Chapter VIII
Service-Oriented Processes: An Introduction to BPEL

Chun Ouyang
Queensland University of Technology, Australia

Wil M.P. van der Aalst
Eindhoven University of Technology, The Netherlands and Queensland University of Technology, Australia

Marlon Dumas
Queensland University of Technology, Australia

Arthur H.M. ter Hofstede
Queensland University of Technology, Australia

Marcello La Rosa
Queensland University of Technology, Australia

ABSTRACT

The Business Process Execution Language for Web Services (BPEL) is an emerging standard for specifying the behaviour of Web services at different levels of details using business process modeling constructs. It represents a convergence between Web services and business process technology. This chapter introduces the main concepts and constructs of BPEL and illustrates them by means of a comprehensive example. In addition, the chapter reviews some perceived limitations of BPEL and discusses proposals to address these limitations. The chapter also considers the possibility of applying formal methods and Semantic Web technology to support the rigorous development of service-oriented processes using BPEL.

INTRODUCTION

Web services are a standardised technology for building and integrating distributed software systems. Web services are an incarnation of a software development paradigm known as Service-Oriented Architectures (SOAs). Although there is no broad consensus around the definition of SOAs, it can be said that SOAs revolve around at least three major principles: (1) software systems are functionally decomposed into independently developed and maintained software entities (known as "services"); (2) services interact through the exchange of messages containing meta-data; and (3) the interactions in which services can or should engage are explicitly described in the form of interfaces.

At present, the first generation of Web service technology has reached a certain level of maturity and is experiencing increasing levels of adoption, especially in the context of business applications. This first generation relies on XML, SOAP and a number of so-called WS-* specifications for message exchange (Curbera, Duftler, Khalaf, Nagy, Mukhi, & Weerawarana, 2002), and on XML Schema and WSDL for interface description. In the meantime, a second generation of Web services, based on richer service descriptions is gestating. Whereas in first-generation Web services, interface descriptions are usually equated to sets of operations and message types, in the second generation the description of behavioural dependencies between service interactions (e.g., the order in which messages must be exchanged) plays a central role.

The Business Process Execution Language for Web Services (BEA Systems, Microsoft, IBM, & SAP, 2003), known as BPEL4WS or BPEL for short, is emerging as a standard for describing the behaviour of Web services at different levels of abstraction. BPEL is essentially a layer on top of WSDL and XML Schema, with WSDL and XML Schema defining the structural aspects of service interactions, and BPEL defining the behavioural aspects. To capture service behaviour, BPEL adopts principles from business process modeling. Indeed, the central idea of BPEL is to capture the business logic and behavioural interface of services in terms of process models. These models may be expressed at different levels of abstraction, down to the executable level. At the executable level, BPEL can be used to describe the entire behaviour of a new Web service that relies on several other services to deliver its functionality. This practice is known as service composition (Casati & Shan, 2001). An example of a composite service is a travel booking system integrating flight booking, accommodation booking, travel insurance, and car rental Web services.

In this chapter, we introduce BPEL by illustrating its key concepts and the usage of its constructs to define service-oriented processes and to model business protocols between interacting Web services. We also discuss some perceived limitations of BPEL and extensions that have been proposed by industry vendors to address these limitations. Finally, we review some research related to BPEL and conclude with a note on future directions.

WHY BPEL?

BPEL supports the specification of service-oriented processes, that is, processes in which each elementary step is either an internal action performed by a Web service or a communication action performed by a Web service (sending and/or receiving a message). They can be executed to implement a new Web service as a concrete aggregation of existing services to deliver its functionality (i.e. composite Web service). For example, a service-oriented process may specify that a when a "Sales" Web service receives a "purchase order" from the "Procurement" Web service of a customer, the Sales service engages in a number of interactions with several the "Procurement" Web service as well as several other

Web services related to invoicing, stock control, and logistics, in order to fulfil the order.

BPEL draws upon concepts and constructs from imperative programming languages including: (1) lexically scoped variables; (2) variable assignment; (3) sequential execution; (4) conditional branching; (5) structured loops; and (6) exception handling (try-catch blocks). However, BPEL extends this basic set of constructs with other constructs related to Web services and business process management, to address the following aspects:

- Messaging: BPEL provides primitive constructs for message exchange (i.e., send, receive, send/receive).
- Concurrency: To deal with concurrency between messages sent and received, BPEL incorporates constructs such as block-structured parallel execution, race conditions, and event-action rules.
- XML typing: To deal with the XML-intensive nature of Web services, BPEL variables have XML types described in WSDL and XML Schema. In addition, expressions may be written in XML-specific languages such as XPath or XSLT.

BPEL process definitions can be either fully executable or they can be left underspecified. Executable BPEL process definitions are intended to be deployed into an execution engine. This deployment results in a new Web service being exposed, which usually relies upon and coordinates several other Web services. This is why BPEL is sometimes referred to as a language for "Web service composition." Meanwhile, underspecified BPEL definitions (called *abstract processes*) capture a non-executable set of interactions between a given (Web) service and several other services. One possible usage of abstract BPEL processes is as a means for specifying the order in which the interactions (or "operations") that a given service supports should occur for the service to deliver its functionality. Such specification of dependencies between interactions is usually called a *business protocol*. Coming back to the above purchase order process, one may capture the fact that an interaction "request for quote" must precede a related interaction "place order." A business protocol can be used for process monitoring, conformance checking and analysis of service compatibility. Importantly, a Web service whose business protocol is described as an abstract BPEL process does not need to be implemented as an executable BPEL process: it may very well be implemented using any other technology (e.g., standard J2EE or .Net libraries and extensions for Web service development). Another usage of an abstract process is as a template of a BPEL process that needs to be refined into an executable implementation. The use of BPEL abstract processes as business protocols or as templates is still the subject of ongoing research as discussed in the Section "BPEL-Related Research Efforts". At present, commercial BPEL technology is mainly focused on fully executable BPEL processes.

To further understand the reason for the emergence of BPEL, it is interesting to view it from a historical perspective. Since 2000 there has been a growing interest in Web services. This resulted in a stack of Internet standards (HTTP, XML, SOAP, WSDL, and UDDI) which needed to be complemented by a process layer. Several vendors proposed competing languages, for example,, IBM proposed WSFL (Web Services Flow Language) (Leymann, 2001) building on FlowMark/MQ-Series and Microsoft proposed XLANG (Web Services for Business Process Design) (Thatte, 2001) building on Biztalk. BPEL emerged as a compromise between both languages, superseding the corresponding specifications. It combines accordingly the features of a block-structured language inherited from XLANG with those for directed graphs originating from WSFL. The first version of BPEL (1.0) has been published in August 2002, and the second version (1.1) has been released in May 2003 as input for the standardization within OASIS. The appropriate technical committee (OASIS Web Services Business Process Execution Language TC, 2006) is working since the time of submission and is in the process of finalizing the appropriate standard specification, namely Web Services Business Process Execution Language (WS-BPEL) version 2.0 (OASIS, 2005).

Currently BPEL is implemented in a variety of tools (see http://en.wikipedia.org/wiki/ BPEL for a compendium). Systems such as BEA WebLogic, IBM WebSphere, Microsoft BizTalk, SAP XI and Oracle BPEL Process Manager support BPEL to various degrees, thus illustrating the practical relevance of this language. Also, there is a relatively complete open-source implementation of BPEL, namely ActiveBPEL.

OVERVIEW OF BPEL

BPEL defines a model and a grammar for describing the behaviour of a business process based on interactions between the process and its partners. A BPEL process is composed of activities that can be combined through structured operators and related through so-called control links. In addition to the main process flow, BPEL provides event handling, fault handling and compensation (i.e., "undo") capabilities. In the long-running business processes, BPEL applies correlation mechanism to route messages to the correct process instance.

BPEL is layered on top of several XML specifications: WSDL, XML Schema and XPath. WSDL message types and XML Schema type definitions provide the data model used in BPEL processes. XPath provides support for data manipulation. All external resources and partners are represented as WSDL services.

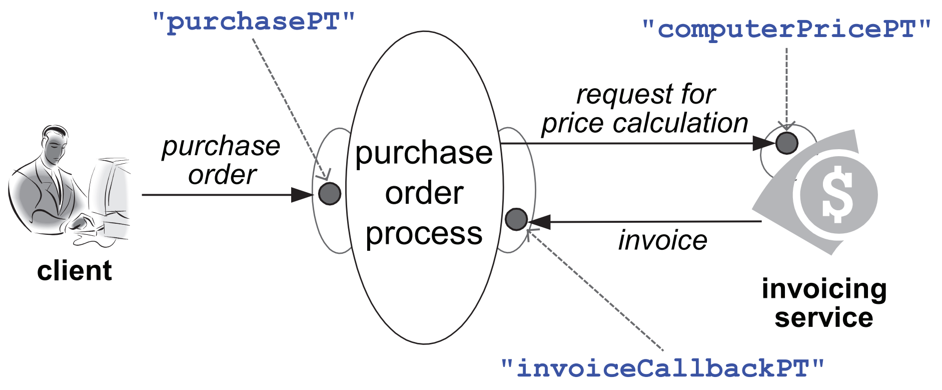
Partners and Partner Links

Business processes that involve Web services often interact with different *partners*. The interaction with each partner occurs through Web service interfaces called *port types*, and the structure of the relationship at the interface level is identified by a *partner link*. A partner link specifies which port type must be supported by each of the partners it connects, and which port type it offers to each partner. A partner link is an instance of a typed connector, known as a *partner link type*, which specifies a set of *roles* and the port type provided by each role.

Consider a simple purchase order process which interacts with two partners: the client and the invoicing service. Each interaction involves one of the parties (i.e., the process or partner) exposing the required functionality via a port type, and the other party making use of that functionality. Figure 1 briefly depicts this process and the interactions between the process and each of its two partners.

For the above process, two partner links are created, namely "purchasing" and "invoicing". Figure 2 shows the XML code snippets defining these two partner links and their link types "purchasingPLT" and "invoicingPLT." The link type "purchasingPLT" includes the definition of a role "purchaseService" (played by the purchase order process) referring to port type "purchasePT" where a "purchase order" is received by the process. Similarly, the link type "invoicingPLT" is defined featuring two roles: "invoiceService" (played by the invoicing service) referring to port type "computerPricePT" where the operation of a "request for price calculation" is called, and "invoiceRequester" (played by the purchase order process) referring to port type "invoiceCallbackPT" where the invoice is received by the process. Following common practice, we define the partner link types in the WSDL document to which the BPEL process definition refers. Meanwhile, the partner links themselves are defined in the BPEL process definition.

Figure 1. A purchase order process interacting with two partners



Activities

A BPEL process definition relates a number of *activities*. Activities are split into two categories: basic and structured activities. *Basic activities* are also called *primitive activities*. They correspond to atomic actions and stand for work being performed within a process. *Structured activities* impose behavioural and execution constraints on a set of activities contained within them. Structured activities can be nested and combined in arbitrary ways, thus enabling the presentation of complex structures.

Basic activities. These contain: *invoke*, invoking an operation on some Web service; *receive*, waiting for a message from an external partner; *reply*, replying to an external partner; *wait*, pausing for a certain period of time; *assign*, copying data from one place to another; *throw*, indicating errors in the execution; *compensate*, undoing the effects of already completed activities; *exit*, terminating the entire service instance; and *empty*, doing nothing. Below, we look closer into three activities: *invoke*, *receive*, and *reply*.

Invoke, *receive*, and *reply* activities are three types of interaction activities defined in BPEL. Interaction activities must specify the partner link through which the interaction occurs, the operation involved, the port type in the partner

Figure 2. Definition of the "purchasing" and "invoicing" partner links and their types

WSDL snippet:

```
...
<partnerLinkType name="purchasingPLT">
  <role name="purchaseService">
    <portType name="purchasePT"/>
  </role>
</partnerLinkType>

<partnerLinkType name="invoicingPLT">
  <role name="invoiceService">
    <portType name="computePricePT"/>
  </role>
  <role name="invoiceRequester">
    <portType name="invoiceCallbackPT"/>
  </role>
</partnerLinkType>
...

BPEL snippet:
...
<partnerLinks>
  <partnerLink name="purchasing"
          partnerLinkType="purchasingPLT"
          myRole="purchaseService"/>
  <partnerLink name="invoicing"
          partnerLinkType="invoicingPLT"
          myRole="invoiceRequester"
          partnerRole="invoiceService"/>
</partnerLinks>
...
```

link that is being used, and the input and/or output variables that will be read from or written to. Note that variables are used to carry data (see Subsection on "Data Handling") and are required only in executable processes.

For an *invoke* activity, the operation and port type that are specified are that of the service being invoked. Such an operation can be a synchronous "request-response" or an asynchronous "one-way" operation. An *invoke* activity blocks to wait for a response if it is calling a request-response operation, whereas in the case of a one-way operation, *invoke* can be viewed as a "send" action. A synchronous invocation requires both an input variable and an output variable. An asynchronous invocation requires only the input variable of the operation because it does not expect a response as part of the operation. For example, in the purchase order process shown in Figure 1, the process initiates a price calculation service by sending a purchase order to the invoicing service. Figure 3 provides the XML definition of this *invoke* activity, which calls a one-way operation "initiatePriceCalculation".

A business process provides services to its partners through *receive* activities and corresponding *reply* activities. A *receive* activity allows the process to block and wait for a matching message to arrive, while a *reply* activity is used to send a response to a request that was previously accepted via a *receive* activity. Such responses are only meaningful for synchronous interactions. Therefore, a pair of *receive* and *reply* activities

Figure 3. An invoke activity for initiating a price calculation service

```
<invoke partnerLink="invoicing"
        portType="computePricePT"
        operation="initiatePriceCalculation"
        inputVariable="PurchaseOrder"/>
```

must map to a request-response operation. In such case, any control flow between these two activities is effectively the implementation of that operation. A *receive* with no corresponding *reply* must map to a one-way operation, and the asynchronous response is always sent by invoking the same one-way operation on the same partner link.

A *receive* activity specifies the partner link it expects to receive from, and the port type and operation that it expects the partner to invoke. In addition, it may specify a variable used to receive the message data being expected. A *receive* activity also plays a key role in the lifecycle of a business process. It is always associated with a specific attribute called *createInstance* with a value of "yes" or "no". The default value of this attribute is "no". A *receive* activity with the *createInstance* attribute set to "yes" must be an initial activity in a business process, which provides the only way to instantiate the process in BPEL (see structured activity *pick* for a variant). A *reply* activity shares the same partner link, port type and operation as the corresponding *receive* activity, but may specify a variable that contains the message data to be sent as the response.

Let's revisit the purchase order process in Figure 1. A process instance is instantiated upon receiving a purchase order from the client, and may be completed by replying to the client with an invoice listing the price for that purchase order. Figure 4 provides the XML definition of the corresponding pair of *receive* and *reply* activities over a request-response operation named "sendPurchaseOrder". In the same process, there is another *receive* activity referring to a one-way operation "sendInvoice". It is used for receiving the invoice produced by the invoicing service. The process defines this activity as the response to the price calculation request sent by the process before (see the *invoke* activity defined in Figure 3). Figure 5 gives the XML definition of this *receive* activity.

Before moving onto structured activities, it is worth mentioning the following two restrictions

Figure 4. An initial receive activity for receiving a purchase order from the client and the corresponding reply activity for replying with an invoice for the order

```
<receive partnerLink="purchasing"
         portType="purchasePT"
         operation="sendPurchaseOrder"
         variable="PurchaseOrder"
         createInstance="yes"/>
...

<reply partnerLink="purchasing"
       portType="purchasePT"
       operation="sendPurchaseOrder"
       variable="Invoice"/>
```

Figure 5. A receive activity for receiving an invoice from the invoicing service

```
<receive partnerLink="invoicing"
         portType="invoiceCallbackPT"
         operation="sendInvoice"
         variable="Invoice"/>
```

that BPEL applies on the above three interaction activities:

- First, BPEL does *not* allow two *receive* activities to be active (i.e., ready to consume messages) at the same time if they have the same partner link, port type, operation, and *correlation set* which is used for routing messages to process instances (see Subsection on "Correlation"). If this happens, a built-in fault named "conflictingReceive" will be raised at runtime.
- Second, BPEL does *not* allow a request to call a request-response operation if an active *receive* is found to consume that request, but a *reply* has not yet been sent to a previous request with the same operation, partner link, and correlation set. If this happens, a built-in fault named "conflictingRequest" will be thrown.

Figure 6. A sequence of activities performed in the purchase order process in Figure 1

```
begin sequence
   receive PurchaseOrder from client;
   invoke  PriceCalculation on invoicing service;
   receive Invoice from invoicing service;
   reply Invoice to client
end sequence
```

Structured activities. BPEL defines six structured activities: *sequence*, *switch, pick*, *while*, *flow*, and *scope*. The use of these activities and their combinations enable BPEL to support most of the workflow patterns described in (Aalst, van der, Hofstede, ter, Kiepuszewski, & Barros, 2003).

A *sequence* activity contains one or more activities that are performed sequentially. It starts once the first activity in the sequence starts, and completes if the last activity in the sequence completes. For example, Figure 6 defines a sequence of activities performed within the purchase order process shown in Figure 1. To improve readability, this and the following code snippets do not use XML syntax. Instead, BPEL element names are written in bold while the level of nestings of elements is captured through indentation.

A *switch* activity supports conditional routing between activities. It contains an ordered list of one or more conditional branches called *case* branches. The conditions of branches are evaluated in order. Only the activity of the first branch whose condition holds true will be taken. There is also a default branch called *otherwise* branch, which follows the list of *case* branches. The *otherwise* branch will be selected if no *case* branch is taken. This ensures that there is always one branch taken in a switch activity. The switch activity completes when the activity of the selected branch completes. For example, consider a supply-chain process which interacts with a buyer and a seller. Assume that the buyer has ordered a volume of 100 items of a certain product. The process needs to check the stock inventory before fulfilment. If the result shows more than 100 items of the product in stock, the process performs the fulfilment work (which may contain a number of activities); if the result shows less than 100 items in stock, a fault is thrown indicating the product is out of stock; otherwise (i.e., no items are in stock), another fault is thrown signalling the product is discontinued. Figure 7 shows how to use a *switch* construct to model these activities.

A *pick* activity captures race conditions based on timing or external triggers. It has a set of branches in the form of an event followed by an activity, and exactly one of the branches is selected upon the occurrence of the event associated with it. If more than one of the events occurs, the selection of the activity to perform depends on which event occurred first. If the events occur almost simultaneously, there is a race and the choice of activity to be performed depends on both timing and implementation. There are two types of events: message events (*onMessage*) which occur upon the arrival of an external message, and alarm events (*onAlarm*) which occur upon a system timeout.

Note that *onMessage* is a *receive*-like construct and is thereby treated in much the same manner as a *receive* activity, for example,, both are used for process instantiation, share the same attributes, and should not violate the constraint on "conflictingReceive." A *pick* activity completes when one of the branches is triggered by the occurrence of its associated event and the corresponding activity completes. Figure 8 shows an example of a typical usage of *pick* for modeling the order entry/completion within a supply-chain process. There are three events: a line item message event whose occurrence will trigger an order entry action; a completion message event whose occurrence will trigger an order completion action; and an alarm event which will occur after a period of 3 days and 10 hours and thus trigger a timeout action.

Figure 7. A switch activity modeling stock inventory check in a supply-chain process

```
begin switch
   case StockResult >100 : perform fulfillment work
   case StockResult > 0 : throw OutOfStock fault
   otherwise : throw ItemDiscoutinued fault
end switch
```

Figure 8. A pick activity modeling order entry/completion in a supply-chain process

```
begin pick
   onMessage LineItem : add line item to order
   onMessage CompletionDetail : perform order
completion
   onAlarm for 'P3DT10H' : handle timeout for
order completion
end pick
```

Figure 9. A while activity modeling a loop of the pick activity defined in Figure 8

```
While MoreOrderEntriesExpected = true
   begin pick
      onMessage LineItem : add line item to order
      onMessage CompletionDetail :
          begin sequence
              perform order completion;
              MoreOrderEntriesExpected := false
          end sequence
      onAlarm for 'P3DT10H' :
          begin sequence
              handle timeout for order completion;
              MoreOrderEntriesExpected := false
          end sequence
   end pick
```

A *while* activity supports repeated performance of an activity in a structured loop, that is, a loop with one entry point and one exit point. The iterative activity is performed until the specified *while condition* (a boolean expression) no longer holds true. For example, the *pick* activity defined in Figure 8 can occur in a loop where the seller is accepting line items for a large order from the buyer. Figure 9 shows how this loop can be specified using a *while* activity. The *pick* activity nested within *while* can be repeated until no more order entries are expected.

A *flow* activity provides parallel execution and synchronization of activities. It also supports the usage of *control link*s for imposing further control dependencies between the activities nested within it. Control links are non-structural constructs in BPEL and will be covered in more detail in the next subsection. Figure 10 shows an example of the simple usage of *flow* construct as equivalent to a nested concurrency construct. In this example, a supply-chain process sends questionnaires to the buyer and seller in parallel, and then blocks to wait for their responses. After both have replied, the process continues to next task (e.g., to generate an evaluation report). In Figure 10, the two *invoke* activities are enabled to start concurrently as soon as the *flow* starts. Assume that both *invoke* activities refer to synchronous request-response operations. The *flow* is completed after the buyer and the seller have both responded.

A *scope* is a special type of structured activity. It is used for grouping activities into blocks, and each block is treated as a unit to which the same event and exception handling can be applied. A *scope* has a primary activity (i.e. main activity) that defines its normal behaviour, and can provide *event handlers*, *fault handlers*, and also a *compensation handler*. Like other structured activities, scopes can be nested to arbitrary depth, and the whole process is implicitly regarded as the top level scope. The usage of *scope* will be discussed in detail further on in Subsections on "Event Handlers", "Fault Handling" and "Compensation".

Control Links

The *sequence*, *flow*, *switch*, *pick,* and *while* described in the previous subsection provide a means of expressing structured flow dependencies. In addition to these constructs, BPEL provides another construct known as *control link*s which, together with the associated notions of *join condition* and *transition condition*, support the definition of precedence, synchronization and conditional dependencies on top of those captured by the structured activities.

A *control link* denotes a conditional transition between two activities. A *join condition*, which is associated to an activity, is a boolean expression in terms of the tokens carried by incoming control links to this activity. Each token, which represents the status of the corresponding control link, may take either a positive (true) or a negative (false) value. For example, a control link between activities A and B indicates that B cannot start before A has either completed or has been skipped (e.g., A is part of an unselected branch of a *switch* or *pick*). Moreover, activity B can only be executed if its associated join condition evaluates to true, otherwise B will not run. A *transition condition*, which is associated to a control link, is a boolean expression over the process variables (just like

Figure 10. A flow activity modeling two concurrent questionnaire interactions in a supply-chain process

```
begin flow
   invoke FillQuestionnaire (request-response) operation on the buyer
   invoke FillQuestionnaire (request-response) operation on the seller
end flow
...
```

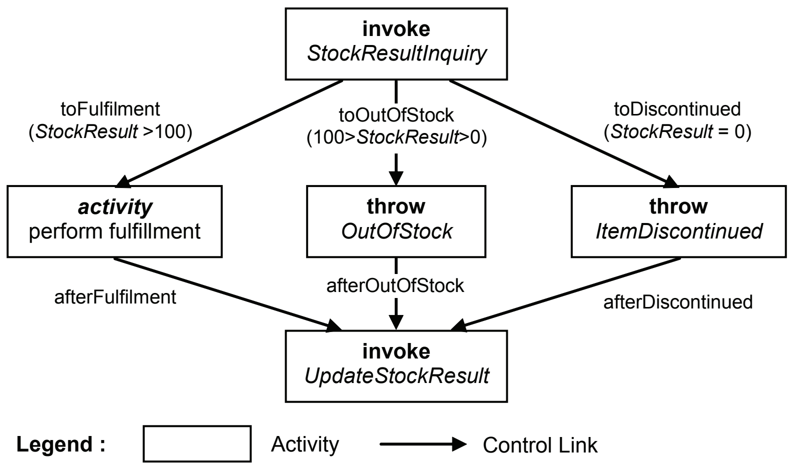
the conditions in a *switch* activity). For example, an activity X propagates a token with a positive value along an outgoing control link L, if and only if X was executed (as opposed to being skipped) and the transition condition associated to L evaluates to true.

As mentioned above, if an activity has incoming control links, one of the enabling conditions for this activity to run is that its associated join condition evaluates to true. Otherwise, a fault called *join failure* occurs. When a join failure occurs at an activity, it can be handled in two different ways as determined by the *suppressJoin-Failure* attribute associated with the activity. This attribute can be set to a value of "yes" or "no". If "yes," it instructs to suppress the join failure. In this case, the activity will be skipped, and the tokens carried by all its outgoing links will take a negative value. The process by which positive and negative tokens are propagated along control links, causing activities to be executed or skipped, is called *dead path elimination*. Otherwise, if the *suppressJoinFailure* is set to "no," the join failure is thrown, which triggers the standard fault handling procedure (see Subsection on "Fault Handling").

Control links are non-structural constructs defined in BPEL, and allow the definition of directed graphs. However, it is important to mention two link restrictions. First, control links must not create cyclic graphs. Second, control links must not cross the boundary of a loop (i.e., a *while* activity) as that would lead to an inconsistent state.

We revisit the example of the stock inventory check within a supply-chain process. This has been previously specified by a *switch* activity shown in Figure 7. Below, we use control links to replace the *switch* construct for the modeling. For completeness, we add two activities: one is that the process inquires the stock result from the seller before the inventory check *switch* activity, and the other is that the process informs the seller about the updated stock result after the *switch* activity. Using structured constructs, the above two activities, together with the previous *switch* activity for stock inventory check, can be specified in a *sequence* construct. Using control link constructs, we obtain a directed graph representation shown in Figure 11. Figure 12 sketches the definition of the corresponding abstract BPEL specification.

Figure 11. A directed graph representing the stock inventory check procedure within a supply-chain process


invoke StockResultInquiry
toFulfilment (StockResult >100)
toOutOfStock (100>StockResult>0)
toDiscontinued (StockResult = 0)
activity perform fulfillment
throw OutOfStock
throw ItemDiscontinued
afterFulfilment
afterOutOfStock
afterDiscontinued
invoke UpdateStockResult
Legend :
Activity
Control Link

From the above example, it can be observed that any control link leaving an unexecuted activity or whose transition condition evaluates to false will have its link status set to false. As a result, each control link will propagate either a true or a false token so that the activities downstream which have a control dependency on the link do not end up waiting for ever. This is the mechanism of dead path elimination that we have mentioned before. Also note that, as a syntactical restriction in BPEL, control links must be used within a *flow* construct.

Event Handlers

The purpose of *event handlers* is to specify logic to deal with events that take place concurrently while the process is running. An event handler is an event-action rule associated with a scope, and is in the form of an event followed by an activity.

Figure 12. Using control links to model the stock inventory check procedure sketched in Figure 11

```
begin flow (suppressJoinFailure =”yes”)
   begin link declaration
      link “toFulfillment”
      link “toOutOfStock”
      link “toDiscontinued”
      link “afterFulfillment”
      link “afterOutOfStock”
      link “afterDiscontinued”
   end link declaration
   invoke StockResultQuery (request-response) operation on the seller
      source of link “toFulfillment” with
         transitionCondition (StockResult >100)
      source of link “toOutOfStock” with
         transitionCondition (StockResult > 0 and StockResult <100)
      source of link “toDiscontinued” with
         transitionCondition (StockResult = 0)
   activity : performing fulfillment work
      joinCondition LinkStatus(“toFulfillment”)
      target of link “toFulfillment”
      source of link “afterFulfillment”
         transitionCondition (true)
   throw OutOfStock fault
      joinCondition LinkStatus(“toOutOfStock”)
      target of link “toOutOfStock”
      source of link “afterOutOfStock”
         transitionCondition (true)
   throw ItemDiscoutinued fault
      joinCondition LinkStatus(“toDiscontinued”)
      target of link “toDiscontinued”
      source of link “afterDiscontinued”
         transitionCondition (true)
   invoke StockResultUpdate (one-way) operation on the seller
      joinCondition LinkStatus(“afterFulfillment”) or
                    LinkStatus(“afterOutOfStock”) or
                    LinkStatus(“afterDiscontinued”)
      target of link “afterFulfillment”
      target of link “afterOutOfStock”
      target of link “afterDiscontinued”
end flow
```

An event handler is enabled when its associated scope is under execution and may execute concurrently with the main activity of the scope. When an occurrence of the event associated with an enabled event handler is registered, the activity within the handler is executed while the scope's main activity continues its execution. Also, the activity within an event handler is invoked concurrently when the corresponding event occurs. For this reason, control links are not allowed to cross the boundary of an event handler.

It is important to emphasize that event handlers are part of the normal behaviour of a scope, unlike fault and compensation handlers (see Subsections on "Fault Handling" and "Compensation"). The event handlers associated with a scope are enabled when that scope commences, and are disabled when the normal processing of the scope is complete. Any event handler that has already started is allowed to finish its execution. An entire scope is not considered to complete until all event handlers associated with the scope have finished their executions.

BPEL allows any type of activity, except the *compensate* activity, to handle events. There are two types of events. One is the message events (*onEvent*) triggered by the arrival of an external message, the other is the alarm events (*onAlarm*) triggered by an alarm which goes off after a user-specified time.

Message event handlers. The semantics of *onEvent* message events is very similar to *receive* activities, except that these message events cannot create process instances. An event handler is not enabled prior to the creation of a process instance, and is capable of processing events only if an instance has been created. The message that triggers an event is identified by the partner link from which the message arrives, the appropriate port type, operation and optional variable and correlation set. This message can be part of either an asynchronous (one-way) or a synchronous (request-response) operation. In the latter case, the event handling logic is expected to have a *reply* activity, in order to fulfil the requirements of the operation.

When a message event is triggered, the activity specified within the corresponding message event handler is carried out. Message event handlers remain active as long as the scope to which they are attached is active. An active message event handler can be triggered multiple times, even simultaneously, if the expected message events occur multiple times. However, it should be noted that simultaneously active instances of a message event handler is permitted, while the semantics of simultaneous *onEvent* from the same partner, port type, operation and with the same correlation set are undefined. The reader may recall that *receive* activities have a similar constraint.

Alarm event handlers. An *onAlarm* event marks a system timeout. It has two alternative attributes: *for* and *until*, and exactly one of them must be specified. These two attributes determine two forms of alarm events. The first specifies duration within *for* attribute. In this form, a timer for calculating the duration is started when the associated scope is activated. As soon as the specified duration is reached, the activity in the corresponding event handler is executed. In the second form, *until* attribute details a specific point in time when the alarm will be fired. As soon as this specific point in time is reached, the alarm event is triggered and the corresponding event handler is executed. It should be noted that, unlike message events, alarm events can be processed at most once while the associate scope is active.

Let's revisit the purchase order process shown in Figure 1. The process may terminate its execution if either a cancel message is received from the client or the process has been running already for two days in processing a purchase order from the client. In the latter case, the process will reply to the client with a cancel message (instead of an invoice). The two event handlers defined in Figure 13 are used to implement the above two scenarios when the sequence activity defined in Figure 6 is running.

Figure 13. Examples of the message and alarm event handlers used for terminating the purchase order process shown in Figure 1

```
begin scope
   onEvent Cancel from client : exit
   onAlarm for 'P2DT' :
      begin sequence
         reply Cancel to client;
         exit
      end sequence
   (* sequence activity defined in Figure 6 *)
end scope
```

Fault Handling

Fault handling in a business process enables the process to recover locally from possible anticipated faults that may arise during its execution. For example, consider a fault caused by insufficient funds in the client's account for payment during a purchase order process. The fault may be handled by requesting the information of another available account from the client, without having to restart the entire process.

BPEL considers three types of faults. These are: *application faults* (or *service faults*), which are generated by services invoked by the process, such as communication failures; *process-defined faults*, which are explicitly generated by the process using the *throw* activity; and *system faults*, which are generated by the process engine, such as "conflictingReceive," "conflictingRequest" and join failures introduced before. Note that the first two types of faults are usually user-defined, while the last one consists of built-in faults defined in BPEL.

Fault handlers specify reactions to internal or external faults that occur during the execution of a scope, and are defined for a scope using *catch* activities. Unlike event handlers, fault handlers do not execute concurrently with the scope's main activity. Instead, this main activity is interrupted before the body of the fault handler is executed. In more detail, if a fault occurs during the normal process of a scope, it will be caught by one of the fault handlers defined for the scope. The scope switches from the normal processing mode to the fault handling mode. Note that it is never possible to run more than one fault handler for the same scope under any circumstances.

A fault handler is defined either explicitly or implicitly. An implicit fault handler is also known as a default fault handler. It is created, using a *catch*-all activity, to catch any fault that is not caught by all explicit fault handlers within the scope. Therefore, one can assume that each scope has at least one (default) fault handler. If a fault handler cannot handle a fault being caught or another fault occurs during the fault handling, both faults need to be re-thrown to the (parent) scope that directly encloses the current scope. A scope in which a fault has occurred is considered to have ended abnormally and thus cannot be compensated, no matter whether or not the fault can be handled successfully (without being re-thrown) by the corresponding fault handler.

Finally, control links may cross the boundary of fault handlers. However, a control link is only allowed to *leave* the boundary of a fault handler, and the converse is not true. Also, if a fault occurred within a scope has been handled successfully, any control link leaving from that scope will be evaluated normally.

Let's refer back to the stock inventory check specified in Figure 7. Figure 14 shows two fault handlers used to catch the two faults that may occur during the inventory check. If a fault occurs indicating the product is out of stock, the process invokes the order pending operation on the buyer. Otherwise, if a fault occurs indicating the product discontinued, the process invokes the order rejection operation on the buyer.

Figure 14. Examples of the fault handlers for catching the corresponding faults occurred during the stock inventory check defined in Figure 7

```
begin scope
   catch OutOfStock fault :
      invoke OrderPending operation on the buyer
   catch ItemDiscontinued fault :
      invoke OrderRejection operation on the buyer
   (* switch activity defined in Figure 7 *)
end scope
```

Compensation

As part of the exception handling, compensation refers to application-specific activities that attempt to undo the already completed actions. For example, consider a client requests to cancel the air ticket reservation with a ticket order process. The process will need to carry out the following compensation actions, which involve the cancellation of the reservation with the airline, and optionally the conduction of fee charges to the client if there are fees applied for the cancellation of a reservation.

In BPEL, compensation actions are specified within a compensation handler. Each scope, except the top level scope (i.e. process scope), provides one compensation handler that is defined either explicitly or implicitly. Similarly to a default fault handler, an implicit (or default) compensation handler is created for a scope, if the scope is asked for compensation but an explicit compensation handler is missing for that scope. A fault handler or the compensation handler of a given scope, may perform a *compensate* activity to invoke the compensation of one of the sub-scopes nested within the given scope. Similarly to the control link restrictions applied to event handlers, control links are not allowed to cross the boundary of compensation handlers.

It is important to mention that whether the compensation handler of a scope is available for invocation depends on the current local state of the scope. For example, it is not possible to conduct the compensation of a scope that has never been executed. BPEL uses a term "*scope snapshot*" to refer to the preserved state of a successfully completed uncompensated scope. In such state, the data to which the scope has access is snapshotted for use when the associated compensation handler is running. Thus, the compensation handler of a scope is available for invocation only if the scope has a scope snapshot. Otherwise, invoking a compensation handler that is unavailable is equivalent to performing an empty activity. Since the compensation of already completed activities is a complex procedure, we decide not to include an example here and the interested reader may refer to (BEA Systems, Microsoft, IBM, & SAP, 2003) for more details.

Data Handling

In the previous subsections, we mainly focus on the control logic of a BPEL process. Careful readers may already notice that the process data is necessary for the process logic to make data-driven decisions (e.g., in a *switch* activity). In the following, we introduce how data is represented and manipulated in BPEL.

Messages. Business protocols specified in BPEL prescribe exchange of *messages* between interacting Web services. These messages are WSDL messages defined in the appropriate WSDL definitions. Briefly, a message consists of a set of named *parts*, and each of these parts is typed generally using XML Schema. For example, in Figure 15 below, the *orderMsg* is shown with three message parts: an *orderNumber* of an integer type, an *orderDetails* of a string type, and a *timeStamp* of a dateTime type. Note that the integer, string and dateTime are all simple XML Schema types. If a complex XML Schema type is needed, it needs to be defined in the corresponding XML Schema file (see Section on "BPEL At Work").

Figure 15. Example of a WSDL message definition

```
<message name="orderMsg"/>
  <part name="orderNumber"
type="integer"/>
  <part name="orderDetails"
type="string"/>
  <part name="timeStamp"
type="dateTime"/>
</message>
```

Variables. In a BPEL process definition, variables are used to hold messages exchanged between the process and its partners as well as internal data that is private to the process. Variables are typed, using WSDL message types, XML Schema simple types or XML Schema elements. Note that if a variable is of WSDL message type, it also consists of a set of named parts (each of which as specified in a *part* attribute). For example, in Figure 16 both *order* and *order_backup* variables are defined as of the *orderMsg* type above, and the *number* variable is defined as of an integer type.

Each variable is declared within a scope and is said to belong to that scope. Variables that belong to the global process scope are called *global variables*, while others are called *local variables*. BPEL follows the same rules as those in imperative programming languages with lexical scoping of variables. A variable is visible in the scope (e.g., namely Q) to which it belongs and all scopes that are nested within Q. Thus, a global variable is visible in the entire process. Also, it is possible to hide a variable in a scope (Q) by defining a variable with the same name in one of the scopes nested in Q.

Expressions. BPEL supports four types of expressions: (1) *boolean-valued* expressions for specifying transition conditions, join conditions, and conditions in *switch* or *while* activities; (2) *deadline-valued* expressions for specifying *until* attribute of *onAlarm* or a *wait* activity; (3)

Figure 16. Examples of variable definitions in BPEL

```
<variables>
  <variable name="order"
messageType="orderMsg"/>
  <variable name="order_backup"
messageType="orderMsg"/>
  <variable name="number"
type="integer"/>
</variables>
```

Figure 17. Examples of expressions used in BPEL

```
... bpws:getLinkStatus('linkL2') ...
... bpws:getVariableData('order','orderNumber')>50
...
... until="'2006-01-31T18:00'" ...
... for="'P40D'" ...
```

duration-valued expressions for specifying *for* attribute of *onAlarm* or a *wait* activity; and (4) general expressions for assignments (see next). BPEL provides an extensible mechanism for specifying the language used to define expressions. This language must have facilities such as to query data from variables, to query the status of control links, and so forth XPath 1.0 is the default language for specifying expressions. Another language that can be used is XSLT. Figure 17 gives four examples of expressions used in BPEL. The first two are both boolean-valued expressions: one indicating the status of a control link, the other indicating whether the *orderNumber* of an *order* message is greater than a given number (e.g. 50). The third one is a deadline-valued expression, and the last one is a duration-based expression.

Assignments. Data can be copied from one variable to another using the *assign* activity. An assign activity may consist of a number of assignments, each of which being defined by a *copy* element with *from* and *to* attributes. The source of the copy (specified by *from* attribute) and the target (specified by *to* attribute) must be type-compatible. BPEL provides a complete set of possible types of assignments. For example, within common uses of assignment, the source of the copy can be a variable, a part of a variable, an XPath expression, and the target of the copy can be a variable or a part of a variable. Figure 18 illustrates copying data from one variable (*order*) to another (*order_backup*) as well as copying data from a variable part (*orderNumber* part of *order*) to a variable of compatible type (*number*), and both assignments are defined within one *assign* activity.

Figure 18. Examples of assignments used in BPEL

```
<assign>
  <copy>
    <from variable="order"/>
    <to variable="order_backup"/>
  </copy>
  <copy>
    <from variable="order"
part="orderNumber"/>
    <to variable="number"/>
  </copy>
</assign>
```

Correlation

Business processes may in practice occur over a long period of time, possibly days or months. In long-running business processes, it is necessary to route messages to the correct process instance. For example, when a request is issued from a partner, it is necessary to identify whether a new business process should be instantiated or the request should be directed to an existing process instance. Instead of using the concept of instance ID as often used in distributed object system, BPEL reuses the information that can be identified from the specifically marked parts in incoming messages, such as *order number* or *client id*, to route messages to existing instances of a business process. This mechanism is known as *correlation*. The concept of *correlation set* is then defined by naming specific combinations of certain parts in the messages within a process. This set can be used in *receive*, *reply* and *invoke* activities, the *onMessage* branch of *pick* activities, and the *onEvent* handlers.

Similarly to variables, each correlation set is defined within a scope. Global correlation sets are declared in the process scope, and local correlation sets are declared in the scopes nested within a process. Correlation sets are only visible for the scope (Q) in which they are declared and all scopes nested in Q. Also, correlation set can only be initiated once during the lifetime of the scope to which it belongs. How to define and use correlation sets will be illustrated through the example in the next section.

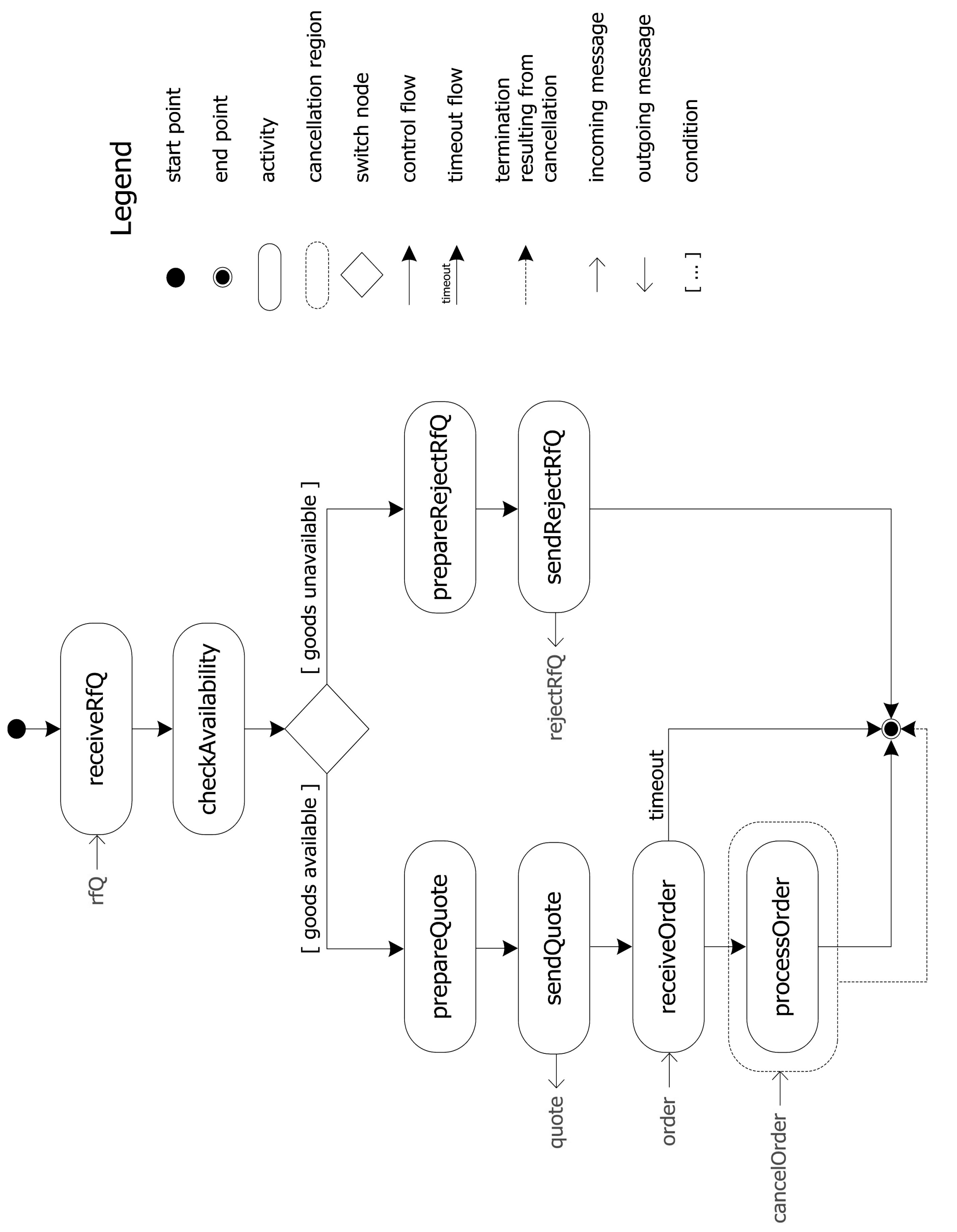
BPEL AT WORK

This section describes the example of a BPEL process which provides sales service. This process, namely salesBP, interacts with a customer process (customerBP) by means of asynchronous messages. The process salesBP enacts the role of service provider, whilst the customer is the service requester.

Process Description

Figure 19 depicts the behaviour of the process salesBP. The process is instantiated upon receiving a request for quote (*rfQ*), which includes the *description* and the *amount* of the goods needed, a unique identifier of the request (*rfQId*), and a deadline (*tO*). Next, the process checks the availability of the amount of the goods being requested. If not available, a *rejectRfQ* is sent back to the customer, providing the *reason* of the rejection. Otherwise, the process prepares a *quote* with the *cost* of the

Figure 19. Flow diagram of the salesBP process



offer and then sends it back to the customer. After sending the *quote*, the process waits for an *order* until time-limit *tO* is reached. If the *order* is not received by that time, no more activities will be performed. Otherwise, if the order is returned before the deadline *tO*, it will be processed. After the order has been processed successfully, the entire process instance will complete. However, the processing of the order may be cancelled at any time upon receiving a *cancelOrder* message from the customer, and as a result, the process will be forced to terminate.

XML Schema Definition

Figure 20 shows the XML Schema file "saleX.xsd" for the process salesBP. It defines the complex XML Schema types for messages *rfQ*, *quote*, *order*, *rejectRfQ* and *cancelOrder* that are involved in salesBP (lines 7-26). In particular, messages *order* and *cancelOrder* have the same structure as message *quote* (lines 4-6). Besides, each message includes an element named *rfQId* (lines 9, 17, 23), through whose value a BPEL-compliant execution engine is able to identify the proper process instance. In this example, this value, which is initially set by the requester (i.e., customerBP) and then propagated to the provider, is supposed to be unique.

WSDL Document

The BPEL process salesBP is layered on top of the WSDL document "sales.wsdl" shown in Figure 21. In particular, the first part of the code concerns the description of the messages exchanged by the service, and their mapping with the related elements in the salesX.xsd schema (lines 2-16). In the

Figure 20. XML Schema definition - salesX.xsd

```
01: <schema ...>
02:   <element name="rfQ" type="rfQMsgType"/>
03:   <element name="quote" type="quoteMsgType"/>
04:   <element name="order" type="quoteMsgType"/>
05:   <element name="cancelOrder" type="quoteMsgType"/>
06:   <element name="rejectRfQ" type="rejectRfQMsgType"/>
07:   <complexType name="rfQMsgType">
08:     <sequence>
09:       <element name="rfQId" type="string"/>
10:       <element name="description" type="string"/>
11:       <element name="amount" type="integer"/>
12:       <element name="tO" type="dateTime"/>
13:     </sequence>
14:   </complexType>
15:   <complexType name="quoteMsgType">
16:     <sequence>
17:       <element name="rfQId" type="string"/>
18:       <element name="cost" type="double"/>
19:     </sequence>
20:   </complexType>
21:   <complexType name="rejectRfQMsgType">
22:     <sequence>
23:       <element name="rfQId" type="string"/>
24:       <element name="reason" type="string"/>
25:     </sequence>
26:   </complexType>
27: </schema>
```

second part, two port types called “providerPT” and “requesterPT” are defined: the former groups all the input messages of the salesBP process, that is, *rfQ*, *order* and *cancelOrder* (lines 17-27), the latter groups all its output messages, that is, *quote* and *rejectRfQ* (lines 28-35). Each message is enclosed in a corresponding operation featuring the same name; each operation is asynchronous (i.e. one-way operation) since it contains only one input message.

BPEL requires the port types involved in an interaction to be included in a partner link type construct together with their corresponding roles. Therefore the partner link type “salesPLT” has been defined, featuring two roles: provider (played by salesBP) and requester (played by customerBP), assigned to port types “providerPT” and “requesterPT,” respectively (lines 36-43).

BPEL Process Definition

Figure 22 shows an excerpt from the BPEL code that defines an executable salesBP process. A BPEL process basically consists of a header, regarding the general process definition, and the process flow, which will be concretely executed by the BPEL engine.

In the initial part, a partner link named “salesPL” is defined (lines 2-5): it refers to the partner link type “salesPLT” previously declared in the WSDL file and is used to allow the process to interact with its partner (customerBP). salesPL has two roles: provider, played by the process itself (line 3), and requester, played by the customer (line 4). A list of variables is also declared, which corresponds to the messages handled by the process (lines 6-13), and a correlation set called “salesCS”, where *rfQId* (set as a value of the so-called *properties* attribute) contains the information for routing messages to the correct process instance (lines 14-16).

The process flow (lines 17-105) is basically a sequence of three activities: a *receive*, a check on some data and a *switch*, corresponding to the

Figure 21. WSDL interface document - sales.wsdl

```
01: <definitions name="sales".../>
02:   <message name="rfQMsg">
03:     <part name="payload" element="rfQ"/>
04:   </message>
05:   <message name="orderMsg">
06:     <part name="payload" element="order"/>
07:   </message>
08:   <message name="quoteMsg">
09:     <part name="payload" element="quote"/>
10:   </message>
11:   <message name="cancelOrderMsg">
12:     <part name="payload" element="cancelOrder"/>
13:   </message>
14:   <message name="rejectRfQMsg">
15:     <part name="payload" element="rejectRfQ"/>
16:   </message>
17:   <portType name="providerPT">
18:     <operation name="rfQ">
19:       <input message="rfQMsg"/>
20:     </operation>
21:     <operation name="order">
22:       <input message="orderMsg"/>
23:     </operation>
24:     <operation name="cancelOrder">
25:       <input message="cancelOrderMsg"/>
26:     </operation>
27:   </portType>
28:   <portType name="requesterPT">
29:     <operation name="quote">
30:       <input message="quoteMsg"/>
31:     </operation>
32:     <operation name="rejectRfQ">
33:       <input message="rejectRfQMsg"/>
34:     </operation>
35:   </portType>
36:   <partnerLinkType name="salesPLT">
37:     <role name="provider">
38:       <portType name="providerPT"/>
39:     </role>
40:     <role name="requester">
41:       <portType name="requesterPT"/>
42:     </role>
43:   </partnerLinkType>
44:   . . .
45: </definitions>
```

Figure 22. BPEL executable process salesBP.bpel

```
<process name="salesBP"...>
 <partnerLinks>
   <partnerLink name="salesPL" myRole="provider"
     partnerRole="requester" partnerLinkType="salesPLT"/>
 </partnerLinks>
 <variables>
   <variable name="rfQ" messageType="rfQMsg"/>
   <variable name="quote" messageType="quoteMsg"/>
   <variable name="order" messageType="orderMsg"/>
   <variable name="cancelOrder" messageType="cancelOrderMsg"/>
   <variable name="rejectRfQ" messageType="rejectRfQMsg"/>
   <variable name="goodsAvailable" type="boolean"/>
 </variables>
 <correlationSets>
   <correlationSet name="salesCS" properties="rfQId"/>
 </correlationSets>
 <sequence>
   <receive name="receiveRfQ" partnerLink="salesPL"
     portType="providerPT" operation="rfQ" variable="rfQ"
     createInstance="yes">
     <correlations>
       <correlation set="salesCS" initiate="yes"/>
     </correlations>
   </receive>
   ...checkAvailability: set variable "goodsAvailable" to true
      or false...
   <switch>
     <case condition="getVariableData('goodsAvailable')">
       <sequence>
         <assign name="prepareQuote">
          <copy>
           <from variable="rfQ" part="payload" query="/rfQ/rfQId"/>
           <to variable="quote" part="payload" query="/quote/rfQId"/>
          </copy>
          <copy>
            <from expression="120"/>
            <to variable="quote" part="payload" query="/quote/cost"/>
          </copy>
         </assign>
         <invoke name="sendQuote" partnerLink="salesPL"
           portType="requesterPT" operation="quote"
           inputVariable="quote">
           <correlations>
             <correlation set="salesCS" initiate="no".../>
           </correlations>
         </invoke>
         <pick name="receiveOrder">
           <onMessage partnerLink="salesPL" portType="providerPT"
             operation="order" variable="order">
             <correlations>
               <correlation set="salesCS" initiate="no"/>
             </correlations>
```

Figure 22. continued

```
              <scope name="processOrder_s">
                <faultHandlers>
                  <catch faultName="forcedTermination">
                    <exit/>
                  </catch>
                </faultHandlers>
                <eventHandlers>
                  <onMessage portType="sales:providerPT"
                    operation="cancelOrder" variable="cancelOrder"
                    partnerLink="salesPL">
                    <correlations>
                      <correlation set="salesCS" initiate="no"/>
                    </correlations>
                    <throw name="forcedTermination"
                      faultName="forcedTermination"/>
                  </onMessage>
                </eventHandlers>
                ...processOrder...
              </scope>
            </onMessage>
            <onAlarm
              until="getVariableData('rfQ','payload','/rfQ/tO')">
              <empty/>
            </onAlarm>
          </pick>
        </sequence>
      </case>
      <otherwise>
        <sequence>
          <assign name="prepareRejectRfQ">
            <copy>
              <from variable="rfQ" part="payload"
                query="/rfQ/rfQId"/>
              <to variable="rejectRfQ" part="payload"
                query="/rejectRfQ/rfQId"/>
            </copy>
            <copy>
              <from expression="'goods unavailable'"/>
              <to variable="rejectRfQ" part="payload"...
                query="/rejectRfQ/reason"/>
            </copy>
          </assign>
          <invoke name="sendRejectRfQ" partnerLink="salesPL"
            portType="requesterPT" operation="rejectRfQ"
            inputVariable="rejectRfQ">
            <correlations>
              <correlation set="salesCS" initiate="no".../>
            </correlations>
          </invoke>
        </sequence>
      </otherwise>
    </switch>
  </sequence>
</process>
```

main flow depicted in Figure 15. Through the first activity, receiveRfQ, a new process instance is created once the initial message *rfQ* is received (line 20). This is then copied into variable *rfQ* (line 19), and the current instance is tagged with the value read from the correlation set "salesCS" (lines 21-23). The second activity, checkAvailability (not shown in the code for simplicity), sets the boolean variable *goodsAvailable* to "true" if the amount of the goods being requested is available, otherwise to "false".

Next, the *switch* activity checks the value of *goodsAvailable* to determine whether or not to continue processing the request for quote. This corresponds to take one of the two branches in the *switch* activity (line 28).

If the goods are available, the *case* branch will be executed leading to a sequence of four sub-activities, as shown in Figure 15 under the [goods available] branch. The first activity is the *assign* prepareQuote (lines 30-39), through which salesBP can generate the *quote* message to be sent to the customer copying into variable *quote* the value of *rfQId* taken out from variable *rfQ* and setting the amount of the offer to *120* (in a real scenario, this value can be read from a proper database). Then the quote is sent by means of the following *invoke* activity, sendQuote (lines 40-46), using the same correlation set "salesCS" (lines 43-45). Now the process should be able to receive an *order* within a given time-limit: if this happens the order can be processed, otherwise the process must end. Accordingly, receiveOrder is realised with a *pick* activity specified to wait for the *order* to arrive (*onMessage* branch – lines 48-72) or for the corresponding timeout alarm to go off (*onAlarm* branch – lines 73-76). In more detail, if the *order* arrives before the timeout expires, it will be processed by the system; otherwise, the process will terminate doing nothing (specified by an *empty* activity – line 75). Assume that the *onMessage* branch is taken. The execution of the activity processOrder may be interrupted when a *cancelOrder* message arrives. For this reason, a scope activity, processOrder_s, is defined (lines 53-71), featuring a fault handler, an event handler and having processOrder as its main activity.

In particular, the event handler (lines 59-69) captures the receipt of a *cancelOrder* message during the processing of the order, and then throws a "forcedTermination" fault, which interrupts the above order processing (lines 66-67). This fault will be immediately caught by the fault handler attached to the scope (lines 54-58), and as a result, the process will be forced terminate. Note that the *exit* activity (within the fault handler – line 56) models the explicit termination, since a BPEL process automatically ends when there is nothing left to do. The timeout value which triggers the *onAlarm* branch is extrapolated from the field *tO* of variable *rfQ* (line 74).

On the other hand, if the goods are unavailable, the otherwise branch of the *switch* is executed incorporating two activities (lines 80-103), as illustrated in Figure 15 under the [goods unavailable] branch. With the first one, prepareRejectRfQ, salesBP copies into variable *rejectRfQ* the value of *rfQId* taken out from variable *rfQ*, and sets the *reason* of the rejection with the string "goods unavailable" (lines 82-94). The second activity, sendRejectRfQ allows the process to send the message *rejectRfQ* back to the customer, reporting the rejection (lines 95-101).

Now, we take closer look into the correlation mechanism used in this example. When the *rfQ* is sent by an instance of customerBP, the BPEL run-time system performs the following tasks step by step. It generates a new instance of the receiving process salesBP (line 20), reads the value of *rfQId* from the input message, initiates the corresponding correlation set "salesCS," and associates a tag with that value to the newly generated instance (lines 21-23). Next, if the amount in the request is available (line 28), the *quote* is sent back to the customer with the same correlation set as *rfQ* (lines 40-46), and hence it will be delivered to the requester process instance that previously sent the *rfQ*. When an *order* is sent by the customer, since it has the same correlation set as the *quote* and *rfQ*, it will be delivered to the process instance of salesBP that previously sent the *quote* (lines 48-52). Analogously, messages *rejectRfQ* and *cancelOrder* are sent to the correct instances of customerBP (lines 95-101) and salesBP (lines 60-65, for the corresponding receipt). Therefore in this example, from a global point of view, customerBP is the initiator for the correlation set salesCS, whilst salesBP is the follower.

BPEL Process Execution

We use Oracle BPEL Process Manager platform 10.1.2 (see http://www.oracle.com/ technology/products/ias/bpel) to edit and execute the salesBP process. Figure 23 provides a graphical view of the BPEL process definition of salesBP in Oracle JDeveloper, a BPEL editor integrated in the Oracle platform. In JDeveloper, both the code and graphical perspectives are available to the user. After compiling the source files (including "salesBP.bpel," "sales.wsdl" and "salesX.xsd"), a BPEL process can be exposed as a Web Service deployed in a compliant run-time environment. A screenshot showing a running instance of salesBP on top of the Oracle BPEL engine is depicted in Figure 24. In the running instance, the amount of the goods being requested by the customer is available and the process salesBP is waiting for an order from the customer.

Note that in the salesBP process pictured in Figures 23 and 24, activity checkAvailability has been implemented by means of a scope (checkAvailability_s) which encloses the necessary activities to interact with another partner link representing an internal Web Service. This service is responsible to check the availability of the goods and to send the result back to salesBP.

BPEL EXTENSIONS

The BPEL specification defines only the kernel of BPEL, which mainly involves the control logic of BPEL, limited definitions on the data handling and even less in the communication aspect. Given the fact that BPEL is already a very complicated language, a complete BPEL specification covering full definitions of BPEL will make the specification less maintainable and the corresponding implementation will become less manageable. For this reason, the OASIS technical committee on WS-BPEL decides to keep the scope of the current specification and allows future extensions to be made in separate documentations. So far, there have been three extensions proposed to BPEL.

BPEL-SPE. BPEL currently does not support the modularization and reuse of "fragments" of a business process. This has driven the publication of a joint proposal of *WS-BPEL Extension for Sub-Processes, known as BPEL-SPE (*Kloppmann, Koenig, Leymann, Pfau, Richayzen, Riegen, et al., 2005 September*), by two major companies involved in Web services standards: IBM and SAP. BPEL-SPE* proposes an extension to BPEL that allows for the definition of *sub-processes* which are fragments of BPEL code that can be reused within the same or across multiple BPEL processes.

BPEL4People. In practice, many business process scenarios require human user interactions. For example, it may be desirable to define which people are eligible to start a certain business process; a process may be stuck because no one has been assigned to perform a particular task; or it is not clear who should perform the task in hand. BPEL currently does not cover human user interactions. To fill in this blank, IBM and SAP have recently proposed an extension to BPEL, namely *BPEL4People* (Kloppmann, Koenig, Leymann, Pfau, Richayzen, Riegen, et al., 2005 July). *BPEL4People* mainly defines how human tasks can be implemented in a process. This can be viewed as to add (human) resource and resource

Figure 23. Oracle JDeveloper 10.1.2: Graphical view of the BPEL process salesBP

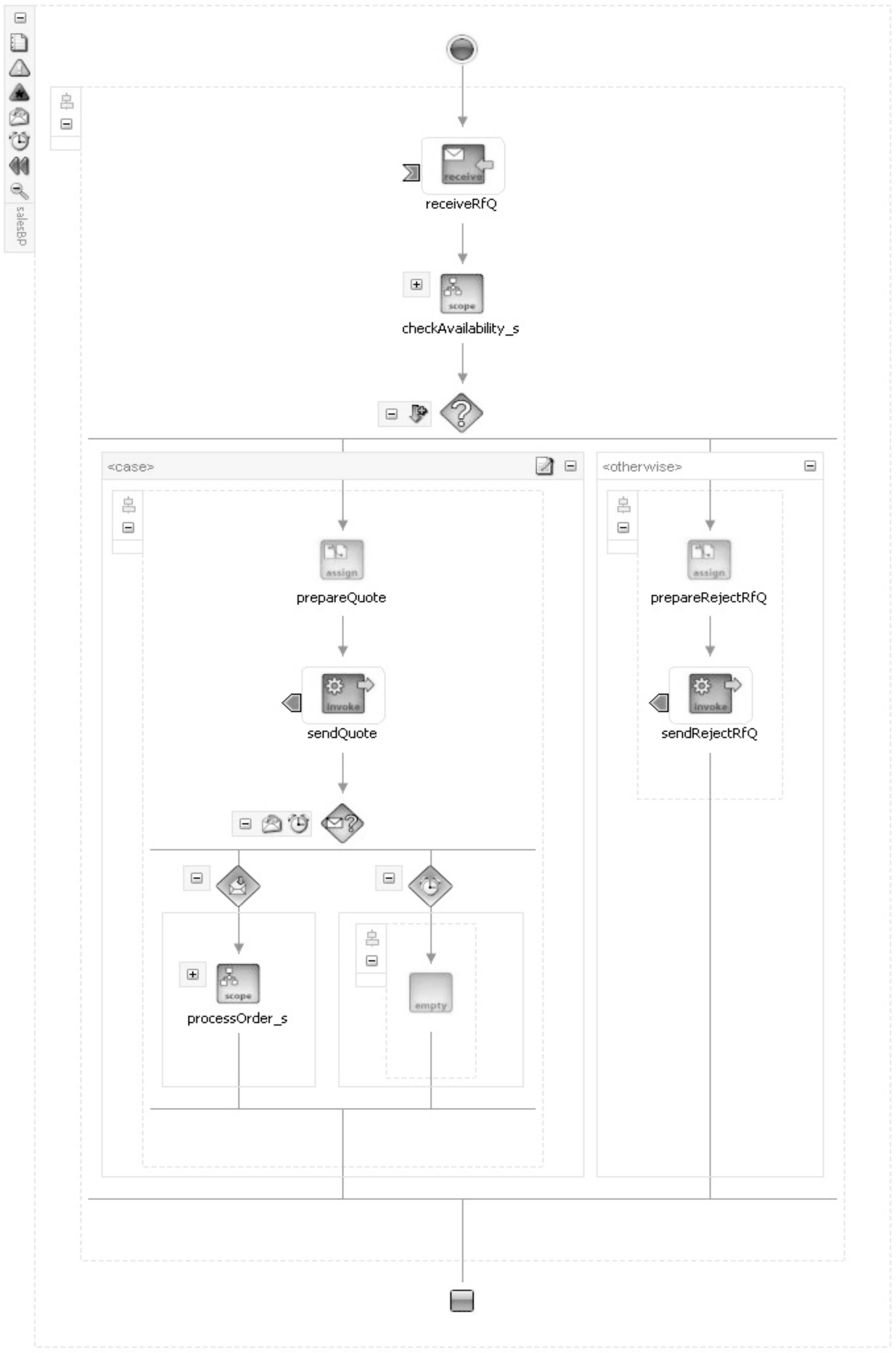
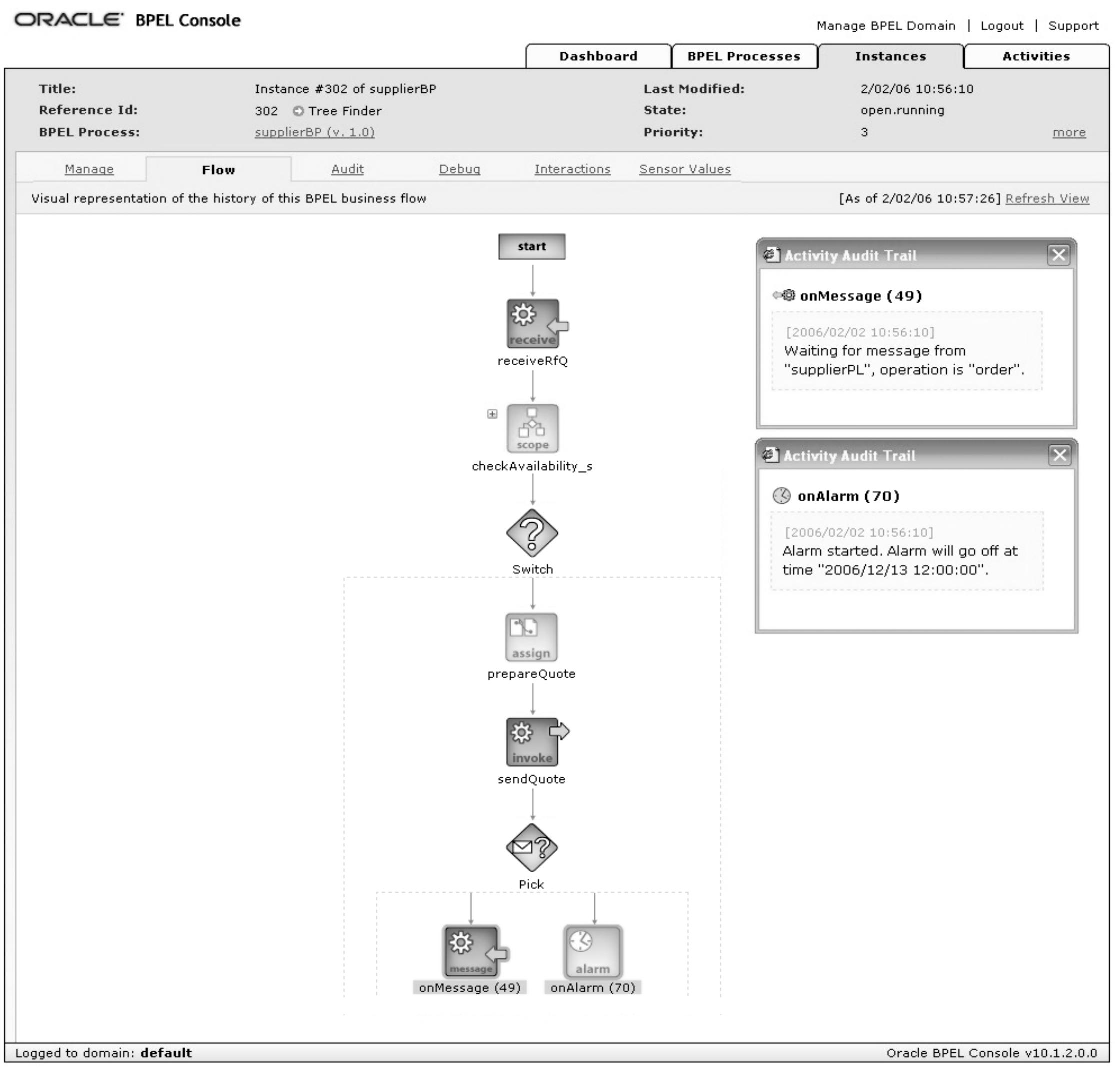


Figure 24. OracleBPEL Process Manager Console 10.1.2: Execution flow of a running instance of the BPEL process salesBP



Title:	Instance #302 of supplierBP	Last Modified:	2/02/06 10:56:10
Reference Id:	302 Tree Finder	State:	open.running
BPEL Process:	supplierBP (v. 1.0)	Priority:	3 more

Visual representation of the history of this BPEL business flow [As of 2/02/06 10:57:26] Refresh View

onMessage (49)
[2006/02/02 10:56:10]
Waiting for message from "supplierPL", operation is "order".

onAlarm (70)
[2006/02/02 10:56:10]
Alarm started. Alarm will go off at time "2006/12/13 12:00:00".

Logged to domain: **default**
Oracle BPEL Console v10.1.2.0.0

allocation considerations to BPEL. In parallel, another tool vendor, Oracle, has implemented its own extension to BPEL for handling human tasks into its BPEL engine.

BPELJ. In BPEL, everything is seen as a service. A BPEL executable process is an implementation of a service that relies on other services. To express that a given service uses certain resources, such as a database, a file or a legacy application, it is necessary to expose the database system, the file system or the legacy application as services. Since BPEL needs to be used in environments where not all resources are exposed as services, it is sometimes handy to be able to break the "all-service" paradigm of BPEL. Driven by this imperative, an extension of BPEL allowing for Java code to be inserted at specific points has been defined, namely *BPELJ* (Blow, Goland, Kloppmann, Leymann, Pfau, Roller, Rowley, 2004 March). This is similar to how Java code can be embedded into HTML code in the context of Java Server Pages (JSPs). As BPEL gains more adoption both in the .Net and the Java platforms, one can expect other similar dialects of BPEL to emerge. Also, as a competing product to BPELJ, Oracle has implemented its own Java Snippet for embedding Java program into a BPEL process.

BPEL-RELATED RESEARCH EFFORTS

There has been a number of research activities conducted on BPEL. These include: systematical evaluation of BPEL based on the so-called workflow patterns (van der Aalst, ter Hofstede, Kiepuszewski, & Barros, 2003), analysis of BPEL process models, generating BPEL code from a "high-level" notations, and choreography conformance checking based on BPEL.

Pattern-based Analysis of BPEL

There are 20 control-flow patterns (van der Aalst, ter Hofstede, Kiepuszewski, & Barros, 2003) and 40 data patterns (Russell, ter Hofstede, Edmond, & van der Aalst, 2005), and accordingly the evaluation has been performed from control-flow perspective (Wohed, van der Aalst, Dumas, & ter Hofstede, 2003) as well as from data perspective (Russel, ter Hofstede, Edmond, & van der Aalst, 2005). The results of the pattern-based evaluation of BPEL show that BPEL is more powerful than most traditional process languages. The control-flow part of BPEL inherits almost all constructs of the block-structured language XLANG and the directed graphs of WSFL. Therefore, it is no surprise that BPEL indeed supports the union of patterns supported by XLANG and WSFL. In particular, the BPEL *pick* construct (namely "deferred choice" in workflow control-flow patterns) is not supported in many existing workflow languages. From the data perspective, BPEL is one of the few languages that fully support the notion of "scope data" elements (one of the workflow data patterns). It provides support for a scope construct which allows related activities, variables and exception handlers to be logically grouped together. The default binding for data elements in BPEL is at process instance level and they are visible to all of the components in a process. In addition to the above evaluation of BPEL, work that has been conducted on the pattern-based evaluation of Oracle BPEL Process Manager (Mulyar, 2005) also involves the evaluation based on a set of 43 workflow resource patterns (Russell, van der Aalst, ter Hofstede, & Edmond, 2005).

Generating and Analyzing BPEL Models

Since BPEL is increasingly supported by various engines, it becomes interesting to link it to other types of models. In this respect, it is insightful to consider the following: (1) BPEL more closely resembles a programming language than a modeling language and (2) BPEL supports the specification of service-oriented processes at various levels of details, down to executable specifications, but it is not designed to support any form of analysis (e.g., behaviour verification, performance analysis, etc.). In other words, BPEL definitions are somewhere in-between the higher-level process models that analysts and designers manipulate in the early phases of the development lifecycle, and fully-functional code. Hence, there are two interesting translations relating to BPEL: (1) a translation from a higher-level notation to BPEL and (2) a translation from BPEL to a model for which established analysis techniques can be applied.

Until now, attention has focused on the second translation. Several attempts have been made to capture the behaviour of BPEL in a formal way. A comparative summary of mappings from BPEL to formal languages can be found in (van der Aalst, Dumas, ter Hofstede, Russell, Verbeek, & Wohed, 2005). The result of comparison shows that the work in (Ouyang, van der Aalst, Breutel, Dumas, ter Hofstede, & Verbeek, 2005) presents the first full formalization of control flow in BPEL that has led to a translation tool called BPEL2PNML and a verification tool called WofBPEL. Both tools are publicly available at http://www.bpm.fit.qut.edu.au/projects/babel/tools. WofBPEL is capable of performing useful and non-syntactic analysis, for example,, detection of unreachable activities and detection of potentially "conflictin-gReceive" faults in a BPEL process. With respect to the verification issues related communication aspects, the work in (Fu, Bultan, & Su, 2004) discusses how to verify the correctness of collection of inter-communicating BPEL processes, and similarly, the work in (Martens, 2005) shows how to check the compatibility of two services with respect to communication.

In industry, various tools and mappings are being developed to generate BPEL code from a graphical representation. Tools such as the IBM WebSphere Choreographer and the Oracle BPEL Process Manager offer a graphical notation for BPEL. However, this notation directly reflects the code and there is no intelligent mapping. This implies that users have to think in terms of BPEL constructs (e.g., blocks, syntactical restrictions on control links, etc.). More interesting is the work of White (2005) that discusses the mapping of Business Process Modelling Notations (BPMN) to BPEL, the work of Mantell (2005) on the mapping from UML Activity Diagrams to BPEL, and the work by Koehler and Hauser (2004) on removing loops in the context of BPEL. However, none of these publications provides a mapping of some (graphical) process modeling language onto BPEL: White (2005) and Mantell (2005) merely present the problem and discusses some issues using examples and Koehler and Hauser (2004) focuses on only one piece of the puzzle. This then motivated the recent work on develop mappings from Workflow nets to BPEL (van der Aalst & Lassen, 2005) and from a core subset of BPMN to BPEL (Ouyang, van der Aalst, Dumas, & ter Hofstede, 2006).

Choreography Conformance Checking based on BPEL

To coordinate a collection of inter-communicating Web services, the concept of "choreography" defines collaborations between interacting parties, that is,, the coordination process of interconnected Web services that all partners need to agree on. A choreography specification is used to describe the desired behaviour of interacting parties. Language such as BPEL and the Web Services Choreography Description Language (WS-CDL) (Kavantzas, Burdett, Ritzinger, Fletcher, & Lafon, 2004 December) can be used to define a desired choreography specification.

Assuming that there is a running process and a choreography specification, it is interesting to check whether each partner (exposed as Web service) is well behaved. Note that partners have no control over each other's services. Moreover, partners will not expose the internal structure and state of their services. This triggers the question of conformance: "Do all parties involved operate as described?" The term "choreography conformance checking" is then used to refer to this question. To address the question, one can assume the existence of both a process model which describes the desired choreography and an event log which records the actual observed behaviour, that is, an actual choreography.

Choreography conformance checking benefits from the coexistence of event logs and process models and may be viewed from two angles. First of all, the model may be assumed to be "correct" because it represents the way partners should work, and the question is whether the events in the log are consistent with the process model. For example, the log may contain "incorrect" event sequences which are not possible according to the definition of the model. This may indicate violations of choreography that all parties previously agreed upon. Second, the event log may be assumed to be "correct" because it is what really happened. In the latter case the question is whether the choreography that has been agreed upon is no longer valid and should be modified.

The work in (van der Aalst, Dumas, Ouyang, Rozinat, & Verbeek, 2005) presents an approach for choreography conformance checking based on BPEL and Petri nets (Murata, 1989). Based on a process model described in terms of BPEL abstract processes, a Petri net description of the intended choreography can be created by using the translation defined in (Ouyang, van der Aalst, Dumas, & ter Hofstede, 2006) and implemented in the tool BPEL2PNML. The conformance checking is then performed by comparing a Petri net and an event log (transformed from SOAP messages under monitoring). To actually measure conformance, a tool called *Conformance Checker* has been developed in the context of the *ProM framework* (see http://www.processmining.org, which offers a wide range of tools related to process mining.

BPEL and Semantic Web Technology

Researchers in the field of Semantic Web have put forward approaches to enhance BPEL process definitions with additional information in order to enable automated reasoning for a variety of purposes. One area in which Semantic Web technology can add value to service-oriented processes is that of *dynamic binding*. The idea of dynamic binding is that rather than hard-coding in the BPEL process definition (or in an associated deployment descriptor) the identities and/or locations of the "partner services" with which the BPEL process interacts, these partner services are determined based on information that is only available at runtime. For example, Verma, Akkiraju, Goodwin, Doshi, and Lee (2004) present an approach to achieve dynamic binding of Web services to service-oriented processes described in BPEL by considering inter-service dependencies and constraints. They present a prototype that can handle BPEL process definitions extended with such dependencies and constraints and can exploit this additional information for runtime discovery of suitable Web services. The paper also discusses another area where Semantic Web technology complements service-oriented process technology: that of semi-automated refinement of process templates described as BPEL abstract processes (see Introduction) into fully executable processes. An example of a tool that implements such refinement techniques is presented in (Berardi, Calvanese, De Giacomo, Hull, & Mecella, 2005).

Mandell and McIlraith (2003) present another approach to dynamic binding of Web services to BPEL processes. Their approach is based on the DAML Web service ontology (DAML-S) (Ankolekar, Burstein, Hobbs, Lasilla, Martin, McDermott, McIlraith, Narayanan, Paolucci, Payne, & Sycara, 2002), the DAML Query Language (DQL) (Fikes, Hayes, & Horrocks, 2002), and the Java Theorem Prover (JTP) (Frank, 1999) which implements DQL. Other approaches to capture Web service semantics include WSDL-S (Akkiraju, Farrell, Miller, Nagarajan, Schmidt, Sheth, & Verma, 2005) and OWL-S (Martin et al., 2005).

While the potential outcomes of these and similar research efforts are appealing, the scalability of the proposed techniques is still unproven and the involved tradeoffs restrict their applicability. Several questions remain open such as: "which languages or approaches to describe service semantics provide the best tradeoffs between expressiveness and computational complexity?" or "How much can a user trust the decisions made by an automated Web service discovery engine, especially at runtime?"

CONCLUSION

In this chapter, we have presented the core concepts of BPEL and the usage of its constructs to describe executable service-oriented processes. We have also discussed extensions to BPEL that have been proposed by tool vendors to address some of its perceived limitations, as well as long-term challenges related to the use of BPEL in the context of rigorous system development methodologies.

Currently, BPEL is being used primarily as a language for implementing Web services using a process-oriented paradigm. In this respect, BPEL is competing with existing enhancements to mainstream programming environments such as WSE and WCF (which enhance the Microsoft .Net framework with functionality for Web service development), or Apache Axis and Beehive (which do the same for the Java platform). Certainly, BPEL is making inroads in this area, and there is little doubt that it will occupy at least a niche position in the space of service implementation approaches. Several case studies related to the use of BPEL in system development projects have been reported in the trade press. These include a report of BPEL use at the European Space Agency and in an outsourcing project conducted by Policy Systems for a state health care service (http://tinyurl.com/zrcje and http://tinyurl.com/krg3o).

However, it must not be forgotten that BPEL can also be used to describe the behaviour of services at a more abstract level. Unfortunately, up to now, tool vendors have given little attention to exploring the possibilities opened by the description of BPEL abstract processes. BPEL abstract processes can be used to represent "service behaviour" at different levels of details. In particular, they enable the representation of temporal, casual and exclusion dependencies between message exchanges. In this respect, BPEL abstract processes can be viewed as adding "behaviour semantics" on top of the basic structural definitions of service interactions provided by WSDL interfaces. Two open questions at the moment are: (1) how to best exploit this additional behaviour semantics in order to support the analysis, testing, operation and maintenance of service-oriented systems; (2) what level of automated support can be realistically provided to aid in refinement of abstract BPEL processes into executable ones. These and the other research directions reviewed in this chapter are only the tip of the iceberg of what can be achieved when richer semantic descriptions of Web services covering behavioural aspects are available.

REFERENCES

Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M., Sheth, A., & Verma, V. (2005, April). *Web Service Semantics – WSDL-S* (Technical note). University of Georgia and IBM. Retrieved October 18, 2006, from http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.html.

Ankolekar, A., Burstein, M., Hobbs, J., Lasilla, O., Martin, D., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Payne, T., & Sycara, K. (2002). DAML-S: Web service description for the Semantic Web. In *Proceedings of the 1st International Semantic Web Conference* (pp. 348-363).

BEA Systems, Microsoft, IBM & SAP (2003, May). Business process execution language for Web services (BPEL4WS). Retrieved October 18, 2006, from ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

Berardi, D., Calvanese, D., De Giacomo, G., Hull, R., & Mecella, M. (2005). Automatic composition of transition-based Semantic Web services with messaging. In *Proceedings of the 31st International Conference on Very Large Data Bases* (pp. 613-624).

Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., & Rowley, M. (2004, March). *BPELJ: BPEL for Java* (White paper). BEA and IBM.

Casati, F., & Shan, M.-C. (2001). Dynamic and adaptive composition of e-services. *Information Systems, 26*(3), 143-162.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web services Web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, *6*(2), 86-93.

Fikes, R., Hayes, P., & Horrocks, I. (2002). DAML Query Language, Abstract Specification. Retrieved October 18, 2006, from *http://www.daml.org/2002/08/dql/dql.*

Frank, G. (1999) A general interface for interaction of special-purpose reasoners within a modular reasoning system. In *Proceedings of the 1999 AAAI Fall Symposium on Question Answering Systems* (pp. 57-62).

Fu, X., Bultan, T., & Su, J. (2004). Analysis of interacting BPEL Web services. In *Proceedings of the 13th International Conference on World Wide Web* (pp. 621-630). New York, NY: ACM Press.

Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., & Lafon, Y. (2004, December). Web services choreography description language version 1.0 (W3C Working Draft 17). Retrieved October 18, 2006, from http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

Kloppmann, M., Koenig, D., Leymann, F., Pfau, G., Richayzen, A., Riegen, von, C., Schmidt, P., & Trickovic, I. (2005, July). WS-BPEL extension for people: BPEL4People. A Joint White Paper by IBM and SAP.

Kloppmann, M., Koenig, D., Leymann, F., Pfau, G., Richayzen, A., von Riegen, C., Schmidt, P., & Trickovic, I. (2005, September). WS-BPEL extension for sub-processes: BPEL-SPE. A Joint White Paper by IBM and SAP.

Koehler, J., & Hauser, R. (2004). Untangling unstructured cyclic flows: A solution based on continuations. In *Proceedings of OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004* (pp. 121–138). Berlin: Springer-Verlag.

Leymann, F. (2001). Web services flow language, version 1.0. Retrieved October 18, 2006, from http://www-306.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf

Mandel, D., & McIlraith S. (2003). Adapting BPEL4WS for the semantic Web: The bottom up approach to Web service interoperation. In *Proceedings of the 2nd International Semantic Web Conference*.

Mantell, K. (2005). From UML to BPEL. Retrieved October, 18, 2006, from http://www.ibm.com/developerworks/Webservices/library/ws-uml2bpel

Martens, A. (2005). Analyzing Web service based business processes. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering* (pp. 19-33). Berlin: Springer-Verlag.

Martin, D., et al. (2005, November). OWL-S: Semantic markup for Web services, W3C Member Submission. Retrieved October 18, 2006, from http://www.w3.org/Submission/OWL-S

Mulyar, N. (2005). *Pattern-based evaluation of Oracle-BPEL* (BPM Center Report BPM-05-24). BPMcenter.org.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 541–580.

OASIS (2005, December 21). Web Services Business Process Execution Language version 2.0 (Committee Draft). Retrieved October 18, 2006, from http://www.oasis-open.org/committees/download.php/16024/wsbpel-specification-draft-Dec-22-2005.htm

OASIS Web Services Business Process Execution Language TC (2006). Retrieved October 18, 2006, from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

Ouyang, C., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M., & Verbeek, H.M.W. (2005). *Formal semantics and analysis of control flow in WS-BPEL* (BPM Center Report BPM-05-15). BPMcenter.org.

Ouyang, C., van der Aalst, W.M.P., Dumas, M., & ter Hofstede, A.H.M. (2006). *Translating BPMN to BPEL* (BPM Center Report BPM-06-02). BPMcenter.org.

Russell, N., ter Hofstede, A.H.M., Edmond, D., & van der Aalst, W.M.P. (2005). Workflow data patterns: Identification, representation and tool support. In *Proceedings of the 24th International Conference on Conceptual Modeling* (pp. 353-368). Berlin: Springer-Verlag.

Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M., & Edmond, D. (2005). Workflow resource patterns: Identification, representation and tool support. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering* (pp. 216-232). Berlin: Springer-Verlag.

Thatte, S. (2001). XLANG Web services for business process design. Retrieved October 18, 2006, from http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

van der Aalst, W.M.P., Dumas, M., Ouyang, C., Rozinat, A., & Verbeek, H.M.W. (2005). *Choreography conformance checking: An approach based on BPEL and Petri nets* (BPM Center Report BPM-05-25). BPMcenter.org.

van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., Russell, N., Verbeek, H.M.W., & Wohed, P. (2005). Life after BPEL? In *Proceedings of European Performance Engineering Workshop and International Workshop on Web Services and Formal Methods* (pp. 35-50). Berlin: Springer-Verlag.

van der Aalst, W.M.P., & Lassen, K.B. (2005). *Translating workflow nets to BPEL* (BETA Working Paper Series). Eindhoven, The Netherlands: Eindhoven University of Technology.

van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., & Barros, A.P. (2003). Workflow patterns. *Distributed and Parallel Databases, 14*(1), 5-51.

Verma, K., Akkiraju, R., Goodwin, R., Doshi, P., & Lee, J. (2004). On accommodating inter-service dependencies in Web process flow composition. In *Proceedings of the American Association for Artificial Intelligence (AAAI) 2004 Spring Symposium.* Stanford, CA: AAAI.

White, S. (2005). Using BPMN to model a BPEL process. *BPTrends, 3*(3), 1-18. Retrieved October, 2006, from http://www.bptrends.com/

Wohed, P., van der Aalst, W.M.P., Dumas, M., & ter Hofstede, A.H.M. (2003). Analysis of Web services composition languages: The case of BPEL4WS. In *Proceedings of the 22nd International Conference on Conceptual Modelling* (pp. 200-215). Chicago: Springer-Verlag.

APPENDEX 1

Exercises

1. Describe two different ways supported by BPEL for describing business processes. What are the differences between them? What are the usages of them?
2. Describe how BPEL uses WSDL, XML Schema, and XPath.
3. Define the partner link between a purchase order process and the external shipping service, and the corresponding partner link type. In this relationship, the purchase order process plays the role of the service requester, and the shipping service plays the role of the service provider. The requester role is defined by a single port type called "shippingCallbackPT". The provider role is defined by a single port type called "shippingPT".
4. Consider the following fragments of a BPEL process definition:
   (a) Write down all possible execution sequences of activities in the above definition.
   (b) Can we add the following *pick* activity in parallel to the two existing *sequence* activities in the above *flow*? If yes, write down all possible execution sequences of activities in this updated process definition, otherwise explain why not.

5. This exercise involves two interacting BPEL processes P1 and P2. P1 consists of a sequence of activities starting with a *receive* activity and ends with a *reply* activity. The pair of *receive* and *reply* defines an interaction with process P2. In P2, there is an *invoke* activity calls a request-response operation on P1, which triggers the executions of the above pair of *receive* and *reply* activities in P1.
   (a) Define an appropriate partner link between P1 and P2 (Assume that P1 plays *myRole*, and P2 plays *partnerRole*).
   (b) Define the pair of *receive* and *reply* activities in P1.
   (c) Define the *invoke* activity in P2.
6. Describe the difference between *switch* and *pick* constructs. Given the four scenarios described below, which of them can be defined using *switch* and which of them can be defined using *pick*?
   (a) After a survey is sent to a customer, the process starts to wait for a reply. If the customer returns the survey in two weeks, the survey is processed; otherwise the result of the survey is discarded.
   (b) Based on the client's credit rating, the client's loan application is either approved or requires further financial capability analysis.
   (c) After an insurance claim is evaluated, based on the findings the insurance service either starts to organize the payment for the claimed damage, or contacts the customer for further details.
   (d) The escalation service of a call centre may receive a storm alert from a weather service which triggers a storm alert escalation, or it may receive a long waiting time alert from the queue management service which triggers a queue alert escalation.
7. The diagram below sketches a process with five activities A0, A1, A2, A3 and A4. A multi-choice node splits one incoming control flow into multiple outgoing flows. Based on the conditions associated with these outgoing flows, one or more of them may be chosen. A sync-merge node synchronises all active incoming control flows into one outgoing flow. Based on the above, sketch

APPENDEX 1. CONTINUED

two possible BPEL definitions for this process using *sequence*, *flow* and *switch* constructs. Also, sketch another BPEL definition of the process using only control link constructs (within a *flow*).

8. The definition below specifies the execution order of the activities within a BPEL process:

    (a) Can we create the following two control links? Justify your answer.
        i) a control link leading from "activityA1" to "activityA3"
        ii) a control link leading from "activityA3" to "activityA5"

    (b) Can we re-define the original process using only control links within the *flow* activity? If so, re-write the process definition, otherwise explain why not.

    (c) Assume that there exist two control links: one leading from "activityA1" to "activityA4", the other from "activityA2" to "activityA4". Both links have a default transition condition, that is, a transition condition that always evaluates to true if the source of the link is executed. Consider the following two scenarios:
        i) "activityA4" has a join condition that is a *disjunction* of all incoming links.
        ii) "activityA4" has a join condition that is a *conjunction* of all incoming links.
        In both scenarios, "activityA4" has its *suppressJoinFailure* attribute set to "yes". Determine whether "activityA4" will be performed in each scenario? Justify your answer and provide a possible execution sequence for each scenario.

    (d) What could verification do when analysing a syntactically correct BPEL process? Argue why automated verification of a BPEL specification is useful.

9. Sketch the control logic of a BPEL process for requesting quotes from an *a priori* known set of N suppliers. The process is instantiated upon receiving a *QuoteServiceRequest* from the Client, and then a *QuoteRequest* is sent in parallel to each of the N suppliers (Supplier1, Supplier2, …, SupplierN). Next, the process waits for *QuoteResponse* from these suppliers. Assume that (a) each supplier replies with at most one response and (b) only M out of N responses are required (M<=N), which means that after receiving the responses from M suppliers, the process can continue without waiting for the responses from the remaining N-M suppliers. To provide the ability to define how many responses are required, a loop is created that repeats until the required number of responses are received. The responses are collected in the order in which they are received. For each response received, the number of responses received (*NofResponse*) is incremented, and the variable containing the result (*Result*) so far is updated. Also, to provide the ability to stop collecting responses after some period of time (e.g., 2 hours), the above loop is contained within a scope activity that has an alarm event handler. If the alarm is triggered, an exception (*TimeOutFault*) is thrown to be caught in the outer scope, thus allowing the process to exit the loop before it finishes. If the exception is thrown, then all that needs to be done is to incorporate a "Timed Out" indication to the *Result*. Finally, the process completes by sending the *Result* to the Client.

10. Below is the BPEL code for the definition of a Supplier abstract process. Since it is an abstract BPEL process, not all elements are fully specified. In particular, you may note that the condition in each *while* loop is omitted, which means that the loop may execute for an arbitrary number of times.

APPENDEX 1. CONTINUED

(a) Given the following sequences of executions, indicate which of them are possible and which of them are not possible based on the above definition. Justify your answer.
   - i) receive *order*;
   - ii) receive *order*, send *orderResponse*;
   - iii) receive *order*, send *orderResponse*, receive *change*;
   - iv) receive *order*, send *orderResponse*, send *orderResponse*, receive *change*, send *order-ChangeResponse*;
   - v) receive *order*, send *orderResponse*, receive *change*, send *orderResponse*, send *order-ChangeResponse*.

(b) In the current process definition, the execution sequence "receive *order*, receive *change*, send *orderChangeResponse*" is not possible. Indicate what minimal changes need to be made to the current process definition, so that this execution sequence becomes possible and all the previous valid execution sequences are preserved.