

Modeling Business Processes with BPEL4WS

Frank Leymann, Dieter Roller

IBM Software Group
Schönaicher Strasse 220
71032 Böblingen
ley1@de.ibm.com
rol@de.ibm.com

Abstract: Business Process Execution Language for Web Services (BPEL4WS) allows defining both, business processes that make use of Web services, and business processes that externalize their functionality as Web services. This short paper introduces the basic language elements of BPEL4WS using a simple example. The concepts underlying the language are briefly explained: Establishing bilateral partnerships, correlating messages and processes, defining the order of the activities of a business process, event handling, handling exceptions via long-running transactions, the resulting programming model, and the usage of BPEL4WS in pure B2B scenarios.

1 Introduction

Web services are components, which are based on the industry standards WSDL [1], UDDI [2], and SOAP [3]. They enable to connect different components even across organizational boundaries in a platform and language independent manner [4].

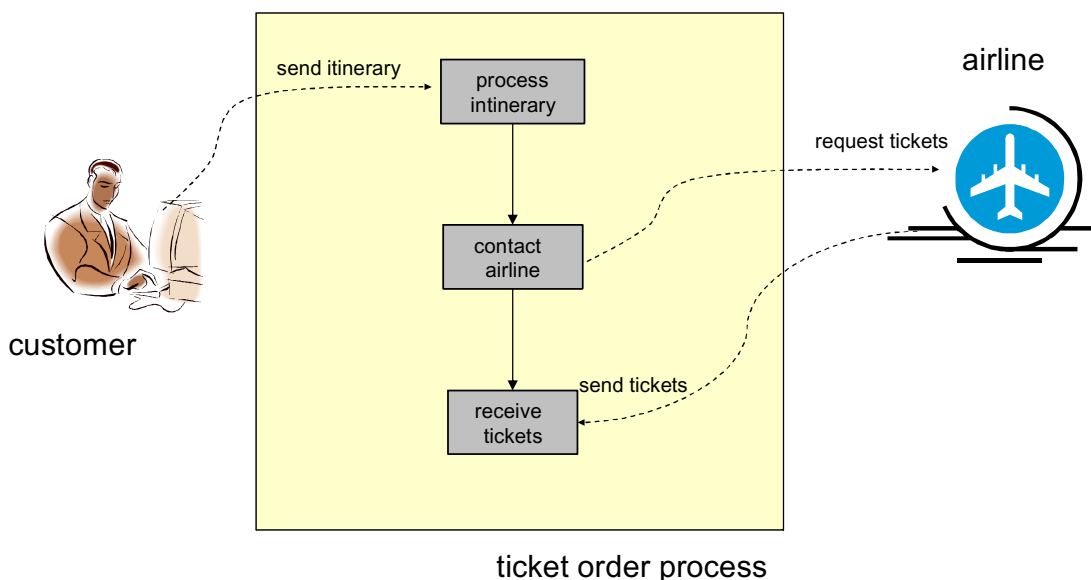
None of these standards for Web services however provides for the definition of the business semantics of Web services, the Web services are isolated and opaque. Breaking isolation means to connect Web services and specify how collections of Web services are jointly used to realize more complex functionality – typically a business process. A “business process” specifies the potential execution order of operations from a collection of Web services, the data shared between these Web services, which partners are involved and how they are involved in the business process, joint exception handling for collections of Web services etc. In particular, the capability of support for long-running transactions between Web services increases consistency and reliability for Web services applications. Breaking opaqueness of Web services means specifying usage constraints of operations of a collection of Web services and their joint behavior – this is obviously very similar to specifying business processes.

Business Process Execution Language for Web Services [5] (BPEL4WS or BPEL for short) allows specifying business processes and how they relate to Web services. This includes specifying how a business process makes use of Web services to achieve its goal, and it includes specifying Web services that are provided by a business process. Business processes specified in BPEL are fully executable and they are portable between BPEL conformant environments. A BPEL business process interoperates with the Web services of its partners, whether these Web services are realized based on BPEL or not. Finally, BPEL supports the specification of business protocols between partners and views on complex internal business processes.

BPEL combines WSFL [6] and XLANG [7], superseding the corresponding specifications. The first version BPEL4WS V. 1.0. has been published in August 2002, a second version BPEL4WS V. 1.1. in May 2003 as input for the standardization within OASIS. The appropriate technical committee [8] is working since the time of submission and has given itself the charter to complete a first version of the standard by middle of 2004.

2 A First Look

The simple business process sketched in the figure helps illustrating the basic elements of BPEL. A travel agent specifies a business process that supports the travel agent in managing airline ticket requests that are requested from customer by sending in an itinerary. After the itinerary has been received, the airline is contacted with an appropriate ticket request. Then the process waits until the airline sends the tickets.



For simplicity of the business process, it is assumed that the tickets will be picked up by the customer. BPEL does not specify the graphical representation of business processes; appropriate methods for visually representing business processes are currently being developed; one of them is Business Process Modeling Notation (BPMN) [9].

This simple process is defined via BPEL as shown in lines 1 to 46. The travel agent gives the business process the name `ticketOrder` (line 1). The different tasks include the receiving of the itinerary (lines 23 to 29), passing the customer's itinerary to an airline requesting corresponding tickets (lines 30 to 37), and finally receiving the requested tickets from the airline (lines 38 to 44). For simplicity of the business process, it is assumed that the tickets will be picked up by the customer in person.

The set of relationships with partners that the agent's process maintains are defined in lines 2 to 11: Lines 3 to 6 introduce the relationship with the partner "customer", and lines 7 to 10 introduce the relationship with the partner "airline". A partner link identifies a relationship between a process and a partner and specifies the Web services mutually used by the partner or process, respectively (see section 3 for more details).

The messages that are persisted by the process are called "variables" (line 12 to 17). Variables are WSDL messages that are typically received from or sent to partners (see section 5 for more details). For example, the process stores an `itineraryMessage` as an `itinerary` variable. The `itineraryMessage` is received from the customer (line 23) when the customer uses the `sendItinerary` operation of the processes `itinerary` port type (lines 25 and 26). This message is stored into the `itinerary` variable (line 27) once received. When the process passes on the `itinerary` message to the airline (line 30) by using the `requestTicket` operation of the `ticketOrder` port type (lines 32 and 33) offered by the airline, this message is a copy of the `itinerary` variable (line 34).

The usage of an operation in a business process is called an "activity" (see section 5 for more details). To define the order in which the activities have to be performed, the `ticketOrder` process structures its activities as a `flow` (line 18): A flow is a directed graph with the activities as nodes and so-called links as edges connecting the activities. The links required to define the flow between different `ticketOrder` process' activities are specified in lines 19 to 22. The activities then specify whether they are the source or the target of one or more links defined via a link. For example, the `receive` activity of line 23 is the source of the `order-to-airline` link (line 20) with the `invoke` activity of line 30 being the target (line 35).

```
1 <process name="ticketOrder">
2   <partnerLinks>
3     <partnerLink name="customer"
4       partnerLinkType="agentLink"
5       myRole="agentService"
6       partnerRole="customer"/>
7     <partnerLink name="airline"
8       partnerLinkType="buyerLink"
9       myRole="ticketRequester"
```

```

10         partnerRole="ticketService"/>
11     </partnerLinks>

12     <variables>
13         <variable name="itinerary"
14             messageType="itineraryMessage"/>
15         <variable name="tickets"
16             messageType="ticketsMessage"/>
17     </variables>

18     <flow>

19         <links>
20             <link name="order-to-airline"/>
21             <link name="airline-to-agent"/>
22         </links>

23         <receive name="processItinerary" ,
24             partnerLink="customer"
25             portType="itineraryPT"
26             operation="sendItinerary"
27             variable="itinerary"
28             <source linkName"order-to-airline"/>
29         </receive>

30         <invoke name="contactAirline" ,
31             partnerLink="airline"
32             portType="ticketOrderPT"
33             operation="requestTickets"
34             variable="itinerary">
35             <target linkName"order-to-airline"/>
36             <source linkName"airline-to-agent"/>
37         </invoke>

38         <receive name="receiveTickets" ,
39             partnerLink="airline"
40             portType="itineraryPT"
41             operation="sendTickets"
42             variable="tickets"
43             <target linkName"airline-to-agent"/>
44         </receive>

45     </flow>

46 </process>

```

The interactions between the partners in the travel agents process are different for the interactions with the customer and the interactions with the airline. In the case of the customer, the customer uses the `sendItinerary` operation on the `itineraryPT` port type provided by the process; this request is then processed by the `<receive>` activity in line 23. No response is being sent back to the customer. In the case of the airline, the process uses the `requestTickets` operation on the `ticketOrderPT` port type offered by the airline to send a request to the airline (lines 30 to 37). The airline sends its response back by using the `sendTickets` operation on the `itineraryPT` port type, which is processed by the process via the appropriate `<receive>` activity (lines 38 to 44).

3 Partners

As already shown in the travel agent example, business processes that involve Web services often interact with different partners. Partners are connected to a process in a bilateral manner called “partner link type”. A partner link type specifies two port types that are mutually provided and required by the two connected partners; i.e. each partner provides one of the port types. These port types are referred to as “roles”. Here is the definition for the partner link type between the process and the airline:

```
47 <partnerLinkType name="buyerLink">
48   <role name="ticketRequester">
49     <portType name="itineraryPT"/>
50   </role>
51   <role name="ticketService">
52     <portType name="ticketOrderPT"/>
53   </role>
54 </partnerLinkType>
```

The partner link type `buyerLink` consists of two roles. The role `ticketRequester` (line 48 to 50) provides a port of port type `itineraryPT` (line 49), and the role `ticketService` (lines 51 to 53) provides a port of port type `ticketOrderPT` (line 52). The port types are defined somewhere else in appropriate WSDL definitions. When defining a partner within a business process a reference to the partner link type underlying the corresponding bilateral relation between the process and the partner is made (see lines 3 and 7). For example, the `airline` partner link in the travel agent process refers to the `buyerLink` partner link type defined in lines 47 to 54. A partner link definition further specifies which role of the underlying partner link type the process itself accepts (“`myRole`”) and which role has to be accepted by the partner (“`partnerRole`”). Accepting a role comes with the obligation to provide the corresponding Web services, i.e. to provide an implementation of the port types of the role. The Web services that are expected by the process from the partner are referenced by the `partnerRole` attribute (e.g. line 10) and the Web services provided by the process and that the partner can rely on and use are referred to by the `myRole` attribute (e.g. line 9).

In other words, the process defines via the `myRole` construct the Web service that represents itself to the outside world; the `partnerRole` construct allows specifying the dependencies of a business process on Web services provided by the outside, i.e. the Web services the business process require and will use.

Multiple partners that implement the same partner link type can be defined in a business process by defining each partner via a separate partner link as shown in the following BPEL fragment.

```
55 <partnerLink name="airline1"  
56     partnerLinkType="buyerLink"  
57     myRole="ticketRequester"  
58     partnerRole="ticketService"/>  
59 <partnerLink name="airline2"  
60     partnerLinkType="buyerLink"  
61     myRole="ticketRequester"  
62     partnerRole="ticketService"/>
```

This would allow the travel agent process to communicate with two different airlines at the same time using the same operations and port types.

4 Variables, Properties, and Correlations

Business processes specified via BPEL prescribe the exchange of messages between Web services. These messages are WSDL messages of operations of the port types associated with the roles of the partner links established between the process and its partners. Some of the messages exchanged may be included in the so-called “business context” of the business process. This context is a collection of WSDL messages called “variables” that represent data that is important for the correct execution of the business process, e.g. for routing decisions to be made or for the construction of messages to be sent.

For example, line 27 specifies that the message received from the `customer` via the `sendItinerary` operation of the process’ `itineraryPT` port type has to be copied to the `itinerary` variable. And line 34 specifies that the message sent to the `airline`’s `ticketOrderPT` port type as input of the `requestTickets` operation stems from the `itinerary` variable.

Often, the business context is stored persistently to avoid loss of the context, thus, ensuring the correct execution of a business process even in case of planned or unplanned system outages. As the likelihood of such outages increases with the lifetime of a business process, and business processes are typically lasting for long time periods, it is a good practice to make the context persistent.

When messages are exchanged between business partners they typically carry some data that is used to correlate a message with the appropriate business process. For example, the `ticketsMessage` may carry an `orderNumber` that is used by the travel agent and the airline to identify the purchase of tickets for a submitted itinerary of a specific customer; that means it allows the travel agent and the airline to correlate a received message with a particular business process. This kind of correlation data is referred to as “property” in BPEL. Very often the same property is used within different messages as data to be used for correlation. For this purpose, BPEL supports the definition of properties as separate entities. The following BPEL fragment defines the `orderNumber` as a property:

```
63 <property name="orderNumber" type="xsd:int"/>
```

Because a property is used by different messages as correlation data, a mechanism is needed that allows identifying the appropriate field within the message that represents this property. In BPEL, this mechanism is called “aliasing”. The following example shows how the `orderNumber` property (line 64) is defined to be the `orderID` field of the `orderInfo` part of the `ticketsMessage`.

```
64 <propertyAlias propertyName="orderNumber"  
65     messageType="ticketsMessage"  
66     part="orderInfo"  
67     query="/orderID"/>
```

5 Activities

Activities are the actions that are being carried out within a business process. The travel agent process already showed some of the activities that can be used within a business process, such as `<receive>`, `<invoke>`, or `<flow>`.

An important action in a business process is to simply wait for a message to be received from a partner. This kind of action is specified via a `<receive>` activity. It identifies the partner from which the message is to be received, as well as the port type and operation provided by the process used by the partner to pass the message (lines 23 to 27).

A more powerful mechanism is provided by the `<pick>` activity. This kind of activity specifies a whole set of messages that can be received from the same or different partners. Whenever one of the specified messages is received, the `<pick>` activity is completed, and processing of the business process continues. Additionally, one may specify that processing should continue if no message is received in a given time. The following BPEL snippet replaces the `<receive>` activity (lines 38 to 44) that waits for the response from the airline with a `<pick>` activity.

```
68 <pick>
69   <onMessage partnerLink="buyerLink"
70             portType="itineraryPT"
71             operation="sendTickets"
72             variable="tickets" >
73     <empty/>
74   </onMessage>
75   <onAlarm for="P1DT" >
76     <invoke partnerLink="customer"
77            portType="travelPT"
78            operation="answerRequest"
79            variable="unableToHonorRequest" />
80   </onAlarm>
81   <target linkName="airline-to-agent" />
82 </pick>
```

The `<onMessage>` element (line 69) is used to define receiving a particular message from a partner via a port type and operation that the process provides. Thus the structure of the `<onMessage>` specification is the same as for a `<receive>` activity; the only difference is the mandatory specification of an enclosed activity that is being carried out when the message has been received. As there is nothing to do when the airline responds, the `<empty>` activity has been chosen.

The `<onAlarm>` element (line 75) is used to specify that the activity should wait for some time or until a specified period in time has been reached. If none of the specified messages has been received when the alarm goes off, the enclosed activity is being carried out. The example specifies that the alarm should go off one day after the `<pick>` activity has started. If this happens, the customer is informed, that the travel agent is unable to handle the customer request.

The start activities of a business process must be `<receive>` or `<pick>` activities. Flagging them with `createInstance="yes"` (lines 87 and 93) indicates that an instance of the specified business process should be created if none exists already. The following illustrates this behavior using a business process that needs to accept the requests from two different partners. The sequence in which the appropriate messages arrive is unclear.

```
83 <receive partnerLink="hotel",
84         portType="roomPT",
85         operation="sendBooking",
86         variable="stayInfo"
87         createInstance="yes"/>
88
89 <receive partnerLink="rentalCar",
90         portType="carPT",
91         operation="sendBooking",
92         variable="rentalInfo"
93         createInstance="yes"/>
```

Regardless which message arrives first, a process instance is created. After the initial message the business process waits for the second one. For example, if the first message is received from a `hotel` partner, a process instance is created and then the business process waits for the message to arrive from a `rentalCar` partner.

This approach eliminates the need to have explicit life cycle commands, for example a command to create a process instance. Having no explicit life cycle commands makes life very easy for the requestors of Web services that represent business processes: There is no need to know whether a process instance has already been created or not. As a result, requestors can interact with Web services representing business processes as with any other Web service.

As already pointed out earlier, the travel agent process does not send a response back to the customer; however in most practical cases a response must be returned. As illustrated in the following example, the `<reply>` activity is used to specify a synchronous response to the request corresponding to a `<receive>` activity.

```
94 <receive partnerLink="customer",
95         portType="itineraryPT",
```

```

96         operation="sendItinerary",
97         variable="itinerary"
98         createInstance="yes"/>
99
100    <reply    partnerLink="customer",
101           portType="travelPT",
102           operation="sendTickets",
103           variable="tickets"/>

```

In this example, the process provides an in-out operation: The input message of this operation is consumed by the `<receive>` activity, and the output message of this operation is produced via the `<reply>` activity.

If the response to the original request is to be sent asynchronously, the response is delivered via the invocation of a Web service provided by the requester. Consequently, the `<invoke>` activity is used within the process that produces the asynchronous response. The original requester will use a `<receive>` activity to consume the response delivered by the `<invoke>` activity.

Furthermore, the `<invoke>` activity can be used within a process to synchronously invoke an in-out operation of a Web service provided by a partner. As shown in the following example, the `<invoke>` activity needs to identify an input as well as an output variable.

```

104    <invoke  partnerLink="airline"
105           portType="ticketOrderPT"
106           operation="requestTickets"
107           inputVariable="itinerary"
108           outputVariable="tickets"/>

```

All activities discussed so far (except `<pick>`), are called “simple activities” indicating that they have no structure and do not allow to enclose other activities. Other simple activities, called “command” activities for obvious reasons, are: `<wait>` that indicates that the business process should wait for a specified time period or until a specified point in time has been reached, `<empty>` which has no action associated and serves as a means to specify that nothing should be done or to synchronize parallel processing within the process, `<terminate>` to indicate the business process should be terminated immediately, `<throw>` to signal the occurrence of an error, `<assign>` to copy fields from variables into other variables, and `<compensate>` to undo the effects of already completed activities (see section 7).

The travel agent process showed the usage of `<flow>`, one of the two most important structured activities. It allows defining sets of activities (including other flow activities) that are wired together via `<link>`s, providing for the potential parallel execution of parts of the flow. Each link may be associated with a transition condition, which is a Boolean expression using values in the different variables of the process. When the business process is being carried out, a particular link is being followed when the associated transition condition evaluates to true.

Other structured activities are: `<sequence>` that causes the enclosed activities to be carried out in the order they are listed, `<switch>` to have one path selected out of many paths using selection criteria that references values in containers, and `<while>` that causes the enclosed activities to be carried out as long as the condition associated with the while-activity evaluates to true.

6 Scopes

In the previous section, `<flow>` has been identified as one of the most important structured activities. The other one is `<scope>` which allows building groups of activities and assign certain characteristics to the group of activities. There are no limitations to the type of activities that are enclosed in a scope. The process by default is a scope. A scope has the following characteristics:

```
109     <scope variableAccessSerializable="yes|no" >
110
111         <variables>
112             ...
113         </variables>
114
115         <faultHandlers>
116             ...
117         </faultHandlers>
118
119         <compensationHandlers>
120             ...
121         </compensationHandlers>
122
123         <eventHandlers>
124             ...
125         </eventHandlers>
126
127         activity
128
129     </scope>
```

The `<variableAccessSerializable>` property controls how two parallel scopes access variables that are defined outside the individual scopes. When set to yes, access to the variables are serialized. This means when the first scope accesses such a variable that is accessed by both scopes, processing of the second scope is suspended until the first scope has completed processing of the last variable that is accessed by both scopes.

Scopes can have their local variables identified via the `<variables>` element. Only activities within the scope have access to those variables. If a variable with the same name exists in an outer scope, the local variable is used when the name of the variable is used inside the scope.

BPEL processes interact with WSDL ports and such ports may send fault messages back to the process. Furthermore, a process itself might detect erroneous situations that result in internal faults. BPEL provides mechanisms that allow trying to recover from such faulty situations. Central to these mechanisms are so-called “fault handlers” that can catch and deal with faults. They are identified via the `<faultHandlers>` element. A more detailed discussion is provided in the following section 7.

In the process of correcting faults, previously completed activities or set of activities need to be undone. This is the purpose of compensation handlers identified via the `<compensationHandlers>` element. A compensation handler can contain any kind of activity (simple or structured).

When BPEL processes are being carried out, the individual activities interact with partners only at appropriately defined activities. However, in many cases it is important that requests from partners can be accepted at any time or when attached to a scope just as long as the process is running within the scope. This is defined by establishing event handlers, identified via the `<eventHandlers>` element. Event handlers are further discussed in section 8.

As identified via activity in line 127, a scope can contain a single activity; which may be either simple or structured. If structured, the activity may contain another `<scope>` activity as shown in the following example; thus scopes may be nested.

```
130 <scope>
131   <flow>
132     <scope>
133       ...
134     </scope>
135     ...
136   </flow>
137 </scope>
```

7 Fault and Compensation Handlers

A fault handler (lines 138 to 145) defines the set of faults it attempts to handle via a corresponding set of `<catch>` elements (line 139). Within such an element any kind of activity (simple or structured) may be nested. This activity will be performed when the corresponding fault occurs. In the example below, the fault handler catches a `noSeatsAvailable` fault returned by an airline partner. When this fault occurs a corresponding rejection message is sent to the customer via the nested `<invoke>` activity (lines 140 to 143).

```
138 <faultHandlers>
139   <catch faultName="noSeatsAvailable">
140     <invoke partnerLink="customer"
141           portType="sendItinerary"
142           operation="sendRejection"
143           inputVariable="rejection"/>
144   </catch>
145 </faultHandlers>
```

When a fault occurs within a scope, the regular processing within the scope is interrupted and the signaled fault is passed to the catching fault handler. The activity nested within this fault handler tries to correct the situation such that regular processing can continue outside the scope or alternate ways to complete the process can be taken.

All of this might require undoing actions that have already been completed within the scope. For example, if the tickets required for a trip are not available, already made reservations for hotel rooms or rental cars must be canceled. The actions required to undo already completed activities are defined via compensation handlers. That means, a fault handler of a scope may make use of compensation handlers to undo actions performed within this scope. It does so via a `<compensate>` activity. The `<compensate>` activity may reference a particular scope (inside the scope that faulted) which causes the compensation handler of the scope to be carried out. If no scope is specified, the appropriate compensation handlers are invoked in the reverse order of execution of the scopes. If the exceptional situation cannot be corrected, the fault handler will re-throw the fault or signal the occurrence of another fault, which will be finally caught by a fault handler of another enclosing scope.

Thus, BPEL allows via its scope mechanism the definition of sets of activities that can be collectively undone in erroneous situations. I.e. such a set of activities is some sort of unit of work, some sort of transaction: Activities that are performed within a scope either all complete or are all compensated [10]. In contrast to this, the well-known “traditional” transactions (like database transactions) are implemented based on locks, i.e. allocating resources to a particular transaction for the duration of the transaction. This takes for granted that transactions are short-lived units of work such that locks can be release fast. Because BPEL scopes are typically long running locking resources doesn’t work in practice but one has to use compensation actions instead. This allows releasing locks once an enclosed activity completes, but one has to run compensation logic to undo already completed actions. The resulting units of work or transactions are referred to as “long running transactions”.

Long running transactions in BPEL are centered on scopes, and scopes can be nested. There is an agreement protocol between a scope and its parent scope to determine the outcome of the long running transaction represented by a scope. The corresponding protocol has been described in WS-Transaction [12]. While BPEL long-running transactions are currently assuming that a scope and all its nested scopes are contained within a single process and are hosted by a single BPEL engine, the agreement protocol in WS-Transaction does not assume this. Thus, a future extension of BPEL may support long running transactions that are distributed across processes and even across BPEL engines.

WS-Transaction also specifies protocols for coordinating distributed atomic transactions. A future extension of BPEL may support distributed atomic transactions consisting of activities of a single process or even of different processes.

8 Event Handlers

The purpose of event handlers is to carry out some processing that is not part of the main part of the business process. An event handler is activated when the control flow enters the scope the event handler is attached to or if the event handler is associated with the process, when the process is started. An event handler is deactivated when the control flow leaves the scope the event handler is attached to or when the process finishes in case the event handler is associated with the process.

The event handlers shown in the following example are attached to the process is used to terminate the process if either an appropriate message is being received from the customer or the process is running already for two days.

```
146 <eventHandlers>
147   <onMessage partnerLink="customer"
148     portType="itineraryPT"
149     operation="cancel"
150     variable="cancelMessage">
151 <reply partnerLink="customer"
152   portType="itineraryPT"
```

```

153         operation="cancel"
154         variable="cancelAcceptedMessage" />
155     <terminate/>
156 </onMessage>
157 <onAlarm for="P1DT">
158     <invoke partnerLink="customer"
159           portType="travelPT"
160           operation="request"
161           variable="unableToHonorRequest" />
162     <terminate/>
163 </onAlarm>
164 </eventHandlers>

```

The first event handler, identified via the `<onMessage>` element (line 147) is carried out when a cancel request is received from the customer as operation `cancel` on port type `itineraryPT`, which is defined as an in-out operation. When the request is received, the customer is informed via an appropriate `<reply>` operation on the receiving port type and operation indicating that the request has been received and is being processed. The `<terminate>` activity causes the termination of the process.

Such a message-based event handler is being carried out, whenever a message is received. If a message arrives when the event handler is being carried out, a new instance of the event handler is created.

The second event handler, identified via the `<onAlarm>` element (line 157) is carried out when the process has been executing for a day. In this case, the customer is informed that the request can not be processed and the process is terminated.

9 Process-based Applications

Applications created with BPEL are so-called “process-based applications” [13], [14]. This kind of application structure split an application into two strictly separated layers: The top layer, the business process, is written in BPEL and represents the flow logic of the application, whereas the bottom layer, the Web services, represents the function logic of the application.

This structure has several advantages over more conventional approaches: (1) the underlying business process as well as the invoked Web services can be changed without any impact on the other Web services within the application or on the Web services that the business process represents, (2) the application can be developed and tested in two separate stages: the business process is developed and tested independent from the development and test of the individual Web services. This approach provides for great flexibility in changing the application. These advantages have been recognized by the UML community too; especially, mappings from UML to BPEL and corresponding relations to model driven architecture (MDA) are on their way [15].

Applications written in BPEL have another major advantage over conventional approaches as they allow tailoring the ready application to the needs and circumstances of a particular environment without touching the application itself. This is achieved by separating the definition of the partners that a business process deals with from the characteristics of the actually involved partners. Within BPEL, one specifies only the port types and operations the different partners are expected to provide.

When such a business process is being carried out, the information about the actual ports or Web services that a concrete partner chosen provides, need to be available. The information about the Web services or ports is collectively subsumed in BPEL under the notion of a “service reference”. Concrete mechanisms of providing service references for the different partners within BPEL have been deliberately left out of the specification (aside from a few exceptions). One of the exceptions deals with the situation that a requestor provides the provider with its own service reference so that the provider can respond back to the requestor.

The typical approach for providing service references is to provide this information when the business process is installed (“deployed”) in the form of a deployment descriptor. Assigning a service reference to a partner comes in many flavors. In the simplest approach a partner would be assigned a service reference containing fixed information. When the business process is being carried out, this fixed service reference is used to invoke the Web service. In the most complex case, the deployment information could just point to some mechanism, that when the business process is being carried out, determines the appropriate service reference, and possibly invokes the selected Web service right away. This mechanism could, for example, go to UDDI, get all the detail information about potential service providers, and then based on that information selects the most appropriate service provider.

Applications created based on BPEL are portable between environments supporting BPEL and Web services: The BPEL processes can be executed by any BPEL engine, and during their execution a BPEL engine will interact with the Web services that are discovered based on the deployment information.

Besides using BPEL for specifying executable processes, BPEL can be used for specifying “business protocols”. A business protocol specifies the potential sequencing of messages exchanged by one particular partner with its other partners to achieve a business goal. I.e. a business protocol defines the ordering in which a particular partner sends messages to and expects messages from its partners based on actual business context. An example for business protocols is the RosettaNet PIPs (see [16]).

Typically, the messages exchanged result from performing activities within internal business processes. Thus, a business protocol may be perceived as a view on a private business process: Internal details like access to backend systems, complete structure of the messages making up the context, complex data manipulation steps, business rules determining branch selection etc are omitted from such a view.

In BPEL the language for specifying business protocols is a subset of the language used for specifying executable processes. This enables to specify an internal executable process together with its views within the same language. It supports an outside-in approach starting with a view and extending it into an internal process, as well as an inside-out approach starting with an internal process projecting it onto its views.

In general, a business protocol (or view, respectively) is not executable: For example, the messages making up the context may be a simple projection of the real internal context messages, it may not be completely specified how messages are constructed that are sent to a partner, branching conditions may not be defined precisely in terms of the data making up the visible context of the business protocol. This is resulting from the fact that a business protocol hides internal details and complexity by will.

Because a business protocol may neither be executable nor deterministic but still expressed as a process, BPEL refers to it as “abstract process”: It abstracts away complex details of an internal executable process. In this sense, abstract processes may be perceived as simple or easy to comprehend processes. And while an abstract process is not guaranteed to be executable, abstract process can be easily specified in a manner such that they are in fact executable! This allows beginning with simple variants of a process and refining them iteratively into the final complex business process.

Finally, an abstract process may be used to easily specify constraints on the usage patterns of a collection of port types: The port types to be constrained are the port types provided by the abstract process, and the operations that are to be constrained are used within activities of the abstract process.

10 Summary

BPEL supports the specification of a broad spectrum of business processes: From fully executable complex business processes over more simple business protocols to usage constraints of Web services. It provides a long-running transaction model that allows increasing consistency and reliability of Web services applications. Correlation mechanisms are supported that allow identifying stateful instances of business processes based on business properties. Partners and Web services can be dynamically bound based on service references

References

- [1] W3C, Web Services Definition Language (WSDL) 1.1. <http://www.w3.org/TR/wsd.html>
- [2] OASIS, Universal Description, Discovery & Integration. <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3>
- [3] W3C, SOAP Version 1.2. <http://www.w3.org/TR/SOAP12>
- [4] F. Leymann, Web Services: Distributed Applications without Limits, Proc. BTW'03 (Leipzig, Germany, February 26-28, 2003), Springer 2003.

- [5] BEA Systems, IBM Corporation, Microsoft Corporation., SAP AG, Siebel Systems. Business Process Execution Language for Web Services.
<http://www.ibm.com/developerworks/webservices/library/ws-bpel>
- [6] F. Leymann, Web Services Flow Language (WSFL 1.0), IBM Corporation.
<http://www-4.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf>
- [7] S. Thatte, XLANG, Microsoft Corporation.
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c
- [8] OASIS Web Services Business Process Execution Language TC.
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.
- [9] BPMI.org, Business Process Modeling Notation, <http://www.bpmi.org>
- [10] F. Leymann, Supporting business transactions via partial backward recovery in workflow management systems, Proc. BTW'95 (Dresden, Germany, March 22-24, 1995), Springer 1995.
- [11] BEA Systems, IBM Corporation, Microsoft Corporation. Web Services Coordination (WS-Coordination).
<http://www.ibm.com/developerworks/webservices/library/ws-coor>.
- [12] BEA Systems, IBM Corporation, Microsoft Corporation. Web Services Transaction (WS-Transaction).
<http://www.ibm.com/developerworks/webservices/library/ws-transpec>.
- [13] F. Leymann and D. Roller, Workflow-based applications, IBM Systems Journal Vol.26 No. 1., 1997.
- [14] F. Leymann and D. Roller, Production Workflow: Concepts and Techniques, Prentice Hall, 2000.
- [15] T. Gardner, UML Modeling of Automated Business Processes with a Mapping to BPEL4WS,
<http://www.cs.ucl.ac.uk/staff/g.piccinelli/eoows/documents/paper-gardner.pdf>
- [16] RosettaNet.
<http://www.rosettanel.org>