
abstractProcess="yes" ... >
...
</process>
```

Because abstract processes do not specify the exact process implementation (and are thus not executable), they differ from executable processes in several syntactical details. On one hand, they are not allowed to use certain BPEL constructs. On the other hand, some BPEL constructs can only be used in abstract processes. We will list the most important differences, starting with the functionality not allowed in abstract processes:

- The function `getVariableData` cannot be used.
- The assignment activity has a special variant that cannot be used in abstract processes. This is the variant where we specify the variable, part, and query expression.
- There is no checking for conflicting receives in abstract processes.
- The `<terminate>` activity cannot be used.

Important functionality allowed only in abstract processes is:

- The `inputVariable` and `outputVariable` attributes for the `<invoke>` activity and the `input` attribute for the `<receive>` and `<reply>` activities are optional.
- The `inputVariable` attribute of the `<onMessage>` activity is optional.
- Variables do not have to be initialized before they are used.
- In abstract processes the type checking is not strictly enforced.
- Property aliases can be used for addressing message parts and locations.
- Abstract processes allow opaque values to be assigned to variables based on non-deterministic choice.

Abstract processes follow the choreography approach of web services composition. However, at the time of writing this book, it seemed that in the majority of cases, BPEL will be used for executable processes.

User name:
Book: Business Process Execution Language for Web Services

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.

Model Driven Approach: Generating BPEL from UML Activity Diagrams

Through the chapter we have seen that BPEL is a high-level language for specifying business processes. Because business processes are at such a high level, it is reasonable to think about defining process models using a modeling language and transforming these models to the BPEL code automatically. This is similar to the Model Driven Architecture (MDA) developed by the Object Management Group (OMG). The main objective of MDA is to raise the level of abstraction of development. Business processes are a perfect candidate for this approach.

The MDA approach is not to provide a simple graphical notation and a tool that enables developers to graphically build BPEL processes. The MDA approach is really about:

- Defining a platform-independent model (PIM) of the business process
- Defining exact rules that allow automatic mapping of the PIM to platform-specific models (PSM); these mappings can be done by a tool

In other words, this means that an independent business protocol model could be automatically mapped to several business process specification languages, BPEL being one of them. Such an approach would raise the abstraction level even further and make business process modeling independent of the underlying execution language, thus stimulating companies to invest more in business process modeling.

The fact that the described approach is not a dream has been demonstrated by researchers at IBM/Rational. They have based business process model development on the Unified Modeling Language (UML) and have defined UML extensions (a UML profile) for automated business processes. Based on the profile they have also developed a tool that generates BPEL process definitions and the corresponding WSDL descriptions based on the UML activity diagrams. This tool and the UML profile are part of the IBM Emerging Technologies Toolkit version (ETTK), which can be downloaded from alphaWorks: http://www.alphaworks.ibm.com/tech/ettk.

User name:
Book: Business Process Execution Language for Web Services

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.

Conclusion

We have seen that BPEL is an efficient language for describing business processes. It provides support for the complexities of real-world business process implementations but is still relatively easy to learn and use. In this chapter we have become familiar with the advanced concepts of BPEL, such as loops, process termination, delays, and deadline and duration expressions. We have addressed fault handling, which is a very important aspect of each business process. Particularly in BPEL processes, which use loosely coupled web services for partner operations, faults can occur quite often. We have discussed scopes, which enable us to break the process into several parts. Each part or scope can have its own variables, correlation sets, fault handlers, compensation handlers, and event handlers. In addition scopes can provide concurrency control through serialization.

Another very important aspect of business processes is compensation. In business processes consistency has to be preserved even if a process is abandoned. Because business processes are often long running and span several partners the usage of ACID transactions is not reasonable. BPEL therefore supports the concept of compensation. The goal of compensation is to reverse the effects of previous activities that have been carried out as part of a business process that is being abandoned. We have become familiar with compensation handlers and how to invoke them. Next we have discussed events and have seen that a business process has to react on message events, which happen when an operation is invoked on the process, and on alarm events, which can occur at a specific time or after certain duration.

We have also addressed complex business processes with many concurrent activities and have seen that BPEL provides links, which enable concurrency control and synchronization using source and target links. Then we have discussed transition and join conditions, and link statuses. We have seen why and when join failures are thrown and how to eliminate dead paths using join failure suppression.

We have discussed the business process lifecycle and process instances and have focused on correlation of messages, another important aspect of BPEL processes. Correlation uses correlation sets to associate messages with business process instances, and is related to message properties. Message properties have global significance in business processes and are mapped to multiple messages. We have become familiar with dynamic partner links. Finally we have discussed abstract business protocols and mentioned the model-driven approach to BPEL process definition. With this we have covered all the advanced aspects of BPEL. The coming chapters discuss important BPEL servers.