

User name:

Book: Essential Business Process Modeling

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.

Part II: Standards

A detailed survey of current BPM standard models and specifications, including: BPEL; the BPMI standards (BPMN and BPEL) and the BPMI reference architecture; the WfMC standards (XPDL, WFXML, WAPI) and the WfMC reference model; web services choreography, its standards (WS-CDL, WSCI, WSCL), and the distinction between choreography and orchestration; the OMG model-driven architecture for BPM, and its RFPs for BPDM and BPRI; BPSS and collaboration; and the influence of XLANG and WSFL on BPEL.

Chapter 5: Business Process Execution Language (BPEL)

Chapter 6: BPMI Standards: BPMN and BPML

Chapter 7: The Workflow Management Coalition (WfMC)

Chapter 8: World Wide Web Consortium (W3C): Choreography

Chapter 9: Other BPM Models

Chapter Five. Business Process Execution Language (BPEL)

The Business Process Execution Language for Web Services (BPEL4WS, usually shortened to BPEL, which rhymes with "people") is, as its name suggests, a language for the definition and execution of business processes. Though it is not the only standard process language, BPEL is the most popular, and is beginning to saturate the process space.

There are two common ways to represent business processes: XML and notational. BPEL competes in the XML arena with BPML, XPDL, and other approaches. Notational languages include Business Process Modeling Notation (BPMN) and UML activity diagrams. Each type of representation has its merits and, as discussed in [Chapter 2](#), a good BPM architecture requires both of them.

IBM, Microsoft, and BEA wrote the BPEL specification and subsequently handed it over to the WSBPEL technical committee of the OASIS organization (of which they are members) for standardization. The conceptual roots of BPEL coincide exactly with earlier BPM initiatives of each of the three companies: IBM's WSFL, Microsoft's XLANG and BEA's Process Definition for Java (PD4J). As discussed in [Chapter 3](#), WSFL is based on Petri nets and XLANG uses concepts of the pi-calculus; BPEL, consequently, is a mixture of these two theories. PD4J, as discussed later in this chapter, is the basis for the Java extension to BPEL, known as BPELJ.

This chapter explores several aspects of BPEL:

- Its authors and maintainers
- How to develop a BPEL process
- Java extensions to BPEL
- BPEL's support for common BPM patterns
- A substantial example of BPEL in action

NOTE

OASIS, or the Organization for the Advancement of Structured Information Standards (<http://www.oasis-open.org>), is a nonprofit consortium that develops, maintains, and promotes e-business standards, including ebXML, SGML, UDDI, PKI, and BPEL. Members include Adobe, AMD, BEA, BMC, Citrix, Computer Associates, Cyclone Commerce, Dell, Documentum, EDS, Entrust, Fujitsu, FundSERV, HP, Hitachi,

IBM, IDS Scheer, Intel, IONA, Microsoft, NEC, Netegrity, Nokia, Novell, Oracle, PeopleSoft, Reuters, SAP, SeeBeyond, Sun, Tibco, Verisign, Vignette, Visa, webMethods, Wells Fargo, and Xerox. The BPEL 1.1 specification is published on the corporate web sites of each of its major authors.

5.1. Anatomy of a Process

The BPEL specification^[*] is positioned as a business process extension to existing web services standards. In the past, web services were limited to stateless interactions; BPEL and other process and choreography languages show how to build stateful, conversational business processes from web services. BPEL is a rigorous language that builds on and extends web services for interacting processes.

[*] T. Andrews, M. Curbera, et al., "Business Process Execution Language for Web Services," Version 1.1. <http://www.oasis-open.org>, May 2004. Available at the following URLs: <http://dev2dev.bea.com/technologies/webservices/BPEL4WS.jsp>, <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbiz2k2/html/bpel1-1.asp>, <http://ifr.sap.com/bpel4ws/>, <http://www.siebel.com/bpel>.

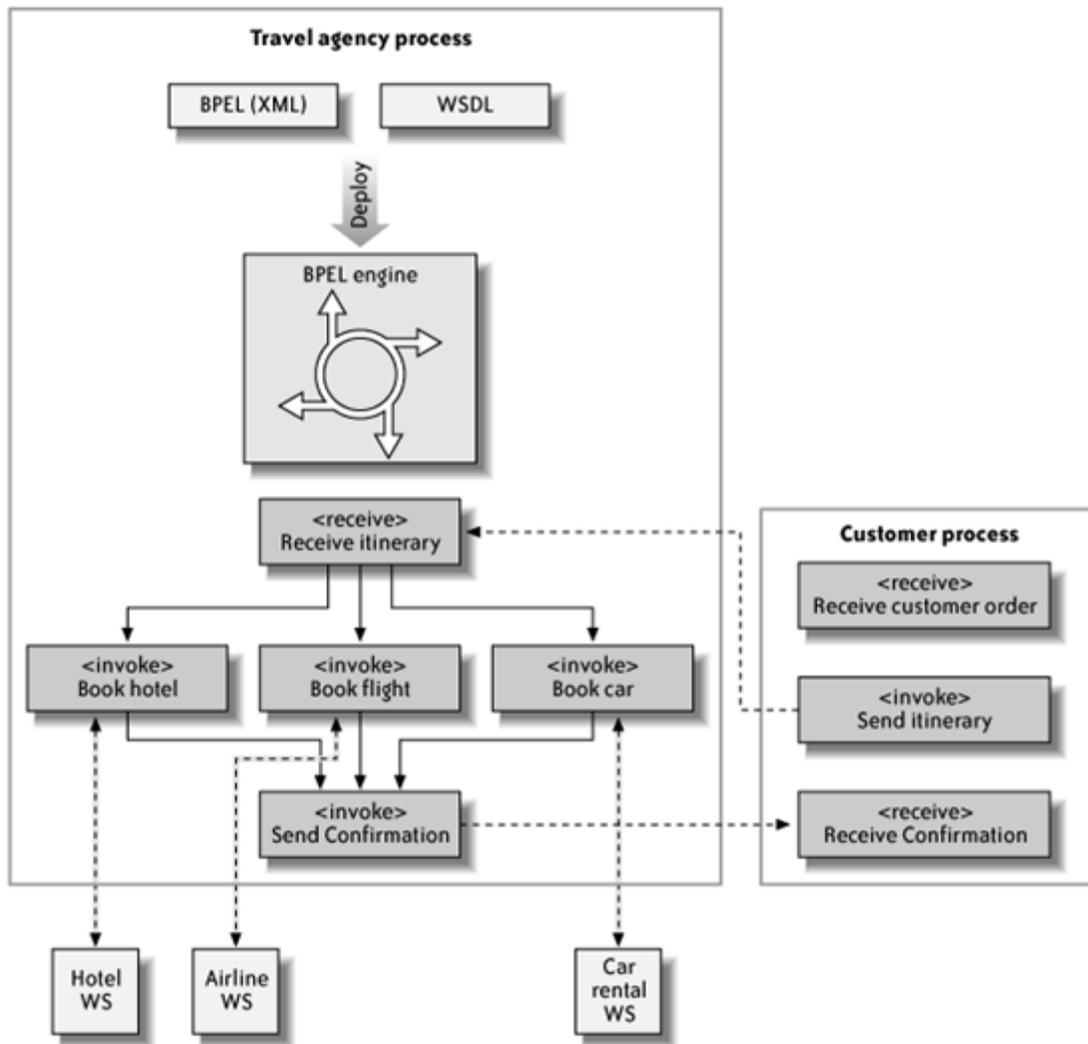
A BPEL process definition consists of two types of files:

- Web Services Definition Language (WSDL) files specifying the web service interfaces—partner link types, properties, port types and operations, message and part—of interest to the process, including services implemented by and called by the process. WSDL is a well-known technology with many uses besides process definition.
- BPEL files, each of which encodes in XML form the definition of a process, including its main activities, partner links, correlation sets, variables, and handlers for compensation, faults, and events.

When combined, the WSDL and the process definition form a business control flow that can act as an interface with external parties through web services. In a sense, the main steps in a process—the ones that drive the flow—are its service touchpoints, represented by `receive`, `pick`, `invoke`, and `reply` activities. The most rigid rule of BPEL programming is that a process must begin with a `receive` or `pick` activity, implying that a process must start by being called as a service of a particular type. Thereafter, the process's logic is less constrained; it does what is required to meet its internal business requirements and its public message interchange agreements.

Figure 5-1 depicts a BPEL travel agency process and its surroundings in a typical BPEL architecture.

Figure 5-1. Anatomy of a BPEL travel agency process



The source code is a BPEL XML file and one or more WSDLs. The source is deployed on a BPEL execution engine, which oversees the running of the process logic. The travel agency process starts by receiving a customer's itinerary. It then attempts hotel, flight, and rental car bookings, and finally sends a confirmation to the customer. A corresponding customer process works in concert with the travel agency process; actions in one trigger the other. Interactions with the booking systems of the hotel, airline, and car rental agency are web service-driven, but, transparently to the travel agency application, the booking services are traditionally stateless, not process-oriented. (Chapter 2 develops a comprehensive architectural model featuring a BPEL runtime engine. Chapters 10 and 11 demonstrate the development, deployment, and testing of BPEL processes on the Oracle BPEL Process Manager platform.)

Figure 5-2 shows a UML class diagram of BPEL's object model, and the overall structure of a process.

Figure 5-2. BPEL overall object model

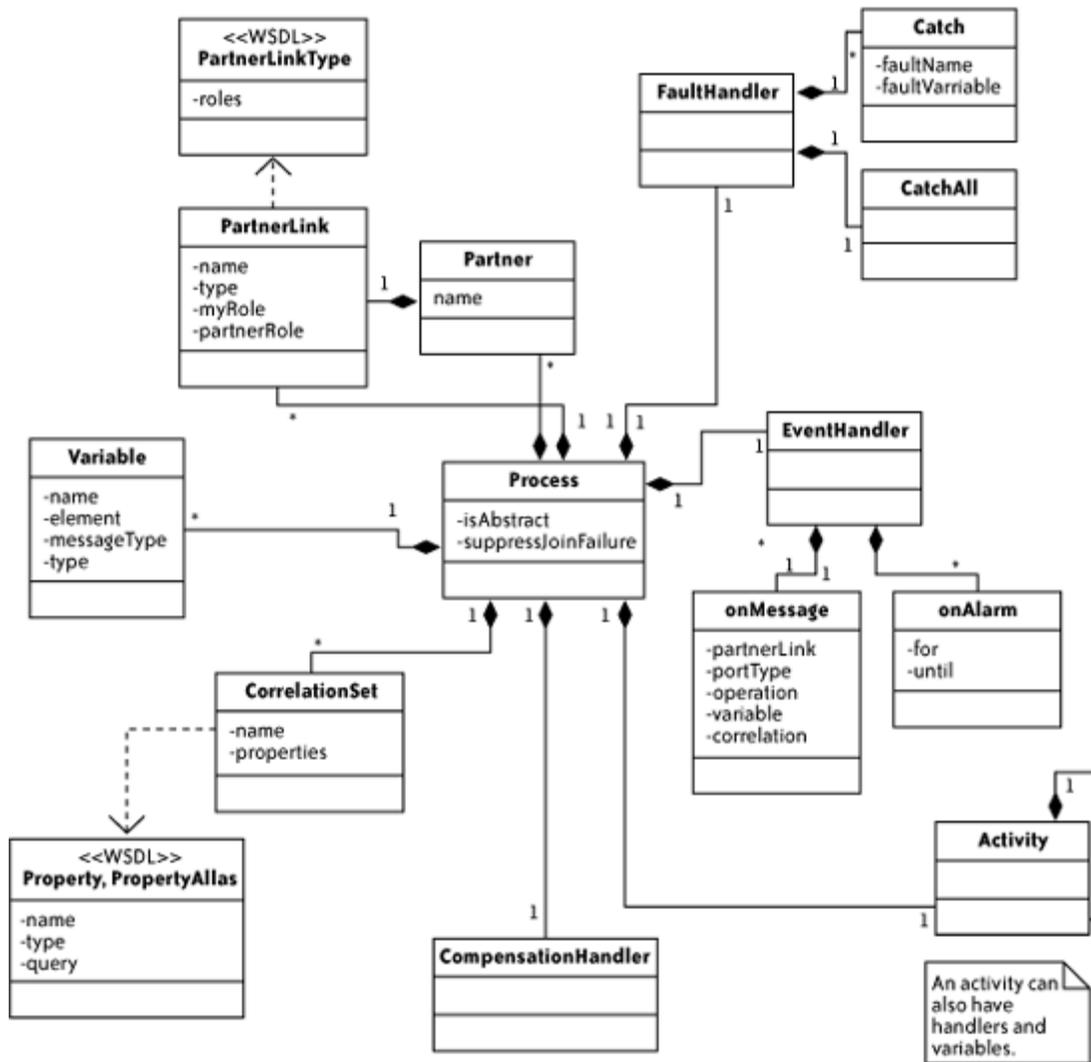


Figure 5-3, another UML diagram, describes the types of process activities.

Table 5-1 summarizes the objects depicted in Figure 5-4.

Figure 5-3. BPEL activity object model

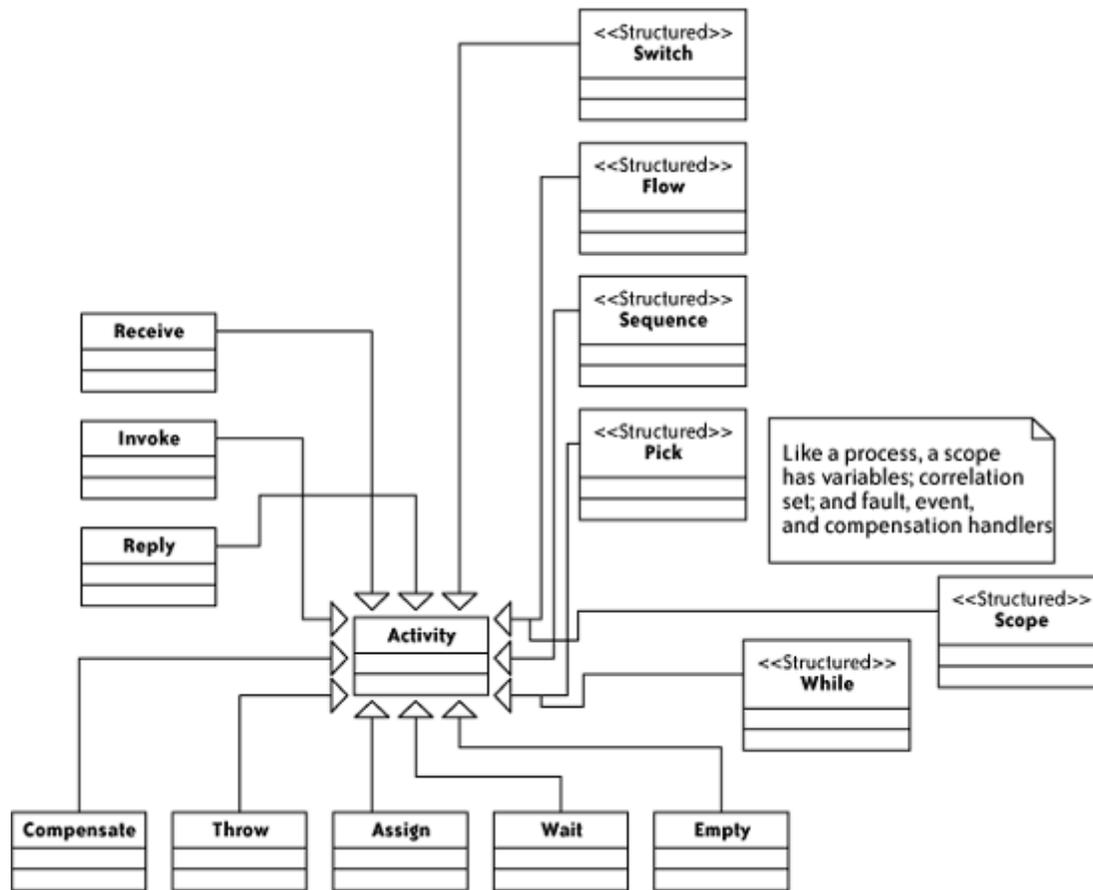


Table 5-1. BPEL objects

Name	Description
Process	A business process containing one or more of the subsequent objects.
Variable	A variable for use in a process or a scope, with a type based on a WSDL message type, an XSD element, or an XSD basic type. A process or scope can have zero or more variables.
Property, Property Alias (from WSDL)	A property is a token of data from a WSDL message. A property alias is an XPath expression to find the value of the property.
CorrelationSet	A set of one or more properties used to correlate message data with the conversational state of the process. A process or scope can have zero or one correlation sets.
Partner Link Type (from WSDL)	A mapping of web service port types to partner roles.
Partner Link	A process' declaration of which partner links it supports and, for each, which role it performs and which role its partner is expected to perform. A process can have one or more partner links.
Partner	Not commonly used; a set of partner links. A process can have zero or more partners.
Compensation Handler	An activity, containing cancellation or rewind logic, to be executed in case a scope or process that has already completed needs to be reverted back to its initial state. A process or scope can have zero or one compensation handlers.
Fault Handler, Catch, CatchAll	A set of handlers to process exceptions, based on fault type, in a process or scope. A process or scope can have zero or one fault handlers; there is no limit on the number of catches within the handler.
EventHandler, onMessage, onAlarm	A set of handles to process unsolicited events, based on event type, in a process or scope. A process or scope can have zero or one event handlers; there is no limit on the number of event detectors within the handler.
Activity	Base type for a BPEL activity. A process or scope has exactly one activity, though that activity can be a structured activity that is broken down into smaller pieces.

Name	Description
Receive	An activity that receives a SOAP message on an inbound web service.
Invoke	An activity that calls a partner's web service, either synchronously or asynchronously.
Reply	An activity that returns a synchronous reply to an inbound web service call triggered by a receive.
Compensate	An activity that triggers the compensation of a given scope or process.
Throw	An activity that generates a fault, triggering the fault handler for the given process or scope.
Assign	An activity that copies data from one variable to another.
Wait	An activity that pauses the process for a specified duration, or until a specified time.
Empty	No-op. Performs no action.
Switch	An exclusive-OR structure. Executes the activity for the conditional case that evaluates to true.
Flow	A distinctive parallel activity execution structure with support for directed graph-based flow.
Sequence	Runs a set of activities sequentially.
Pick	Waits for exactly one of several events (including timeouts) to occur. Executes the activity corresponding to the first event that fires.
While	Runs an activity in a loop for as long as a given XPath-valued expression is true.
Scope	An activity with its own set of handlers, variables, and correlation sets.

EXECUTABLE AND ABSTRACT PROCESSES

A BPEL process can be executable or abstract. An *executable process* is built to actually run in a process engine. An *abstract process* is a protocol definition or an account of the publicly observable behavior of a given participant. Though in this book we use BPEL strictly for executable purposes, the motivation for the abstract approach is important to understand. WSDL in isolation describes only the static structure of a partner's interface: its inbound and outbound services. An abstract process, in contrast, is behavioral; it describes the control flow exhibited by the partner as its interfaces with its partners. A BPEL abstract process, in this regard, serves the same purpose as a WSCI interface (described in [Chapter 8](#)).

The BPEL code for an abstract process resembles that of an executable process, except that the abstract process contains only activities that model public interaction or drive control flow; an abstract process can use process data but only for the evaluation of conditions that affect control flow.

Abstract processes set the attribute `abstractProcess="yes"` in their process elements.

User name:

Book: Essential Business Process Modeling

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.

5.2. BPEL Example

The best way to learn a language is to dive into it. This section provides an extended example to illustrate some of the main elements of the BPEL language. The BPEL process in [Example 5-1](#), which is similar to that presented in [Chapter 10](#) (and also described in the discussion of state machines in [Chapter 3](#)), manages the processing of an insurance claim.

Example 5-1. BPEL example: InsuranceClaim.bpel

```

1  <process name="InsuranceClaim"
2    targetNamespace="http://acm.org/samples"
3    suppressJoinFailure="yes"
4    xmlns:tns=http://acm.org/samples
5    xmlns=http://schemas.xmlsoap.org/ws/2003/03/business-process/
6    xmlns:xsd=http://www.w3.org/2001/XMLSchema
7    xmlns:addressing=http://schemas.xmlsoap.org/ws/2003/03/addressing
8    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
9
10   <!--
11     Partners in the process:
12     client - app that can initiate and kill
13     worklist - service that manages manual activities
14   -->
15   <partnerLinks>
16     <!--
17     <partnerLink name="client" partnerLinkType="tns:InsuranceClaim"
18       myRole="InsuranceClaimProvider" />
19     <partnerLink name="worklist" partnerLinkType="task:TaskManager"
20       partnerRole="TaskManager" myRole="TaskManagerRequester" />
21   </partnerLinks>
22
23   <!-- Process-level variables -->
24   <variables>
25     <variable name="status" type="xsd:string" />
26     <variable name="initiateMsg" messageType="tns:InsuranceClaimMsg" />
27     <variable name="killEv" messageType="tns:InsuranceClaimMsg" />
28     <variable name="taskResponse" messageType="task:taskMessage" />
29   </variables>
30
31   <!-- Message correlation to be performed on the ClaimID field -->
32   <correlationSets>
33     <correlationSet name="claim" properties="tns:claimID" />
34   </correlationSets>
35
36   <!-- Catch any errors and fix manually -->
37   <faultHandlers>
38     <catchAll>
39       <empty name="PlaceholderForManualFix" />
40     </catchAll>
41   </faultHandlers>
42
43   <!-- Globally receive a kill event (correlated with the claim ID from the
44     original initiate) and terminate the process. -->
45   <eventHandlers>
46     <onMessage partnerLink="client" portType="tns:InsuranceClaim"
47       operation="kill" variable="killEv">
48       <correlations>
49         <correlation set="claim" initiate="no" />
50       </correlations>
51       <sequence>
52         <empty /><!-- Do something, like notify internal systems of kill -->
53         <terminate name="killClaim" />

```

```

54     </sequence>
55 </onMessage>
56 </eventHandlers>
57
58 <sequence>
59
60     <!-- We start with a receive activity: get the initiate message. Will
61     correlate on claim set defined earlier
62     -->
63     <receive partnerLink="client" portType="tns:InsuranceClaim"
64         operation="initiate" variable="initiateMsg" createInstance="yes"
65         name="initiateEvent">
66         <correlations>
67             <correlation set="claim" initiate="yes"/>
68         </correlations>
69     </receive>
70
71     <!-- Let an agent evaluate it. Call worklist partner to do this -->
72     <invoke name="evalClaim" partnerLink="worklist" portType="task:TaskManager"
73         operation="evalClaim" inputVariable="initiateMsg"/>
74
75     <!-- Get either the response or a timeout -->
76     <pick name="analyzePick">
77         <onMessage partnerLink="worklist" portType="task:TaskManagerCallback"
78             operation="onTaskResult" variable="taskResponse">
79             <!-- From response extract status and set to variable 'status' -->
80             <assign name="setStatus">
81                 <copy>
82                     <from variable="taskResponse" part="payload"
83                         query="/tns:taskMessage/tns:result="/>
84                     <to variable="status"/>
85                 </copy>
86             </assign>
87         </onMessage>
88         <!-- Timeout! 10 days have passed. Escalate -->
89         <onAlarm for="PT10D">
90             <sequence>
91                 <!-- Call partner service to escalate -->
92                 <invoke name="evalClaim" partnerLink="worklist"
93                     portType="task:TaskManager" operation="escalateClaim"
94                     inputVariable="initiateMsg"/>
95                 <!-- Get the escalation response -->
96                 <receive name="receiveTaskResult" partnerLink="worklist"
97                     portType="task:TaskManagerCallback"
98                     operation="onTaskResult" variable="taskResponse"/>
99                 <!-- From response extract status and set to variable 'status' -->
100                <assign name="setStatus">
101                    <copy>
102                        <from variable="taskResponse" part="payload"
103                            query="/tns:taskMessage/tns:result="/>
104                        <to variable="status"/>
105                    </copy>
106                </assign>
107            </sequence>
108        </onAlarm>
109    </pick>
110
111    <!-- Look at result of claim process and act accordingly:
112    'rejected' and 'accepted' are good. Anything else, throw a fault -->
113    <switch name="resultEval">
114        <case condition="bpws:getVariableData('status')='rejected'">
115            <empty> <!-- perform rejection actions -->
116        </case>
117        <case condition="bpws:getVariableData('status')='accepted'">
118            <empty> <!-- perform acceptance actions -->
119        </case>
120        <otherwise>
121            <throw name="illegalStatus" faultName="illegalStatus"/>
122        </otherwise>
123    </switch>
124 </sequence>
125 </process>

```

The underlying business logic for this process is straightforward: when a claim arrives, an agent evaluates it and determines whether to accept or reject it. If the agent does not respond within 10 days, the activity is escalated to a manager, who then makes an accelerated accept/reject decision. At any point, the processing can be terminated with a kill event.

The code implementing this logic is an XML document with root element `process`; from that element, we learn that the name of the process is `InsuranceClaim` (line 1). The heart of the process is the `sequence` activity in lines 58-124, which in turn contains a set of child activities and runs them sequentially. In this example, those children are a `receive` activity (lines 63-69), an `invoke` activity (lines 72-73), a `pick` activity (lines 76-109), and a `switch` activity (lines 113-123).

The purpose of the `receive` activity (lines 63-69) is to listen for an inbound message containing the claims request. The process, as you will see, has a WSDL interface and implements a web service interface; the `receive` represents an inbound operation of that service (in this case, `initate`, as stated in line 64). The attribute `createInstance="yes"` in line 64 means that this `receive` activity is the trigger that starts the process.

In the `invoke` activity in lines 72-73, the process calls a web service operation, offered by another partner (in this example, operation `evalClaim` for partner `worklist`), to evaluate the claim. The partner service chooses an insurance agent to evaluate the claim. Several days might pass before the agent makes a decision about the claim, but the `invoke` call is asynchronous; it returns immediately, and the process waits for a response in the `pick` in lines 76-109.

The `pick` activity waits for one of two events to occur: the response from the worklist (handled by the `onMessage` element in lines 77-87) or a timeout (in `onAlarm` in lines 89-108). In the former case, the response arrives as another inbound message; the worklist service calls back the process by invoking its `onTaskResult` operation (line 78). The `assign` activity in lines 80-86 extracts the `result` field from the worklist message and copies it into a process variable called `status` (declared in the `variables` section in line 25), which is used later in the process as a decision point for acceptance or rejection processing.

CORRELATION METHOD 1: WS-ADDRESSING

How does the process correlate the response with the request made in the `invoke`? In other words, how can the process be sure that the response is for the right insurance claim? The answer, in this case, is the WS-Addressing standard: when the process sends a request to the worklist, it embeds a unique message ID into the SOAP header; when the worklist responds, it passes back that ID. The logic is handled at the web services container level, transparent to the BPEL code.

The BPEL specification does not require that the container support WS-Addressing. To use this method of correlation, check the capabilities of your BPEL platform. This example was test on Oracle's BPEL Process Manager, which encourages the WP-Addressing approach.

The timeout occurs after 10 days, an interval determined by the condition `for="PT10D"` in line 89. The timeout triggers escalation: if, after ten days, no response has arrived from the worklist on the original claim request, the process uses an `invoke` to call the worklist's `escalateClaim` service operation (lines 92-94). Like the earlier `invoke`, this call is asynchronous. The `receive` activity in lines 96-98 waits for the result, which might take several hours or days to arrive. The `assign` in lines 100-106 captures the result in the `status` variable.

Finally, the `switch` activity in lines 113-132 is an exclusive-OR construct that performs either acceptance or rejection logic based on the value of the `status` variable, which records the result returned by the worklist. The acceptance case is handled in lines 114-116; its logic (not shown in the code example for brevity) probably involves sending an acceptance letter and a check to the subscriber. The rejection handler in lines 117-119 may involve sending a rejection letter. If the result is neither an accept nor a reject (the `otherwise` case in lines 120-122), the activity throws a fault called `illegalStatus`, which triggers the `faultHandler` in lines 37-42. The handler in this example does very little, but it can have arbitrarily complex exception handling logic.

The requirement to kill the claim at any point is met by the `eventHandlers` construct in lines 45-56. The `onMessage` handler listens on the inbound `kill` web service operation (lines 46-47), using a BPEL correlation set (lines 48-50) to ensure that the kill event is for the same claim as the one under consideration in this instance of the process. The event handler explicitly terminates the process using the `terminate` activity in line 53.

CORRELATION METHOD 2: BPEL CORRELATION SET

Unlike WS-Addressing, which matches IDs in the message header, BPEL's correlation mechanism matches particular data fields embedded in the message body. The `receive` activity in lines 63-69 populates a correlation set, to be used for subsequent correlations, with data embedded in its message. In the kill event handler, the data from the kill event is compared with the data in the correlation set; if it matches, the kill event is processed; if it does not match, the kill event is rejected.

The `partnerLinks` element, in lines 15-21, specifies the web service interface offered by the process, as well as interfaces that are used by the process. Two partner links are listed: `client` represents the interface implemented by the process, which includes the inbound operations to initiate and kill a claim. The client interface is specified in a WSDL, a portion of which is displayed here:

```
<portType name="InsuranceClaim">
  <operation name="initiate">
    <input message="tns:InsuranceClaimMsg" />
  </operation>
  <operation name="kill">
    <input message="tns:InsuranceClaimMsg" />
  </operation>
</portType>

<plnk:partnerLinkType name="InsuranceClaim">
  <plnk:role name="InsuranceClaimProvider">
    <plnk:portType name="tns:InsuranceClaim" />
  </plnk:role>
</plnk:partnerLinkType>
```

The WSDL includes the definition of a `partnerLinkType` called `InsuranceClaim`, which defines a `role` called `InsuranceClaimProvider`, which maps to the port type `InsuranceClaim`, which in turn defines the initiate and kill operations. The `client` partner link supports the `InsuranceClaimProvider` role, implying that it implements the kill and initiate operation of that role's port type.

The `worklist` partner link is a service that this process calls to evaluate and escalate the claim; `worklist` calls the process back with results. The WSDL for `worklist` has the following port types and partner link types:

```
<portType name="TaskManager">
  <operation name="evalClaim">
    <input message="tns:taskMessage" />
  </operation>
  <operation name="escalateClaim">
    <input message="tns:taskMessage" />
  </operation>
</portType>
<portType name="TaskManagerCallback">
  <operation name="onTaskResult">
    <input message="tns:taskMessage" />
  </operation>
</portType>

<plnk:partnerLinkType name="TaskManager">
  <plnk:role name="TaskManager">
    <plnk:portType name="tns:TaskManager" />
  </plnk:role>
  <plnk:role name="TaskManagerRequester">
    <plnk:portType name="tns:TaskManagerCallback" />
  </plnk:role>
</plnk:partnerLinkType>
```

The expression `myRole="TaskManagerRequester"` in line 20 of the process means that the process implements the `TaskManagerCallback` service, which defines an `onTaskResult` operation. The expression `partnerRole="TaskManager"` means that the `worklist` partner implements the port type `TaskManager` with operations `evalClaim` and `escalateClaim`.

User name:

Book: Essential Business Process Modeling

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.

5.3. BPEL in a Nutshell

The following sections examine the essential language constructs designers will need to understand to create a BPEL process: the basic process structure, variables and assignments, exception handling and compensation, split and join, loops, participant exchange, transactions, and extensions.

5.3.1. Basic Process Structure: Start, End, Activities, Sequence

A developer's first BPEL process is not easy to write. Two puzzles face the beginner:

- A BPEL process has exactly one activity (which, of course, can consist of any number of subactivities to any level of hierarchy). Which activity should that be? Which are allowed? Which are disallowed? Which are recommended?
- How is the initial inbound event handler set up? Where do you place the `receive` or `pick` in the activity described in the first problem?

The simplest approach, and the one recommended to most developers, is to use a `sequence` whose first activity is a `receive` with `createInstance="yes"`, as in the following code example :

```
<sequence>
  <receive . . . createInstance="yes" . . .> . . . </receive>
  <!-- other activities -->
</sequence>
```

This process starts when the `receive` triggers, then executes the remaining steps sequentially, and exits when the last activity has completed. Error handling complicates the processing, of course; see the section "[Exception Handling and Compensation](#)" for a discussion.

Another approach is to use a `receive` within a `flow`. The `receive` should not have any inbound links. For example:

```
<flow>
  <receive . . . createInstance="yes" . . .> . . . </receive>
  <!-- other activities -->
</flow>
```

This process starts when the `receive` triggers, whereupon the remaining activities in the flow run in parallel, or, if links are used, in a directed-graph style of execution. (See the section "[Split and Join](#)" later in this chapter for more on `flow`.) The process finishes when the flow has merged all its activities.

In the advanced category, the BPEL specification includes an example (Section 16.3 of the BPEL 1.1 specification) with two `receive` nodes in a `flow`. The intent is not to choose one or the other events (as with a `pick`), but to require both in order to proceed with the remainder of the process. Both nodes compete to start the process, but because they are in a flow, when one wins, it must wait for the other to trigger and join it in the process. As the following code example shows, the `flow` is enclosed in a `sequence`, so when the `flow` completes, the other activities in the process run sequentially, and when they complete, the process exits normally:

```
<sequence>
  <flow>
    <receive . . . createInstance="yes" . . .>
      <correlations>. . .</correlations> <!-- corr required for multi-start -->
    </receive>
    <receive . . . createInstance="yes" . . .>
      <correlations>. . .</correlations> <!-- corr required for multi-start -->
    </receive>
  </flow>
```

```

    <!-- other activities -->
</sequence>

```

Here are some novelties to avoid at all costs:

- Do not put basic activities (e.g., `assign`, `empty`, or `wait`) before the initial `receive` or `pick`.
- Do not use `switch` or `while` as the main activity of the process.
- Do not use a `scope` as the main activity of the process. The process has everything that a scope has—handlers, variables, correlation sets..

5.3.2. Variables and Assignments

Most processes need to maintain application data during the course of their execution. The data is initialized when the process begins and is subsequently read and modified. A BPEL process can define a set of variables, pass them to web service touchpoints as input or output parameters, and assign all or part of one variable to another.

Formally, a process variable has a name that is unique for its scope and a type that is either a WSDL message type or an XML Schema element or basic type. A variable is set in one of the following ways:

- Bound to the input of an inbound activity, such as a `receive`, `pick`, or `eventHandler`.
- Bound to the output of a synchronous `invoke`.
- Assigned a value with the `assign` activity

The `assign` activity is defined as a copy of data from a source to a target. The source can be a literal value, an expression, the value of the whole or part of another process variable, or part of a process variable. [Table 5-2](#) shows an example for each type.

Table 5-2. Assignment examples

Usage	Code
From literal	<pre> <variable name="x" type="xsd:int"/> <assign> <copy> <from>1</from> <to variable="x"/> </copy> </assign> </pre>
From expression	<pre> <variable name="x" type="xsd:int"/> <assign> <copy> <from expression="bpws:getVariableData('x') + 1"/> <to variable="x"/> </copy> </assign> </pre>
Whole copy	<pre> <variable name="x" type="xsd:int"/> <variable name="y" type="xsd:int"/> <assign> <copy> <from variable="y"/> <to variable="x"/> </copy> </assign> </pre>

Usage**Code**

Partial copy

```

In WSDL:
<message name="person">
  <part name="name" type="xsd:string"/>
  <part name="address" type="xsd:string"/>
</message>

In BPEL process:
<variable name="person" messageType="person"/>
<variable name="personAddress" type="xsd:string"/>

<assign>
  <copy>
    <from variable="person" part="address"/>
    <to variable="personAddress"/>
  </copy>
</assign>

```

The built-in BPEL function `bpws:getVariableData` is used to get the value of a variable. For an XSD element type, the function can use an XPath expression to extract a particular data token from the XML document. For a WSDL message type, the function can extract values from any part of the message.

In addition, see the section "The Life Event process" in Chapter 11 for an example of assigning dynamic endpoint information to a partner link.

5.3.3. Exception Handling and Compensation

A scope is a process code block having its own set of activities and corresponding variables; correlation sets; and handlers for fault, compensation, and events. A scope is a localized execution context: its variables are visible only within its boundaries, and its handlers apply only to its activity flow. Scopes are hierarchical; a scope can have multiple nested subscopes, each of which can in turn have additional subscopes, and so on down the chain; a process is itself a top-level scope.

5.3.3.1. Compensation handler

Compensation is a transaction that reverses the effects of a previously completed transaction. In many online transactional applications, updates to one or more systems are made within a local or distributed transaction, and are not finalized until the transaction is committed; to negate the updates, the application simply rolls back the transaction. However, business processes often run for such long periods of time that keeping open transactions for the duration is infeasible. If an earlier step needs to be negated, rather than rolling back its transaction, the process executes its compensation handler. The following code example shows a handler that invokes a `rescind` web service, presumably informing its partner to cancel some activity. The `rescind` is intended to reverse the `update` service invocation in the main flow of the scope block.

```

<scope name="s">
  <compensationHandler>
    <invoke operation="rescind" . . . />
  </compensationHandler>

  <invoke operation="update" . . . />
</scope>

```

The compensation handler for a scope is invoked, using the `compensate` activity, from the fault handler or compensation handler of the parent scope. In the following case, the compensation handler for scope `inner` is called from the compensation handler of its parent scope `outer`. The operations `update` and `addToStatement` are compensated by `rescind` and `removeFromStatement`:

```

<scope name="outer">
  <compensationHandler>
    <sequence>
      <invoke operation="rescind" . . . />
      <compensate scope="s2"/>
    </sequence>
  </compensationHandler>

  <sequence>
    <invoke operation="update" . . . />

```

```

    <scope name="inner">
      <compensationHandler>
        <invoke operation="removeFromStatement" . . . />
      </compensationHandler>
      <invoke operation="addToStatement" . . . />
      . . .
    </scope>
  </sequence>
</scope>

```

5.3.3.2. Fault handler

Compensation is the reversal of a completed scope, and *fault handling* is the processing of a break in a scope that is in-flight. When a fault is generated, either implicitly by the BPEL engine or explicitly by a `throw` activity, control jumps to the fault handler defined for the given fault type.^[5] The fault handler is a set of `catch` structures, resembling the following:

[5] If no such handler is defined, the fault is propagated to the parent scope. If the parent has no suitable handler, the fault is propagated to the parent's parent, and so on, until the topmost scope level—that is, the process level—is reached. If the fault is not handled at the process level, the process is terminated.

```

<scope name="s1">
  <faultHandlers>
    <catch faultName="x:invalidAccount">
      . . .
    </catch>
    <catch faultName="x:closedAccount">
      . . .
    </catch>
    <catchAll">
      . . .
    </catchAll>
  </faultHandlers>
</scope>

```

BPEL faults are uniquely identified by name. Some are standard error types documented in the BPEL specification, such as `uninitializedVariable`, which is thrown by the engine when code tries to read the value of an uninitialized part of a variable. Others are application-specific, such as `x:invalidAccount` and `x:closeAccount`, which are used in the code example to represent illegitimate accesses of an account. Catch handlers are defined for both of these faults, and the `catchall` structure handles any faults not accounted for in the other catch structures. Each handler lists the activities to be performed to handle the fault. The handler can swallow the fault, leading to the resumption of processing in the scope, or it can rethrow the fault or throw a different fault, thereby propagating the fault to the parent scope.

The `throw` activity generates a fault, causing control to be passed to the fault handler defined for the given scope. In the following code example:

```

<switch name="routeRequest">
  <case name="checking" . . . > . . . </case>
  <case name="savings" . . . >
    <switch name="CheckAcctStatus">
      <case name="Open" . . . > . . . </case>
      <case name="Closed" . . . > <throw faultName="closedAccount"> </case>
    </switch>
  </case>
  <case name="trust" . . . > . . . </case>
  <otherwise><throw faultName="invalidAccount"/></otherwise>
</switch>

```

the `terminate` activity is used to immediately abort the process, skipping any defined fault handling:

```
<terminate name="Unrecoverable condition met">
```

5.3.3.3. Event handler

An event handler enables the scope to react to events, or to the expiration of timers, at any point during the scope's execution. Two obvious uses are:

Cancellation

The scope defines a handler for a cancellation event. If it receives it, the scope can be terminated, no matter where it is in its execution.

Escalation

A timer is set on the scope. If it expires, special activities are executed to perform "business escalation."

The next code example demonstrates these uses. Cancellation is triggered by the message defined by partner link `Customer`, port type `controller` and operation `cancel`; the handler throws a fault to terminate the scope. Escalation occurs after two days (`PT2D`), at which point the handler performs logic through an `invoke`.

```
<scope name="s1">
  <eventHandlers>
    <onMessage partnerLink="Customer" portType="controller"
      operation="cancel" variable="cancelEvent">
      <correlations>
        <correlation set="controllerSet" initiate="no"/>
      </correlations>
      <throw faultName="x:cancelled"/>
    </onMessage>

    <onAlarm until="PT2D">
      <invoke name="escalation" . . . />
    </onAlarm>
  </eventHandlers>
</scope>
```

5.3.4. Split and Join

BPEL's two activities for split and join—`switch` and `flow`—exhibit contrasting styles. `switch` is a traditional programmatic control structure for conditional logic. `flow` is an unusual graph structure with support for parallel activities connected by guarded links.

5.3.3.4. switch

`switch` is an exclusive-OR structured activity that consists of one or more `case` structures, each having a conditional expression and an associated activity. The activity performed by the switch is that of the first case whose condition evaluates to true. An optional `otherwise` clause can be defined with an activity but no condition; that activity is run only if none of the preceding cases have a true condition.

In the following example, a `flow` activity is performed if the first condition (variable `i` has value 1) holds; a `sequence` with an `assign` and `switch` is executed if the second condition (`i = 2`) is true. An `invoke` is run by default:

```
<switch>
  <case condition="bpws:getVariableData('i')=1">
    <flow> . . . </flow>
  </case>

  <case condition="bpws:getVariableData('i')=2">
    <sequence>
      <assign . . . />
      <switch> . . . </switch>
    </sequence>
  </case>

  <otherwise>
    <invoke . . . />
  </otherwise>
</switch>
```

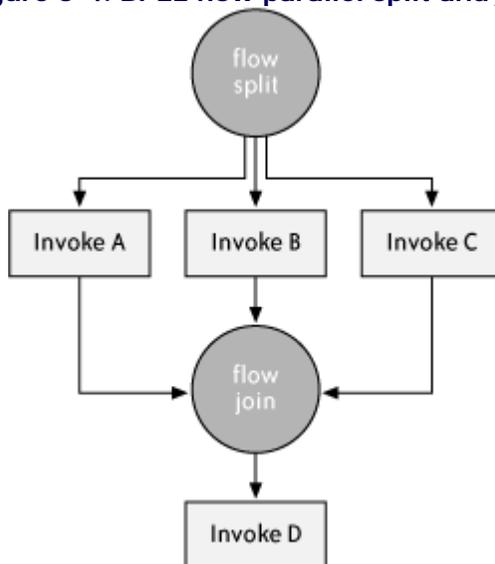
5.3.3.5. flow

The `flow` activity—BPEL's most interesting, unusual, and pedantic construct—models parallel activity execution and activity synchronization. Depending on how it is configured, `flow` exhibits a variety of behaviors; to understand `flow`, it is best to learn each case—the parallel split and join behavior, link synchronization and dependencies behaviors, and dead path elimination behavior—using an example and a diagram.

5.3.3.5.1. Parallel split and join

The first case for `flow` is perfectly suited to the P4 patterns for parallel split and join. In the following example (see Figure 5-4 and the following code), the process invokes in parallel (i.e., splits) web services for partner links A, B, and C respectively; the order of execution is unpredictable. The `flow` waits for each contained activity to complete (i.e., joins them) before exiting. The invocation of the service for partner link D does not occur until each of the three previous invocation finishes.

Figure 5-4. BPEL flow parallel split and join



```

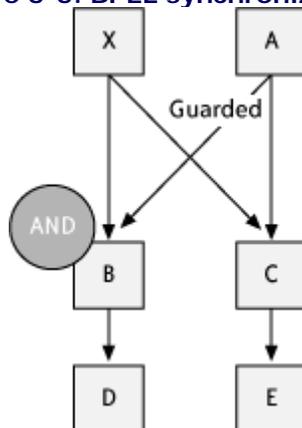
<flow>
  <invoke partnerLink="A" . . ./>
  <invoke partnerLink="B" . . ./>
  <invoke partnerLink="C" . . ./>
</flow>
<invoke partnerLink="D" . . ./>

```

5.3.3.5.2. Links and synchronization dependencies

The BPEL flow mechanism offers several features to model the situation where one activity cannot start until one or more activities on which it depends complete. Specifically, a flow can define a set of links, each originating from a `source` activity in the flow and terminating at a `target` activity in the flow. An activity can be the source as well as the target of multiple links. In Figure 5-5, for example, A has links to B and C, X has links to B and C, B has a link to D, and C has a link to E.

Figure 5-5. BPEL synchronization



Understanding flow in BPEL requires understanding the order in which such activities are executed, and whether a particular activity is executed at all.

In the current example, initially X and A are run in parallel. Activities B and C must wait for both X and A to

complete because, according to BPEL's flow rules, an activity must wait for each of its incoming links. To complicate matters, a source activity can define a transition condition on any of its outgoing links (if not defined for a given link, the condition defaults to true), and a target activity can define a join condition evaluated based on the transition conditions each of its incoming links. The join condition is optional; if not defined, it defaults to an OR condition, which is true if any of the link transition conditions is true. The target activity executes only if its join condition evaluates to true. (The behavior when the join condition is false is examined in the next section.)

Continuing with the example, C does not define a join condition, and hence defaults to OR. That is, C executes if either X or A (or both) has a true transition condition on its link. Note that C must wait for both source activities to complete, even though it requires only one to send out a true link. Activity B defines an explicit join condition, requiring both of its incoming links, from X and A, to be true; B executes only if both X and A have a true transition condition. The link from A to B has an explicit transition condition (it is "guarded"), whereas the link from X to B is implicitly true. As for activities D and E: D waits for B to complete, and E waits for C.

The BPEL encoding of this scenario is as follows:

```
<flow>
  <links>
    <link name="AB" />
    <link name="AC" />
    <link name="XB" />
    <link name="XC" />
    <link name="BD" />
    <link name="CE" />
  </links>

  <invoke partnerLink="A" . . .>
    <source linkName="AB" transitionCondition=". . ."/>
    <source linkName="AC" />
  </invoke>

  <invoke partnerLink="X" . . .>
    <source linkName="XB" />
    <source linkName="XC" />
  </invoke>

  <invoke partnerLink="B" . . .
    joinCondition="bpws:getLinkStatus('XB') and bpws:getLinkStatus('AB')">
    <source linkName="BD" />
    <target linkName="AB" />
    <target linkName="XB" />
  </invoke>

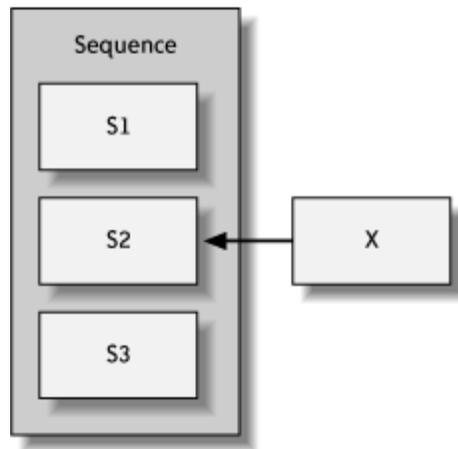
  <invoke partnerLink="C" . . .>
    <source linkName="CE" />
    <target linkName="AC" />
    <target linkName="XC" />
  </invoke>

  <invoke partnerLink="D" . . .>
    <target linkName="BD" />
  </invoke>

  <invoke partnerLink="E" . . .>
    <target linkName="CE" />
  </invoke>
</flow>
```

A link can cross activity boundaries. In [Figure 5-6](#), activity S2 is the second activity in the sequence of S1-S2-S3, but it is also the target of a link from activity X. According to BPEL's flow rules, S2 cannot execute until both S1 and X have executed. The overall order of execution is S1-X-S2-S3 or X-S1-S2-S3.

Figure 5-6. BPEL cross-boundary link



5.3.3.5.3. Dead path elimination

By default, if an activity's join condition evaluates to false, BPEL generates a fault called `bpws:joinFailure`; the activity is not executed, and control is diverted to a fault handler. In the first of the examples presented in the previous section, B's join condition fails if either X or A sends a false link to B.

BPEL also supports the semantics of *dead path elimination*: if the join condition of an activity is false, the activity doesn't execute but completes immediately and sends false on each of its outgoing links. For example, if B's join condition fails, and dead path elimination is enforced, B doesn't execute, and its outgoing link to D is set of false, which, because D's implicit join condition requires the link from B to be true, in turn prevents D's execution. In this case, the execution sequence is either X-A-C-E or A-X-C-E.

Deciding whether to use dead path elimination semantics is as simple as setting a flag. The attribute `suppressJoinFailure` can be set to `yes` or `no` (default is `no`) for any activity. If set to `yes`, that activity uses dead path elimination semantics; otherwise, it uses the semantics of join fault. In the following code example, the flag is enabled for the entire flow, which means that B, a part of that flow, will use dead path elimination if its join condition is false.

```
<flow suppressJoinFailure="yes">
  <links>
    <link name="AB"/>
    <link name="AC"/>
    <link name="XB"/>
    <link name="XC"/>
    <link name="BD"/>
    <link name="CE"/>
  </links>

  <invoke partnerLink="A" . . .>
    <source linkName="AB" transitionCondition=". . ."/>
    <source linkName="AC"/>
  </invoke>

  <invoke partnerLink="X" . . .>
    <source linkName="XB" />
    <source linkName="XC" />
  </invoke>

  <invoke partnerLink="B" . . .
    joinCondition="bpws:getLinkStatus('XB') and bpws:getLinkStatus('AB')">
    <source linkName="BD" />
    <target linkName="AB" />
    <target linkName="XB" />
  </invoke>

  <invoke partnerLink="C" . . ./>
    <source linkName="CE" />
    <target linkName="AC" />
    <target linkName="XC" />
  </invoke>

  <invoke partnerLink="D" . . .>
    <target linkName="BD"/>
  </invoke>

  <invoke partnerLink="E" . . .>
    <target linkName="CE"/>
  </invoke>
</flow>
```

```

    </invoke>
  </flow>

```

The semantics of dead path elimination are discussed further in [Chapter 3](#).

5.3.5. Loops

BPEL's sole looping construct is the `while` activity. Unlike BPML, BPEL does not offer a `foreach` loop, but an example of how to iterate through a repeating XML element, a typical use of `foreach`, is provided in this section.

5.3.5.1. while

The `while` activity executes a child activity in a loop and evaluates the continuation condition before each iteration. The child activity is executed if the condition is true; otherwise, the loop exits.

In the following example, the loop iterates over a counter variable `i`. The integer variable is initially set to 0 and is incremented by 1 at the end of each loop iteration. The loop runs until `i` is 5, executing as part of a sequence an `invoke` activity and the `assign` to increment `i`:

```

<variable name="i" type="xsd:integer"/>
<assign>
  <copy>
    <from expression="0"/>
    <to variable="i"/>
  </copy>
</assign>

. . .

<while condition="bpws:getVariableData(i) != 5">
  <sequence>
    <invoke . . . />
    <assign>
      <copy>
        <from expression="bpws:getVariableData(i) + 1"/>
        <to variable=";i"/>
      </copy>
    </assign>
  </sequence>
</while>

```

5.3.5.2. Implementing foreach

Iterating over a set of XML data in BPEL is a comparatively difficult development chore, thanks to BPEL's omission of, or decision not to include, a `foreach` loop. The following example shows how iterate over all instances of the `input` element described by the following schema:

```

<element name="ForLoopRequest">
  <complexType>
    <sequence>
      <element name="input" type="string" maxOccurs="5" />
    </sequence>
  </complexType>
</element>

```

Here is an XML document based on this schema:

```

<ForLoopRequest>
  <input>Foreach</input>
  <input> is</input>
  <input> possible</input>
  <input> after all</input>
</ForEachRequest>

```

[Example 5-2](#) is an excerpt of a BPEL process that iterates over the repeating elements of such a document.

Example 5-2. Iterating BPEL process

```

1  <!-- input is a ForLoopRequest.
2      numItems is a count of the "input" elements in the input message
3      currItem is a loop counter that starts at 0 and increases by 1 to numItems
4      theItem is used in the loop to store the value in the currItem position
5      tempExpr is a string used to build XPath in the loop -->
6      <variables>
7          <variable name="input" messageType="tns:ForLoopRequestMessage"/>
8          <variable name="numItems" type="xsd:int"/>
9          <variable name="currItem" type="xsd:int"/>
10         <variable name="theItem" type="xsd:string"/>
11         <variable name="trace" type="xsd:string"/>
12         <variable name="tempExpr" type="xsd:string"/>
13     </variables>
14     . . .
15     <!-- Initialize currItem to 0 and use XPath count( ) function to get the number of
16         "input" elements. -->
17     <assign name="getNumItems">
18         <copy>
19             <from expression="bpws:getVariableData('&quot;input&quot;',&quot;payload&quot;',
20                 &quot;count(/tns:ForLoopRequest/tns:input)&quot;)'></from>
21             <to variable="numItems"/>
22         </copy>
23         <copy>
24             <from expression="number(0)"></from>
25             <to variable="currItem"/>
26         </copy>
27     </assign>
28     . . .
29     <!-- the while loop, ironically named "foreach"
30         It runs until currItem = numItems -->
31     <while name="foreach"
32         condition="bpws:getVariableData('currItem') &lt;
33         bpws:getVariableData('numItems')">
34         <sequence>
35             <assign name="doForEach">
36                 <!-- Increment currItem. Need this for loop condition, as well
37                     as XPath index -->
38                 <copy>
39                     <from expression="bpws:getVariableData('&quot;currItem&quot;')
40                         + 1"></from>
41                     <to variable="currItem"/>
42                 </copy>
43                 <!-- The XPath expression is of the form:
44                     "/tns:ForLoopRequest/tns:input[currItem]" (index is 1-based)
45                     Build it and store in tempExpr. -->
46                 <copy>
47                     <from expression="concat('&quot;/tns:ForLoopRequest/tns:input[&quot;',
48                         bpws:getVariableData('&quot;currItem&quot;'), '&quot;]&quot;)'></from>
49                     <to variable="tempExpr"/>
50                 </copy>
51                 <!-- Evaluate the expression and store in "theItem" -->
52                 <copy>
53                     <from expression="bpws:getVariableData('&quot;input&quot;',
54                         &quot;payload&quot;',bpws:getVariableData('&quot;tempExpr&quot;'))">
55                     </from>
56                     <to variable="theItem"/>
57                 </copy>
58             </assign>
59             <!-- Now, do something with "theItem" -->
60         </sequence>
61     </while>

```

Intuitively, this code performs the following logic:

```

numItems = count(/ForLoopRequest/input)
for currItem = 1 to numItems
    theItem = /ForLoopRequest/input[currItem]

```

Table 5-5 maps this intuitive code to the corresponding section in the thorny BPEL code.

Table 5-3. ForEach logic

Intended code	Actual code
numItems = count(/ForLoopRequest/input)	Assign rule, lines 19-20.
For currItem = 1 to numItems	First, <code>currItem</code> is assigned to 0, lines 24-25. The <code>while</code> condition appears in lines 32-33. <code>currItem</code> is incremented in lines 39-41.
theItem = /ForLoopRequest/input[currItem]	Two assign copies in lines 46-57.

5.3.6. Participant Exchange

One of BPEL's strongest notions is that processes communicate with each other as business partners. The partnering relationships are represented declaratively in the definition of process links, as well as in the actual communication touch points of the process flow.

5.3.6.1. Partner link types

To begin with, in the WSDL, *partner link types* map web service port types to partner roles. More formally, a partner link type has a name and one or two roles, each of which has a name and a reference by name to a port type. A partner link type with two roles represents a relationship in which partners, such as a buyer and seller, exchange service calls:

```
<partnerLinkType name="BuyerSeller">
  <role name="buyer"> <portType name="buyerPT" /> </role>
  <role name="seller"> <portType name="sellerPT" /> </role>
</partnerLinkType>
```

A partner link type with one role is suitable for interactions where the service does not need to know about its callers:

```
<partnerLinkType name="Server">
  <role name="server"> <portType name="serverPT" /> </role>
</partnerLinkType>
```

5.3.6.2. Partner links

In the BPEL process definition, *partner links* declare which partner link type roles defined in the WSDL are performed by the process and which are performed by partners. For example, a buyer process defines the link `BuyerSellerLink`, referencing the type `BuyerSeller`, with `myRole` set to `buyer` and `partnerRole` to `seller`:

```
<partnerLink name="BuyerSellerLink" partnerLinkType="BuyerSeller"
  myRole="buyer" partnerRole="seller" />
```

If the process invokes the `server` service, it declares a partner link called `ServerLink` with `partnerRole` set to `server`:

```
<partnerLink name="ServerLink" partnerLinkType="Server" partnerRole="server" />
```

NOTE

In most BPEL examples, a partner link is mapped statically to a particular service endpoint. See [Chapter 11](#) (the section "[The Life Event process](#)") for an example of a dynamic partner link, whose endpoint information is determined at runtime.

5.3.6.3. Partners

BPEL also offers a construct known as a business *partner*, which has a name and a set of partner links. For example, a given process participant that is both a seller and a shipper can be defined as `SellerShipper`:

```
<partner name="SellerShipper">
  <partnerLink name="Seller" />
```

```
<partnerLink name="Shipper" />
</partner>
```

5.3.6.4. Partner interactions

BPEL process flow contains the activities *receive*, *reply*, *invoke*, and *pick*, which implement the actual interpartner communication. *receive* and *pick* represent links implemented by the process and called by partners (e.g., the buyer). *reply* and *invoke* represent partner services called by this process (e.g., the seller, the server).

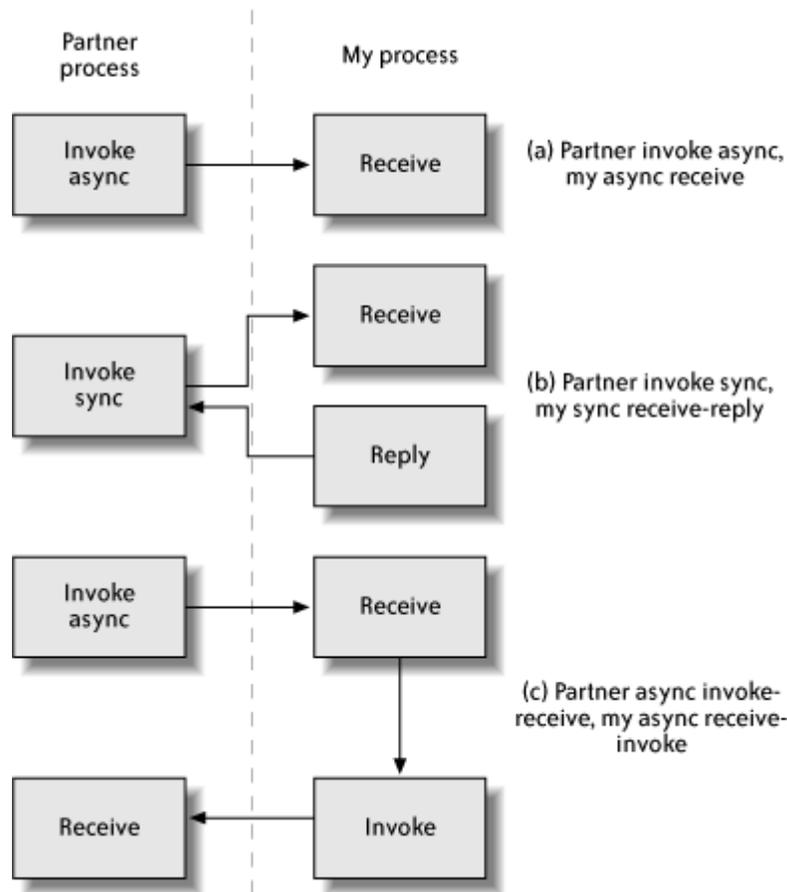
Table 5-4 describes the possible types of partner interactions. The pronoun *my* is used to distinguish a process from its partners.

Table 5-4. Partner communication patterns

Pattern	Partner roles	Description
Async receive	MyRole = receiveWS	A partner calls my web service, which triggers logic in my process. The partner resumes control immediately.
Sync receive-reply	MyRole = receive WS PartnerRole = reply WS	A partner calls my web service, which triggers logic in my process. My process eventually sends back a response to the partner. The partner blocks for the duration; to the partner, the entire interaction is a single two-way web service invocation.
Async receive-invoke	MyRole = receive WS PartnerRole = invoke WS	A partner calls my web service, which triggers logic in my process; the partner resumes control immediately. My process eventually calls the partner's callback web service through an <i>invoke</i> .
Sync invoke	PartnerRole = invoke WS	My process calls the partner's web service with an <i>invoke</i> . The two-way web service returns a result.
Async invoke	PartnerRole = invoke WS	My process calls the partner's web service with an <i>invoke</i> . The "one-way" web service does not return a result.
Async invoke-receive	MyRole = receive WS PartnerRole = invoke WS	My process calls the partner's web service with an <i>invoke</i> . The web service later calls back by triggering my <i>receive</i> .

As Figure 5-7 shows, each interaction, considered from the perspective of the current process, has a corresponding partner interaction. A partner *async invoke* is an *async receive* for my process, whereas a partner *sync invoke* is a *sync receive-reply* for my process. A partner *async invoke-receive* is asymmetrically an *async receive-invoke* for my process.

Figure 5-7. Partner interactions



The four partner interaction activities are discussed in the following sections.

5.3.6.4.1. invoke

The `invoke` activity calls a partner web service either synchronously or asynchronously. The web service is identified by partner link type and WSDL port type and operation. A synchronous web service requires both input and output variables to be passed in; an asynchronous service requires only an input variable. In case the service generates a fault, the `invoke` activity can define one or more fault handlers, and it can also specify a compensation handler.

The following example shows a call to a synchronous web service:

```
<invoke partnerLink="myPartner" portType="service" operation="syncRequest"
  inputVariable="request" outputVariable="response"/>
```

5.3.6.4.2. receive

Whereas `invoke` lets a process call a partner's web service's operation, `receive` is a web service operation implemented by the process for use by partners. The idea, as was discussed previously, is that partners trigger the execution of a process by calling its services; for the process, the service is an event that set it in action.

Like `invoke`, `receive` uses partner link type and WSDL port type and operation to identify the service. The arguments passed in by the caller are bound to a specified variable. The following code example implements the `initialRequest` service, whose input is bound to the `request` variable. The `createInstance` attribute indicates that when this service is called, a new instance of the process is launched:

```
<receive partnerLink="myPartner" portType="service" operation="initialRequest"
  variable="request" createInstance="yes" />
```

The next example shows the `secondRequest` service, which does *not* create a new instance but listens for an event in an existing instance:

```
<receive partnerLink="myPartner" portType="service" operation="secondRequest"
  variable="request" createInstance="no" />
```

5.3.6.4.3. reply

The `reply` activity sends back a response *synchronously* to an earlier `receive`. The effect is that of a single web service call in which the `receive` accepts the input, and the `reply` passes back the output; the process can perform arbitrary processing logic in between. As the following code sample shows, the `reply` matches the `partnerLink`, `portType`, and `operation` attributes of the `receive`; the output is specified in the `variable` attribute

```
<reply partnerLink="client" portType="c:AccountOpenPT" operation="submitApplication"
  variable="request" />

<!-- do stuff in between -->

<reply partnerLink="client" portType="c:AccountOpenPT" operation="submitApplication"
  variable="confirmationNumber" />
```

5.3.6.4.4. pick

`pick` waits for one of several events to occur, then executes the activity associated with that event. Like `receive`, `pick` has a `createInstance` attribute that determines whether to start a new process instance when the event occurs. `pick` also has an optional timer (`onAlarm`) that executes a specified activity if none of the specified events occurs within a given duration or until a given deadline. The list of candidate events is a set of `onMessage` elements; the event is defined as a web service operation for a given port type and partner link, similar to the `receive` event definition earlier.

In the following example, the `pick` triggers a new process instance when it gets events `ev1` or `ev2`. In each case, it executes some sequence of activities. The `onAlarm` condition fires if neither event occurs in 3 days and 10 hours. (The XPath 1.0 syntax is `P3DT10H`.)

```
<pick createInstance="yes">
  <onMessage partnerLink="pl" portType="pt" operation="ev1"
    variable="v">
    <sequence>. . .</sequence>
  </onMessage>

  <onMessage partnerLink="pl" portType="pt" operation="ev2"
    variable="v">
    <sequence>. . .</sequence>
  </onMessage>

  <onAlarm for="P3DT10H">
    <sequence>. . .</sequence>
  </onAlarm>
</pick>
```

5.3.6.5. Properties

A standard WSDL extension allows for the definition of *message properties*, which can be thought of as macros to access parts of WSDL messages. The definition of a message property has two parts: a *property*, which has a name and a WSDL message or XML schema type; and a *property alias*, which stipulates from which WSDL message and part the property comes. The property alias includes an XPath expression that extracts data to be represented by the property from the WSDL message. In the following code example, the property `nameProp` represents the name of a `person` message type:

```
<property name="nameProp" type="xsd:string"/>
<propertyAlias propertyName="nameProp" messageType="person" part="name"
  query="/personalData/name"/>
```

5.3.6.6. Correlation

The main use of properties in BPEL is *message correlation*, a large topic in its own right, as will be apparent shortly. The definition of a process can specify any number of correlation sets, each of which has a name and a list of message properties. In the following code sample, the correlation set `USPersonCorrSet` includes properties `nameProp` and `SSNProp`:

```
<correlationSets>
  <correlationSet name="USPersonCorrSet" properties="nameProp SSNProp"/>
</correlationSets>
```

The purpose of the correlation set is to tie together a partner conversation. A fundamental principal of BPEL is that instances of partner processes communicate with each other simply by calling each other's web services, passing application data whose structure is transparent to the BPEL engine. In BPEL, partners do not address each other by ID, but rather pass around messages containing key fields that can be correlated for the lifetime of the exchange.

The protocol governing a correlation set, from the perspective of one of the participating processes, is as follows:

- The first step in the conversation is the **initiating** step. This step is normally a **receive** or an **invoke**. The correlation set is populated with values based on the initiating message. In the case of a **receive**, the values come from the inbound message. For an **invoke**, the values are those sent to the partner, those received back (if the invocation is synchronous), or both.
- Each subsequent step matches the data in the set with the initialized data. A **receive** is triggered only if the data in the message matches the correlation set. An **invoke** or **reply** must send out data that agrees with the data in the set.

The following example illustrates the protocol. Assuming that the conversation in question is the opening of an account, the first step is a **receive** that, as the code shows, initiates the `USPersonCorrSet` set with contents of the variable `request`. The attribute `initiate` is set to `Yes`, indicating that this step is the first in the conversation. The `createInstance` attribute is also set to `Yes`, indicating that this step is the first in the process. If it is set to `No`, this activity can still begin a conversation because a process can have multiple conversations; of course, a fundamental rule of BPEL is that every process begins with the initiation of a new conversation.

```
variable="request" createInstance="Yes">
  <correlations>
    <correlation set="USPersonCorrSet" initiate="Yes"/>
  </correlations>
</receive>
```

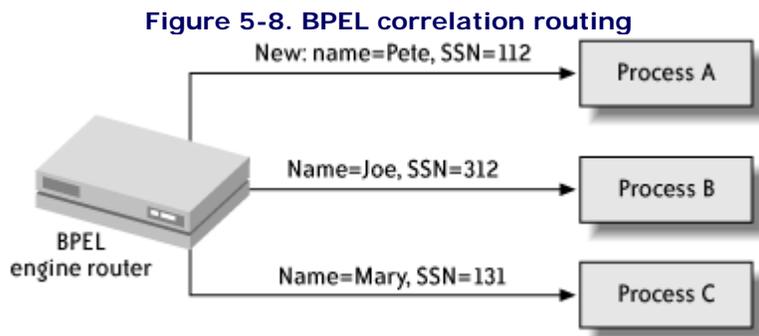
The conversation is continued with the following asynchronous **invoke**. The `initiate` attribute is set to `No`. BPEL requires that the data sent match the correlation set of the previous set.

```
<invoke partnerLink="l" portType="pt" operation="approveAccount"
  inputVariable="request" >
  <correlations>
    <correlation set="USPersonCorrSet" initiate="No" pattern="out"/>
  </correlations>
</invoke>
```

The final step in the conversation is a **receive** that retrieves the asynchronous response to the previous invoke. BPEL triggers this activity only if the data in the message matches the correlation set.

```
<receive partnerLink="l" portType="pt" operation="accountApproval"
  variable="accountApproval" >
  <correlations>
    <correlation set="USPersonCorrSet" initiate="No"/>
  </correlations>
</receive>
```

Based on this information, it is clear that any implementation of a BPEL engine requires a router component that, upon receipt of a message from a web service invocation, triggers the process instance that correlates the message. The engine is depicted in [Figure 5-8](#). Different name/SSN combinations are routed to different process instances; process A, not yet initiated, is started by the message with `name=Pete` and `SSN=112`; subsequent messages matching those criteria are routed to process A.



5.3.7. Transactions

Several factors make transactions hard to manage in a business process. First, a process that is long-running cannot run in a single ACID transaction; instead, it must necessarily divide its function into a number of smaller transactions. Second, the specification of standards for distributing transactions across multiple partner web services and processes is a work in progress. BPEL 1.1 anticipates, but does not support, distributed transaction coordination protocols such as WS-Transaction. Consequently, if a partner invokes a BPEL process, or a BPEL process invokes a partner, the BPEL process cannot, using a standard protocol, work together with the partner in the event of an error to reverse the effects of the interaction. The best that can be achieved is for the partners to agree on particular cancellation operations.

BPEL's strategy for the first problem is compensation (described earlier in the section "Exception Handling and Compensation"), which provides a structured and application-directed way to reverse the effects of the smaller activities that have already completed. This happens strictly locally; the second problem remains an open issue. The BPEL specification uses the term Long Running Transaction (LRT) to refer to the use of a hierarchical nesting of scoped compensation handlers to unwind previously committed work in a controlled fashion. An LRT is not a specific language feature but rather a way to describe the use of compensation to undo work performed over potentially a long period of time.

BPEL developers beware: the LRT strategy helps guide the effort, but the actual design and coding of compensation logic is still a difficult chore. The hard work is application-specific!

5.3.8. Extensions

Core BPEL is extended by adding attributes or elements, qualified by namespace, to BPEL elements that support extensibility. An important new BPEL extension, called BPELJ, is discussed in the next section.

User name:

Book: Essential Business Process Modeling

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.

5.4. BPELJ

"Pure" BPEL, examined previously, emphasizes "programming in the large," or the activity of defining the big steps of a process in a clean XML form. But a process that is

BPEL WISH LIST

As comprehensive as BPEL is, it can still be improved. The following standard enhancements would make BPEL more powerful and easier to code:

foreach

Include an activity that can iterate over a data set. (An example of iterating over a repeating XML element using a `while` loop and XPath is presented in the section "Implementing `foreach`." earlier in this chapter.)

XML creation and update

Enhance the `assign` activity to support the construction of XML elements or documents. For example, build a reply message using data from a request message.

Lightweight subprocesses or macros

Provide a mechanism to factor out a chunk of code into a modular piece, and provide the ability to call that piece with parameters in a lightweight fashion.

More sophisticated correlation

In addition to basic correlation set matching, allow a message listener activity (`receive`, `pick`, `eventHandler`) to filter on an arbitrary Boolean condition written in XPath or XQuery. In the example at the beginning of this chapter, the kill claim event handler triggers only if the kill message has the same claim ID as the original claim request. Supporting more complex queries would be desirable; for example, allowing the kill only if the claim amount is less than \$1,000; the claim has been active for less than two business days; and the claim is not currently in escalation, activation, or rejection states.

Business calendars

Provide a standard way to reference business calendars in the calculation of timeout conditions in `wait` and `onAlarm` activities (e.g., have the process wait five business days for a particular event to occur).

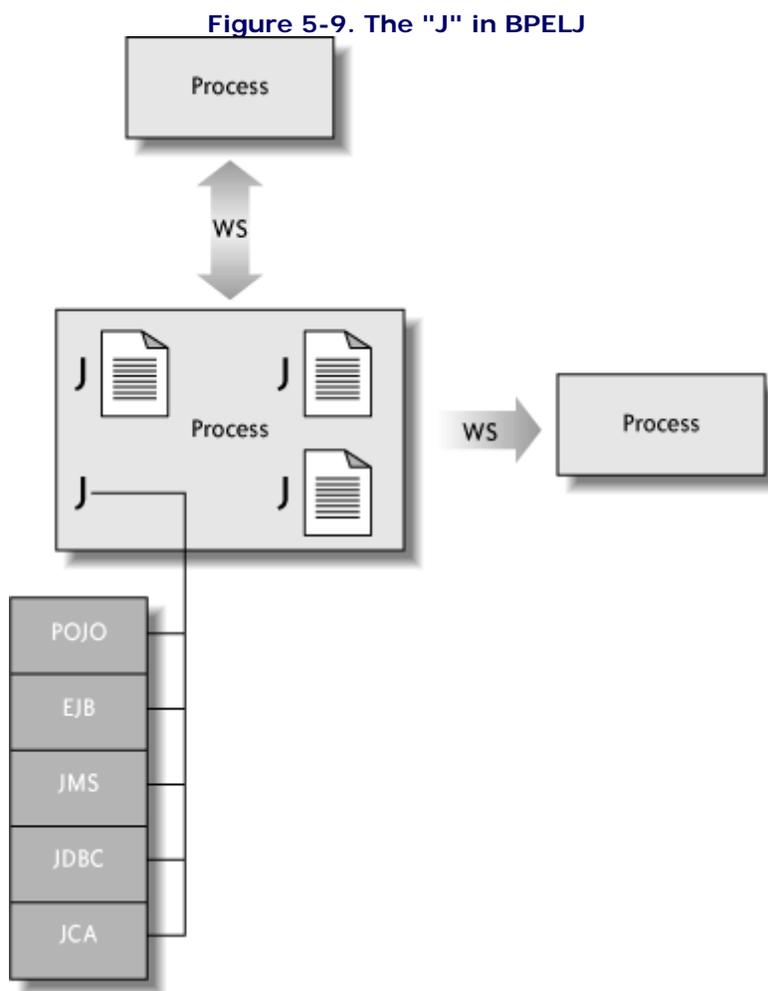
meant to actually run and be useful invariably has countless little steps, which are better implemented as lines of code than as activities in a process. Functions such as processing input, building output, interfacing with in-house

systems, making decisions, and calculating dates either drive or are driven by the process but are often too complex to encode as part of the process flow.

"Programming in the small" is essential to the development of real-world processes, but is hard to accomplish with pure BPEL. The best pure BPEL can offer is web services: if a piece of logic is hard, develop it as a web service and call it from the process with an `invoke` activity. This approach works, but it is grossly inappropriate. First, the reason for BPEL's emphasis on web services is the goal of communicating processes; partners call each other as services, but must a partner call pieces of itself as services? Second, the performance overhead of calling a service is obviously a showstopper; the process needs fast local access to the small logic components, as if they were an extension of the BPEL machine.

With these factors in mind, IBM and BEA have written a white paper that presents a standard Java extension of BPEL called BPEL for Java (BPELJ).^[5] A BPELJ process, depicted in [Figure 5-9](#), has chunks of embedded Java code, as well as invocations of separate plain old Java objects (POJOs), Enterprise Java Beans (EJBs), or other Java components. Though it still interfaces with its partner process through web services, the process internally leverages Java to perform much of the hard work.

[5] M. Blow, Y. Goland, M. Kloppman, F. Leymann, G. Phau, D. Roller, M. Rowley, "BPELj: BPEL for Java," <http://www-106.ibm.com/developerworks/webservices/library/wl-bpelj>, March 2004.



BPELJ is an obvious technology choice for companies that intend to deploy their processes on J2EE platforms such as BEA Weblogic and IBM WebSphere; the platform is already Java-enabled, so it is best to use Java capabilities in the construction of the process. Luckily, BEA and IBM are building to BPELJ.

NOTE

Current BPELJ vendor implementations are difficult to find. Two BPEL toolkits—the Oracle BPEL Process Manager 2.2 and IBM's WebSphere Application Developer Integration Edition 5.1.1—have Java extensions (Oracle's is used in an example in [Chapter 10](#)), but they are proprietary. BEA is planning to develop a reference implementation of BPELJ and possibly release it as Open Source. Expect major Java application server vendors like BEA, IBM, and Oracle to be the earliest adopters and to evolve the language.

5.4.1. A Glimpse of BPELJ

Example 5-3 demonstrates some of the core features of BPELJ. The bold italicized parts are BPELJ-specific.

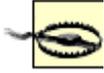
Example 5-3. BPELJ insurance automated claim

```

1  <!-- Process attributes:
2      - expressionLanguage is Java by default. Can be overridden to,
3      say, XPath at the element level
4      - Java code embedded in process goes in to the Java package "com.mike.claim"
5      - The BPELJ namespace is referenced below.
6  <process name="InsuranceClaim"
7      suppressJoinFailure="yes"
8      expressionLanguage="http://jcp.org/java"
9      bpelj:package="com.mike.claim"
10     targetNamespace="http://mike.com/claim"
11     xmlns:tns="mike.com/claim"
12     xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
13     xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
14     xmlns:bpelj="http://schemas.xmlsoap.org/ws/2003/03/business-process/java">
15
16     <!-- Three partner links: one a web service client interface, the others
17         Java internal stuff -->
18     <partnerLinks>
19         <partnerLink name="client" partnerLinkType="tns:Claim"
20             myRole="ClaimProvider"/>
21         <partnerLink name="claimProcessor"
22             partnerLinkType="bpelj:com.mike.claim.ClaimProcessorEJB">
23         <partnerLink name="jmsPublisher"
24             partnerLinkType="bpelj:javax.jms.TopicPublisher">
25     </partnerLinks>
26
27     <!-- Two variables, one an XML, the other Java -->
28     <variables>
29         <variable name="input" messageType="tns:ClaimsMessage"/>
30         <variable name="jmsMessage" messageType="bpelj:javax.jms.TextMessage">
31         <variable name="claimOK" messageType="bpelj:java.lang.Boolean ">
32     </variables>
33
34     <sequence name="main">
35         <!-- process starts by receiving a claim through the client web service -->
36         <receive name="receiveClaim" partnerLink="client" portType="tns:Claim"
37             operation="initiate" createInstance="yes">
38             <output part="input" variable="input"/>
39         </receive>
40
41         <!-- now invoke the Java claims processor as a partner link! -->
42         <invoke name="processClaim" partnerLink="claimProcessor" operation="execute">
43             <input part="input" variable="input"/>
44             <output variable="claimOK"/>
45         </invoke>
46
47         <!-- if claim is ok, publish the original input on a JMS topic -->
48         <switch name="pubIfOK">
49             <case>
50                 <condition>claimOK</condition>
51                 <bpelj:snippet name="createJMSMessage">
52                     <!-- Use partner link topic publisher to allocate a JMS message
53                         and populate it with the claim input message.
54                         Note "p_jmsPublisher" is the way to reference the partner
55                         link "jmsPublisher" in BPELJ -->
56                     <bpelj:code>
57                         jmsMessage=p_jmsPublisher.getSession( ).createTextMessage(input);
58                     </bpelj:code>
59                 </bpelj:snippet>
60                 <invoke name="PubClaim" partnerLink="jmsPublisher" operation="publish"
61                     <input part="message" variable="jmsMessage"/>
62                 </invoke>
63             </case>
64             <otherwise>
65                 <empty/> <!-- do nothing in this case -->
66             </otherwise>
67         </switch>
68
69     </sequence>
70 </process>

```

The process receives a claim by a web service request from a client application, then processes it using an EJB, and finally, if the claim was successful, publishes the request to a JMS topic for consumption by interested subscribers. The process extends core BPEL by defining partner links for Java components (lines 16-19), declaring Java variables (lines 25-26), using the `invoke` activity to call Java components (lines 37-40 and 51-53), using Java expressions to make decisions that affect flow (line 45), and embedding a snippet of Java code (lines 46-50). BPELJ features not included in this example include Java correlation, Java exception handling, and XML-Java binding.



Some of these changes are not supported by BPEL 1.1! The Java snippet in lines 46-50, for example, is illegal because BPEL does not support the addition of new activity types. Similarly, the `input` and `output` elements (lines 38-39) are not permitted in `invoke` and `receive` activities, and the `condition` (line 45) in the `switch` activity should be an attribute rather than a child element of `case`. In their joint whitepaper, BEA and IBM admit these incompatibilities and even suggest alternative approaches that are supported. For example, the snippet could be overloaded in the `empty` activity. The alternatives are onerous and unintuitive, which explains why the authors chose to cheat. *BPELJ won't be ready for prime time until these issues are resolved.*

5.4.2. BPELJ Source Code

The source code of a pure BPEL process is a set of XML files containing WSDL and BPEL process definitions. BPELJ source code can be either XML files with embedded Java code or Java source files annotated with XML. The former approach is the one adopted in the examples above; likewise, most of the samples in the BPELJ specification are XML with embedded Java.

The latter approach, documented in the BPELJ specification as a viable alternative, makes sense only if the number of lines of Java code is close to the number of lines of XML. [Example 5-4](#) shows a Java source file (*MyProcessImpl.java*) containing a comment in lines 1-15 that defines the BPEL process XML. The source file implements the class `MyProcessImpl`; the source code begins on line 16. Lines 17-20 implement a method that is called in the process on line 10.

Example 5-4. BPELJ sample

```

1  /**
2   * @bpelj:process process::
3   * <process name="MyProcess">
4   *   <variables>
5   *     <variable name="x" type="bpelj:Integer"/>
6   *   </variables>
7   *   . . .
8   *   <bpelj:snippet>
9   *     <bpelj:code>
10  *       x = self.getRandomValue ( );
11  *     </bpelj:code>
12  *   </bpelj:snippet>
13  *   . . .
14  * </process>
15  **/
16  public class MyProcessImpl implements Serializable
17  {
18      Integer getRandomValue( )
19      {
20          return new Integer(myRand.nextInt( ));
21      }
22  }

```

A well-designed BPELJ process should be mostly pure BPEL with a smattering of Java. (Analogously, a well-designed Java Server Page is mostly markup with minimal embedded Java.) Significant Java processing can be factored out to special Java partner link types, whose source code resides in traditional Java source files, separate from the process. Consequently, XML-driven BPELJ is preferable to Java-driven BPELJ.

NOTE

BPELJ is conceptually similar to Process Definition for Java (PD4J),^[5] a specification proposed by BEA to the

Java Community Process (JSR 207) for mixing XML and Java for process definition. PD4J is the design model for BEA's WebLogic Integration 8.1. BEA, along with IBM, authored BPELJ (also submitted to JSR 207), and is building to it for its Version 9 release of WebLogic Integration. PD4J favors the Java-with-annotated-XML approach, so perhaps WebLogic Integration 9 will adopt annotated Java as its development model.

[5] JSR 207, <http://www.jcp.org/en/jsr/detail?id=207>.

5.4.3. Other Language Implementations

BPELJ is the first language extension of BPEL, extending BPEL's capabilities on Java platforms. The same approach is suitable for other programming languages, notably those that figure prominently into Microsoft's .NET platform, such as C#. Expect to see such implementations soon, as BPEL increases in popularity.

User name:**Book:** Essential Business Process Modeling

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.

5.5. BPEL and Patterns

BPEL fares exceptionally well in its support for support for P4 patterns. As rated by P4 members in its paper,^[5] BPEL directly supports 13 of the 20 patterns, and it indirectly supports another one. The results of the paper are captured in Table 5-5.

[5] P. Wohed, W. van der Aalst, M. Dumas, A. H. M. ter Hofstede, "Pattern Based Analysis of BPEL4WS," FIT Technical Report, FIT-TR-2002-04.

Table 5-5. BPEL support for the P4 patterns

Pattern	Compliance (+, +-, -)	Approach	Notes
Sequence	+	<code>sequence</code> activity	
Parallel Split	+	<code>flow</code> activity	
Synchronization	+	<code>flow</code> activity followed by another activity, which will not execute until all parallel paths in the flow have completed.	
Exclusive Choice	+	<code>switch</code> activity.	
Simple Merge	+	<code>switch</code> activity followed by another activity, which will not execute until the one activity selected in the switch has completed.	
Multi-Choice	+	<code>flow</code> with conditional links to the activities to be chosen.	See P4 paper, p.8f.
Sync Merge	+	Use dead path elimination to join the results of the multiple choice.	
Multi Merge	-	No.	
Discriminator	-	No.	See P4 paper, p.9.
Arbitrary cycles	-	Only structured loops are allowed. No goto-like constructs.	
Implicit Termination	+	<code>rflow</code> with a link out.	
Multiple Instances (MI) Without Synchronization	+	<code>invoke</code> in a <code>while</code> loop.	
MI With Design Time Knowledge	+	Run each instance as a separate activity in a <code>flow</code> activity.	
MI With Runtime Knowledge	-	Onerous.	See P4 paper, p. 11.
MI Without Runtime Knowledge	-		
Deferred Choice	+	<code>pick</code> activity	

Pattern	Compliance (+, +-, -)	Approach	Notes
Interleaved Parallel Routing	+ -	Multiple <code>scopes</code> within a <code>flow</code> that compete for a single shared variable whose access is serialized (<code>variableAccessSerializable</code> is set to <code>yes</code>). The order of the scopes is arbitrary but serial.	See P4 paper, p. 12f.
Milestone	-	Poll for milestone in a <code>while</code> loop.	See P4 paper, p.14.
Cancel Activity	+	<code>Fault</code> out of the activity to be cancelled.	
Cancel Case	+	<code>terminate</code> activity	

User name:

Book: Essential Business Process Modeling

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.

5.6. Summary

The main points of this chapter include the following:

- BPEL is an XML-based process definition language. The XML approach has several merits, including programmability, executability, exportability, and easy web services integration capabilities.
- BPEL was originally written by IBM, Microsoft, and BEA, but has been handed over to OASIS for standardization. BPEL is based on IBM's WSFL and Microsoft's XLANG.
- The source code for a BPEL process is a set of WSDL files and a BPEL XML file. WSDL is the standard web service definition format, specifying port types, partner link types, message types, and properties. The process definition references the WSDL, creating partner links based on WSDL partner link types and variables based on WSDL-defined message types. Compensation, fault and event handlers can be defined for the process or any of its scope levels. The flow of a BPEL process includes service touchpoints (`receive`, `invoke`, `reply`) and control flow elements (`wait`, `while`, `switch`, `flow`, `sequence`, `scope`).
- BPELJ introduces Java extensions to "pure" BPEL, such as the ability to define Java process variable, evaluate dates and conditions with Java code, and embed code snippets. Other powerful features include Java partner links (enabling `invoke` calls to local Java classes in addition to pure BPEL's partner web services) and correlation based on Java classes.
- BPELJ source code can be XML with embedded Java or Java with annotated XML. The former approach is arguably better from a design perspective. The latter approach is influenced by BEA's PD4J model, used in WebLogic Integration 8.
- BPEL has built-in or easily attainable support for 14 of the 20 P4 patterns.

User name:

Book: Essential Business Process Modeling

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.

5.7. References

1. S. Dietzen, "Standards for Service-Oriented Architecture," *Weblogic Pro*. May/June 2004.
2. C. McDonald, "Orchestration, Choreography, and Collaboration,"
<http://lists.w3.org/Archives/Public/www-archive/2003May/0009.html>
3. S. White, "Business Process Modeling Notation, " Version 1.0.
<http://www.bpmn.org/Documents/Introduction%20to%20BPMN.pdf>, May 2004.