OASIS

Web Services Business Process Execution Language Version 2.0

Primer

9 May 2007

Document identifier:

wsbpel-primer

Location:

http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.doc

http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf

http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.htm

Editors and Contributors:

Charlton Barreto, Adobe Systems, Inc. <cbarreto@adobe.com>
Vaughn Bullard, Amberpoint <vbullard@amberpoint.com>
Thomas Erl, SOA Systems <thomas.erl@soasystems.com>
John Evdemon, Microsoft <John.Evdemon@microsoft.com>
Diane Jordan, IBM <drj@us.ibm.com>
Khanderao Kand, Oracle, <khanderao.kand@oracle.com>
Dieter König, IBM <dieterkoenig@de.ibm.com>
Simon Moser, IBM <smoser@de.ibm.com>
Ralph Stout, iWay Software <ralph_stout@ibi.com>
Ron Ten-Hove, Sun <Ronald.Ten-Hove@sun.com>
Ivana Trickovic, SAP <ivana.trickovic@sap.com>
Danny van der Rijn, TIBCO Software <dannyv@tibco.com>
Alex Yiu, Oracle <alex.yiu@oracle.com>

Abstract:

The WS-BPEL 2.0 specification [WS-BPEL 2.0] provides a language for formally describing business processes and business interaction protocols. WS-BPEL was designed to extend the Web Services interaction model to support business transactions.

The WS-BPEL Primer is a *non-normative* document intended to provide an easy to read explanation of the WS-BPEL 2.0 specification. The goal of this document is to help readers understand the concepts and major components of the WS-BPEL language. This document will also assist readers in recognizing appropriate scenarios for using WS-BPEL. This document describes several features of WS-BPEL using examples and extensive references to the normative specification.

Status:

This is the approved version of the WS-BPEL 2.0 Primer.

Notices

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS's procedures with respect to rights in OASIS specifications can be found at the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification, can be obtained from the OASIS Executive Director.

OASIS invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to implement this specification. Please address the information to the OASIS Executive Director.

Table of Contents

1. Introduction

1.1. Terminology

WS-BPEL is an acronym for Web Services Business Process Execution Language. WS-BPEL 2.0 is a revision of the original acronym BPEL4WS (Business Process Execution Language for Web Services) 1.0 and 1.1.

1.2. Objective

WS-BPEL 2.0 specification [WS-BPEL 2.0] defines a language for business process orchestration based on web services. This document, WS-BPEL primer, is a supplementary document to WS-BPEL 2.0 specifications. The primer provides a brief explanation of all the key features of WS-BPEL with the help of a practical use case and numerous examples. The primer is intended towards business process analysts, software developers/architects and system integrators who want to know the basics features of WS-BPEL. A basic knowledge of XML, WSDL and any programming language is essential for a better understanding of this document.

1.3. Non-Normative Status

The primer is a non-normative document and not a definitive specification of WS-BPEL. The primer contains examples and other information for a better understanding of WS-BPEL. However, these examples and information would not cover all possible scenarios that are syntactically expressed and covered in WS-BPEL specifications. For any specific information, one is advised to refer to the WS-BPEL specification.

1.4. Relationship to the WS-BPEL Specification

WS-BPEL 2.0 specification [WS-BPEL 2.0] provides a complete normative description to business process execution language. This Primer provides an overview of developing business processes using WS-BPEL2.0 specifications. This document should be considered as supplementary and in terms of its scope and completeness, it is not a replacement to the original specifications.

1.5. Outline of this Document

The primer begins with the background of WS-BPEL, progressively unfolds the language features from basic to advanced concepts, and finally provides a complete picture with an example of a business process for a use case.

The second chapter covers a history of WS-BPEL, and its design goals. Since WS-BPEL is a result of OASIS standardization process for BPEL4WS, this chapter also covers key feature additions and changes in WS-BPEL.

The third chapter covers basic language concepts of WS-BPEL and their usages.

The fourth and fifth chapters cover more advanced features related to compensation, parallel flows, events, concurrency, correlations, and life cycle managements.

The sixth chapter covers the use case and the use of many key features to explain a business process handling Timesheet Submission.

2. History of WS-BPEL

2.1. Design Goals

The WS-BPEL 2.0 effort began with a set of design goals in mind, that helped serve as an initial scope to the specification work. Goals included the definition of Web service orchestration concepts and enabling limited data manipulation.

These initial design goals may be found at http://www.oasis-open.org/committees/download.php/3249/Original%20Design%20Goals%20for%20the%20BPEL4WS%20Specification.doc).

2.2. XLANG and WSFL

The Business Process Execution Language for Web Services (BPEL4WS) was first conceived in July, 2002 with the release of the BPEL4WS 1.0 specification, a joint effort by IBM, Microsoft, and BEA. This document proposed an orchestration language inspired by previous variations, such as IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG specification.

Joined by other contributors from SAP and Siebel Systems, version 1.1 of the BPEL4WS specification was released less than a year later, in May of 2003. This version received more attention and vendor support, leading to a number of commercially available BPEL4WS-compliant orchestration engines. Just prior to this release, the BPEL4WS specification was submitted to an OASIS technical committee so that the specification could be developed into an official, open standard.

3. Basic Concepts

3.1. The Structure of BPEL Processes

First of all, a BPEL Process is a container where you can declare relationships to external partners, declarations for process data, handlers for various purposes and, most importantly, the activities to be executed. On top, the process container has a couple of attributes, i.e. a (mandatory) name and a (also mandatory) declaration of a namespace – as shown in the example below. You should note that not all possible attributes of the process element are shown in this example.

```
<process name="PrimerProcess"
   targetNamespace="http://oasis-open.org/WSBPEL/Primer/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable" />
```
Example 3-1: A process.

The namespace declaration "http://docs.oasis-open.org/wsbpel/2.0/process/executable" specifies that this is an Executable Process. An additional namespace is available for Abstract Processes. Abstract processes describe process behavior partially without covering every detail of execution. That behavior typically encompasses a process template or the externally visible behavior of a process towards business partners without exposing the internal business logic. Executable Processes, by contrast, define the complete business behavior both the externally visible part and internal processing part.

The `process` element is the outermost container. Therefore, any partners, process data or handlers that are declared on the process container can be considered as global. BPEL also supports the concept of declaring all these things in a local way. The syntactic element to do this is called a `scope`.

```
<scope name="Scope" />
```
Example 3-2: A scope.

With the help of scopes, you can divide up your business process, each holding a portion of the overall business logic. For example, process data that is declared local to a scope is not visible to anything outside of that scope, i.e. it is only valid within such a specific part of the process.

3.2. Relationship to Business Partners

BPEL Business Processes offer the possibility to aggregate web services and define the business logic between each of these service interactions. It is also said that BPEL orchestrates such web service interactions. Each service interaction can be regarded as a communication with a business partner. The interaction is described with the help of partner links. Partner links are instances of typed connectors which specify the WSDL port types the process offers to and requires from a partner at the other end of the partner link.
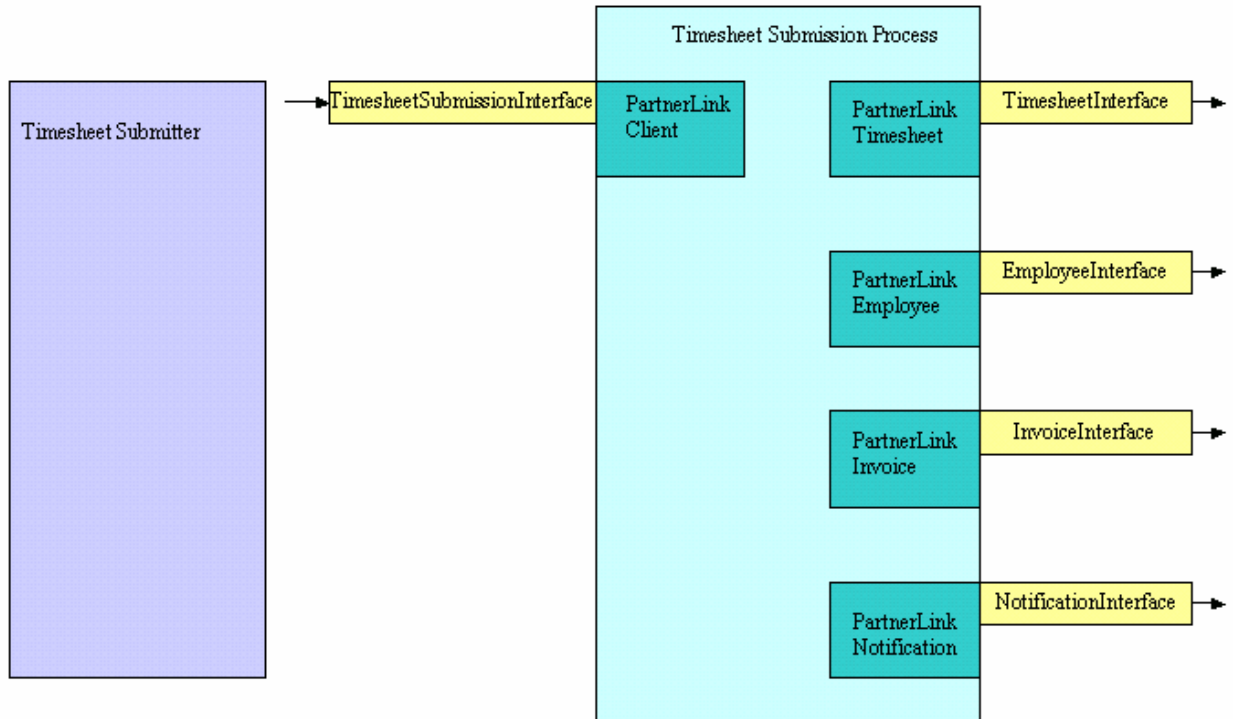
Note that for one partner, there can be a set of partner links. You can regard one partner link as one particular communication channel. Such an interaction is potentially two sided: the process invokes the partner and the partner invokes the process. Therefore, each `partnerLink` is characterized by a partner link type and a role name. This information identifies the functionality that must be provided by the business process and by the partner service.

```
<partnerLinks>
   <partnerLink name="ClientStartUpLink"
      partnerLinkType="wsdl:ClientStartUpPLT" myRole="Client" />
</partnerLinks>
```
Example 3-3: Partner links.

Partner link declarations can take place directly under the `process` element, which means that they are accessible by all BPEL constructs within the BPEL process, or under a `scope` element, which means they are only accessible from the children of that scope.

3.3. State of a BPEL Process

In Section 3.1, the term process data was introduced. Technically, process data are variables that are declared on a process or on a scope within that process. Variables hold the data that constitute the state of a BPEL business process during runtime. Data in BPEL is written to and read from typed variables. The values contained in such variables can be of two sources: either they come from messages exchanged with a partner, or it is intermediate data that is private to the process. In order to fulfill type-contracts with the partner, all variables in BPEL must either be WSDL message types, XML schema simple types, XML schema complex types or XML schema elements. In order to change the state of a process by changing the content of its

variables, an expression language for manipulating and querying variables is required. In BPEL, the default language for that is XPath 1.0.

You can declare a variable by specifying a name and one of the three types mentioned above, written down as one of the three attributes `type`, `messageType` or `element`.

```
<variables>
   <variable name="myVar1" messageType="myNS:myWSDLMessageDataType" />
   <variable name="myVar1" element="myNS:myXMLElement" />
   <variable name="myVar2" type="xsd:string" />
   <variable name="myVar2" type="myNS:myComplexType" />
</variables>
```
Example 3-4: Variables.

Variable declarations can appear directly under the `process` element, which means that they are visible to all BPEL constructs within the BPEL process, or under a `scope` element, which means it is only visible to the children of that scope.

3.4. Behavior of a BPEL Process

The major building blocks of BPEL processes are activities. There are two types: structured activities can contain other activities and define the business logic between them. In contrast, basic activities only perform their intended purpose (like receiving a message from a partner, or manipulating data) and have no means to define any other logic by containing other activities.

3.4.1.   Providing and Consuming Web Services

In BPEL, there exist a couple of simple activities with the purpose of consuming messages from and providing messages to web service partners. These activities are the receive activity, the reply activity and the invoke activity. All these activities allow exchanging messages with external partners (services).

The purpose of the `receive` activity is receiving messages from an external partner. Therefore, a receive activity always specifies the partner link and operation of the partners the web service. You need also to specify a variable (or a set of variables) that holds the requested data that will be received from the partner. A receive activity may have an associated `reply` activity if it is used to provide a WSDL request-response operation.

```
<receive name="ReceiveRequestFromPartner"
   createInstance="yes"
   partnerLink="ClientStartUpPLT"
   operation="StartProcess" ... />
```
Example 3-5: Receive activity.

The attribute called `createInstance` on the receive activity means that you can use a receive activity with `createInstance="yes"` to create a new process instance, whereas `createInstance="no"` means that the incoming message will be consumed by the (already running) process instance.

As already mentioned, one receive activity can have an associated `reply` activity. You might think of a client that wants to order a book from a book selling process. The client would send a request to the receive activity of the book selling process, the process then would do some internal logic (like determining whether the book is available, checking if the delivery address and the provided credit card number are correct etc.). After that logic is done, it would be natural for the process to respond to the client to let him know whether the order was successful or not.

You should be aware that a `reply` activity can come in two flavors: It can reply normal data (which would yield to a normal reply), or it can reply faulted data (like a "the book is out of Stock" exception in the example above), which would then yield to a "faulted reply". If so, you should specify an additional `faultName` attribute on the reply activity.

As you see, the `reply` activity is typically used in conjunction with the receive activity to implement a WSDL request-response operation on a particular communication channel (partner link). It provides means to return data to the caller by specifying a `partnerLink` and operation for the Web service. The specified variable holds the response data or fault data returned to the caller of the Web service. If fault data is returned, the `faultName` identifies the corresponding WSDL fault.

```
<reply name="ReplyResponseToPartner"
   partnerLink="ClientStartUpPLT"
   operation="StartProcess" ... />
```
Example 3-6: Reply activity.

The third web-service related activity is the `invoke` activity. The `invoke` activity is used to call a web service provided by a partner. A `partnerLink` and operation of the web service to be called must be specified.

```
<invoke name="InvokePartnerWebService"
   partnerLink="BusinessPartnerServiceLink"
   operation="partnerOperation" ... />
```
Example 3-7: Invoke activity.

In WSDL 1.1 there exist multiple types of operations. Two of them are supported by BPEL: one-way operations and request-response operations. An `invoke` activity can therefore either call a one-way operation (and would then continue with the process logic without waiting for the partner to reply), or a request-response operation (which would block the process (or a part of it) until it receives a response from the partner service. If the operation that is invoked is of type request-response operation, you must provide both an input and output variable.

```
<invoke name="RequestResponseInvoke"
   partnerLink="BusinessPartnerServiceLink"
   operation="RequestResponseOperation"
   inputVariable="Input"
   outputVariable="Output" />
```
Example 3-8: Invoke activity for a WSDL request-response operation.

In case it is a one-way operation, you only have to specify an input variable.

```
<invoke name="OneWayInvoke"
```

```
   partnerLink="BusinessPartnerServiceLink"
   operation="OneWayOperation"
   inputVariable="Input" />
```
Example 3-9: Invoke activity for a WSDL one-way operation.

Besides, you can specify various handlers on an `invoke` activity. This is because an `invoke` activity can be regarded as a shorthand notation of a scope activity that contains the handlers allowed as well as the `invoke` activity itself. The handlers allowed within an `invoke` activity are fault handlers and a compensation handler. For more information on the various types of handlers, see section 4.1.

For the sake of completeness, it should be mentioned that there are more web-service related constructs like the Pick Activity and a handler called Event Handler which will be described in sections 4.2.1 and 4.2.3, respectively.

3.4.2.   Structuring the Process Logic

BPEL provides means to structure the business logic according to your needs. If you need a number or activities executed in a sequential order (e.g. first receive the order, then check the payment, then ship the goods, then send a confirmation) you can do so by using BPEL's `sequence` activity. In other words, the `sequence` activity is used to define a collection of activities which are executed sequentially in lexical order.

```
<sequence name="InvertMessageOrder">
   <receive name="receiveOrder" ... />
   <invoke name="checkPayment" ... />
   <invoke name="shippingService" ... />
   <reply name="sendConfirmation" ... />
</sequence>
```
Example 3-10: Sequence activity.

Another activity used for structuring the business logic is the `if-else` activity. The construct might be known from traditional programming languages. The `if-else` activity allows you to select exactly one branch of the activity from a given set of choices. For each choice, the behavior is to check a condition and if that condition evaluates to true, the associated branch is executed, otherwise an alternative path is taken. As with all expressions in BPEL, you can use XPath expressions to formulate your condition. Note that only the first branch with a true condition is executed. If no condition evaluates to true, then a default choice can be specified using the `else` branch.

```
<if name="isOrderBiggerThan5000Dollars">
   <condition>
      $order &gt; 5000
   </condition>
   <invoke name="calculateTenPercentDiscount" ... />
   <elseif>
      <condition>
         $order &gt; 2500
      </condition>
      <invoke name="calculateFivePercentDiscount" ... />
   </elseif>
```

```
    <else>
        <reply name="sendNoDiscountInformation" ... />
    </else>
</if>
```
Example 3-11: If-else activity.

3.4.3. Repetitive Activities

BPEL offers three activities that allow the repeated execution of a piece of business logic. One of these activities is the `while` activity. The `while` activity is a structured activity, i.e. it has a child nested within. The `while` activity allows you to repeatedly execute the child activity as long as a given condition evaluates to true. The condition is specified on the `while` activity and gets evaluated at the beginning of each iteration, which means consequently that the body of the `while` activity might not be executed at all (if the condition does not evaluate to true at all).

```
<while>
    <condition>
        $iterations &gt; 3
    </condition>
    <invoke name="increaseIterationCounter" ... />
</while>
```
Example 3-12: While activity.

In contrast, the `repeatUntil` activity has the difference that the body of the activity is performed at least once, since the condition is evaluated at the end of each iteration.

```
<repeatUntil>
    <invoke name="increaseIterationCounter" ... />
    <condition>
        $iterations &gt; 3
    </condition>
</repeatUntil>
```
Example 3-13: RepeatUntil activity.

The third activity in the group of repetitive activities is the `forEach` activity. In its default behavior, the `forEach` activity iterates sequentially N times over a given set of activities. A use-case that can be imagined is that it iterates over an incoming order message where the order message consists of N order items. Technically, the `forEach` activity iterates its child scope activity exactly N times where N equals the `finalCounterValue` minus the `startCounterValue` plus 1.

```
<forEach parallel="no" counterName="N" ...>
    <startCounterValue>1</startCounterValue>
    <finalCounterValue>5</finalCounterValue>
    <scope>
        <documentation>check availability of each item ordered</documentation>
        <invoke name="checkAvailability" ... />
    </scope>
</forEach>
```
Example 3-14: ForEach activity (serial).

There are two variants `forEach` activity: sequential and parallel, as specified by the parallel attribute. The parallel variant is discussed in more detail in section 4.3.

One restriction applies to the `forEach` activity, as it is shown in the example: All other structured activities introduced so far can have any arbitrary activity as its child activity, whereas the `forEach` activity can only have a scope (see Section 3.1).

3.4.4. Parallel Processing

So far, you have been introduced to various concepts how business processes can be structured in a sequential fashion. However, often it is desirable or even necessary to execute things in parallel. For this purpose, BPEL offers the `flow` activity. Note that besides the flow activity, the parallel variant of forEach activity and event handlers allow multiple instances of its child activity to run in parallel – they are discussed later.

In the following example, a set of three activities (*checkFlight*, *checkHotel* and *checkRentalCar*) are executed in parallel, i.e. their corresponding web services would be invoked concurrently. All three activities are started concurrently when the `flow` activity starts.

```
<flow ...>
   <links> ... </links>
   <documentation>
      check availability of a flight, hotel and rental car concurrently
   </documentation>
   <invoke name="checkFlight" ... />
   <invoke name="checkHotel" ... />
   <invoke name="checkRentalCar" ... />
</flow>
```
Example 3-15: Flow activity.

Sometimes, even in a mainly parallel process, it is necessary to synchronize between some of these activities. Let's assume a fourth activity, *bookFlight*, is added to the example given above. If you would just add this activity to the `flow` activity, it would also be executed in parallel when the `flow` activity starts. However, booking a flight makes only sense after you checked that there is actually one available. Therefore, you can add a `link` between these two activities. Adding a link introduces a control dependency which means that the activity which is the target of the link will only be executed if the activity that is the source of the link has completed. The corresponding BPEL would look as follows:

```
<flow ...>
   <links>
      <link name="checkFlight-To-BookFlight" />
   </links>
   <documentation>
      check availability of a flight, hotel and rental car concurrently
   </documentation>
   <invoke name="checkFlight" ...>
      <sources>
         <source linkName="checkFlight-To-BookFlight" />
      </sources>
   </invoke>
```

```
      <invoke name="checkHotel" ... />
      <invoke name="checkRentalCar" ... />
      <invoke name="bookFlight" ...>
         <targets>
            <target linkName="checkFlight-To-BookFlight" />
         </targets>
      </invoke>
</flow>
```
Example 3-16: Flow activity with links.

The semantics of link elements are richer then indicated in this little example. A link can have a transition condition associated with it which influences its status. If no transitionCondition is specified, the status of the link is true. If a transitionCondition is specified, it will set the status of the link. Take a look at the following example:

```
<flow ...>
   <links>
      <link name="request-to-approve" />
      <link name="request-to-decline" />
   </links>
   <receive name="ReceiveCreditRequest"
      createInstance="yes"
      partnerLink="creditRequestPLT"
      operation="creditRequest"
      variable="creditVariable">
      <sources>
         <source linkName="request-to-approve">
            <transitionCondition>
               $creditVariable/value &lt; 5000
            </transitionCondition>
         </source>
         <source linkName="request-to-decline">
            <transitionCondition>
               $creditVariable/value >= 5000
            </transitionCondition>
         </source>
      </sources>
   </receive>
   <invoke name="approveCredit" ...>
      <targets>
         <target linkName="request-to-approve" />
      </targets>
   </invoke>
   <invoke name="declineCredit" ...>
      <targets>
         <target linkName="request-to-decline" />
      </targets>
   </invoke>
</flow>
```
Example 3-17: Flow activity with links and transition conditions.

The link *request-to-approve* has a transition condition that checks if the part *value* of variable *creditVariable* has a value that is less than 5000. If that is the case, the link status of the request-to-approve link will be set to true, otherwise to false. Since the transition condition of the

request-to-decline link is the exact opposite (greater than or equal to 5000), this means that exactly one of the two successor activities *approveCredit* or *declineCredit* will be executed.

Transition conditions offer a mechanism to split the control flow based on certain conditions. Therefore, a mechanism to merge it again must be offered, too. BPEL does that with join conditions. Join conditions are associated with activities, usually if the activities have any incoming links. A `joinCondition` specifies for an activity something like a "start condition", e.g. all incoming links must have the status of `true` in order for the activity to execute, or at least one incoming link must have the status `true`. The following example illustrates this:

```
<flow ...>
   <links>
      <link name="request-to-approve" />
      <link name="request-to-decline" />
      <link name="approve-to-notify" />
      <link name="decline-to-notify" />
   </links>
   <receive name="ReceiveCreditRequest"
      createInstance="yes"
      partnerLink="creditRequestPLT"
      operation="creditRequest"
      variable="creditVariable">
      <sources>
         <source linkName="request-to-approve">
            <transitionCondition>
               $creditVariable/value &lt; 5000
            </transitionCondition>
         </source>
         <source linkName="request-to-decline">
            <transitionCondition>
               $creditVariable/value >= 5000
            </transitionCondition>
         </source>
      </sources>
   </receive>
   <invoke name="approveCredit" ...>
      <source linkName="approve-to-notify" />
      <targets>
         <target linkName="request-to-approve" />
      </targets>
   </invoke>
   <invoke name="declineCredit" ...>
      <source linkName="approve-to-notify" />
      <targets>
         <target linkName="request-to-decline" />
      </targets>
   </invoke>
   <reply name="notifyApplicant" ...>
      <targets>
         <joinCondition>
            $approve-to-notify or $decline-to-notify
         </joinCondition>
         <target linkName="approve-to-notify" />
         <target linkName="decline-to-notify" />
      </targets>
   </invoke>
```

```
      </reply>
</flow>
```
Example 3-18: Flow activity with links, transition conditions and join conditions.

Let's imagine that an error occurs in activity *approveCredit*, the outgoing link of this activity
(which is link *approve-to-notify*) will be set to `false` by the execution environment. This may
lead to a situation where the join condition is evaluated to `false` as well. WS-BPEL provides
two mechanisms for dealing with false join conditions. By default, a joinFailure fault is thrown
that may be caught by an appropriate fault handler (see also section 3.4.6). Alternatively, when
the attribute `suppressJoinFailure` on the process or an enclosing activity is set to `yes`,
the activity associated with the `false` join condition is skipped and the `false` link status is
propagated along links leaving that activity. In other words, a `false` link status will be
propagated transitively along entire paths formed by successive links until a join condition is
reached that evaluates to `true`. This approach is called *Dead-Path Elimination* (DPE).

3.4.5.   Data Manipulation

In the previous sections it has been shown how business data can be received into a process (see
section 3.4.1). However, in real-life business processes, this data sometimes needs to be split up
in various parts, or joined from various sources. Imagine an input message to a process which
contains multiple parts:

- the name of the customer
- an article number of the item that is ordered
- a shipping address
- credit card information (number, expiration date etc.)

Now also imagine that your enterprise is set up in a way that you have a department that handles
the shipping, one that handles the availability, and you have a business partner that checks the
validity of credit cards. Now you can easily imagine that the partner service that checks for credit
cards doesn't really care for what the customer ordered, it only cares for the name of the
customer, and for the credit card information. What your process would need to do is: take these
two information parts out of the input message of the process, assemble a new message just
containing of these two parts, and then use this new message as the input message for the
creditCardCheck Service. Analogous examples can be set up with the name shipping department
etc. It becomes apparent that a mechanism for data manipulation is needed within BPEL
Business processes.

BPEL therefore offers the `assign` activity. The `assign` activity contains one or more `copy`
operations. Each copy operation has a `from`–spec and a `to`–spec, indicating the source and
target elements where data gets copied from and to, respectively. In the following example, a
whole variable gets copied into another variable of the same data type:

```
<assign>
   <copy>
      <from variable="TimesheetSubmissionFailedMessage" />
      <to variable="EmployeeNotificationMessage" />
   </copy>
</assign>
```

Example 3-19: Assign activity.

However, the textual example above showed that is not sufficient, often you want only to copy parts of a variable into another variable, or even parts of a variable into a part of another variable. Therefore, you need a mean to reference such parts. One of the goals of BPEL was not to reinvent yet another XML-based data manipulation language, but to reuse other standard like XPath and XSLT in order to achieve this.

```
<assign>
   <copy>
      <from variable="Input" part="operation1Parameter">
         <query>
            creditCardInformation
         </query>
      </from>
      <to variable="CreditCardServiceInput" />
   </copy>
</assign>
```

Example 3-20: Assign activity with query language attribute.

As it can be seen in the example, when addressing a part of a variable, a query language can be specified in order to do this. XPath 1.0 is used in the example, as indicated in the queryLanguage attribute of the `query` element. (Note: XPath 1.0 is the default expression and query language in WS-BPEL 2.0. Typically users do not explicitly specify the usage of XPath 1.0 in their process definition through queryLanguage or expressionLanguage attribute.)

You can not only assign from a variable or a part of a variable to another variable (or its parts), but also from one of the following:

Variable Property:
```
 <from>
    bpel:getVariableProperty("Input", "a:orderNo")
 </from>
```

Partner Link / EPR:
  `<from partnerLink="Supplier"/>`, see also section 3.2

Expression:
  `<from>count($po/poline)</from>`

3.4.6.   Exception Handling

So far, almost all explained concepts, along with the given examples, assumed that everything goes fine when executing a business process. In reality, a language like BPEL must be able to cope with exceptional situations, e.g. calling a web service that is currently unavailable. In other words, a mechanism to detect that such an exceptional situation occurred must be offered, and also means to handle such situation must be provided.

BPEL therefore offers the concept of fault handlers. A fault handler can be attached to a `scope` (see section 4.1.2 for more details), a `process`, or, as outlined in section 3.4.1, even directly as

a shorthand notation on an `invoke` activity. In this section, we deal with the fault handling on process level only.

A fault handler gets installed as soon as the scope, it is associated to, gets started. As an example, the process level fault handler gets installed when the process starts. If the process completes normally, the installed fault handler then gets discarded, but if a fault situation occurs, that fault gets propagated to the fault handler.

```
<faultHandlers>
   <catch faultName="BookOutOfStockException"
      faultVariable="BookOutOfStockVariable">
      ...
   </catch>
   <catchAll>...</catchAll>
</faultHandlers>
```
Example 3-21: Fault handler for a process.

As it can be seen in the example, a fault handler can have two types of children: One or more `catch` constructs, and at most one `catchAll`. Each `catch` construct then has to provide an activity (indicated by the "…" in the example) that performs exception handling for a specific type of error condition. In the given example, you call a checkAvailability web service for books to see whether a book that was ordered is in stock, and the response of the web service might be to throw a BookOutOfStockException (which would have be declared on the WSDL interface of that service).

A `catch` construct has optional additional attributes: a `faultName` that refers to the name of the fault that should get caught, and a `faultVariable` attribute. When `faultVariable` attribute is used, either `faultMessageType` or `faultElement` attribute must be specified. The `faultVariable` would point to a variable locally declared within the `catch` construct based on the `faultMessageType` or `faultElement` attribute.

Optionally, the fault hander can end with a `catchAll` construct. The intention is to provide a mean for default fault handling, e.g. if the checkAvailability web service would not only throw a BookOutOfStockException, but also a BookOutOfPrintException and a BookTitleNotFoundException. If you don't want to distinguish the error handling of the latter two, you would just fill the `catchAll` with the appropriate exception handling activities.

4. Advanced Concepts I

4.1. Refining the Process Structure

In BPEL, you may structure your business process into a hierarchy of nested scopes. Each scope can have its own definitions of variables, partner links, message exchanges, correlation sets, and handlers. This limits the visibility of these definitions to the enclosed activities and provides the context in which they are executed. The outermost context is the process definition itself.

Two constructs of the BPEL language require the use of scopes. The primary activity in the `onEvent` event handler and in the `forEach` activity must be a scope. These scopes own the definition of the event handler variable and the loop counter variable (the `counterName` attribute of the `forEach` activity in section 3.4.3), respectively. In the event handler case, the enclosed scope may also own the definition of the partner link, message exchange, or correlation set definition used by the event handler.

If an `invoke` activity contains the definition of a fault handler or compensation handler then it is equivalent to an implicit `scope` immediately enclosing the `invoke` activity. This implicit `scope` activity assumes the `name` of the `invoke` activity it encloses, its `suppressJoinFailure` attribute and its `sources` and `targets` elements (see section 3.4.4).

4.1.1.   Lifecycle of a Scope

A scope can be used like a regular activity, for example, as a child element of a loop. The lifecycle of a scope begins with the following initialization sequence for entities defined locally within the scope:

- Initialize variables and partner links
- Instantiate correlation sets
- Install fault handlers, termination handler, and event handlers

The steps listed above are performed in an all-or-nothing fashion, that is, either all succeed or the fault `bpel:scopeInitializationFailure` is thrown to the parent scope. After scope initialization, the primary activity of the scope is executed and all event handlers are enabled in parallel, except an initial start activity which is executed before event handlers are enabled.

A scope finishes its work either successfully or unsuccessfully – three cases must be distinguished:

- [Normal completion] If the primary activity completes without throwing a fault and no orphaned inbound message activities are detected then all event handlers are disabled (running event handler instance are allowed to finish), the compensation handler is installed. The scope finishes successfully.

- [Internal fault] If a fault is thrown within the scope then all other running activities and event handler instances within the scope are terminated and a matching fault handler (see section 3.4.6) is executed. The scope finishes unsuccessfully.
- [External termination] If a running scope received a termination signal (because of an external fault or completion condition) then all other running activities and event handler instances within the scope are terminated. The scope finishes unsuccessfully.

The following sections describe the different types of handlers in more detail.

4.1.2.   Scoped Fault Handling

BPEL fault handlers for processes have been introduced in section 3.4.6. They may also be associated with a scope in order to deal with exceptional situations more locally to the place where they occurred.

```
<scope>
   <faultHandlers>
      <catch faultName="xyz:anExpectedError">...</catch>
      <catchAll><!-- deal with other errors -->
         ...
      </catchAll>
   </faultHandlers>
   <sequence>
      <!-- do work -->
   </sequence>
</scope>
```
Example 4-1: Scoped fault handling.

When a fault happens within a scope then the scope completes unsuccessfully. Before the scope's processing ends, a local fault handler can deal with the fault. Fault handlers may themselves throw new faults or rethrow the fault they caught to the next enclosing scope, or to the process if no more enclosed scope is present.

4.1.3.   Terminating Running Work

Before a fault handler actually begins its processing, all other running work within its associated scope is terminated. Structured activities are terminated, and the termination is propagated into their contained activities. Some basic activities (assign, empty, throw, rethrow, exit) are allowed to complete while others are interrupted.

Scopes themselves may influence their termination behavior. Typical use cases include performing cleanup work or sending a message to a business partner. After terminating the scope's primary activity and all running event handler instances, the scope's (custom or default) terminationHandler is executed.

```
<scope>
   <terminationHandler>
      <!-- clean up resources in case of forced termination -->
   </terminationHandler>
   <sequence>
```

```
        <!-- do work -->
    </sequence>
</scope>
```
Example 4-2: Termination handling.

Custom termination handlers can contain any BPEL activities, including `compensate` and `compensateScope`. They cannot propagate faults to their enclosing business logic because the termination was either caused by another fault or by a completion condition of a `forEach` activity.

4.1.4.   Undoing Completed Work

Business processes typically represent long-running work which cannot be completed within a single atomic transaction. Already committed ACID transactions create persistent effects before the process is completed. Application-specific compensation steps undo these effects when required. In a BPEL scope, the language construct for reversing previously completed process steps is the `compensationHandler`. It can be invoked after successful completion of its associated scope, using the `compensate` or `compensateScope` activities.

```
<scope name="S1">
    <faultHandlers>
        <catchAll>
            <compensateScope target="S2" />
        </catchAll>
    </faultHandlers>
    <sequence>
        <scope name="S2">
            <compensationHandler>
                <!-- undo work -->
            </compensationHandler>
            <!-- do some work -->
        </scope>
        <!-- do more work -->
        <!-- a fault is thrown here; results of S2 must be undone -->
    </sequence>
</scope>
```
Example 4-3: Compensation handling.

A scope's compensation handler has visibility to the current state of the process instance. The state of a successfully completed scope is saved such that a compensation handler can "continue" working on it later. The state of enclosing scopes is shared with other concurrent work, so all concurrency and isolation considerations (see section 4.8) apply to compensation handlers in the same way as for the primary activity of a scope.

Compensation handlers are invoked by the `compensate` or `compensateScope` activity, which may reside in a fault handler, compensation handler, or termination handler (together referred to as FCT-handlers) of an immediately enclosing scope. A `compensate` activity causes the compensation handler of all successfully completed and not yet compensated child scopes to be executed in default order. A `compensateScope` activity causes the compensation handler of one specified successfully completed scope to be executed. If a compensation handler's associated scope is contained in a repeatable construct or event handler then multiple

compensation handler instances exist – each one associated with a corresponding scope instance. Finally, each compensation handler instance may in turn cause compensation handlers of nested scopes to be invoked. The set of all compensation handler instances invoked by one `compensate` or `compensateScope` activity is called a *compensation handler instance group*.

If a fault is thrown in a compensation handler then the fault is propagated to the scope containing the `compensate` or `compensateScope` activity that invoked the compensation handler instance group. Before the corresponding fault handler begins its work, the termination of all running work includes all still running compensation handlers of the group.

4.1.5. Event Handling

Each scope as well as the process itself may define event handlers. They are used to process Web service request messages arriving in parallel to the primary activity of the scope or process. As one of multiple advanced interaction scenarios, event handlers are described in section 4.2.3.

4.2. Advanced Web Service Interactions

In the previous chapter, we explained how the `receive` activity is used to perform a blocking wait for a particular incoming message. There are also scenarios where the behavior of a business process is more complex. In some cases, one out of a set of different messages will arrive. In other cases, multiple incoming messages are expected before further steps in the business logic can be performed. Sometimes messages arrive in parallel to the main flow of the business process. All of these scenarios can be modeled using BPEL constructs. In the specification, these constructs are jointly referred to as *inbound message activities (IMA)*. In the following section, we will discuss each construct in detail.

4.2.1. Selective Event Processing

In the first advanced scenario, there is a set of multiple suitable messages where each one of them can trigger subsequent steps in the business process. Optionally, one is able to specify the behavior for the exceptional case where none of these expected messages arrive within a certain amount of time. The `pick` activity contains one or more `onMessage` elements and zero or more `onAlarm` elements. Each `onMessage` element points to a Web service operation exposed by the business process and to a variable that holds the received message. Each `onAlarm` element has a specified point in time or a time interval. Both elements contain the business logic to be performed when the specified message or timeout event occurs.

```
<pick>
   <onMessage partnerLink="buyer"
      operation="inputLineItem"
      variable="lineItem">
      <!-- activity to add line item to order -->
   </onMessage>
   <onMessage partnerLink="buyer"
      operation="orderComplete"
      variable="completionDetail">
```

```
       <!-- activity to perform order completion -->
   </onMessage>
   <onAlarm>
      <for>'P3DT10H'</for>
      <!-- handle timeout for order completion -->
   </onAlarm>
</pick>
```
Example 4-4: The pick activity.

A `pick` activity with two `onMessage` branches and one `onAlarm` branch is shown in the previous example. Either the *inputLineItem* operation, the *orderComplete* operation, or the expiration of the timeout interval of 3 days and 10 hours will cause the associated activity to be processed and the `pick` activity to complete. The first event to occur triggers the associated logic. Only one event is processed, that is, the `pick` activity completes when the business logic associated with this event has been processed.

Similar to the `receive` activity; `createInstance="yes"` can be specified for the `pick` activity in order to cause a new process instance to be created upon the receipt of message. Note that a new process instance can only be created by receiving a message and not with an `onAlarm` specification.

4.2.2. Multiple Event Processing

In the previous section, we showed how one message can be received when there are multiple possible choices. We now take a look at a scenario where more than one message is needed to trigger subsequent steps. Consider a broker's business process that does its work after both a buyer and a seller have reached agreement on a particular trade. Both trading partners inform the broker who then initiates the settlement. The broker process waits for a message from both partners without knowing which one arrives first.

```
<flow>
   <links>
      <link name="buyToSettle" />
      <link name="sellToSettle" />
   </links>
   <receive name="receiveBuyerInformation" createInstance="yes" ...>
      <sources>
         <source linkName="buyToSettle" />
      </sources>
      <correlations>
         <correlation set="tradeID" initiate="join" />
      </correlations>
   </receive>
   <receive name="receiveSellerInformation" createInstance="yes" ...>
      <sources>
         <source linkName="sellToSettle" />
      </sources>
      <correlations>
         <correlation set="tradeID" initiate="join" />
      </correlations>
   </receive>
   <invoke name="settleTrade" ...>
      <targets>
```

```
            <joinCondition>$buyToSettle and $sellToSettle</joinCondition>
            <target linkName="buyToSettle" />
            <target linkName="sellToSettle" />
        </targets>
    </invoke>
    ...
</flow>
```
Example 4-5: Multiple start activities.

You noticed two `receive` activities with the `createInstance="yes"` attribute shown in
the above example. Such activities are also called *start activities*. In our broker scenario, we
certainly don't want to create two different process instances. Both messages from buyer and
seller must be processed by the same one business process instance. Exactly this is achieved by
using two start activities. If both incoming messages fit to the respective receive activity and
both are designated for the same business process instance then the second message will not
create a new process instance. In a subsequent section, we explain how messages are correlated
with particular process instance. Although there are two start activities, only the first message
actually causes a process instance to be created and the second one will be received by the same
instance. After both receive activities have been executed, processing of the broker's business
logic continues at the *settleTrade* activity that joins the two parallel control flow branches into
one.

4.2.3.   Concurrent Event Processing

So far, we only talked about activities that perform a blocking wait until a certain message or
timeout event occurs. However, it is not always possible or appropriate to interrupt the business
logic. Consider a purchase order process that is initiated by a buyer's purchase request message.
The order may be completely processed without further interaction; ultimately the goods are
shipped and the invoice is returned to the buyer. In some cases, however, the buyer wants to
inquire the status of the purchase order, modify or even cancel the order while it is being
processed. Such interactions can not be expected to happen only at particular points in the order
processing. The business process must be enabled to accept requests to arrive in parallel to its
"normal" flow of control. In BPEL, this kind of asynchronous execution is called *event handling*.

```
<process name="purchaseOrderProcess" ...>
    ...
    <eventHandlers>
        <onEvent partnerLink="purchasing"
            operation="queryOrderStatus" ...>
            <scope>...</scope>
        </onEvent>
        <onEvent partnerLink="purchasing"
            operation="cancelOrder" ...>
            <scope>...</scope>
        </onEvent>
    </eventHandlers>
    ...
</process>
```
Example 4-6: Event handlers.

Event handlers are associated with the whole process or a scope. They are enabled when their
associated scope is initialized and disabled when their associated scope terminates. When

enabled, any number of events may occur. They are processed in parallel to the scope's primary activity and in parallel to each other. Message events also represent Web services operations exposed by a process and are modeled as `onEvent` elements. Timer events are modeled as `onAlarm` elements, similar to `pick` activities. In event handlers, timer events can be processed multiple times. Event handlers can never be used to create new process instances, so message events are always received by a process instance that is already active.

4.2.4.   Message Correlation

In the previous sections, we have seen several scenarios in which a process receives more than one message or exposes ("implements") more than one Web service operation. Some of the associated inbound message activities can be used to created new process instances; others are used to model situations where a running process instance receives additional requests.

If a Web service request message does not lead to the creation of a new process instance, how does it "find" the running process instance it is designated for? You may be aware of stateful Web service implementations where the target instance of the stateful service is identified reference parameters of WS-Addressing [WS-Addressing] endpoint references. This mechanism may also be used to identify process instances but BPEL does not mandate this approach. BPEL provides a portable correlation mechanism called *correlation sets*.

The major observation behind this concept is the fact that most messages exchanged between business processes and the outside world already carry the key data required to uniquely identify a process instance. For example, consider again a purchase order process. The customer sending a purchase order is usually identified by a customer id. What if multiple orders submitted by this customer are in progress? In this case, one would also associate an order number with each order. Typically such correlation data is always part of the message payload. BPEL allows defining *properties* that represent pieces of correlation information. The BPEL implementation is made aware of these properties and their location in a message by using *property alias* definitions. Finally, each inbound message activity can be associated with a set of properties that together correlate an inbound request with a unique process instance.

```
<wsdl:definitions ...>
  ...
  <wsdl:message name="sendPOResponse">
    <wsdl:part name="confirmation"
      element="po:sendPurchaseOrderResponse" />
  </wsdl:message>
  <wsdl:message name="queryPORequest">
    <wsdl:part name="query" element="po:queryPurchaseOrderStatus" />
  </wsdl:message>
  <wsdl:message name="cancelPORequest">
    <wsdl:part name="cancellation" element="po:cancelPurchaseOrder" />
  </wsdl:message>
  ...

  <vprop:property name="customerID" type="xsd:string" />
  <vprop:property name="orderNumber" type="xsd:int" />

  <vprop:propertyAlias propertyName="tns:customerID"
    messageType="tns:sendPOResponse" part="confirmation">
```

```
            <bpel:query>CID</bpel:query>
      </vprop:propertyAlias>
      <vprop:propertyAlias propertyName="tns:orderNumber"
            messageType="tns:sendPOResponse" part="confirmation">
            <bpel:query>Order</bpel:query>
      </vprop:propertyAlias>
      <vprop:propertyAlias propertyName="tns:customerID"
            messageType="tns:queryPORequest" part="query">
            <bpel:query>CID</bpel:query>
      </vprop:propertyAlias>
      <vprop:propertyAlias propertyName="tns:orderNumber"
            messageType="tns:queryPORequest" part="query">
            <bpel:query>Order</bpel:query>
      </vprop:propertyAlias>
      <vprop:propertyAlias propertyName="tns:customerID"
            messageType="tns:cancelPORequest" part="cancellation">
            <bpel:query>CID</bpel:query>
      </vprop:propertyAlias>
      <vprop:propertyAlias propertyName="tns:orderNumber"
            messageType="tns:cancelPORequest" part="cancellation">
            <bpel:query>Order</bpel:query>
      </vprop:propertyAlias>
      ...

</wsdl:definitions>
```
Example 4-7: Correlation property and property alias definitions.

The definition of the WSDL messages that carry correlation data is shown above. The two correlation properties are used to uniquely identify a process instance for a particular purchase order. For each WSDL message and each property, property alias definitions specify where the property can be located within the message.

```
<process name="purchaseOrderProcess" ...>
   <correlationSets>
      <correlationSet name="PurchaseOrder"
         properties="cor:customerID cor:orderNumber" />
   </correlationSets>

   ...
   <eventHandlers>
      <onEvent partnerLink="purchasing"
         operation="queryPurchaseOrderStatus" ...>
         <correlations>
            <correlation set="PurchaseOrder" initiate="no" />
         </correlations>
         <scope>...</scope>
      </onEvent>
      <onEvent partnerLink="purchasing"
         operation="cancelPurchaseOrder" ...>
         <correlations>
            <correlation set="PurchaseOrder" initiate="no" />
         </correlations>
         <scope>...</scope>
      </onEvent>
   </eventHandlers>
   ...
   <sequence>
```

```
        <receive partnerLink="purchasing"
            operation="sendPurchaseOrder" ...
            createInstance="yes">
        </receive>
        ...
        <reply partnerLink="purchasing"
            operation="sendPurchaseOrder" ...>
            <correlations>
                <correlation set="PurchaseOrder" initiate="yes" />
            </correlations>
        </reply>
        ...
        <!-- process the purchase order -->
        ...
    </sequence>
</process>
```
Example 4-8: Correlation sets.

In the process shown above, the process instance is created by receiving a new purchase order. The receipt of the purchase order is confirmed by returning a reply. This reply message contains the correlation information that must be presented in subsequent requests that are targeted at this process instance. Each event handler therefore refers to the same correlation set. The correlation set is initiated when the purchase order confirmation reply is sent. It is immutable after that point in time and identifies this process instance. The event handlers for purchase order status inquiries and purchase order cancellation specify `initiate="no"`. When these operations are invoked, the correlation set properties (*customerId* and *orderNumber*) must have exactly the same values as in the reply activity, otherwise, the request can not be correlated with the correct process instance.

4.2.5.   Concurrent Message Exchanges

Each inbound message activity, that is, `receive`, `onMessage`, or `onEvent`, may have an associated `reply` activity in order to implement a WSDL request-response operation. If a process has multiple `receive` and `reply` activities that point to the same partner link and WSDL operation then the association between individual `receive` and `reply` activities is ambiguous. In this case, you use the `messageExchange` attribute to establish the relationship.

```
<process ...>
    ...
    <messageExchanges>
        <messageExchange name="receiveBuyerInformation" />
        <messageExchange name="receiveSellerInformation" />
    </messageExchanges>
    ...
    <flow>
        ...
        <receive messageExchange="receiveBuyerInformation"
            partnerLink="tradingPartner"
            operation="tradingPartnerInfo" ... />
        <receive messageExchange="receiveSellerInformation"
            partnerLink="tradingPartner"
            operation="tradingPartnerInfo" ... />
        ...
        <reply messageExchange="receiveBuyerInformation"
```

```
         partnerLink="tradingPartner"
         operation="tradingPartnerInfo" ... />
      <reply messageExchange="receiveSellerInformation"
         partnerLink="tradingPartner"
         operation="tradingPartnerInfo" ... />
      ...
   </flow>
</process>
```
Example 4-9: Explicit message exchange declarations.

Two `receive-reply` pairs point to the same partner link and Web service operation, shown in the previous example. In order to disambiguate the relationship between these activities, message exchanges have been declared explicitly and referenced on the respective `receive` and `reply` activities.

Consider a message exchange and/or a partner link defined within a scope. If an inbound message activity (IMA, `receive`, `pick/onMessage`, `eventHandlers/onEvent`) referencing such a message exchange or partner link has been executed but the associated reply activity has not been executed when the scope ends then the IMA is called *orphaned*. If an orphaned IMA is detected then a `bpel:missingReply` standard fault is thrown.

4.3. More Parallel Processing

Earlier, we introduced different forms of repetitive processing – the `while`, `repeatUntil`, and `forEach` activities. In addition to these sequential loops, there is a variation of the `forEach` activity. Instead of performing each loop iteration in a sequence, all loop iterations are started at the same time and processed in parallel. Besides the `flow` activity and event handlers, this is the third form of parallel processing in BPEL.

Scenarios where the parallel `forEach` activity is useful include cases where sets of independent data are processed or where independent communication with different partners can be performed in parallel. The main difference compared to the `flow` activity is that the number of parallel branches is not known at modeling time, so a parallel `forEach` behaves like a flow with N similar child activities not constrained by links. The specified counter variable is used to iterate through a number of parallel branches, controlled by a start value and a final value. It can be used for indexed access to a specific element when a set of elements is defined having the values of the XML schema `minOccurs` and `maxOccurs` attributes greater than one.

The `forEach` activity allows specifying a completion condition. You would use it if not all branches are required to complete, for example, when parallel requests are sent out and a sufficiently large subset of the recipients have responded.

In the following example, we show how the `forEach` activity can be used. The example has the following structure:

- Obtain a list of business partners (Endpoint References)
- Initialize a list of quotes
- For each business partner EPR, perform the following in parallel:
  - Assign the EPR to a partner link local to the forEach scope

- o Send out a request for a quote
- o Receive quote
- Leave the parallel forEach when 50% of the responses have arrived
- Work with the received quotes

The example uses an XSL transformation "initQuotes.xsl" in order to initialize the output variable for the list of quotes: for each partnerEPR, an empty quote element is created.

```xml
<xsl:transform version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:sref="http://docs.oasis-open.org/wsbpel/2.0/serviceref"
   xmlns:bo="http://example.com/bo">
   <xsl:template match="bo:partnerEPRs">
      <foo:responses>
         <xsl:for-each select="sref:service-ref">
            <bo:quote />
         </xsl:for-each>
      </foo:responses>
   </xsl:template>
</xsl:transform>
```
Example 4-10: XSL transformation used in the forEach activity.

The WS-BPEL code looks as follows:

```xml
<invoke name="retrieveBusinessPartners"
   partnerLink="localDirectory"
   operation="retrieveBusinessPartners"
   inputVariable="filterCriteria"
   outputVariable="partnerEPRs" />

<assign><!-- initialize list of quotes -->
   <copy>
      <from>
         bpel:doXslTransform("initQuotes.xsl",$partnerEPRs)
      </from>
      <to>$quotes</to>
   </copy>
</assign>

<forEach parallel="yes" countername="n">
   <startCounterValue>1</startCounterValue>
   <finalCounterValue>
      count($partnerEPRs/sref:service-ref)
   </finalCounterValue>
   <completionCondition>
      <branches>ceiling(0.5*count($partnerEPRs/sref:service-ref))</branches>
   </completionCondition>
   <scope>
      <partnerLinks>
         <partnerLink name="address" partnerLinkType="..."
            partnerRole="..." myRole="..." />
      </partnerLinks>

      <sequence>
         <assign><!-- get one business partner's EPR -->
            <copy>
```

```
                    <from>$partnerEPRs[$n]</from>
                    <to partnerLink="address" />
                </copy>
            </assign>

            <invoke name="requestQuote"
                partnerLink="address"
                operation="requestQuote"
                inputVariable="quoteRequest" />

            <receive name="receiveQuote"
                partnerLink="address"
                operation="receiveQuote"
                variable="quote" />

            <assign>
                <copy>
                    <from variable="quote" />
                    <to>$quotes[$n]</to>
                </copy>
            </assign>
        </sequence>

    </scope>
</forEach>

<!-- ... work with the list of received quotes ... -->
```
Example 4-11: ForEach activity (parallel).

4.4. Delayed Execution

In some situations, the execution of the business logic cannot continue immediately. The process has to wait for a specified time period or until a certain point in time is reached. The wait activity indicates that processing will be suspended. Note that concurrent branches in the process are not affected.

```
<wait><!-- wait for three days and ten hours to go by ... -->
    <for>'P3DT10H'</for>
</wait>
```
Example 4-12: Wait activity.

Duration-valued or deadline-valued XPath expressions can be specified in the wait activity. If a point in time is specified that has already passed, or if a zero or negative duration is specified, then the wait activity completes immediately.

4.5. Immediately Ending a Process

When a process encounters exceptional situations, it may deal with them using fault handling, termination handling, and compensation handling mechanisms introduced earlier. In case of unexpected severe failures, there may not always be a reasonable way of dealing with them. The exit activity can be used to immediately end all currently running activities, on all parallel branches, without involving any termination handling, fault handling, or compensation behavior.

```
<if>
   <condition>...</condition><!-- the good case -->
   ...
   <elseif>
      <condition>...</condition><!-- handle expected error -->
      ...
   </elseif>
   ...
   <else><!-- unexpected error – unable to handle ... -->
      <exit />
   </else>
</if>
```
Example 4-13: Exit activity.

When using the `exit` activity, you must be aware that any open conversations are also affected, that is, other partners interacting with the process may be waiting for a response that will never arrive.

4.6. Doing Nothing

The `empty` activity is used to specify that no action is to be taken, that is, this is the BPEL rendering of a no-op activity. This activity is used for fault handlers that consume a fault without acting on it. Other use cases for the empty activity include synchronization points in a flow, or placeholders for activities that are to be added later.

```
<empty>
   <targets>
      <target linkName="left-branch" />
      <target linkName="right-branch" />
   </targets>
</empty>
```
Example 4-14: Empty activity.

4.7. Data Validation

A business process receives data from partners via inbound message activities. The `assign` activity introduced earlier provides means for simple data manipulation. The result of such updates performed on BPEL variables cannot always be guaranteed to be valid according to the WSDL message or XML schema type/element with which the variable is declared. In case of complex-typed variables, multiple assignment steps may be required to create a value, so it may not even be avoidable that intermediate results are not valid. In order to make sure that variable contents are valid according to the variable declaration, BPEL provides two explicit approaches – the `validate` activity and the `validate` attribute of the `assign` activity.

```
<scope>
   <faultHandlers>
      <catch faultName="bpel:invalidVariables">
         <reply name="invalidPurchaseOrder" ... />
      </catch>
   </faultHandlers>
   <sequence>
```

```
      <receive name="receivePurchaseOrder" variable="purchaseOrder" ... />
      ...
      <validate name="validatePurchaseOrder" variables="purchaseOrder" />
      <reply name="acknowledgeReceipt" ... />
      ...
   </sequence>
</scope>
```
Example 4-15: Validate activity.

The `validate` activity validates a list of specified variables against their corresponding XML definition. If the `validate` attribute of the `assign` activity is set to `"yes"` then the `assign` activity validates all the variables being modified by the activity. In case of a WSDL message variable, each message part is validated. The standard fault `bpel:invalidVariables` is thrown, if one of the variables is invalid against its corresponding XML definition.

Explicit variable validation at runtime implies additional resource consumption that is not always desirable. It may be necessary to validate certain variables during a test phase or in exceptional situations without modifying the business process each time. For this purpose, the BPEL implementation may provide means to turn on/off explicit validation.

4.8. Concurrent Data Manipulation

We have discussed several different ways for modeling concurrent behavior in earlier sections. The process model may contain a fixed number of parallel activities in a `flow` activity or a variable number of concurrent branches in a `forEach` activity or event handler instances. It is not new news that one should be careful when such parallel activities access the same global data.

The BPEL scope provides an approach for controlling global data access. Consider multiple scopes containing activities reading or updating the same variables. When the attribute `isolated` is set to "yes" on each of these scopes then they no longer step on each other's feet. If activities in concurrent scopes access shared variables then these accesses are protected in the same way as if all such activities within one scope are completed before any in another scope.

```
<process ...>
   <variables>
      <variable name="global" element="..." />
   </variables>
   <flow>
      <scope name="S1" isolated="yes">
         <sequence>
            ...
            <invoke ... outputVariable="global" />
            ...
         </sequence>
      </scope>
      <scope name="S2" isolated="yes">
         <sequence>
            ...
            <assign>
```

```
                <copy>
                    <from>...</from>
                    <to variable="global" />
                </copy>
            </assign>
            ...
        </sequence>
    </scope>
    </flow>
</process>
```
Example 4-16: Isolated scopes accessing a global variable.

The same isolation semantics applies to variable accesses via properties and to endpoint reference assignments from and to partner links.

The isolation domain of a scope also includes the execution of event handlers, fault handlers, and termination handlers. A compensation handler of an isolated scope is also isolated.

If a isolated scope contains child scopes then these child scopes must have their `isolated` attribute set (or defaulted) to `"no"`. Access to shared variables from within such child scopes is controlled by their enclosing isolated scope.

4.9. Dynamic Business Partner Resolution

Before a BPEL partner link can be used to invoke a partner's Web service operation, the partner link must be associated with a concrete Web service endpoint address. This can be done during the deployment of the process (which is out of scope of the BPEL specification), or dynamically at runtime.

A variant of the BPEL assign activity is used to assign an endpoint reference to a partner link. Endpoint references can be those defined in [WS-Addressing], enclosed by a neutral container element `sref:service-ref`.

```
<sref:service-ref
    reference-scheme="http://www.w3.org/2005/08/addressing">
    <wsa:EndpointReference
        xmlns:wsa="http://www.w3.org/2005/08/addressing">
        ...
    </wsa:EndpointReference>
</sref:service-ref>
```
Example 4-17: Service reference.

Note that the optional `reference-scheme` attribute would need to be used when the child element of the `sref:service-ref` is ambiguous.

Let's have a look at an example. Consider a process that receives an endpoint reference which is then used for a later invocation of a Web service.

```
<process name="purchaseOrderProcess" ...>
    <partnerLinks>
        <partnerLink name="myCustomer"
```

```
            partnerLinkType="lns:purchaseOrderLT"
            myRole="purchaseOrderService" />
         partnerRole="customer"/>
   </partnerLinks>
   ...
   <receive partnerLink="myCustomer"
       operation="sendPurchaseOrder"
       variable="purchaseOrder" />
   ...
   <assign>
      <copy>
         <from>$purchaseOrder/callback</from>
         <to partnerLink="myCustomer" />
      </copy>
   </assign>
   ...
   <invoke partnerLink="myCustomer"
       operation="sendInvoice"
       variable="invoice" />
   ...
</process>
```

Example 4-18: Endpoint reference assignment to a partner link.

In the example above, a purchase order process exposes a one-way operation for submitting an order. The order contains an endpoint reference (wrapped in a `sref:service-ref` element) that points to another one-way operation provided by the caller. After assigning this endpoint reference to the partner role side of the partner link associated with the caller, the callback operation can be invoked in order to return an invoice.

Now what would it look like if the calling application is itself implemented as a BPEL process? This customer process has to submit a purchase order, containing a callback endpoint reference. In addition, it has to provide the callback operation itself in order to receive the invoice.

```
<process name="customerProcess" ...>
   <partnerLinks>
      <partnerLink name="myPurchaseOrderService"
          partnerLinkType="lns:purchaseOrderLT"
          partnerRole="purchaseOrderService" />
         myRole="customer"/>
   </partnerLinks>
   ...
   <assign>
      <copy>
         <from partnerLink="myPurchaseOrderService"
             endpointReference="myRole" />
         <to>$purchaseOrder/callback</to>
      </copy>
   </assign>
   ...
   <invoke partnerLink="myPurchaseOrderService"
       operation="sendPurchaseOrder"
       variable="purchaseOrder" />
   ...
   <receive partnerLink="myPurchaseOrderService"
       operation="sendInvoice"
       variable="invoice" />
```

```
   ...
</process>
```
Example 4-19: Endpoint reference assignment from a partner link.

The customer process is shown in the previous example. The endpoint reference pointing to the callback operation is assigned to an element contained in the purchase order request. It is then sent to the purchase order service, which eventually uses this endpoint reference to return the invoice.

Note that in this example, it is required that the callback operation is performed on the same customer's process instance that submitted the purchase order in the first place. This may either be done by using BPEL correlation sets. Alternatively, the implementation may choose to use stateful endpoint references carrying instance identification information, for example, within a WS-Addressing `ReferenceParameters` element.

5. Advanced Concepts II

5.1. Language Extensibility

BPEL is extensible. Every BPEL implementation may support BPEL processes containing expression languages and/or query languages other than XPath 1.0 which is mandated by the BPEL specification. Moreover, a BPEL implementation may support adding XML elements and/or attributes of non-BPEL namespaces to a BPEL process.

5.1.1. Expression Languages and Query Languages

For the purpose of data access and data manipulation, many elements in BPEL contain an expression or query. By default, this is specified as an XPath 1.0 literal. Note that XPath 1.0 is not only the default but also the only expression/query language mandated by BPEL.

In the example below, a boolean XPath expression "`$counter < 42`" is used in a `condition` in order to determine whether a `while` loop performs another iteration.

```
<while>
   <condition>$counter &lt; 42</condition>
   ...
</while>
```
Example 5-1: Condition with XPath 1.0 expression.

Alternatively, the BPEL implementation may support other expression languages or query languages. As an example, consider a boolean expression written in Java. For simplification, we just use "`http://www.example.com/java`" as the value of the expressionLanguage attribute and assume that the value of the BPEL variable `counter` is represented by the Java variable of the same name.

```
<while>
   <condition expressionLanguage="http://www.example.com/java">
      counter &lt; 42
   </condition>
   ...
</while>
```
Example 5-2: Condition with Java expression.

5.1.2. Elements and Attributes of Other Namespaces

Elements and attributes of XML namespaces other than those provided by BPEL itself can be added to almost every element of a BPEL process. In the following example, the `invoke` activity has been extended with an element used for rendering the `invoke` activity in a process modeling tool.

```
<process name="purchaseOrderProcess" ...
   xmlns:tool="http://example.com/bpel/editorElements">
```

```
    ...

    <invoke partnerLink="shipping"
        operation="requestShipping"
        inputVariable="shippingRequest"
        outputVariable="shippingInfo">
        <documentation>decide on shipper</documentation>
        <tool:myIcon>shipping/requestShipping.gif</tool:myIcon>
    </invoke>

    ...
</process>
```
Example 5-3: Extension without runtime semantics.

In order to avoid ambiguities with respect to the XML schema definition for BPEL, certain language extension points require the use of explicit BPEL wrapper elements, such as `extensionActivity` and `extensionAssignOperation`. Another explicit wrapper for elements from a namespace other than BPEL – `literal` – is used to wrap literal XML data elements in assignments to BPEL variables.

The following example shows a non-standard activity used to perform a user interaction; it is wrapped in a BPEL `extensionActivity` element.

```
<process name="purchaseOrderProcess" ...
    xmlns:user="http://example.com/bpel/userInteractions">
    ...

    <if>
        <condition>$amount &gt; 1000000</condition>
        <extensionActivity>
            <user:userInteraction type="user:approval">
                <user:userResolution role="manager" />
            </user:userInteraction>
        </extensionActivity>
    </if>

    ...
</process>
```
Example 5-4: Extension with runtime semantics.

BPEL provides a declaration element for the extension namespace used to specify whether the extension elements must be understood by a BPEL implementation. The example below refers to the two previously introduced extension namespaces. Some of these extension elements (namespace prefixed with `tool`) are used in a BPEL process editor only and carry no particular runtime semantics, therefore, they can declared as `mustUnderstand="no"` and ignored when the process is executed. Other extension elements (namespace prefixed with `user`) represent an integral part of the business logic and are declared as `mustUnderstand="yes"` such that the process must be rejected if a BPEL implementation doesn't know how to interpret them.

```
<process ...
    xmlns:tool="http://example.com/bpel/editorElements"
    xmlns:user="http://example.com/bpel/userInteractions">
```

```
   <extensions>
      <extension namespace="http://example.com/bpel/editorElements"
         mustUnderstand="no" />
      <extension namespace="http://example.com/bpel/userInteractions"
         mustUnderstand="yes" />
   </extensions>

   ...
</process>
```
Example 5-5: Extension declarations.

5.2. Abstract Processes

Abstract processes describe process behavior partially without covering every detail of execution. An Abstract Process can be "implemented" by a set of executable processes. The "implementation" is also known as executable completion. Elements and attributes of an executable process may be hidden in an Abstract Process, by either just omitting them or by replacing them with opaque constructs. The common language constructs of abstract and executable processes have the same semantics.

Usage patterns include

- Abstraction – using an Abstract Process for showing only certain aspects of an executable process, for example, external interactions using a particular partner link.
- Refinement – using an Abstract Process as a starting point for developing an executable process.
- Protocol matching – using Abstract Processes for determining whether processes provided by two business partners can interact with each other, potentially in a conversation involving multiple steps.

The following sections describe two key Abstract Process concepts: the Common Base and Abstract Process Profiles. The Common Base defines the basic syntax requirements for all Abstract Processes. Abstract Process Profiles specify the allowed subset of the Common Base and allowed executable completions.

5.2.1.   The Common Base

The Common Base provides the syntax rules for Abstract Processes. An Abstract Process may contain the same language constructs as an executable process. In addition, constructs of an executable process may be omitted or replaced by explicit opaque tokens, according to the constraints defined in the specified Abstract Process Profile. The Common Base defines four types of opaque tokens: activities, expressions, attributes and from-specs.

The example snippet below shows a `receive` activity in an Abstract Process where both the operation and the variable of a corresponding executable process are replaced by an opaque token.

```
<receive partnerLink="shippingRequester"
   operation="##opaque"
```

```
      variable="##opaque" />
```
Example 5-6: Abstract receive activity.

5.2.2.　　Abstract Process Profile for Observable Behavior

The externally observable behavior of a business process is the sequence of inbound and outbound interactions with other partners. An Abstract Process may only contain such external interactions referring to a particular partner link and thereby define the contract of the business process with e.g. another enterprise. An executable process must then adhere to this contract – it could be modeled by refining the Abstract Process according to the executable completion rules defined in this profile.

As an example scenario, consider the following sequence of steps. The result is a pair of executable processes EP1 and EP2 that are compatible in the sense that every outbound interaction in one process has a corresponding inbound operation in the other process.

We begin with an executable process EP1.

```
<process name="purchaseOrderProcess" ...
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

   <partnerLinks>
      <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT"
         myRole="purchaseService" />
      ...
   </partnerLinks>

   <sequence>
      <receive partnerLink="purchasing"
         operation="sendPurchaseOrder"
         variable="PO"
         createInstance="yes" />

      <flow>
         ... shipping, invoicing and scheduling ...
      </flow>

      <invoke partnerLink="purchasing"
         operation="returnInvoice"
         inputVariable="invoice" />
   </sequence>
</process>
```
Example 5-7: Executable process EP1.

A corresponding Abstract Process AP1 is created such that EP1 is a valid executable completion of AP1. Activities of EP1 not relevant for the externally observable behavior over one partner link of EP1 are omitted or replaced by opaque activities. In this case, we replace the flow by an opaque activity.

```
<process name="purchaseOrderProcess" ...
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
   abstractProcessProfile="http://docs.oasis-
open.org/wsbpel/2.0/process/abstract/ap11/2006/08">
```

```
   <partnerLinks>
      <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT"
         myRole="purchaseService" />
      ...
   </partnerLinks>

   <sequence>
      <receive partnerLink="purchasing"
         operation="sendPurchaseOrder"
         variable="PO"
         createInstance="yes" />

      <opaqueActivity>
         ... shipping, invoicing and scheduling ...
      </opaqueActivity>

      <invoke partnerLink="purchasing"
         operation="returnInvoice"
         inputVariable="invoice" />
   </sequence>
</process>
```

Example 5-8: Abstract process AP1, created from executable process EP1.

For the Abstract Process AP1, a compatible Abstract Process AP2 is created by mirroring all inbound and outbound activities of AP1 to corresponding outbound and inbound activities of AP2, respectively.

```
<process name="customerProcess" ...
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
   abstractProcessProfile="http://docs.oasis-
open.org/wsbpel/2.0/process/abstract/ap11/2006/08">

   <partnerLinks>
      <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT"
         myRole="customer" />
      ...
   </partnerLinks>

   <sequence>
      <opaqueActivity>... create process instance ...</opaqueActivity>

      <invoke partnerLink="purchasing"
         operation="sendPurchaseOrder"
         inputVariable="PO" />

      <opaqueActivity>... do other work ...</opaqueActivity>

      <receive partnerLink="purchasing"
         operation="returnInvoice"
         variable="invoice" />
   </sequence>
</process>
```

Example 5-9: Abstract process AP2, compatible with Abstract Process AP1.

Finally, the Abstract Process AP2 is then handed out to a customer who refines it to a valid executable completion EP2. Note that the added start activity uses a different partner link.

```
<process name="customerProcess" ...
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

   <partnerLinks>
      <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT"
         myRole="customer" />
      ...
   </partnerLinks>

   <sequence>
      <receive partnerLink="internal" ... createInstance="yes" />

      <invoke partnerLink="purchasing"
         operation="sendPurchaseOrder"
         inputVariable="PO" />

      <sequence>... prepare additional orders ...</sequence>

      <receive partnerLink="purchasing"
         operation="returnInvoice"
         variable="invoice" />
   </sequence>
</process>
```
Example 5-10: Executable process EP2, created from Abstract Process AP2.

5.2.3.   Abstract Process Profile for Templates

Templates are Abstract Processes with explicit opaque extension points. The main usage scenario is to allow process developers to complete execution details at these extension points at a later point in time. It is particularly useful to separate works between process analysts and process developers. While the former can focus the higher business flow design, the latter can focus the system concern of the process execution (for example, atomic transaction and system failure recovery). An Abstract Process template can be used as a mean to round-tripping the changes from both sides as well.

If a start activity is hidden in a template then it is represented by an opaque activity with the `template:createInstance="yes"` attribute.

For further details, please see section "13.4. Abstract Process Profile for Templates" and the example of an Abstract Process based on template profile in section "15.2. Ordering Service" in WS-BPEL 2.0 specification [WS-BPEL 2.0].

6. Using WS-BPEL

6.1. Applying WS-BPEL to our scenario

This section proceeds step-by-step through the process of developing a WS-BPEL definition that describes a case study scenario outlined below.

6.1.1.   Introduction

The case study in this primer follows the step-by-step process which a consulting firm named Perspective Technology Corp. (hereafter referred to as "PTC") uses to build a WS-BPEL process definition for their new Timesheet Submission business process.

When a consultant submits a timesheet, four services are used to process it. A timesheet typically contains a number of entries corresponding to the time spent by a consultant working for a client. Each of these entries is validated with the Timesheet service. PTC employs consultants of varying skills/grades, on which their rates are based. The Employee service is used to access detailed information on a consultant in order to determine rates. PTC then sends an invoice(s) to the client(s) and notifies a payment receivable service for follow-up. The Timesheet Submission process finally sends a reply back to the consultant.
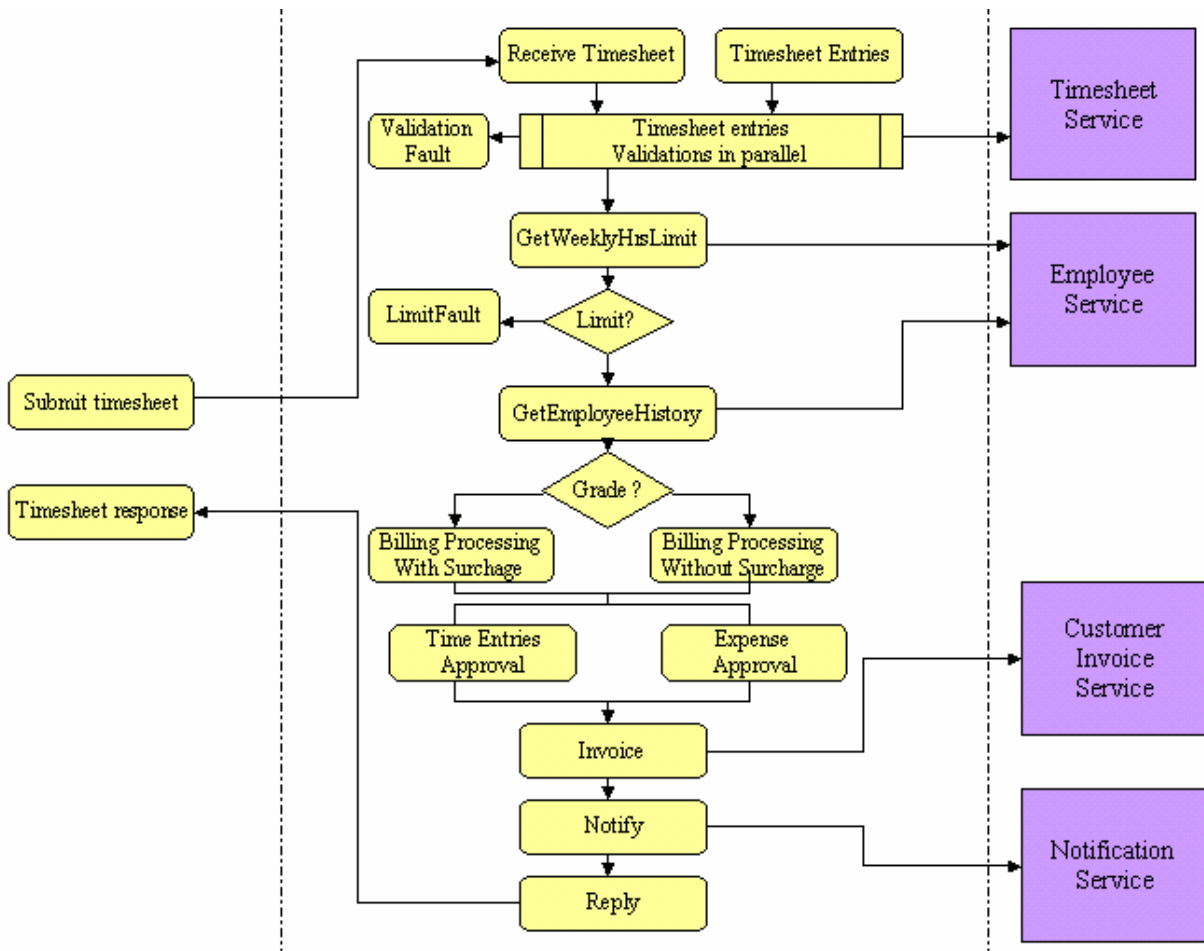
6.1.2.   The TimesheetSubmission Process

Prior to walking through the development of the WS-BPEL, the process definition in its entirety is shown below to provide the context for the scenario.

```
<process name="TimesheetSubmission"
   targetNamespace="http://www.xmltc.com/ptc/process/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable/"
   xmlns:bpl="http://www.xmltc.com/ptc/process/"
   xmlns:emp="http://www.xmltc.com/ptc/employee/"
   xmlns:inv="http://www.xmltc.com/ptc/invoice/"
   xmlns:tst="http://www.xmltc.com/ptc/timesheet/"
   xmlns:not="http://www.xmltc.com/ptc/notification/">

   <partnerLinks>
      <partnerLink name="client"
         partnerLinkType="tns:TimesheetSubmissionType"
         myRole="TimesheetSubmissionServiceProvider" />
      <partnerLink name="Invoice"
         partnerLinkType="inv:InvoiceServiceType"
         partnerRole="InvoiceServiceProvider" />
      <partnerLink name="Timesheet"
```

```
            partnerLinkType="tst:TimesheetServiceType"
            partnerRole="TimesheetServiceProvider" />
        <partnerLink name="Employee"
            partnerLinkType="emp:EmployeeServiceType"
            partnerRole="EmployeeServiceProvider" />
        <partnerLink name="Notification"
            partnerLinkType="not:NotificationServiceType"
            partnerRole="NotificationServiceProvider" />
    </partnerLinks>

    <variables>
        <variable name="ClientSubmission"
            messageType="bpl:receiveSubmitMessage" />
        <variable name="EmployeeHoursRequest"
            messageType="emp:getWeeklyHoursRequestMessage" />
        <variable name="EmployeeHoursResponse"
            messageType="emp:getWeeklyHoursResponseMessage" />
        <variable name="EmployeeHistoryRequest"
            messageType="emp:updateHistoryRequestMessage" />
        <variable name="EmployeeHistoryResponse"
            messageType="emp:updateHistoryResponseMessage" />
        ...
    </variables>

    <scope name="mainScope">
        <faultHandlers>
            <catch faultName="SomethingBadHappened"
                faultVariable="TimesheetFault" faultElement="...">
                ...
            </catch>
            <catchAll>
                <compensateScope target="invoiceSubmissionScope" />
            </catchAll>
        </faultHandlers>
        <sequence>
            <receive name="receiveInput"
                partnerLink="client"
                operation="Submit"
                variable="ClientSubmission"
                createInstance="yes" />
            ...
            <assign>
                <copy>
                    <from variable="EmployeeData" part="grade">
                        <query>employeeGradeInformation</query>
                    </from>
                    <to variable="EmployeeRateInput />
                </copy>
            </assign>
            ...
            <flow>
                <links>
                    <link name="timesheetEntriesApproval" />
                    <link name="timesheetExpensesApproval" />
                </links>
                <receive name="receiveEntriesApproval" ...>
                    <sources>
                        <source linkName="timesheetEntriesApproval" />
```

```xml
                </sources>
                <correlations>
                    <correlation set="tradeID" initiate="no" />
                </correlations>
            </receive>
            <receive name="receiveExpensesApproval" ...>
                <sources>
                    <source linkName="timesheetExpensesApproval" />
                </sources>
                <correlations>
                    <correlation set="timesheetID" initiate="no" />
                </correlations>
            </receive>
            <scope name="invoiceSubmissionScope">
                <targets>
                    <joinCondition>
                        $timesheetEntriesApproval and
                        $timesheetExpensesApproval
                    </joinCondition>
                    <target linkName="timesheetEntriesApproval" />
                    <target linkName="timesheetExpensesApproval" />
                </targets>
                <compensationHandler>
                    <invoke name="withdrawInvoiceSubmission" ... />
                </compensationHandler>
                <invoke name="submitInvoice" ... />
            </scope>
            ...
        </flow>
        ...
        <invoke name="retrieveMonthlyAccountingPostDateTime"
            partnerLink="AccountingService"
            operation="getMonthlyPostDate"
            inputVariable="monthAndYear"
            outputVariable="monthlyPostDateTime" />
        <wait>
            <until>$monthlyPostDateTime</until>
        </wait>
        <invoke partnerLink="AccountingService"
            operation="postAccountingEntries"
            inputVariable="accountingEntries" />

        <invoke name="ValidateWeeklyHours"
            partnerLink="Employee"
            operation="GetWeeklyHoursLimit"
            inputVariable="EmployeeHoursRequest"
            outputVariable="EmployeeHoursResponse" />
        ...
        <assign>
            <copy>
                <from variable="TimesheetSubmissionFailedMessage" />
                <to variable="EmployeeNotificationMessage" />
            </copy>
            <copy>
                <from variable="TimesheetSubmissionFailedMessage" />
                <to variable="ManagerNotificationMessage" />
            </copy>
        </assign>
```

```
        ...
        <reply partnerLink="client"
           operation="SubmitTimesheet"
           variable="TimesheetSubmissionResponse" />
        ...
     </sequence>
  </scope>

</process>
```
Example 6-1: The TimesheetSubmission process.

6.1.3. Getting Started: Defining the process element

Before writing the WS-BPEL process definition, a WS-BPEL target namespace URI for it needs to be defined. The WS-BPEL process definition can then be started with the following skeleton:

```
<process name="TimesheetSubmission"
   targetNamespace="http://www.xmltc.com/ptc/process/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable/"
   xmlns:bpl="http://www.xmltc.com/ptc/process/"
   xmlns:emp="http://www.xmltc.com/ptc/employee/"
   xmlns:inv="http://www.xmltc.com/ptc/invoice/"
   xmlns:tst="http://www.xmltc.com/ptc/timesheet/"
   xmlns:not="http://www.xmltc.com/ptc/notification/">

   <partnerLinks>
      ...
   </partnerLinks>
   <variables>
      ...
   </variables>
   <sequence>
      ...
   </sequence>
   ...
</process>
```
Example 6-2: A skeleton process definition.

At this point, one can start defining the TimesheetSubmission process.

6.1.4. Defining partnerLinks

The TimesheetSubmission process will interact with other parties during execution, so it is good to start with defining the partnerLink definitions in the partnerLinks section.

The four partnerLink definitions shown below correspond to the consumer of the Timesheet Submission process (client), as well as the providers of invoicing (Invoice), timesheet data (Timesheet), employee data (Employee), and notification services (Notification).

```
<partnerLinks>
   <partnerLink name="client"
      partnerLinkType="tns:TimesheetSubmissionType"
      myRole="TimesheetSubmissionServiceProvider" />
   <partnerLink name="Invoice"
```

```
      partnerLinkType="inv:InvoiceServiceType"
      partnerRole="InvoiceServiceProvider" />
  <partnerLink name="Timesheet"
      partnerLinkType="tst:TimesheetServiceType"
      partnerRole="TimesheetServiceProvider" />
  <partnerLink name="Employee"
      partnerLinkType="emp:EmployeeServiceType"
      partnerRole="EmployeeServiceProvider" />
  <partnerLink name="Notification"
      partnerLinkType="not:NotificationServiceType"
      partnerRole="NotificationServiceProvider" />
</partnerLinks>
```
Example 6-3: The partnerLinks construct containing one partnerLink for an invoking external partner and four partnerLinks for partners invoked by the process service.

6.1.5.   Defining partnerLinkTypes

In Example 7-2, each partnerLink contains a partnerLinkType attribute, which references a partnerLinkType. In the below example WSDL, the partnerLinkType represents the interaction between the timesheet submission process and the employee data service:

```
<definitions name="Employee"
   targetNamespace="http://www.xmltc.com/ptc/employee/wsdl/"
   xmlns:emp="http://www.xmltc.com/ptc/employee/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype/" ...>
   ...
   <plnk:partnerLinkType name="EmployeeServiceType">
      <plnk:role name="EmployeeServiceProvider"
         portType="emp:EmployeeInterface" />
   </plnk:partnerLinkType>
   ...
</definitions>
```
Example 6-4: The employee data service WSDL definition containing partnerLinkType definitions.

The below example WSDL, the partnerLinkType represents the interaction between the timesheet submission process and the notification service:

```
<definitions name="Notification"
   targetNamespace="http://www.xmltc.com/ptc/notification/wsdl/"
   xmlns:not="http://www.xmltc.com/ptc/notification/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype/" ...>
   ...
   <plnk:partnerLinkType name="NotificationServiceType">
      <plnk:role name="NotificationServiceProvider"
         portType="not:NotificationInterface" />
   </plnk:partnerLinkType>
   ...
</definitions>
```
Example 6-5: The notification service WSDL definition containing partnerLinkType definitions.

6.1.6.   Defining variables

At this point, one needs to decide how state is maintained between message exchanges, in the Variables section. The data variables that are globally used by the process to store state information, along with any other state information to be maintained for the process logic itself, are defined here.

In the example below, variable elements for WSDL message types are defined using messageType. The type, or element attributes could likewise be used, to define variables for simple/complex XML Schema types or for XML Schema elements, respectively.

```
<variables>
   <variable name="ClientSubmission"
      messageType="bpl:receiveSubmitMessage" />
   <variable name="EmployeeHoursRequest"
      messageType="emp:getWeeklyHoursRequestMessage" />
   <variable name="EmployeeHoursResponse"
      messageType="emp:getWeeklyHoursResponseMessage" />
   <variable name="EmployeeHistoryRequest"
      messageType="emp:updateHistoryRequestMessage" />
   <variable name="EmployeeHistoryResponse"
      messageType="emp:updateHistoryResponseMessage" />
   ...
</variables>
```
Example 6-6: The variables section hosting a selection of the variable elements used by the Timesheet Submission process.

6.1.7.   Using getVariableProperty function

Next, one will need to process data stored in or associated with variables. The bpel:getVariableProperty function is used here to retrieve property values from variables. For example, in the Timesheet Submission process, the employee's grade needs to be extracted from the EmployeeHistoryResponse value. The getVariableProperty function is used in this case as follows:

```
   bpel:getVariableProperty("EmployeeHistoryResponse", "emp:grade")
```
Example 6-7: Using bpel:getVariableProperty() to extract an employee's grade from the EmployeeHistoryResponse value in a Timesheet Submission process.

6.1.8.   Defining Process Logic

Now that partners and global variables have been specified, the Timesheet Submission process needs to do some meaningful work with them. Here, WS-BPEL constructs that structure process logic to express control patterns, handling of faults and external events, and coordination of message exchanges will be used in the following sections.

6.1.9.   Defining a sequence activity

In the Timesheet Submission process, activities need to be organized so that they are executed in a pre-defined, sequential order, using the Sequence activity.

```
<sequence>
```

```
   <receive>...</receive>
   <assign>...</assign>
   <invoke>...</invoke>
   <reply>...</reply>
</sequence>
```
Example 6-8: A sequence activity containing some activity elements provided by WS-BPEL. WS-BPEL provides numerous activities that can be used to express process logic within a process definition.

6.1.10.   Defining an if activity

In the Timesheet Submission process, conditional behavior is needed for certain activities to decide between two or more branches, using the If activity. In this case, which checks an employee's grade for performing specific logic, the condition is exercised using data extracted by the bpel:getVariableProperty() function:

```
<if name="isEmployeeGradeGreaterThan10">
   <condition>
      bpel:getVariableProperty( "EmployeeHistoryResponse","emp:grade") > 10
   </condition>
   <invoke name="calculateSurcharge" ... />
   <else>
      <reply name="sendNoSurchargeInformation" ... />
      ...
   </else>
</if>
```
Example 6-9: An if activity containing conditions for evaluating a branch to process billing surcharges for employees above grade "10", using the getVariableProperty function.

In the next case, which checks the hourType for a Timesheet Submission, the condition is exercised using a WS-BPEL function:

```
<if name="isAdministriativeHours">
   <condition>$EmployeeHoursResponse/hourType = 6</condition>
   <invoke name="validateAdministrativeHours" ... />
   <else>
      <reply name="processNoAdministrativeHours" ... />
      ...
   </else>
</if>
```
Example 6-10: An if activity containing conditions for evaluating a branch to process hour limits for administrative tasks.

6.1.11.   Defining a while activity

In the Timesheet Submission process, certain bits of process logic will need to be repeatedly executed, using the While activity. In this case, invoice validation is repeatedly performed as long as the defined invoice status condition is met.

```
<while>
   <condition>
      bpel:getVariableProperty( "InvoiceStatusResponse","inv:status")
      &lt; 9
   </condition>
```

```
    <sequence>
        <invoke name="invoiceValidation" ... />
        <receive name="receiveInvoiceValidation"
            partnerLink="InvoiceValidation"
            operation="returnInvoiceValidation"
            variable="InvoiceStatusResponse" />
    </sequence>
</while>
```
Example 6-11: A while activity containing a condition for evaluating the invoice status for a timesheet submission, and processing invoice validation as long as the status is not "9".

It is important to reiterate that the condition is evaluated at the beginning of each While iteration; as such, the body of the while may not be executed at all, if the condition never holds. The variable must have been initialized or an error will occur.

6.1.12. Defining a repeatUntil activity

If the body of the activity must be performed at least once, the repeatUntil activity would be used in place of while. In this case, timesheet validation needs to be performed at least once, then repeatedly depending on completion status.

```
<repeatUntil>
    <sequence>
        <invoke name="timesheetValidation" ... />
        <receive name="receiveValidation"
            partnerLink="TimesheetValidation"
            operation="returnTimesheetValidation"
            variable="TimesheetStatusResponse" />
    </sequence>
    <condition>
        bpel:getVariableProperty(
        "TimesheetStatusResponse","tst:completionStatus") < 5
    </condition>
</repeatUntil>
```
Example 6-12: A repeatUntil activity processing timesheet completion validation, containing a condition for evaluating the timesheet completion status for a timesheet submission until the status is updated to "5".

6.1.13. Defining a forEach activity

In the Timesheet Submission process, each timesheet record entry is required to validate employee grade and expense type for the client receiving the service (which may restrict certain types for certain employee grades), using the forEach activity.

For this example, the Timesheet Submission process obtains the list of timesheet expense entries, and for each entry, perform the following in parallel:

- Assign the EPR to a partner link local to the forEach scope
- Send out a request for validation
- Receive a boolean response
- Leave the parallel forEach when all responses have arrived

```
<invoke name="retrieveExpenseList"
```

```
            partnerLink="TimesheetService"
            operation="query"
            inputVariable="filterCriteria"
            outputVariable="expenseList" />

<forEach parallel="yes" countername="n">
    <startCounterValue>1</startCounterValue>
    <finalCounterValue>
        count($expenseList.payload/expenseItem)
    </finalCounterValue>
    <scope>
        <partnerLinks>
            <partnerLink name="ExpenseValidation"
                partnerLinkType="exp:ExpenseValidationType"
                partnerRole="ExpenseValidationProvider" myRole="consumer" />
        </partnerLinks>
        <variables>
            <variable name="expenseDetailsMsg" element="..." />
            <variable name="validated" element="..." />
        </variables>
        <sequence>
            <assign>
                <copy>
                    <from>
                        $expenseList.payload/expenseItem[$n]/validatorEPR
                    </from>
                    <to partnerLink="ExpenseValidation" />
                </copy>
                <copy>
                    <from>
                        $expenseList.payload/expenseItem[$n]/expenseDetails
                    </from>
                    <to variable="expenseDetailMsg" />
                </copy>
            </assign>
            <invoke name="requestValidation"
                partnerLink="ExpenseValidation"
                operation="validateExpense"
                inputVariable="expenseDetailMsg" />
            <receive name="receiveValidation"
                partnerLink="ExpenseValidation"
                operation="returnValidation"
                variable="validated" />
            <assign>
                <copy>
                    <from variable="validated" />
                    <to>$expenseList.payload/expenseItem[$i]/validation</to>
                </copy>
            </assign>
        </sequence>
    </scope>
</forEach>

... work with received validations ...
```

Example 6-12: A forEach performing validation on a list of timesheet expense entries, where all validations must be completed before proceeding.

In this example, the forEach activity iterates the child scope for the number of expense entries returned by the retrieveExpenseList service. As forEach iterates its child scope (finalCounterValue - startCounterValue) + 1 times, the startCounterValue is set to 1 and the finalCounterValue is set to the count() of expense entries. No completionCondition is set since it is desired that all branches of the parallel complete before proceeding.

6.1.14. Defining assign activities

In the Timesheet Submission process, data returned from interactions with partners will need to be copied to variables so that the process can manipulate them, using Assign. In this case, timesheet submission messages (reflecting submission failures) are copied to notification messages (to be sent to their respective recipients).

```
<assign>
   <copy>
      <from variable="TimesheetSubmissionFailedMessage" />
      <to variable="EmployeeNotificationMessage" />
   </copy>
   <copy>
      <from variable="TimesheetSubmissionFailedMessage" />
      <to variable="ManagerNotificationMessage" />
   </copy>
</assign>
```

Example 6-14: Within the assign construct, the contents of the TimesheetSubmissionFailedMessage variable are copied to two different message variables.

Note that <copy> can process a variety of data transfer functions (for example, only a part of a message can be extracted and copied into a variable). Also, from and to elements can contain optional part and query attributes, allowing for specific parts or values of the variable to be referenced.

New data can also be constructed and inserted using expressions in assign – these expressions can operate on variables, properties and constants to produce a new value.

In the next case, the Timesheet Submission process needs to determine the pay rate (which varies per scale based upon employee grade), where grade is a part of the EmployeeData variable. Here, the grade is referenced using an XPath 1.0 query:

```
<assign>
   <copy>
      <from variable="EmployeeData" part="grade">
         <query>employeeGradeInformation</query>
      </from>
      <to variable="EmployeeRateInput" />
   </copy>
</assign>
```

Example 6-15: Within the assign construct, the grade part of the EmployeeData variable is copied to an input message used for rate calculation. .

6.1.15. Interacting with Partners

Through the previous sections, one has dealt with how to structure the business logic for the Timesheet Submission process. The process will need to interact with partners to perform various services external to the process logic. The following sections cover how this is done in the Timesheet Submission process.

6.1.16. Defining an invoke activity

The Timesheet Submission process needs to invoke external services, such as for validating hours submitted for a weekly timesheet, via the Invoke activity.

```
<invoke name="ValidateWeeklyHours"
    partnerLink="Employee"
    operation="GetWeeklyHoursLimit"
    inputVariable="EmployeeHoursRequest"
    outputVariable="EmployeeHoursResponse" />
```
Example 6-16: The invoke activity, identifying the target partner service details, for accessing the limit on weekly hours for an employee from the employee data service.

6.1.17. Defining a receive activity

Above, the Timesheet Submission process defined a relationship with an consumer service, through the "client" partnerLink, and as such expects to receive requests from that consumer to begin processing. As such, a Receive activity is specified to process those requests.

```
<receive name="receiveInput"
    partnerLink="client"
    operation="Submit"
    variable="ClientSubmission"
    createInstance="yes" />
```
Example 6-17: The receive activity used in the Timesheet Submission Process definition to indicate the client partner service responsible for launching the process with the submission of a timesheet document.

6.1.18. Defining a reply activity

In the Timesheet Submission process, data that is processed will be returned to the consumer in a synchronous exchange, using the Reply activity.

```
<reply partnerLink="client"
    operation="SubmitTimesheet"
    variable="TimesheetSubmissionResponse" />
```
Example 6-16: A companion reply to the receive in Example 7-15 above.

6.1.19. Elaborating and Refining Process Logic

Through the previous sections, one has covered basic structuring of the Timesheet Submission business logic and interacting with partners to perform services external to that logic. The behavior of the process will require addressing cases where messages are processed selectively, multiple messages are processed in parallel (say, by the same process instance), external services are performed in parallel (either by branching or through a loop condition), process execution is

delayed, process execution is immediately ended, and/or exception conditions are handled. These are covered in the following sections.

6.1.20. Defining a pick activity

The Timesheet Submission process is required to loop through the input of individual timesheet entries, used to produce a complete timesheet record, whose completion is controlled by a timeout event, using the Pick activity.

In this case, the Timesheet Submission process uses a pick activity to loop through the input of entries for the timesheet record, along with a timeout for timesheet completion as enabled by the `<onAlarm>` event.

```
<pick>
   <onMessage partnerLink="Timesheet"
      operation="inputTimesheetEntry"
      variable="timesheetEntry">
   </onMessage>
   <onMessage partnerLink="Timesheet"
      operation="timesheetComplete"
      variable="completionDetail">
   </onMessage>
   <onAlarm>
      <for>'P0DT12H'</for>
   </onAlarm>
</pick>
```
Example 6-18: A loop and timeout pick activity for the Timesheet Submission process. The loop completion is set to timeout in 12 hours in the date time expression specified in the onAlarm.

6.1.21. Defining a flow activity

In the Timesheet Submission process, an invoice can only be submitted once both the timesheet entries and the timesheet expenses have been approved, each in parallel by different signature authorities. When the Timesheet Submission process has completed pre-processing timesheet entries, it submits these for approval to its appropriate authority (e.g. a Project Manager), and when it has completed pre-processing expenses, it submits these for approval to its appropriate authority (e.g. a direct-reports Manager). These approval submissions are asynchronous. Once these submissions are processed, both authorities essentially inform the Timesheet Submission process that they have approved their respective line items, and the process then initiates the invoice submission. This parallel processing is performed using the Flow activity.

The Timesheet Submission process starts two Invoke activities concurrently when the flow starts – the TimesheetEntriesSignatory and the TimesheetExpensesSignatory – both of which are one-way operations. The completion of the flow occurs after both the TimesheetEntriesSignatory and the TimesheetExpensesSignator have been invoked.

```
<flow>
   <documentation>
      process the timesheet entries and expenses sign-offs concurrently
   </documentation>
   <invoke partnerLink="TimesheetEntriesSignatory" ... />
```

```
      <invoke partnerLink="TimesheetExpensesSignatory" ... />
</flow>
```
Example 6-19: A flow processing the timesheet entries and expenses sign-offs.

Now the Timesheet Submission process needs to wait for a message from both partners without knowing which one arrives first. As such another flow is defined for the process to handle the asynchronous replies/notifications, represented as receives. The synchronization dependencies between these concurrent receives are expressed by links which connect them.

```
<flow>
   <links>
      <link name="timesheetEntriesApproval" />
      <link name="timesheetExpensesApproval" />
   </links>

   <receive name="receiveEntriesApproval" ...>
      <sources>
         <source linkName="timesheetEntriesApproval" />
      </sources>
   </receive>
   <receive name="receiveExpensesApproval" ...>
      <sources>
         <source linkName="timesheetExpensesApproval" />
      </sources>
   </receive>
   <scope name="invoiceSubmissionScope">
      <targets>
         <joinCondition>
            $timesheetEntriesApproval and $timesheetExpensesApproval
         </joinCondition>
         <target linkName="timesheetEntriesApproval" />
         <target linkName="timesheetExpensesApproval" />
      </targets>
      <compensationHandler>
         <invoke name="withdrawInvoiceSubmission" ... />
      </compensationHandler>
      <invoke name="submitInvoice" ... />
   </scope>
   ...
</flow>
```
Example 6-20: A flow using multiple receives to process timesheet entries and expenses approval, and to then perform invoice submission.

Managing the waiting for a message from both partners is achieved by using two receive activities. Both messages from the timesheet entries signatory and the timesheet expenses signatory are processed by the same business process instance.

```
<flow>
   <links>
      <link name="timesheetEntriesApproval" />
      <link name="timesheetExpensesApproval" />
   </links>

   <receive name="receiveEntriesApproval" ...>
      <sources>
         <source linkName="timesheetEntriesApproval" />
```

```
            </sources>
            <correlations>
                <correlation set="tradeID" initiate="no" />
            </correlations>
        </receive>
        <receive name="receiveExpensesApproval" createInstance="yes" ...>
            <sources>
                <source linkName="timesheetExpensesApproval" />
            </sources>
            <correlations>
                <correlation set="timesheetID" initiate="no />
            </correlations>
        </receive>
        <scope name="invoiceSubmissionScope">
            <targets>
                <joinCondition>
                    $timesheetEntriesApproval and $timesheetExpensesApproval
                </joinCondition>
                <target linkName="timesheetEntriesApproval" />
                <target linkName="timesheetExpensesApproval" />
            </targets>
            <compensationHandler>
                <invoke name="withdrawInvoiceSubmission" ... />
            </compensationHandler>
            <invoke name="submitInvoice" ... />
        </scope>
        ...
</flow>
```

Example 6-21: The flow from 6-19 adding correlation to ensure the messages from both receives are processed by the same process instance.

6.1.22.   Defining a wait activity

The Timesheet Submission process, in some situations, needs to suspend processing until a certain point in time, using the Wait activity to specify an intentional delay for a certain period of time (duration) or until a certain deadline is reached.

In this case, a sub-process is invoked (one-way) which handles reconciling submitted invoices and expense reports in the accounting records. This sub-process has a branch which posts all of these entries at the end of the month:

```
<invoke name="retrieveMonthlyAccountingPostDateTime"
    partnerLink="AccountingService"
    operation="getMonthlyPostDate"
    inputVariable="monthAndYear"
    outputVariable="monthlyPostDateTime" />

<wait>
    <documentation>
        The monthlyPostDateTime is a deadline expression,
        e.g. '2006-11-30T22:00+01:00'
    </documentation>
    <until>$monthlyPostDateTime</until>
</wait>

<invoke partnerLink="AccountingService"
```

```
operation="postAccountingEntries"
inputVariable="accountingEntries" />
```
Example 6-22: A wait (using the response from the above invoke to populate its deadline criterion in until) defined for a sub-process of the Timesheet Submission process for posting accounting data at the end of a particular month.

6.1.23. Defining faultHandlers - catch, and catchAll

The Timesheet Submission process will need to address error conditions at the process level, using the FaultHandlers.

In this case, faultHandlers are used to catch specific faults and provide exception handling for the error conditions, along with a catchall element to house default error handling activities.

```
<faultHandlers>
   <catch faultName="SomethingBadHappened"
      faultVariable="TimesheetFault" faultElement="...">
      ...
   </catch>
   <catchAll>...</catchAll>
</faultHandlers>
```
Example 6-23: The faultHandlers section, hosting catch and catchAll child constructs, specified for timesheet service faults.

6.1.24. Defining a validate activity

The Timesheet Submission process will need to address the validation of data, in this case with timesheet entries. In order to make certain that variable contents are valid according to the variable declaration, the Validate activity is used.

```
<scope>
   <faultHandlers>
      <catch faultName="bpel:invalidVariables">
         <reply name="invalidTimesheetSubmission" ... />
      </catch>
   </faultHandlers>

   <sequence>
      <receive name="receiveTimesheetEntry" variable="timesheetEntry" ... />
      <validate name="validateTimesheetEntry"
         variables="timesheetEntry" />
      ...
      <reply name="acknowledgeReceipt" ... />
      ...
   </sequence>
</scope>
```
Example 6-24: A validate defined for validating timesheet entries.

6.1.25. Defining a compensationHandler

The Timesheet Submission process will need to address the case where under certain error conditions; the submission of an invoice to a client needs to be undone. In this case a compensation steps need to be perform to undo this action.

```xml
<scope name="mainScope">
   <faultHandlers>
      <catchAll>
         <compensateScope target="invoiceSubmissionScope" />
      </catchAll>
   </faultHandlers>
   <sequence>
      ...
      <scope name="invoiceSubmissionScope">
         ...
         <compensationHandler>
            <invoke name="withdrawInvoiceSubmission" ... />
         </compensationHandler>
         <invoke name="submitInvoice" ... />
      </scope>
      ...
      <!-- do additional work -->
      <!-- a fault is thrown here;
           results of invoiceSubmissionScope must be undone -->
   </sequence>
</scope>
```
Example 6-25: A compensationHandler to undo invoicing in the case of a downstream fault.

7. What's new in WS-BPEL 2.0

As a result of the OASIS Technical Committee's issues process, the original BPEL4WS 1.1 specification has received several updates. The following list summarizes the major changes that have been incorporated in the WS-BPEL 2.0 specification.

Data Access

- Variables can now be declared using XML schema complex types
- XPath expressions are simplified by using the '$' notation for variable access, for example, `$myMsgVar.part1/po:poLine[@lineNo=3]`
- Access to WSDL messages has been simplified by mapping directly mapping WSDL message parts to XML schema element/type variables
- Several clarifications have been added to the description of the `<assign>` activity's `<copy>` semantics
- The `keepSrcElementName` option has been added to `<copy>` in order to support XSD substitution groups or choices
- The `ignoreMissingFromData` has been added to automatically some of `<copy>` operation, when the from data is missing.
- An extension operation has been added to the `<assign>` activity
- A standardized XSLT 1.0 function has been added to XPath expressions
- The ability to validate XML data has been added, both as an option of the `<assign>` activity and as a new `<validate>` activity
- Variable initialization as part the of variable declaration has been added

Scope Model

- New scope snapshot semantics have been defined
- Fault handling during compensation has been clarified
- The interaction between scope isolation and control links have been clarified
- Enrichment of fault catching model
- A `<rethrow>` activity has been added to fault handlers
- The `<terminationHandler>` has been added to scopes
- The `exitOnStandardFault` option has been added to processes and scopes

Message Operations

- The `join` option has been added to correlation sets in order to allow multiple participants to rendezvous at the same process with a deterministic order
- Partner link can now be declared local to a scope
- The `initializePartnerRole` option has been added to specify whether an endpoint reference must be bound to a partner link during deployment
- The `messageExchange` construct has been added to pair up concurrent `<receive>` and `<reply>` activities

New Activities

- Added serial and parallel `<forEach>` with optional completion condition
- Added `<repeatUntil>`
- Added new extension activity
- Changed `<switch>` to `<if>`-`<elseif>`-`<else>`
- Changed `<terminate>` to `<exit>`
- Differentiate different cases of `<compensate>` by renaming them to `<compensate>` and `<compensateScope>`

Miscellaneous Changes

- Added `repeatEvery` alarm feature to event handlers
- Clarified resources resolution (e.g. variable, partner link) for event handlers
- Added formal `<documentation>` support
- Added extension namespace declarations in order to specify what extension must be understood
- Add `<import>` support to import WSDL and XSD formally

Abstract Processes

- Clarified Abstract Process usage patterns
- Introduced Abstract Profiles to address different needs in Abstract Processes, and two profiles "Observable Behavior" and "Process Template" listed in the specification

8. Summary

8.1. Benefits of WS-BPEL

WS-BPEL provides the orchestration service layer for Service Oriented Architecture (SOA) by which the following benefits can be realized:

- Industry standard language for expressing business processes: As With its rich and comprehensive semantics, WS-BPEL represents a standard which has undergone rigorous development by the industry toward addressing complex requirements, resulting in a comprehensive orchestration solution.
- Leverage a common skill set and language: Standards enable lower total cost of ownership through knowledge portability - instead of using complex proprietary technologies, WS-BPEL enables best practices, patterns, experience and training to be leveraged from a variety of vendors, as well as access to resources knowledgeable in the WS-BPEL model and technology.
- Abstracts business logic and responsibility: Application and business services can be designed to be process-agnostic and reusable. The business process assumes the management and coordination of state, freeing constituent services from a number of design constraints. Additionally, the business process logic is centralized in one location, as opposed to being distributed across and embedded within multiple services.
- Designed to fit naturally into the Web services stack: In WS-BPEL a business processes interact with services through Web services invocations, and are themselves externalized as Web services. This recursive composition enables a BPEL process to leverage the interoperability provided by the lower levels of the Web Services stack, such as WSDL, SOAP, and WS-Addressing.
- Expressed entirely in XML: As WS-BPEL business processes are expressed in XML, they are human-readable and can be used by any XML processing facilities, enabling them to be produced and consumed within the XML stack.
- Uses and extends WSDL 1.1: WS-BPEL uses and extends WSDL to both provide and consume Web services in an abstract way, using WSDL to define service interfaces.
- Uses XML Schema 1.0 type definitions for the data model
- Portable across platform and vendor: WS-BPEL provides for standards-based platforms which reduce vendor lock-in and facilitate migration from one vendor platform to another. WS-BPEL processes will run on any WS-BPEL-compliant engine.

Interoperable between interacting processes: Many deployments will have multiple orchestration platforms due to embedding in tools and applications, organizational purchases, etc. WS-BPEL provides a common standard which provides for interoperability between the different platforms and the processes that execute on them.

Appendices

A. References

[BPEL4WS 1.1]   BEA, IBM, Microsoft, SAP and Siebel, "Business Process Execution
                    Language for Web Services Version 1.1", S. Thatte, et al., May 2003.
                    http://www.oasis-
                    open.org/committees/download.php/2046/BPEL%20V1-
                    1%20May%205%202003%20Final.pdf

[Infoset]   W3C Recommendation, "XML Information Set (Second Edition)", J.
                    Cowan, R. Tobin, February 4, 2004.
                    http://www.w3.org/TR/2004/REC-xml-infoset-20040204

[RFC 2119]   IETF, "Key words for use in RFCs to Indicate Requirement Levels",
                    RFC 2119, S. Bradner, March 1997.
                    http://www.ietf.org/rfc/rfc2119.txt

[RFC 2396]   IETF, "Uniform Resource Identifiers (URI): Generic Syntax", RFC
                    2396, T. Berners-Lee, R. Fielding, L. Masinter, August 1998.
                    http://www.ietf.org/rfc/rfc2396.txt

[Sagas]   Garcia-Molina H. and Kenneth Salem, "SAGAS", Proceedings of the
                    ACM SIGMOD International Conference on Management of Data,
                    pages 249--259, May 1987.

[SOAP 1.1]   W3C Note, "Simple Object Access Protocol (SOAP) 1.1", D. Box, D.
                    Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen,
                    S. Thatte, D. Winer, May 8, 2000.
                    http://www.w3.org/TR/2000/NOTE-SOAP-20000508

[Trends]   Traiger I. L., "Trends in System Aspects of Database Management",
                    Proceeding of the 2nd International Conference on Database (ICOD-
                    2), pages 1-21, Wiley & Sons, 1983.

[UDDI]   OASIS, "UDDI Version 3.0.2", L. Clement, A. Hately, C. V. Riegen,
                    T. Rogers, October 19, 2004. http://uddi.org/pubs/uddi-v3.0.2-
                    20041019.htm

[WS-Addressing]   W3C, "Web Services Addressing 1.0 - Core", Martin Gudgin, Marc
                    Hadley, Tony Rogers, May 9, 2006.
                    http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/

[WS-BPEL 2.0]   Web service business Process Execution Language Version 2.0 Specification, OASIS Standard, 11 April 2007, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[WSDL 1.1]   W3C Note, "Web Services Definition Language (WSDL) 1.1", E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, March 15, 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[WSFL]   IBM, "Web Service Flow Language (WSFL 1.0)", F. Leymann, May 2001. http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[WS-I Basic Profile 1.1 Errata] Web Services Interoperability Organization, "Basic Profile Version 1.1 Errata", Revision 1.8, A. Karmarkar , October 25, 2005. http://www.ws-i.org/Profiles/BasicProfile-1.1-errata-2005-10-25.html

[WS-I Basic Profile]   Web Services Interoperability Organization, "Basic Profile Version 1.1", K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, P. Yendluri, April 16, 2004. http://www.ws-i.org/Profiles/BasicProfile-1.1.html

[XLANG]   Microsoft, "XLANG Web Services for Business Process Design", S. Thatte, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[XML Namespace]   W3C Recommendation , "Namespaces in XML", T. Bray, D. Hollander, A. Layman, January 14, 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114

[XML Schema Part 1]   W3C Recommendation, "XML Schema Part 1: Structures Second Edition", H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, October 28, 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[XML Schema Part 2]   W3C Recommendation, "XML Schema Part 2: Datatypes Second Edition", P. V. Biron, A. Malhotra, October 28, 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[XMLSpec]   W3C Recommendation, "Extensible Markup Language (XML) 1.0 (Third Edition)", T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, February 4, 2004. http://www.w3.org/TR/2004/REC-xml-20040204

[XPATH 1.0]   W3C Recommendation, "XML Path Language (XPath) Version 1.0", J. Clark, S. DeRose, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116

[XSLT 1.0]   W3C Recommendation, "XSL Transformations (XSLT) Version 1.0", J. Clark, November 16, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116

B. Acknowledgements

The authors of this Primer would like to extend their thanks and appreciation to Thomas Erl for contributing sections of his book "Service-Oriented Architecture: Concepts, Technology, and Design" to this effort.