Web Services Business Process Execution Language Version 2.0

Committee Draft, 01 September 2005

Document identifier:

wsbpel-specification-draft-01

Location:

http://www.oasis-open.org/apps/org/workgroup/wsbpel/

Editors:

Assaf Arkin <arkin@intalio.com>
Sid Askary <saskary@nuperus.com>
Ben Bloch <ben_b54@hotmail.com>
Francisco Curbera, IBM <curbera@us.ibm.com>
Yaron Goland, BEA <ygoland@bea.com>
Neelakantan Kartha, Sterling Commerce <N_Kartha@stercomm.com>
Canyang Kevin Liu, SAP <kevin.liu@sap.com>
Satish Thatte, Microsoft <satisht@microsoft.com>
Prasad Yendluri, webMethods <pyendluri@webmethods.com>
Alex Yiu, Oracle <alex.yiu@oracle.com>

Contributors:

{FirstName} {Last Name}, {Organization}

Editor's Notes – KevinL – this section should be consolidated with Appendix H

Abstract:

This document defines a notation for specifying business process behavior based on Web Services. This notation is called Web Services Business Process Execution Language (abbreviated to WS-BPEL in the rest of this document). Processes in WS-BPEL export and import functionality by using Web Service interfaces exclusively.

Business processes can be described in two ways. Executable business processes model actual behavior of a participant in a business interaction. Business protocols, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The process descriptions for business protocols are called abstract processes. WS-BPEL is meant to be used to model the behavior of both executable and abstract processes.

WS-BPEL provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions. WS-BPEL defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces.

Status:

This is a draft version of the WS-BPEL TC specification, updated from the orignal BPEL4WS V1.1 specification dated May 5, 2003 that was submitted to the WS BPEL TC. See: http://www.oasis-open.org/apps/org/workgroup/wsbpel/download.php/2046/BPEL%20V1-1%20May%205%202003%20Final.pdf

If you are on the <wsbpel@lists.oasis-open.org> list for committee members, send comments there. If you are not on that list, subscribe to the <wsbpel-comment@lists.oasis-open.org> list and send comments there. To subscribe, send an email message to <mailto:wsbpel-comment-request@lists.oasis-open.org> with the word "subscribe"as the body of the message.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the WS-BPEL TC web page http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

---

Table of Contents

Appendixes

---

1. Introduction

The goal of the Web Services effort is to achieve universal interoperability between applications by using Web standards. Web Services use a loosely coupled integration model to allow flexible integration of heterogeneous systems in a variety of domains including business-to-consumer, business-to-business and enterprise application integration. The following basic specifications originally defined the Web Services space: SOAP, Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). SOAP defines an XML messaging protocol for basic service interoperability. WSDL introduces a common grammar for describing services. UDDI provides the infrastructure required to publish and discover services in a systematic way. Together, these specifications allow applications to find each other and interact following a loosely coupled, platformindependent model.

Systems integration requires more than the ability to conduct simple interactions by using standard protocols. The full potential of Web Services as an integration platform will be achieved only when applications and business processes are able to integrate their complex interactions by using a standard process integration model. The interaction model that is directly supported by WSDL is essentially a stateless model of synchronous or uncorrelated asynchronous interactions. Models for business interactions typically assume sequences of peer-to-peer message exchanges, both synchronous and asynchronous, within stateful, long-running interactions involving two or more parties. To define such business interactions, a formal description of the message exchange protocols used by business processes in their interactions is needed. The definition of such business protocols involves precisely specifying the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal implementation. There are two good reasons to separate the public aspects of business process behavior from internal or private aspects. One is that businesses obviously do not want to reveal all their internal decision making and data management to their business partners. The other is that, even where this is not the case, separating public from private process provides the freedom to change private aspects of the process implementation without affecting the public business protocol.

Business protocols must clearly be described in a platform-independent manner and must capture all behavioral aspects that have cross-enterprise business significance. Each participant can then understand and plan for conformance to the business protocol without engaging in the process of human agreement that adds so much to the difficulty of establishing cross-enterprise automated business processes today.

What are the concepts required to describe business protocols? And what is the relationship of these concepts to those required to describe executable processes? To answer these questions, consider the following::

- Business protocols invariably include data-dependent behavior. For example, a supply-chain protocol depends on data such as the number of line items in an order, the total value of an order, or a deliver-by deadline. Defining business intent in these cases requires the use of conditional and time-out constructs.
- The ability to specify exceptional conditions and their consequences, including recovery sequences, is at least as important for business protocols as the ability to define the behavior in the "all goes well" case.
- Long-running interactions include multiple, often nested units of work, each with its own data requirements. Business protocols frequently require cross-partner coordination of the outcome (success or failure) of units of work at various levels of granularity.

If we wish to provide precise predictable descriptions of service behavior for crossenterprise business protocols, we need a rich process description notation with many features reminiscent of an executable language. The key distinction between public message exchange protocols and executable internal processes is that internal processes handle data in rich private ways that need not be described in public protocols.

In thinking about the data handling aspects of business protocols it is instructive to consider the analogy with network communication protocols. Network protocols define the shape and content of the protocol envelopes that flow on the wire, and the protocol behavior they describe is driven solely by the data in these envelopes. In other words, there is a clear physical separation between protocol-relevant data and "payload" data. The separation is far less clear cut in business protocols because the protocol-relevant data tends to be embedded in other application data.

WS-BPEL uses a notion of message properties , which are a type of variable property, to identify protocol-relevant data embedded in messages. Properties can be viewed as "transparent" data relevant to public aspects as opposed to the "opaque" data that internal/private functions use. Transparent data affects the public business protocol in a direct way, whereas opaque data is significant primarily to back-end systems and affects the business protocol only by creating nondeterminism because the way it affects decisions is opaque. We take it as a principle that any data that is used to affect the behavior of a business protocol must be transparent and hence viewed as a property.

The implicit effect of opaque data manifests itself through nondeterminism in the behavior of services involved in business protocols. Consider the example of a purchasing protocol. The seller has a service that receives a purchase order and responds with either acceptance or rejection based on a number of criteria, including availability of the goods and the credit of the buyer. Obviously, the decision processes are opaque, but the fact of the decision must be reflected as behavior alternatives in the external business protocol. In other words, the protocol requires something like an if activity in the behavior of the seller's service but the selection of the branch taken is nondeterministic. Such nondeterminism can be modeled by allowing the assignment of a nondeterministic or opaque value to a message property, typically from an enumerated set of possibilities. The property can then be used in defining conditional behavior that captures behavioral alternatives without revealing actual decision processes. WS-BPEL explicitly allows the use of nondeterministic data values to make it possible to capture the essence of public behavior while hiding private aspects.

The basic concepts of WS-BPEL can be applied in one of two ways. A WS-BPEL process can define a business protocol role, using the notion of abstract process. For example, in a supply-chain protocol, the buyer and the seller are two distinct roles, each with its own abstract process. Their relationship is typically modeled as a partner link. Abstract processes use all the concepts of WS-BPEL but approach data handling in a way that reflects the level of abstraction required to describe public aspects of the business protocol. Specifically, abstract processes handle only protocol-relevant data. WS-BPEL provides a way to identify protocol-relevant data as message properties. In addition, abstract processes use nondeterministic data values to hide private aspects of behavior.

It is also possible to use WS-BPEL to define an executable business process. The logic and state of the process determine the nature and sequence of the Web Service interactions conducted at each business partner, and thus the interaction protocols. While a WS-BPEL process definition is not required to be complete from a private

implementation point of view, the language effectively defines a portable execution format for business processes that rely exclusively on Web Service resources and XML data. Moreover, such processes execute and interact with their partners in a consistent way regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Even where private implementation aspects use platform-dependent functionality, which is likely in many if not most realistic cases, the continuity of the basic conceptual model between abstract and executable processes in WS-BPEL makes it possible to export and import the public aspects embodied in business protocols as process or role templates while maintaining the intent and structure of the protocols. This is arguably the most attractive prospect for the use of WS-BPEL from the viewpoint of unlocking the potential of Web Services because it allows the development of tools and other technologies that greatly increase the level of automation and thereby lower the cost in establishing cross-enterprise automated business processes.

In summary, we believe that the two usage patterns of business protocol description and executable business process description require a common core of process description concepts. In this specification we clearly separate the core concepts from the extensions required specifically for the two usage patterns. The WS-BPEL specification is focused on defining the common core, and adds only the essential extensions required for each usage pattern.

WS-BPEL defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. The interaction with each partner occurs through Web Service interfaces, and the structure of the relationship at the interface level is encapsulated in what we call a partner link. The WS-BPEL process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination. WS-BPEL also introduces systematic mechanisms for dealing with business exceptions and processing faults. Finally, WS-BPEL introduces a mechanism to define how individual or composite activities within a process are to be compensated in cases where exceptions occur or a partner requests reversal.

WS-BPEL is layered on top of several XML specifications: WSDL 1.1, XML Schema 1.0, and XPath1.0. WSDL messages and XML Schema type definitions provide the data model used by WS-BPEL processes. XPath provides support for data manipulation. All external resources and partners are represented as WSDL services. WS-BPEL provides extensibility to accommodate future versions of these standards, specifically the XPath and related standards used in XML computation.

2. Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119].

Namespace URIs of the general form "some-URI" represent some application-dependent or context-dependent URI as defined in [RFC 2396].

This specification uses an informal syntax to describe the XML grammar of the XML fragments that follow:

- The syntax appears as an XML instance, but the values indicate the data types instead of values.
- Grammar in bold has not been introduced earlier in the document, or is of particular interest in an example.
- <-- description --> is a placeholder for elements from some "other" namespace (like ##other in XSD).
- Characters are appended to elements, attributes, and as follows: "?" (0 or 1), "*" (0 or more), "+" (1 or more). The characters "[" and "]" are used to indicate that contained items are to be treated as a group with respect to the "?", "*", or "+" characters.
- Elements and attributes separated by "|" and grouped by "(" and ")" are meant to be syntactic alternatives.
- The XML namespace prefixes (defined below) are used to indicate the namespace of the element being defined.
- Examples starting with <?xml contain enough information to conform to this specification; other examples are fragments and require additional information to be specified in order to conform.
- XSD schemas and WSDL definitions are provided as a formal definition of grammars [XML Schema Part 1] and [WSDL 1.1].

This specification uses a number of namespace prefixes throughout; their associated URIs are listed below. Note that the choice of any namespace prefix is arbitrary and not semantically significant.

- **xsi** - "http://www.w3.org/2001/XMLSchema-instance"
- **xsd** - "http://www.w3.org/2001/XMLSchema"
- **wsdl** - "http://schemas.xmlsoap.org/wsdl/"
- **plnk** – http://schemas.xmlsoap.org/ws/2004/03/partner-link/”
- **bpws** – “http://schemas.xmlsoap.org/ws/2004/03/business-process/”

3. Relationship with Other Specifications

WS-BPEL depends on the following XML-based specifications: WSDL 1.1, XML Schema 1.0 and XPath 1.0.

Among these, WSDL has the most influence on the WS-BPEL language. The WS-BPEL process model is layered on top of the service model defined by WSDL 1.1. At the core

of the WS-BPEL process model is the notion of peer-to-peer interaction between services described in WSDL; both the process and its partners are modeled as WSDL services. A business process defines how to coordinate the interactions between a process instance and its partners. In this sense, a WS-BPEL process definition provides and/or uses one or more WSDL services, and provides the description of the behavior and interactions of a process instance relative to its partners and resources through Web Service interfaces. That is, WS-BPEL defines the message exchange protocols followed by the business process of a specific role in the interaction.

The definition of a WS-BPEL business process also follows the WSDL model of separation between the abstract message contents used by the business process and deployment information (messages and portType versus binding and address information). In particular, a WS-BPEL process represents all partners and interactions with these partners in terms of abstract WSDL interfaces (portTypes and operations); no references are made to the actual services used by a process instance.

However, the abstract part of WSDL does not define the constraints imposed on the communication patterns supported by the concrete bindings. Therefore a WS-BPEL process may define behavior relative to a partner service that is not supported by all possible bindings, and it may happen that some bindings are invalid for a WS-BPEL process definition.

While WS-BPEL attempts to provide as much compatibility with WSDL 1.1 as possible there are three areas where such compatibility has proven impossible. One area, discussed later in this document, is in fault naming. Another area is in support for overloaded operation names in WSDL portTypes. A BPEL processor MUST reject any WSDL portType definition that includes overloaded operation names. This restriction was deemed appropriate as overloaded operations are rare, they are actually banned in the WS-I Basic Profile and supporting them was felt to introduce more complexity than benefit. Finally a WS-BPEL processor MUST reject a WS-BPEL that refers to a portType that contain solicit-response or notification operations as defined in the WSDL 1.1 specification, this requirement MUST be statically enforced.

BPEL does not make any assumptions about the WSDL binding. Constraints, ambiguities, provided or missing capabilities of WSDL bindings are out of scope of this specification.

A WS-BPEL process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them. Note that the description of the deployment of a WS-BPEL process is out of scope for this specification.

Introduction of service reference container (“bpws:service-ref”) is meant to avoid inventing a private WS-BPEL mechanism for web service endpoint references and to provide pluggability of different versions of service referencing or endpoint addressing schemes being used within a BPEL program without having explicit dependency to a particular version of specification.

With respect to [WS-I Basic Profile] (Basic Profile 1.1) all BPEL implementations SHOULD be configurable such that they can participate in Basic Profile 1.1 compliant interactions. A BPEL implementation MAY allow the Basic Profile 1.1 configuration to be disabled, even for scenarios encompassed by the Basic Profile 1.1. At the time this specification was completed, the WSDL v2.0 work was ongoing and not ready for consideration for WS-BPEL v2.0. Future versions of WS-BPEL may provide support for WSDL v2.0.

4. This Section Has Been Deleted

5. Core Concepts and Usage Patterns

As noted in the introduction, we believe that the two usage patterns of business protocol description and executable business process description require a common core of process description concepts. In this specification we clearly separate the core concepts from the extensions required specifically for the two usage patterns. The WS-BPEL specification is focused on defining the common core, and adds only the essential extensions required for each usage pattern. These extensions are described in separate sections (*Extensions for Executable Processes* and *Extensions for Business Protocols*).

In a number of cases, the behavior of a process in a certain combination of circumstances is undefined, e.g., when a variable is used before being initialized. In the definition of the core concepts we simply note that the semantics in such cases is not defined.

WS-BPEL takes it as a general principle that compliant implementations MAY choose to perform static analysis to detect and reject process definitions that may have undefined semantics. Such analysis is necessarily pessimistic and therefore might in some cases prevent the use of processes that would not, in fact, create situations with undefined semantics, either in specific uses or in any use.

In the executable usage pattern for WS-BPEL, situations of undefined semantics always result in standard faults in the WS-BPEL namespace. These cases will be described as part of the **Extensions for Executable Processes** in the specification. However, it is important to note that WS-BPEL uses exactly one standard *internal* fault for its core control semantics, namely, bpws:joinFailure. This is the only standard fault that plays a role in the core concepts of WS-BPEL. Of course, the occurrence of faults specified in WSDL portType definitions during web service invocation is accounted for in the core concepts as well.

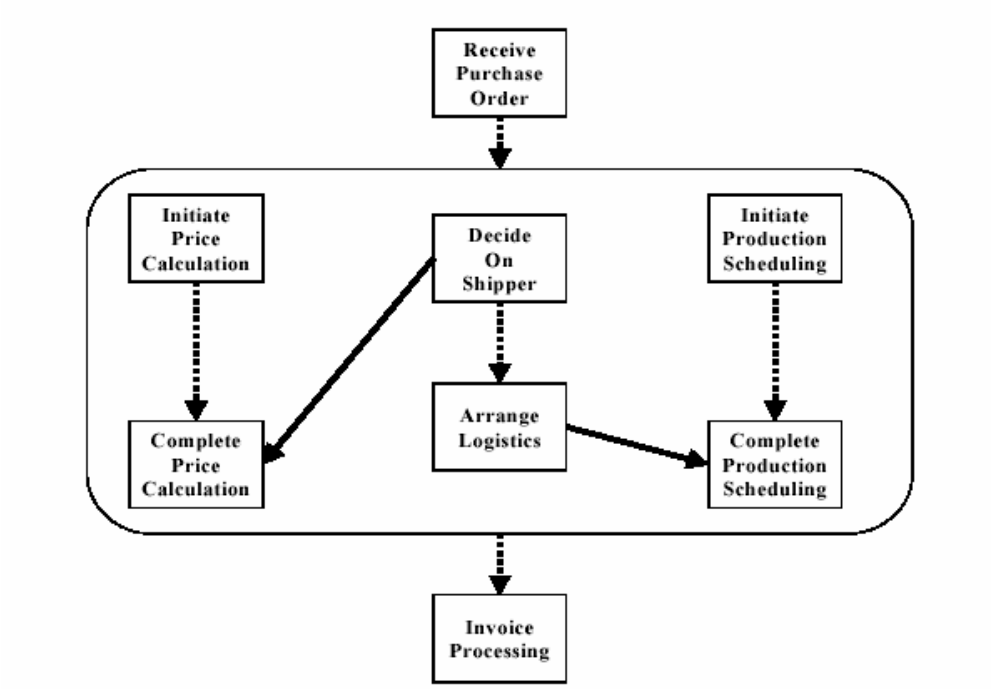
6. Defining a Business Process

6.1. Initial Example

Before describing the structure of business processes in detail, this section presents a simple example of a WS-BPEL process for handling a purchase order. The aim is to introduce the most basic structures and some of the fundamental concepts of the language.

The operation of the process is very simple, and is represented in the following figure. Dotted lines represent sequencing. Free grouping of sequences represents concurrent sequences. Solid arrows represent control links used for synchronization across concurrent activities. Note that this is not meant to be a definitive graphical notation for WS-BPEL processes. It is used here informally as an aid to understanding.

On receiving the purchase order from a customer, the process initiates three tasks concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer.



The WSDL portType offered by the service to its customers (purchaseOrderPT) is shown in the following WSDL document. Other WSDL definitions required by the business process are included in the same WSDL document for simplicity; in particular, the portTypes for the Web Services providing price calculation, shipping selection and scheduling, and production scheduling functions are also defined there. Observe that

there are no bindings or service elements in the WSDL document. A WS-BPEL process is defined "in the abstract" by referencing only the portTypes of the services involved in the process, and not their possible deployments. Defining business processes in this way allows the reuse of business process definitions over multiple deployments of compatible services.

The partner link types included at the bottom of the WSDL document represent the interaction between the purchase order service and each of the parties with which it interacts (see *Partner Link Types, Partner Links, and Endpoint References*). Partner link types can be used to represent dependencies between services, regardless of whether a WS-BPEL business process is defined for one or more of those services. Each partner link type defines up to two "role" names, and lists the portTypes that each role must support for the interaction to be carried out successfully. In this example, two partner link types, "purchasingLT" and "schedulingLT", list a single role because, in the corresponding service interactions, one of the parties provides all the invoked operations: The "purchasingLT" partner link represents the connection between the process and the requesting customer, where only the purchase order service needs to offers a service operation ("sendPurchaseOrder"); the "schedulingLT" partner link represents the interaction between the purchase order service and the scheduling service, in which only operations of the latter are invoked. The two other partner link types, "invoicingLT" and "shippingLT", define two roles because both the user of the invoice calculation and the user of the shipping service (the invoice or the shipping schedule) must provide callback operations to enable asynchronous notifications to be asynchronously sent ("invoiceCallbackPT" and "shippingCallbackPT" portTypes).

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6
<definitions targetNamespace="http://manufacturing.org/wsdl/purchase"
        xmlns:sns="http://manufacturing.org/xsd/purchase"
        xmlns:pos="http://manufacturing.org/wsdl/purchase"
        xmlns="http://schemas.xmlsoap.org/wsdl/">

        <types>
          <xsd:schema >
           <xsd:import
                namespace="http://manufacturing.org/xsd/purchase"
           schemalocation="http://manufacturing.org/xsd/purchase.xsd"/>
          </xsd:schema>
        </types>

        <message name="POMessage">
             <part name="customerInfo" type="sns:customerInfo"/>
             <part name="purchaseOrder" type="sns:purchaseOrder"/>
        </message>
        <message name="InvMessage">
             <part name="IVC" type="sns:Invoice"/>
        </message>
        <message name="orderFaultType">
              <part name="problemInfo" element=”sns:OrderFault"/>
        </message>
        <message name="shippingRequestMessage">
```

```
             <part name="customerInfo" element="sns:customerInfo"/>
       </message>
       <message name="shippingInfoMessage">
             <part name="shippingInfo" element="sns:shippingInfo"/>
       </message>
       <message name="scheduleMessage">
            <part name="schedule" element="sns:scheduleInfo"/>
       </message>

<!-- portTypes supported by the purchase order process -->
       <portType name="purchaseOrderPT">
            <operation name="sendPurchaseOrder">
              <input message="pos:POMessage"/>
              <output message="pos:InvMessage"/>
              <fault name="cannotCompleteOrder"
                 message="pos:orderFaultType"/>
            </operation>
       </portType>
       <portType name="invoiceCallbackPT">
             <operation name="sendInvoice">
       <input message="pos:InvMessage"/>
       </operation>
       </portType>
       <portType name="shippingCallbackPT">
            <operation name="sendSchedule">
              <input message="pos:scheduleMessage"/>
            </operation>
       </portType>

<!-- portType supported by the invoice services -->
       <portType name="computePricePT">
            <operation name="initiatePriceCalculation">
              <input message="pos:POMessage"/>
            </operation>
            <operation name="sendShippingPrice">
              <input message="pos:shippingInfoMessage"/>
            </operation>
       </portType>

<!-- portType supported by the shipping service -->
       <portType name="shippingPT">
            <operation name="requestShipping">
              <input message="pos:shippingRequestMessage"/>
              <output message="pos:shippingInfoMessage"/>
              <fault name="cannotCompleteOrder"
                    message="pos:orderFaultType"/>
            </operation>
       </portType>

<!-- portType supported by the production scheduling process -->
       <portType name="schedulingPT">
             <operation name="requestProductionScheduling">
              <input message="pos:POMessage"/>
             </operation>
            <operation name="sendShipingSchedule">
              <input message="pos:scheduleMessage"/>
            </operation>
```

```
       </portType>

<plnk:partnerLinkType name="purchasingLT">
     <plnk:role name="purchaseService"
                portType="pos:purchaseOrderPT"/>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="invoicingLT">
     <plnk:role name="invoiceService"
               portType="pos:computePricePT"/>

     <plnk:role name="invoiceRequester"
               portType="pos:invoiceCallbackPT"/>

</plnk:partnerLinkType>

<plnk:partnerLinkType name="shippingLT">
     <plnk:role name="shippingService"
               portType="pos:shippingPT"/>

     <plnk:role name="shippingRequester"
               portType="pos:shippingCallbackPT"/>

</plnk:partnerLinkType>

<plnk:partnerLinkType name="schedulingLT">
     <plnk:role name="schedulingService"
               portType="pos:schedulingPT"/>

</plnk:partnerLinkType>
</definitions>
```

The business process for the order service is defined next. There are four major sections in this process definition:

- The <variables> section defines the data variables used by the process, providing their definitions in terms of WSDL message types, XML Schema types (simple or complex), or XML Schema elements. Variables allow processes to maintain state data and process history based on messages exchanged.
- The <partnerLinks> section defines the different parties that interact with the business process in the course of processing the order. The four partnerLinks shown here correspond to the sender of the order (customer), as well as the providers of price (invoicingProvider), shipment (shippingProvider), and manufacturing scheduling services (schedulingProvider). Each partner link 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 for the relationship to succeed, that is, the portTypes that the purchase order process and the partner need to implement.
- The <faultHandlers> section contains fault handlers defining the activities that must be performed in response to faults resulting from the invocation of the assessment and approval services. In WS-BPEL, all faults, whether internal or resulting from a service invocation, are identified by a qualified name. In

particular, each WSDL fault is identified in WS-BPEL by a qualified name formed by the target namespace of the WSDL document in which the relevant portType and fault are defined, and the ncname of the fault. It is important to note, however, that because WSDL 1.1 does not require that fault names be unique within the namespace where the operation is defined, all faults sharing a common name and defined in the same namespace are indistinguishable. In spite of this serious WSDL limitation, WS-BPEL provides a uniform naming model for faults, in the expectation that future versions of WSDL will provide a better fault-naming model.

- The rest of the process definition contains the description of the normal behavior for handling a purchase request. The major elements of this description are explained in the section following the process definition.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<process name="purchaseOrderProcess"
        targetNamespace="http://acme.com/ws-bp/purchase"
        xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
        xmlns:lns="http://manufacturing.org/wsdl/purchase">

<documentation xml:lang="EN">A simple example of a WS-BPEL process for
handling a purchase order.</documentation>

<partnerLinks>
     <partnerLink name="purchasing"
        partnerLinkType="lns:purchasingLT"
        myRole="purchaseService"/>
     <partnerLink name="invoicing"
        partnerLinkType="lns:invoicingLT"
        myRole="invoiceRequester"
        partnerRole="invoiceService"/>
     <partnerLink name="shipping"
        partnerLinkType="lns:shippingLT"
        myRole="shippingRequester"
        partnerRole="shippingService"/>
     <partnerLink name="scheduling"
        partnerLinkType="lns:schedulingLT"
        partnerRole="schedulingService"/>
</partnerLinks>

<variables>
        <variable name="PO" messageType="lns:POMessage"/>
        <variable name="Invoice"  messageType="lns:InvMessage"/>
        <variable name="POFault"  messageType="lns:orderFaultType"/>
        <variable name="shippingRequest"
messageType="lns:shippingRequestMessage"/>
        <variable name="shippingInfo"
messageType="lns:shippingInfoMessage"/>
        <variable name="shippingSchedule"
messageType="lns:scheduleMessage"/>
</variables>
```

```
<faultHandlers>
     <catch faultName="lns:cannotCompleteOrder"
            faultVariable="POFault"
            faultMessageType="lns:orderFaultType">
       <reply partnerLink="purchasing"
          portType="lns:purchaseOrderPT"
          operation="sendPurchaseOrder"
          variable="POFault"
          faultName="cannotCompleteOrder"/>
     </catch>
</faultHandlers>

<sequence>
     <receive partnerLink="purchasing"
       portType="lns:purchaseOrderPT"
       operation="sendPurchaseOrder"
       variable="PO">
     </receive>

     <flow>
       <documentation>
         A parallel flow to handle shipping, invoicing and scheduling
       </documentation>
       <links>
           <link name="ship-to-invoice"/>
           <link name="ship-to-scheduling"/>
       </links>
       <sequence>
           <assign>
               <copy>
                   <from>$PO.customerInfo</from>
                   <to>$shippingRequest.customerInfo</to>
               </copy>
           </assign>
           <invoke partnerLink="shipping"
               portType="lns:shippingPT"
               operation="requestShipping"
               inputVariable="shippingRequest"
               outputVariable="shippingInfo">
               <sources>
                   <source linkName="ship-to-invoice"/>
               </sources>
           </invoke>
           <receive partnerLink="shipping"
               portType="lns:shippingCallbackPT"
               operation="sendSchedule"
               variable="shippingSchedule">
               <sources>
                   <source linkName="ship-to-scheduling"/>
               </sources>
           </receive>
       </sequence>
       <sequence>
           <invoke partnerLink="invoicing"
               portType="lns:computePricePT"
               operation="initiatePriceCalculation"
               inputVariable="PO">
```

```
            </invoke>
            <invoke partnerLink="invoicing"
                portType="lns:computePricePT"
                operation="sendShippingPrice"
                inputVariable="shippingInfo">
                <targets>
                    <target linkName="ship-to-invoice"/>
                </targets>
            </invoke>
            <receive partnerLink="invoicing"
                portType="lns:invoiceCallbackPT"
                operation="sendInvoice"
                variable="Invoice"/>
        </sequence>
        <sequence>
            <invoke partnerLink="scheduling"
                portType="lns:schedulingPT"
                operation="requestProductionScheduling"
                inputVariable="PO">
            </invoke>
            <invoke partnerLink="scheduling"
                portType="lns:schedulingPT"
                operation="sendShippingSchedule"
                inputVariable="shippingSchedule">
                <targets>
                    <target linkName="ship-to-scheduling"/>
                </targets>
            </invoke>
        </sequence>
     </flow>
     <reply partnerLink="purchasing"
        portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder"
        variable="Invoice"/>
    </sequence>
</process>
```

6.2. The Structure of a Business Process

This section provides a quick summary of the WS-BPEL syntax. It provides only a brief overview; the details of each language construct are described in the rest of this document.

The basic structure of the language is:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<process name="ncname" targetNamespace="uri"
        queryLanguage="anyURI"?
        expressionLanguage="anyURI"?
        suppressJoinFailure="yes|no"?
        abstractProcess="yes|no"?
        xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">

<import namespace="uri" location="uri" importType="uri"/>*
```

```
<partnerLinks>?
<!-- Note: At least one role must be specified. -->
    <partnerLink name="ncname" partnerLinkType="qname"
       myRole="ncname"? partnerRole="ncname"?>+
    </partnerLink>
</partnerLinks>

<variables>?
     <variable name="ncname" messageType="qname"?
       type="qname"? element="qname"?/>+
</variables>

<correlationSets>?
      <correlationSet name="ncname" properties="qname-list"/>+
</correlationSets>

<faultHandlers>?
<!-- Note: There must be at least one fault handler or default. -->
     <catch faultName="qname"? faultVariable="ncname"?
                               faultMessageType="qname"?
                               faultElement="qname"?>*
       activity
    </catch>
    <catchAll>?
       activity
    </catchAll>
</faultHandlers>

<eventHandlers>?
<!-- Note: There must be at least one onEvent or onAlarm handler. -->
       <onEvent partnerLink="ncname" portType="qname"?
         operation="ncname" messageType="qname" variable="ncname"
         messageExchange="ncname"? >*
             <correlationSets>?
                   <correlationSet name="ncname"
                      properties="qname-list"/>+
             </correlationSets>
             <correlations>?
               <correlation set="ncname"
                       initiate="yes|join|no"?/>+
             </correlations>
             activity
       </onEvent>
       <onAlarm>*
         ( <for expressionLanguage="anyURI"?>duration-expr</for> |
           <until expressionLanguage="anyURI"?>deadline-expr</until> )
           <repeatEvery expressionLanguage="anyURI"?>
             duration-expr
           </repeatEvery>?
           activity
       </onAlarm>
</eventHandlers>
       activity
</process>
```

The top-level attributes are as follows:

- queryLanguage. This attribute specifies the default XML query language used for selection of nodes in assignment, property definition, and other uses. The default value for this attribute is: "urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0", which represents the usage of XPath 1.0 within WS-BPEL 2.0. The URI of the corresponding XPath 1.0 specification is: http://www.w3.org/TR/1999/REC-xpath-19991116.
- expressionLanguage. This attribute specifies the expression language used in the process. The default value for this attribute is: "urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0", which represents the usage of XPath 1.0 within WS-BPEL 2.0. The URI of the corresponding XPath 1.0 specification is: http://www.w3.org/TR/1999/REC-xpath-19991116.
- suppressJoinFailure. This attribute determines whether the joinFailure fault will be suppressed for all activities in the process. The effect of the attribute at the process level can be overridden by an activity using a different value for the attribute. The default for this attribute is "no" at the process level. When this attribute is not specified for an activity, it inherits its value from its closest enclosing activity or from the process if no enclosing activity specifies this attribute.
- exitOnStandardFault. This attribute specifies if the process must exit if a WS-BPEL standard fault other than bpws:joinFailure is encountered[1]. If the value of this attribute is set to “yes”, then the process MUST exit immediately as if an <exit> activity has been reached, when a WS-BPEL standard fault other than bpws:joinFailure is enountered. If the value of this attribute is set to “no”, then the process can handle a standard fault using a fault handler. The default value for this attribute is “no”. When this attribute is not specified on a <scope> it inherits its value from its enclosing <scope> or <process>.

  If the value of exitOnStandardFault of a <scope> or <process> is set to “yes”, then a fault handler that explicitly targets the WS-BPEL standard faults MUST NOT be used in that scope. A process definition that violates this condition MUST be detected and rejected by static analysis.

- abstractProcess. This attribute specifies whether the process being defined is abstract (rather than executable). The default for this attribute is "no".

The value of the queryLanguage and expressionLanguage attributes on the <process> element are global defaults and can be overridden on specific activities like <assign> using the mechanisms defined later in this specification. In addition the queryLanguage attribute is also available for use in defining BPEL property aliases in WSDL. BPEL processors MUST:

[1] bpws:joinFailure generally represents a modeling error and hence is excluded from other standard faults in this case.

- statically determine which languages are referenced by queryLanguage or expressionLanguage attributes either in the BPEL process definition itself or in any BPEL property definitions in associated WSDLs and
- if any referenced language is unsupported by the BPEL processor then the processor MUST NOT process the submitted BPEL process definition.

Note that: <documentation> construct may be added to virtually all WS-BPEL constructs as the formal way to annotate processes definition with human documentation. Examples of <documentation> construct can be found in previous section. Detailed description of <documention> is provided in next section, as it is a part of “Language Extensibility”.

The token "**activity**" can be any of the following:

- <receive>
- <reply>
- <invoke>
- <assign>
- <throw>
- <exit>
- <wait>
- <empty>
- <sequence>
- <if>
- <while>
- <repeatUntil>
- <pick>
- <flow>
- <scope>
- <compensate>
- <rethrow>
- <validate>
- <extensionActivity>

The syntax of each of these elements, except <exit>, <compensate> and <rethrow>, is considered in the following paragraphs.

- Although <exit> is permitted as an interpretation of the token activity, it is only available in executable processes and as such is defined in the section on **Extensions for Executable Processes**.
- <compensate> activity can be used ONLY within a fault handler or a compensation handler (i.e. <catch>, <catchAll> and <compensationHandler> elements).
- <rethrow> activity can be used ONLY within a fault handler (i.e. <catch> and <catchAll> elements).

The <receive> construct allows the business process to do a blocking wait for a matching message to arrive. The portType attribute on the <receive> activity is optional. If the portType attribute is included for readability, the value of the portType attribute MUST match the portType value implied by the combination of the specified partnerLink and the role implicitly specified by the activity (See also partnerLink description in the next section). The optional messageExchange attribute is used to associate a <reply> activity with a <receive> activity. (For details of messageExchange, please see "Providing Web Service Operations" section).

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<receive partnerLink="ncname" portType="qname"? operation="ncname"
    variable="ncname"? createInstance="yes|no"?
     messageExchange="ncname"?
    standard-attributes>
    standard-elements
    <correlations>?
        <correlation set="ncname" initiate="yes|join|no"?/>+
    </correlations>
    <fromPart part="ncname" toVariable="ncname"/>*
</receive>
```

The <reply> construct allows the business process to send a message in reply to a message that was received through a <receive>. The combination of a <receive> and a <reply> forms a request-response operation on the WSDL portType for the process. The portType attribute on the <reply> activity is optional. If the portType attribute is included for readability, the value of the portType attribute MUST match the portType value implied by the combination of the specified partnerLink and the role implicitly specified by the activity (See also partnerLink description in the next section). The optional messageExchange attribute is used to associate a <reply> activity with an inbound message activity, such as, <receive>, <onMessage> and <onEvent>. (For details of messageExchange, please see "Providing Web Service Operations" section).

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<reply partnerLink="ncname" portType="qname"? operation="ncname"
    variable="ncname"? faultName="qname"?
    messageExchange="ncname"?
    standard-attributes>
    standard-elements
    <correlations>?
        <correlation set="ncname" initiate="yes|join|no"?/>+
    </correlations>
    <toPart part="ncname" fromVariable="ncname"/>*
</reply>
```

The <invoke> construct allows the business process to invoke a one-way or requestresponse operation on a portType offered by a partner. The portType attribute on the <invoke> activity is optional. If the portType attribute is included for readability, the

value of the portType attribute MUST match the portType value implied by the combination of the specified partnerLink and the role implicitly specified by the activity (See also partnerLink description in the next section).

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<invoke partnerLink="ncname" portType="qname"? operation="ncname"
    inputVariable="ncname"? outputVariable="ncname"?
    standard-attributes>
    standard-elements
    <correlations>?
        <correlation set="ncname" initiate="yes|join|no"?
            pattern="in|out|out-in"/>+
    </correlations>
    <catch faultName="qname"? faultVariable="ncname"?
                            faultMessageType="qname"?>*
        activity
    </catch>
    <catchAll>?
        activity
    </catchAll>
    <compensationHandler>?
        activity
    </compensationHandler>
    <toPart part="ncname" fromVariable="ncname"/>*
    <fromPart part="ncname" toVariable="ncname"/>*
</invoke>
```

The <assign> construct can be used to update the values of variables with new data. An <assign> construct can contain any number of elementary assignments, including <copy> assign elements or data update operations defined as extension under other namespaces. The syntax of the assignment activity is:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<assign validate="yes|no"? standard-attributes>
    standard-elements
    (<copy>
       from-spec
        to-spec
    </copy> |
    <extensibleAssign>
       ...assign-element-of-other-namespace...
    </extensibleAssign>) +
</assign>
```

The <validate> construct can be used to validate the values of variables against their associated XML and WSDL data definition. The construct has a variables attribute, which points to the variables being validated. The syntax of the validate activity is:

```
<validate variables="ncnames" />
```

The <throw> construct generates a fault from inside the business process.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<throw faultName="qname" faultVariable="ncname"? standard-attributes>
     standard-elements
</throw>
```

The <wait> construct allows you to wait for a given time period or until a certain time has passed. Exactly one of the expiration criteria must be specified.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<wait standard-attributes>
     standard-elements
        ( <for expressionLanguage="anyURI"?>duration-expr</for> |
          <until expressionLanguage="anyURI"?>deadline-expr</until> )
</wait>
```

The <empty> construct allows you to insert a "no-op" instruction into a business process. This is useful for synchronization of concurrent activities, for instance.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<empty standard-attributes>
       standard-elements
</empty>
```

New activities can be introduced for use in BPEL by placing them inside of the `extensionActivity` element. The contents of an `extensionActivity` element MUST be a single element that MUST make available BPEL's standard-attributes and standard-elements.

```
<extensionActivity>
    <???? standard-attributes>
       standard-elements
    </????>
</extensionActivity>
```

The <sequence> construct allows you to define a collection of activities to be performed sequentially in lexical order.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<sequence standard-attributes>
     standard-elements
     activity+
</sequence>
```

The <if> construct allows you to select exactly one branch of activity from a set of potential choices.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6
<if standard-attributes>
     standard-elements
     <condition expressionLanguage="anyURI"?>
          ... bool-expr ...
     </condition>
     <then>
          activity
     </then>
     <elseif>*
        <condition expressionLanguage="anyURI"?>
          ... bool-expr ...
        </condition>
        activity
     </elseif>
     <else>?
        activity
     </else>
</if>
```

The <while> construct allows you to indicate that an activity is to be repeated until a certain success criteria has been met.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<while standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>
     ... bool-expr ...
   </condition>

     activity
</while>
```

The <repeatUntil> constructs allows you to indicate the repeated performance of a specified iterative activity. The iterative activity will continue to be performed so long as after it executes the given Boolean <repeatUntil> condition holds true.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<repeatUntil standard-attributes>
       standard-elements
       activity
       <condition expressionLanguage="anyURI"?>
          ... bool-expr ...
       </condition>
```

```
</repeatUntil>
```

The <pick> construct allows you to block and wait for a suitable message to arrive or for a time-out alarm to go off. When one of these triggers occurs, the associated activity is performed and the pick completes.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<pick createInstance="yes|no"? standard-attributes>
     standard-elements
     <onMessage partnerLink="ncname" portType="qname"?
          operation="ncname" variable="ncname"?
          messageExchange="ncname"? >+
         <correlations>?
             <correlation set="ncname" initiate="yes|join|no"?/>+
        </correlations>
        <fromPart part="ncname" toVariable="ncname"/>*
        activity
    </onMessage>
    <onAlarm>*
          ( <for expressionLanguage="anyURI"?>duration-expr</for> |
            <until expressionLanguage="anyURI"?>deadline-expr</until> )

         activity
    </onAlarm>
</pick>
```

The portType attribute on the <onMessage> activity is optional. If the portType attribute is included for readability, the value of the portType attribute MUST match the portType value implied by the combination of the specified partnerLink and the role implicitly specified by the activity (See also partnerLink description in the next section). The optional messageExchange attribute is used to associate a <reply> activity with a <onMessage> activity. (For details of messageExchange, please see “Providing Web Service Operations” section).

The <flow> construct allows you to specify one or more activities to be performed concurrently. Links can be used within concurrent activities to define arbitrary control structures.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<flow standard-attributes>
     standard-elements
     <links>?
          <link name="ncname">+
     </links>
     activity+
</flow>
```

The <scope> construct allows you to define a nested activity with its own associated variables, fault handlers, and compensation handler.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<scope isolated="yes|no" standard-attributes>
    standard-elements
    <partnerLinks>?
         ... see above under <process> for syntax ...
    </partnerLinks>
    <variables>?
         ... see above under <process> for syntax ...
    </variables>
    <correlationSets>?
         ... see above under <process> for syntax ...
    </correlationSets>
    <faultHandlers>?
         ... see above under <process> for syntax ...
    </faultHandlers>
    <compensationHandler>?
         ... see above under <process> for syntax ...
    </compensationHandler>
    <terminationHandler>?
         ...
    </terminationHandler>
    <eventHandlers>?
          ...
    </eventHandlers>
    activity
</scope>
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6
```

The <compensate> construct is used to invoke compensation on an inner scope that has already completed normally. This construct can be invoked only from within a fault handler or another compensation handler.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<compensate scope="ncname"? standard-attributes>
        standard-elements
</compensate>
```

Note that the "*standard-attributes*" referred to above are:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

name="ncname"? suppressJoinFailure="yes|no"?
```

where the default values are as follows:

- name: No default value (that is, the default is unnamed)
- suppressJoinFailure: When this attribute is not specified for an activity, it inherits its value from its closest enclosing activity or from the process if no enclosing activity specifies this attribute.

and that the "*standard-elements*" referred to above are:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6
<targets>?
   <joinCondition expressionLanguage="anyURI"?>?
     ... bool-expr ...
   </joinCondition>
   <target linkName="ncname" />+
</targets>
<sources>?
   <source linkName="ncname">+
      <transitionCondition expressionLanguage="anyURI"?>?
        ... bool-expr ...
      </transitionCondition>
   </source>
</sources>
```

6.3. Language Extensibility

WS-BPEL contains constructs that are generally sufficient for expressing abstract and executable business processes. In some cases, however, it might be necessary to “extend” the WS-BPEL language with additional constructs from other XML namespaces.

WS-BPEL supports extensibility by allowing namespace-qualified attributes to appear on any WS-BPEL element and by allowing elements from other namespaces to appear within WS-BPEL defined elements. This is allowed in the XML Schema specifications for WS-BPEL.

If, during the processing of a BPEL process an element is encountered that is not recognized by the processor then the element and its children MUST be treated as if they were not present in the BPEL process instance. The previously defined ignore semantics make it possible to add optional attributes or elements to BPEL that can be safely ignored if not recognized. In the case of unsupported mandatory extensions (see section 13.7) the previous logic is unnecessary as the entire process instance will be rejected.

Extensions MUST NOT contradict the semantics of any element or attribute defined by the WS-BPEL specification. Extensions that affect the semantics of WS-BPEL processes MUST appear in the WS-BPEL process definition or its directly referenced WSDL portTypes, property alias definitions or property definitions.

Apart from core WS-BPEL process constructs, extensions are also allowed in WSDL-based WS-BPEL constructs, such as, <partnerLinkType>, <role>, <property> and <propertyAlias>. The same syntax pattern and semantic rules for extensions of core BPEL constructs are applied to these extensions as well. Particularly, <extension> directives of a WS-BPEL process are applied to all extensions used in WSDL-based WS-BPEL constructs, of which the WSDL definitions are transitively referenced by the process.

The <documentation> construct is designed to be an integral part of Language Extensibility. The content of <documentation> are for human-consumption. Example types for those content are: plain text, HTML and XHTML. Tool-implementation specific information (e.g. the graphical layout details) should be added through elements and attributes of other namespaces, using the general WS-BPEL Language Extensibility mechanism.

6.4. Document Linking

A WS-BPEL process definition relies on XML Schema and WSDL 1.1 for the definition of datatypes and service interfaces. Process definitions also rely on other constructs such as partner link types, variable properties and property aliases (defined later in this specification) which are defined within WSDL 1.1 documents using the WSDL 1.1 language extensibility feature.

The <import> element is used within a WS-BPEL process to explicitly indicate a dependency on external XML Schema or WSDL definitions. Any number of <import> elements may appear as initial children of the <process> element, before any other child element. Each <import> element contains three mandatory attributes

- namespace. The namespace attribute specifies the URI namespace of the imported definitions.
- location. The location attribute contains a URI indicating the location of a document that contains relevant definitions in the namespace specified. The document located at the URI MUST contain definitions belonging to the same namespace as indicated by the namespace attribute.
- importType. The importType attribute identifies the type of document being imported by providing the URI of the encoding language. The value MUST be set to "http://www.w3.org/2001/XMLSchema" when importing XML Schema 1.0 documents, and to "http://schemas.xmlsoap.org/wsdl/" when importing WSDL 1.1 documents.

The presence of an <import> element should be interpreted as a hint to the WS-BPEL processor. In particular, processors are not required to retrieve the imported document from the location specified on the <import> element.

6.5 The Lifecycle of a Business Process

As noted in the introduction, the interaction model that is directly supported by WSDL is essentially a stateless client-server model of synchronous or uncorrelated asynchronous interactions. WS-BPEL, builds on WSDL by assuming that all external interactions of the business process occur through Web Service operations. However, WS-BPEL business processes represent stateful long-running interactions in which each interaction has a beginning, defined behavior during its lifetime, and an end. For example, in a supply chain, a seller's business process might offer a service that begins an interaction by accepting a purchase order through an input message, and then returns an acknowledgement to the buyer if the order can be fulfilled. It might later send further messages to the buyer, such as shipping notices and invoices. The seller's business process remembers the state of each such purchase order interaction separately from other similar interactions. This is necessary because a buyer might be carrying on many simultaneous purchase processes with the same seller. In short, a WS-BPEL business process definition can be thought of as a template for creating business process instances.

The creation of a process instance in WS-BPEL is always implicit; activities that receive messages (that is, receive activities and pick activities) can be annotated to indicate that the occurrence of that activity causes a new instance of the business process to be created. This is done by setting the createInstance attribute of such an activity to "yes". When a message is received by such an activity, an instance of the business process is created if it does not already exist (see Providing Web Service Operations and Pick).

To be instantiated, each business process must contain at least one such "start activity." That is, a receive/pick activity annotated with a createInstance="yes" attribute. See section 11.4 for more details on start activities.

If more than one start activity is enabled concurrently, then all such activities MUST share at least one common correlation set (see Correlation and the Multiple Start Activities example).

If a process contains exactly one start activity then the use of correlation sets is unconstrained. This includes a pick with multiple onMessage branches; each such branch can use different correlation sets or no correlation sets.

A business process instance is terminated in one of the following ways:

- When the activity that defines the behavior of the process as a whole completes. In this case the termination is normal.
- When a fault reaches the process scope, and is either handled or not handled. In this case the termination is considered abnormal even if the fault is handled and the fault handler does not rethrow any fault. A compensation handler is never installed for a scope that terminates abnormally.
- When a process instance is explicitly terminated by an `exit` activity (see `Terminating the Service Instance`). In this case the termination is abnormal.

The structure of the main processing section is defined by the outer <sequence> element, which states that the three activities contained inside are performed in order. The customer request is received (<receive> element), then processed (inside a <flow> section that enables concurrent behavior), and a reply message with the final approval status of the request is sent back to the customer (<reply>). Note that the <receive> and <reply> elements are matched respectively to the <input> and <output> messages of the "sendPurchaseOrder" operation invoked by the customer, while the activities performed by the process between these elements represent the actions taken in response to the customer request, from the time the request is received to the time the response is sent back (reply).

A receive activity for an inbound request/response operation is said to be *open* if that activity has been performed and no corresponding reply activity has been performed. If the process instance reaches the end of its behavior, and one or more receive activities remain open, then the status of the instance becomes undefined. This condition indicates a modeling error that was not detected by static analysis.

The example makes the implicit assumption that the customer request can be processed in a reasonable amount of time, justifying the requirement that the invoker wait for a synchronous response (because this service is offered as a request-response operation). When that assumption does not hold, the interaction with the customer is better modeled as a pair of asynchronous message exchanges. In that case, the "sendPurchaseOrder" operation is a one-way operation and the asynchronous response is sent by invoking a second one-way operation on a customer "callback" interface. In addition to changing the signature of "sendPurchaseOrder" and defining a new portType to represent the customer callback interface, two modifications need to be made in the preceding example to support an asynchronous response to the customer. First, the partner link type "purchasingLT" that represents the process-customer connection needs to include a second role ("customer") listing the customer callback portType. Second, the <reply> activity in the process needs to be replaced by an <invoke> on the customer callback operation.

The processing taking place inside the <flow> element consists of three <sequence> blocks running concurrently. The synchronization dependencies between activities in the three concurrent sequences are expressed by using "links" to connect them. The links are defined inside the flow and are used to connect a source activity to a target activity. (Note that each activity declares itself as the source or target of a link by using the nested <source> and <target> elements.) In the absence of links, the activities nested directly inside a flow proceed concurrently. In the example, however, the presence of two links introduces control dependencies between the activities performed inside each sequence. For example, while the price calculation can be started immediately after the request is received, shipping price can only be added to the invoice after the shipper information has been obtained; this dependency is represented by the link (named "ship-to-invoice") that connects the first call on the shipping provider ("requestShipping") with sending shipping information to the price calculation service ("sendShippingPrice"). Likewise,

shipping scheduling information can only be sent to the manufacturing scheduling service after it has been received from the shipper service; thus the need for the second link ("ship-to-scheduling").

Observe that information is passed between the different activities in an implicit way through the sharing of globally visible data variables. In this example, the control dependencies represented by links are related to corresponding data dependencies, in one case on the availability of the shipper rates and in another on the availability of a shipping schedule. The information is passed from the activity that generates it to the activity that uses it by means of two global data variables ("shippingInfo" and "shippingSchedule").

Certain operations can return faults, as defined in their WSDL definitions. For simplicity, it is assumed here that the two operations return the same fault ("cannotCompleteOrder"). When a fault occurs, normal processing is terminated and control is transferred to the corresponding fault handler, as defined in the <faultHandlers> section. In this example the handler uses a <reply> element to return a fault to the customer (note the "faultName" attribute in the <reply> element).

Finally, it is important to observe how an assignment activity is used to transfer information between data variables. The simple assignments shown in this example transfer a message part from a source variable to a message part in a target variable, but more complex forms of assignments are also possible.

7. Partner Link Types, Partner Links, and Endpoint References

A very important, if not the most important, use case for WS-BPEL will be in describing cross-enterprise business interactions in which the business processes of each enterprise interact through Web Service interfaces with the processes of other enterprises. An important requirement for realistic modeling of business processing in this environment is the ability to model the required relationship with a partner process. WSDL already describes the functionality of a service provided by a partner, at both the abstract and concrete levels. The relationship of a business process to a partner is typically peer-to-peer, requiring a two-way dependency at the service level. In other words, a partner represents both a consumer of a service provided by the business process and a provider of a service to the business process. This is especially the case when the interactions are based on asynchronous messaging rather than on remote procedure calls. The notion of Partner links is used to directly model peer-to-peer conversational partner relationships. Partner links define the shape of a relationship with a partner by defining the message and port types used in the interactions in both directions. However, the actual partner service may be dynamically determined within the process. WS-BPEL uses a notion of endpoint reference, manifested as a service reference container (“bpws:service-ref”), to represent the dynamic data required to describe a partner service endpoint.

It is important to emphasize that the notions of partner link and endpoint reference used here are preliminary. The specification for these concepts as they relate to Web Services is still evolving, and we expect normative definitions for them to emerge in future. The WS-BPEL specification will be updated to conform to the expected future standards.

7.1. Partner Link Types

A partner link type characterizes the conversational relationship between two services by defining the "roles" played by each of the services in the conversation and specifying the portType provided by each service to receive messages within the context of the conversation. The following example illustrates the basic syntax of a partner link type declaration:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<partnerLinkType name="BuyerSellerLink"
     xmlns="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">
     <role name="Buyer" portType="buy:BuyerPortType"/>
     <role name="Seller" portType="sell:SellerPortType"/>
</partnerLinkType>
```

Each role specifies exactly one WSDL portType.

In the common case, portTypes of the two roles originate from separate namespaces. However, in some cases, both roles of a partner link type can be defined in terms of portTypes from the same namespace. The latter situation occurs for partner link types that define "callback" relationships between services.

The partner link type definition can be a separate artifact independent of either service's WSDL document. Alternatively, the partner link type definition can be placed within the WSDL document defining the portTypes from which the different roles are defined.

The extensibility mechanism of WSDL 1.1 is used to define partnerLinkType as a new definition type to be placed as an immediate child element of a <wsdl:definitions> element in all cases. This allows reuse of the WSDL target namespace specification and, more importantly, its import mechanism to import portTypes. For cases where a partnerLinkType declaration is linking the portTypes of two different services, the partnerLinkType declaration can be placed in a separate WSDL document (with its own targetNamespace).

The syntax for defining a partnerLinkType is:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<definitions name="ncname" targetNamespace="uri"
     xmlns="http://schemas.xmlsoap.org/wsdl/">
        ...
```

```
     <plnk:partnerLinkType name="ncname">
          <plnk:role name="ncname" portType="qname"/>
          <plnk:role name="ncname" portType="qname"/>?

     </plnk:partnerLinkType>
        ...
</definitions>
```

This defines a partner link type in the namespace indicated by the value of the "targetNamespace" attribute of the WSDL document element. The portTypes identified within roles are referenced by using QNames as for all top-level WSDL definitions.

Note that in some cases it can be meaningful to define a partner link type containing exactly one role instead of two. That defines a partner linking scenario where one service expresses a willingness to link with any other service, without placing any requirements on the other service.

Examples of partnerLinkType declarations are found in various business process examples in this specification.

7.2. Partner Links

The services with which a business process interacts are modeled as partner links in WS-BPEL. Each partner link is characterized by a partnerLinkType. More than one partner link can be characterized by the same partnerLinkType. For example, a certain procurement process might use more than one vendor for its transactions, but might use the same partnerLinkType for all vendors.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<partnerLinks>
     <partnerLink name="ncname" partnerLinkType="qname"
          myRole="ncname"? partnerRole="ncname"?
          initializePartnerRole="yes|no"?>+
     </partnerLink>
</partnerLinks>
```

Each partnerLink is named, and this name is used for all service interactions via that partnerLink. This is critical, for example, in correlating responses to different partnerLinks for simultaneous requests of the same kind (see Invoking **Web Service Operations and Providing Web Service Operations**).

The role of the business process itself is indicated by the attribute *myRole* and the role of the partner is indicated by the attribute *partnerRole*. In the degenerate case where a partnerLinkType has only one role, one of these attributes is omitted as appropriate.

Note that the partnerLink declarations specify the shape of the relationships that the WS-BPEL process will employ in its behavior. Before operations on a partner's service can be

invoked via a partnerLink, the binding and communication data for the partner service must be available. The relevant information about a partner service can be set as part of business process deployment. This is outside the scope of WS-BPEL. However, it is also possible to select and assign actual partner services dynamically, and WS-BPEL provides the mechanisms to do so via assignment of endpoint references. In fact, because the partners are likely to be stateful, the service endpoint information needs to be extended with instance-specific information. WS-BPEL allows the endpoint references implicitly present in partnerLinks to be both extracted and assigned dynamically, and also to be set more than once. See **Assignment** for the mechanisms used for dynamic assignment of endpoint references to partner services.

The initializePartnerRole attribute specifies if the BPEL processor is required to initialize a partnerLink's partnerRole value. The attribute has no affect on the partnerRole's value after its initialization. The initializePartnerRole attribute MUST NOT be used on a partnerLink that does not have a partner role; this restriction MUST be statically enforced. If the initializePartnerRole attribute is set to "yes" then the BPEL processor MUST initialize the EPR for the specified partnerLink/partnerRole combination before that partnerRole is first referenced by the BPEL process. If the initializePartnerRole attribute is set to "no" then the BPEL processor MUST NOT initialize the EPR for the specified partnerLink/partnerRole combination before that partnerRole is first referenced by the BPEL process. If the initializePartnerRole attribute is omitted then its value MUST be treated as "no".

A partnerLink can be declared within a process or scope element. The name of a partnerLink should be unique amongst the names of all partnerLinks defined within the same immediately enclosing scope. This requirement MUST be statically enforced.. Access to partnerLink follows common lexical scoping rules, similar to the rules for variables. A partnerLink resolves to the nearest enclosing scope, regardless of the type of the partnerLink. If a local partnerLink declared in an enclosing scope, the local partnerLink will be used in local assignments and message sending/receiving activities. The lifecyle of a partnerLink is same as the lifecycle of the scope declaring the partnerLink. The initial binding information of a partnerLink can be set as a part of business process deployment, regardless whether it is declared on the process or scope element level.

7.4. Endpoint References

WSDL makes an important distinction between portTypes and ports. PortTypes define abstract functionality by using abstract messages. Ports provide actual access information, including communication service endpoints and (by using extension elements) other deployment related information such as public keys for encryption. Bindings provide the glue between the two. While the user of a service must be statically dependent on the abstract interface defined by portTypes, some of the information contained in port definitions can typically be discovered and used dynamically.

The fundamental use of endpoint references is to serve as the mechanism for dynamic communication of port-specific data for services. An endpoint reference makes it possible in WS-BPEL to dynamically select a provider for a particular type of service and to invoke their operations. WS-BPEL provides a general mechanism for correlating messages to stateful instances of a service, and therefore endpoint references that carry instance-neutral port information are often sufficient. However, in general it is necessary to carry additional instance-identification tokens in the endpoint reference itself.

Every partner role in a partnerLink in a WS-BPEL process instance is assigned a unique endpoint reference in the course of the deployment of the process or dynamically by an activity within the process.

Endpoint references associated with parterRole and myRole of partnerLinks are manifested as service reference containers (“bpws:service-ref”). This container is used as envelope to wrap around the actual endpoint reference value, when a WS-BPEL process interacts the endpoint reference of a partnerLink. It provides pluggability of different versions of service endpoint referencing schemes being used within a BPEL program. The design pattern here is similar to those of expression language, also known as open-content models.

The “bpws:service-ref” has an optional attribute called “reference-scheme” to to denote the URI of the reference interpretation scheme of service endpoint, which is the child element of bpws:service-ref.

Most likely, the URI of reference scheme and the namespace URI of the child element of bpws:service-ref are not necessarily the same. Typically, this optional attribute is used ONLY when the child element of the “bpws:service-ref” is ambiguous by itself. The optional attribute supplies further information to disambiguate the usage of the content. One example would be: different treatments of wsdl:service element, if wsdl:service is used as the endpoint reference.

If that attribute is not specified, use the namespace URI of the content element within the wrapper to determine the reference scheme of service endpoint.

If the attribute is specified, use the URI as the reference scheme of service endpoint and treat the content element within the wrapper accordingly. When the "reference-scheme" attribute is specified, the URI value is most likely different from the namespace URI of the content element for EPR.

When the BPEL container fails to interpret the combination of the "reference-scheme" attribute and the content element OR just the content element alone, a standard fault "bpws:unsupportedReference" must be thrown.

When WS-BPEL users interact and manipulate endpoint references of partnerLinks, The “bpws:service-ref” element are NOT exposed to WS-BPEL users in all cases. For example, when people try to do an assignment from the endpoint reference of myRole of

partnerLink-A to that of partnerRole of partner-B. On the contrary, if people try to do assignment from a messageType or element based variable through expression or from a service-ref -form of from-spec, the “bpws:service-ref” is visible to WS-BPEL users.

8. Variable Properties

8.1. Motivation

8.1.1 Motivation for Message Properties

The data in a message consists conceptually of two parts: application data and protocol relevant data, where the protocols can be business protocols or infrastructure protocols providing higher quality of service. An example of business protocol data is the correlation tokens that are used in correlation sets (see **Correlation**). Examples of infrastructure protocols are security, transaction, and reliable messaging protocols. The business protocol data is usually found embedded in the application-visible message parts, whereas the infrastructure protocols almost always add *implicit* extra parts to the message types to represent protocol headers that are separate from application data. Such implicit parts are often called *message context* because they relate to security context, transaction context, and other similar middleware context of the interaction. Business processes might need to gain access to and manipulate both kinds of protocol-relevant data. The notion of message properties is defined as a general way of naming and representing distinguished data elements within a message, whether in application-visible data or in message context. For a full accounting of the service description aspects of infrastructure protocols, it is necessary to define notions of service policies, endpoint properties, and message context. This work is outside the scope of
WS-BPEL. Message properties are defined here in a sufficiently general way to cover message context consisting of implicit parts, but the use in this specification focuses on properties embedded in application-visible data that is used in the definition of business protocols and abstract business processes.

8.1.2 Motivation for Variable Properties

Message properties are an instance of a more generic mechanism, variable properties. All variables in BPEL can have properties defined on them. Properties are useful on non-message variables as a way to isolate the BPEL process’s logic from the details of a particular variable’s definition. Using properties a BPEL process can isolate its variable initialization logic in one place and then set and get properties on that variable in order to manipulate it. If the variable’s definition is later changed the bulk of the BPEL process definition that manipulates that variable can remain unchanged.

8.2. Defining Properties

A property definition creates a globally unique name and associates it with an XML Schema type. The intent is not to create a new type. The intent is to create a name that has greater significance than the type itself. For example, a sequence number can be an integer, but the integer type does not convey this significance, whereas a globally named sequence-number property does. Properties can occur anywhere in a variable.

A typical use for a property in WS-BPEL is to name a token for correlation of service instances with messages. For performance reasons, properties used for correlation MUST be defined using XML Schema simple types, this restriction MUST be statically enforced. For example, a social security number might be used to identify an individual taxpayer in a long-running multiparty business process regarding a tax matter. A social security number can appear in many different message types, but in the context of a tax-related process it has a specific significance as a taxpayer ID. Therefore a global name is given to this use of the type by defining a property, as in the following example:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<definitions name="properties"
        targetNamespace="http://example.com/properties.wsdl"
        xmlns:tns="http://example.com/properties.wsdl"
        xmlns:txtyp="http://example.com/taxTypes.xsd"
        xmlns="http://schemas.xmlsoap.org/wsdl/">

        <! -- import schema taxTypes.xsd -- >
        <!-- define a correlation property -->
        <bpws:property name="taxpayerNumber"
                type="txtyp:SSN"/>
        ...
</wsdl:definitions>
```

In correlation, the property name must have global significance to be of any use. Properties such as price, risk, response latency, and so on, which are used in conditional behavior in a business process, have similar global and public significance. It is likely that they will be mapped to multiple messages, and therefore they need to be globally named as in the case of correlation properties. Such properties are essential, especially in abstract processes.

Even in the general case of properties on XML typed WS-BPEL variables the property name should maintain its generic nature. The name is intended to identify a certain kind of value, often with an implied semantic. Any variable the property is available on is therefore expected to provide a value that meets not just the syntax of the property definition but also its semantics.

The WSDL extensibility mechanism is used to define properties so that the target namespace and other useful aspects of WSDL are available.

The WS-BPEL standard namespace, "`http//schemas.xmlsoap.org/ws/2004/03/business-process/`", is used for

property definitions. The syntax for a property definition is a new kind of WSDL definition as follows:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<wsdl:definitions name="ncname">
       <bpws:property name="ncname" type="qname"? element="qname"?/>
                ...
</wsdl:definitions>
```

Either the type or element attributes MUST be present but not both. Properties used in business protocols are typically embedded in application-visible message data. The notion of aliasing is introduced to map a global property to a field in a specific message part or a value in a XML variable. The property name becomes an alias for the message part and location or XML variable value, and can be used as such in **Expressions** and **Assignment** in abstract business processes. As an example, consider the following WSDL message definition:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<definitions name="messages"
       targetNamespace="http://example.com/taxMessages.wsdl"
       xmlns:txtyp="http://example.com/taxTypes.xsd"
       xmlns="http://schemas.xmlsoap.org/wsdl/">

       <!-- define a WSDL application message -->
       <message name="taxpayerInfo">
         <part name="identification" element="txtyp:socialsecnumber"/>
       </message>
                ...
</definitions>
```

The definition of a global property and its location in a particular field of the message are shown in the next WSDL fragment:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<definitions name="properties"
       targetNamespace="http://example.com/properties.wsdl"
       xmlns:tns="http://example.com/properties.wsdl"
       xmlns:txtyp="http://example.com/taxTypes.xsd"
       xmlns:txmsg="http://example.com/taxMessages.wsdl"

       xmlns="http://schemas.xmlsoap.org/wsdl/">

       <!-- define a correlation property -->
       <bpws:property name="taxpayerNumber" type="txtyp:SSN"/>
                ...
       <bpws:propertyAlias propertyName="tns:taxpayerNumber"
```

```
                messageType="txmsg:taxpayerInfo" part="identification">
                <query>
                /txtyp:socialsecnumber
                </query>
        </bpws:propertyAlias>
</definitions>
```

The `bpws:propertyAlias` defines a globally named property `tns:taxpayerNumber` as an alias for a location in the `identification` part of the message type `txmsg:taxpayerInfo`.

The syntax for a propertyAlias definition is:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
         1         2         3         4         5         6

<definitions name="ncname"
        ...
>

        <bpws:propertyAlias propertyName="qname"
                messageType="qname"? part="ncname"?
                type="qname"? element="qname"?>
           <query queryLanguage="anyURI"?>?
           ... queryString ...
           </query>
        </bpws:propertyAlias>
        ...
</wsdl:definitions>
```

The interpretation of the message and part attributes, as well as the <query> element is the same as in the corresponding from-spec in copy assignments (see **Assignment**).

8.3 Defining Property Aliases

Properties used in business protocols are typically embedded in application-visible variables. The notion of aliasing is introduced to map a global property to a field in a specific message part or variable value. The property name becomes an alias for the message part and/or location, and can be used as such in Expressions and Assignment in abstract business processes. As an example, consider the following WSDL message definition:

```
<definitions name="messages"
        targetNamespace="http://example.com/taxMessages.wsdl"
        xmlns:txtyp="http://example.com/taxTypes.xsd"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
        <!-- define a WSDL application message -->
        <message name="taxpayerInfoMsg">
          <part name="identification"
element="txtyp:taxPayerInfoElem"/>
        </message>
                ...
```

```
</definitions>
```

The definition of a global property and its location in a particular field of the message are shown in the next WSDL fragment:

```
<definitions name="properties"
        targetNamespace="http://example.com/properties.wsdl"
        xmlns:tns="http://example.com/properties.wsdl"
        xmlns:txtyp="http://example.com/taxTypes.xsd"
        xmlns:txmsg="http://example.com/taxMessages.wsdl"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
        <!-- define a correlation property -->
        <bpws:property name="taxpayerNumber" type="txtyp:SSN"/>
               ...
        <bpws:propertyAlias propertyName="tns:taxpayerNumber"
messageType="txmsg:taxpayerInfoMsg" part="identification">
               <query>
                  txtyp:socialsecnumber
               </query>
        </bpws:propertyAlias>
        <bpws:propertyAlias propertyName="tns:taxpayerNumber"
               element="txtyp:taxPayerInfoElem">
               <query>
                 txtyp:socialsecnumber
               </query>
        </bpws:propertyAlias>
</definitions>
```

The first bpws:propertyAlias defines a globally named property tns:taxpayerNumber as an alias for a location in the identification part of the message type txmsg:taxpayerInfo.

The second bpws:propertyAlias provides a second definition for the same globally named property tns:taxpayerNumber but this time as an alias for a location inside of the element txtyp:taxPayerInfoElem.

The presence of both aliases means that it is possible to retrieve the social security number from both a variable holding a message of messageType txmsg:taxpayerInfo as well as an element defined using txtyp:taxPayerInfoElem.

The syntax for a propertyAlias definition is:

```
<definitions name="ncname"
        ... >
        <bpws:propertyAlias propertyName="qname"
               messageType="qname"? part="ncname"?
               type="qname"? element="qname"?>
           <query queryLanguage="anyURI"?>?
           ... queryString ...
           </query>
        </bpws:propertyAlias>
        ...
</wsdl:definitions>
```

A property alias element MUST use one of the three following combinations of attributes:

- messageType and part,
- type or
- element.

When a propertyAlias is used with the messageType/part combination then the property MUST be available on all BPEL variables where the qname value used in the messageType attribute on the declaration of both the variable and the propertyAlias are identical. The part attribute and query element is applied against the BPEL messageType variable to either set or get the property variable in the same way that the part attribute and query element are used in the first from and to specs in copy assignments.

Using the same "`tns:taxpayerNumber`" example from above, for a message variable "`myTaxPayerInfoMsg`" of type "`txmsg:taxpayerInfo`":

```
<from variable="myTaxPayerInfoMsg" property="tns:taxpayerNumber" />
```

and

```
<from>$myTaxPayerInfoMsg.identification/txtyp:socialsecnumber</from>
```

have the same output. Please see **Assignment** for details.

If a propertyAlias is used with a type attribute then the property MUST be available on all BPEL variables where the qname value used in the type attribute on the declaration of both the variable and the propertyAlias are identical. The query is applied against the BPEL variable to either set or get the property variable in the same way that the query is used in the first from and to specs in copy assignments when applied against BPEL variables defined using a type.

If a propertyAlias is used with an element attribute then the property MUST be available on all BPEL variables where the qname value used in the element attribute on the declaration of both the variable and the propertyAlias are identical. The query is applied against the BPEL variable to either set or get the property variable in the same way that the query is used in the first from and to specs in copy assignments when applied against BPEL variables defined using an element definition.

A BPEL process MUST NOT be accepted for processing if it defines two or more propertyAlias's for the same property name and BPEL variable type. For example, it is not legal to define two propertyAlias's for the property taxpayernumber and the messageType txmsg:taxpayerInfoMsg. The same logic would prohibit having two propertyAliases on the same element qname and property name value or two propertyAliases on the same type qname and property name value.

9. Data Handling

Business processes model stateful interactions. The state involved consists of messages received and sent as well as other relevant data such as time-out values. The maintenance of the state of a business process requires the use of state variables, which are called variables in WS-BPEL. Furthermore, the data from the state needs to be extracted and combined in interesting ways to control the behavior of the process, which requires data expressions. Finally, state update requires a notion of assignment. WS-BPEL provides these features for XML data types and WSDL message types. The XML family of standards in these areas is still evolving, and using the process-level attributes for query and expression languages provides for the incorporation of future standards.

The extensions required for abstract and executable processes are concentrated in the datahandling feature set. Executable processes are permitted to use the full power of data selection and assignment but are not permitted to use nondeterministic values. Abstract processes are restricted to limited manipulation of values contained in variable properties but are permitted to use nondeterministic values to reflect the consequences of hidden private behavior. Detailed differences are specified in the following sections.

9.3. Expressions

(Editor note: re-arranging the order of this section later for ease of tracking for this version)

WS-BPEL uses several types of expressions. The kinds of expressions used are as follows (relevant usage contexts are listed in parentheses):

- Boolean-valued expressions (transition conditions, join conditions, while condition, and if conditions)
- Deadline-valued expressions ("until" expression of onAlarm and wait)
- Duration-valued expressions ("for" expression of onAlarm and wait, "repeatEvery" expression of onAlarm)
- General expressions (assignment)

WS-BPEL provides an extensible mechanism for the language used in these expressions. The language is specified by the `expressionLanguage` attribute of the `process` element. In addition, language constructs that require or allow conditional expressions (such as “if”, “while” and others) provide the ability to override the default expression language for individual expressions. Compliant implementations of the current version of WS-BPEL MUST support the use of XPath 1.0 as the expression language. XPath 1.0 is indicated by the default value of the `expressionLanguage` attribute, which is:

urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0

which represents the usage of XPath 1.0 within WS-BPEL 2.0. The URI of the corresponding XPath 1.0 specification is:

http://www.w3.org/TR/1999/REC-xpath-19991116

Given an expression language, it must be possible to query data from variables, to extract property values, and to query the status of links from within expressions. This specification defines those functions for XPath 1.0 only, and it is expected that other expressionlanguage bindings will provide equivalent functionality. The rest of this section is specific to XPath 1.0.

WS-BPEL introduces several extension functions to XPath's built-in functions to enable XPath 1.0 expressions to access information from the process. The extensions are defined in the standard WS-BPEL namespace http://schemas.xmlsoap.org/ws/2004/03/businessprocess/. The prefix "`bpws:`" is associated with this namespace.

Any qualified names used within XPath expressions are resolved by using namespace declarations currently in scope in the WS-BPEL document at the location of the expression.

The arguments to all XPath functions defined in this specification MUST be given as quoted strings. The previous requirement MUST be statically enforced. It is therefore illegal to pass into a BPEL XPath function any XPath variables, the output of XPath functions, a XPath location path or any other value that is not a quoted string. This means, for example, that `getVariableProperty("varA","propB")` meets the previous requirement while `getVariableProperty( $varA, string(getVariableProperty("varB","propB") )` does not. Note that the previous requirement institutes a restriction which does not exist in the XPath standard.

The following functions are defined by this specification:

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

This function extracts global property values from variables. The first argument names the source variable for the data and the second is the qualified name (QName) of the global property to select from that variable (see Variable Properties). If the referenced property is not defined or if there does not exist a propertyAlias to associate the property with the referenced variable then the semantics of the process is undefined. The return value of this function is calculated by applying the appropriate propertyAlias for the requested property to the current value of the submitted variable. If the application of the XPATH in the propertyAlias to the variable value results in a response that contains anything other than exactly one information item and/or a collection of Character information items then a bpws:selectionFailure fault MUST be thrown.

These WS-BPEL-defined extension functions are available for use within all XPath 1.0 expressions.

The syntax of XPath 1.0 expressions for WS-BPEL is considered in the following paragraphs.

9.3.1. Boolean Expressions

These are expressions that conform to the XPath 1.0 *Expr* production where the evaluation results in Boolean values.

9.3.2. Deadline-Valued Expressions

These are expressions that conform to the XPath 1.0 *Expr* production where the evaluation results in values that are of the XML Schema types *dateTime* or *date*. Note that XPath 1.0 is not XML Schema aware. As such, none of the built-in functions of XPath 1.0 are capable of producing or manipulating dateTime or date values. However, it is possible to write a constant (literal) that conforms to XML Schema definitions and use that as a deadline value or to extract a field from a variable (part) of one of these types and use that as a deadline value. XPath 1.0 will treat that literal as a string literal, but the result can be interpreted as a *lexical representation* of a dateTime or date value.

9.3.3. Duration-Valued Expressions

These are expressions that conform to the XPath 1.0 *Expr* production where the evaluation results in values that are of the XML Schema type *duration*. The preceding discussion about XPath 1.0's XML Schema unawareness applies here as well.

9.3.4. General Expressions

These are expressions that conform to the XPath 1.0 *Expr* production where the evaluation results in any XPath value type (string, number, or Boolean).

If the execution of an expression results in an unhandled expression language execution fault, then the WS-BPEL standard fault bpws:subLanguageExecutionFault MUST be thrown.

9.1. Variables

(Editor note: re-arranging the order of this section later for ease of tracking for this version)

Business processes specify stateful interactions involving the exchange of messages between partners. The state of a business process includes the messages that are

exchanged as well as intermediate data used in business logic and in composing messages sent to partners.

Variables provide the means for holding messages that constitute the state of a business process. The messages held are often those that have been received from partners or are to be sent to partners. Variables can also hold data that are needed for holding state related to the process and never exchanged with partners.

The type of each variable may be a WSDL message type, an XML Schema type (simple or complex) or an XML Schema element. The syntax of the `variables` declaration is:

```
<variables>
        <variable name="ncname" messageType="qname"?
                type=”qname”? element=”qname”?/>+
            from-spec?
</variables>
```

The name of a variable MUST be unique amongst the names of all variables defined within the same immediately enclosing scope. This requirement MUST be statically enforced.. Variable names are NCNames (as defined in XML Schema specification) but in addition they MUST NOT contain the “.” character. This restriction is necessary because the “.” character is used as a delimiter in variable names in BPEL's default binding to XPath 1.0 (i.e. the binding identified by "urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"). The delimiter separates the WS-BPEL message type variable name and the name of one of its WSDL message parts. The concatenation of the WSDL message variable name, the delimiter and the WSDL part name is used as an XPath variable reference which manifests the XML Infoset of the corresponding WSDL message part. Variable access follows common lexical scoping rules. A variable resolves to the nearest enclosing scope, regardless of the type of the variable. If a local variable has the same name as a variable defined in an enclosing scope, the local variable will be used in local assignments and/or getVariableProperty functions.

The `messageType`, `type` or `element` attributes are used to specify the type of a variable. Exactly one of these attributes must be used. Attribute `messageType` refers to a WSDL message type definition. Attribute `type` refers to an XML Schema type (simple or complex). Attribute `element` refers to an XML Schema element.

Using [Infoset] terminology, the infoset for a BPEL element variable consists of a document information item (DII) that contains exactly one child, an element information item (EII) which is pointed at by the document element property. The EII is the value of the element variable.

If a BPEL implementation chooses to manifest a simpleType variable as an XML infoset**,** the infoset should consist of a DII that contains exactly one child, an EII which is pointed at by the document element property. The properties of the document element, specifically the namespace name and local name properties, are undefined by this

specification, as such an implementation MUST specify whatever namespace name/local name values it likes. However the children of the document element MUST exclusively consist of a series of character information items (CharIIs) that represent the simpleType value.A BPEL implementation MAY choose to map simpleType variables to non-XML-infoset data-models defined in the expression/query language being used. (e.g. Boolean in XPath 1.0)

A variable can optionally be initialized by using an in-line from-spec. From-spec is defined in section 9.3. Conceptually the in-line variable initializations are modeled as a virtual sequence activity that contains a series of virtual assign activities, one for each variable being initialized in the order they are listed in the variable declarations. The virtual assigns then each contain a single virtual copy whose from-spec is as given in the variable initialization and the to-spec points to the variable being created. The previous discussion of virtual sequences and assigns is only meant for modeling purposes. The description is strictly intended to describe initialization behavior in terms of previously understood concepts but there is no intention that the behavior be seen as actually implying that any of the 'virtual' sequences or assigns exist as anything other than a model of the expected behavior.

Variable initialization logic contained in scopes that contain or whose progeny contain a start activity MUST only use idempotent functions. The use of idempotent functions allows for all the values for such variables to be pre-computed and re-used on each process instance.

The infoset for a complexType variable consists of a document information item that contains exactly one child, an element information item which is pointed at by the document element property. The properties of the document element, specifically the namespace name and local name properties, are undefined by this specification, as such an implementation MUST specify whatever legal property values it likes. However the children of the document element MUST exclusively consist of the complexType values assigned to the variable. In order to simplify data access, WSDL parts of WSDL message variables are manifested in BPEL as infosets, one infoset per WSDL message part. BPEL engines MUST follow the following algorithm when manifesting a WSDL message part as an infoset.
For each part in the WSDL message definition:

Step 1 – Create a synthetic DII which has no children other than as specified below.

Step 2a – If the WSDL message part is defined using the type attribute then create an EII as a child of the document element. The local name and namespace name of the newly created EII are determined by the WS-BPEL 2.0 engine and are not specified by this document. The handling of this EII is similar to how WS-BPEL 2.0 handles the containers for complex and simple type XML variables. The contents of the new EII are required to conform to the contents defined by the referenced type definition.

Step 2b – If the WSDL message part is defined using the element attribute then create an EII as a child of the document element which manifests the element defined by the referenced type definition.

The previous models are conceptual; they define how WS-BPEL 2.0 submits and retrieves XML variable values using infoset definitions. But these models are not intended to require that WS-BPEL 2.0 implementations actually implement an infoset model, only that however variable binding is handled the end result duplicates the behaviors defined using the infoset model. For example, a WS-BPEL 2.0 implementation may choose to bind a simple type BPEL variable of type xs:string directly to a String object in XPath 1.0. The choice of mapping MUST be consistently applied to variables and WSDL message part values of the same XML schema type. E.g. if a xs:string variable is manifested as a string object, a xs:string message part MUST be manifested as a string object also. For detailed definition of manifestation of BPEL variables in XPath 1.0, please see “Binding BPEL Variables In XPath 1.0” section.

In summary, a BPEL variable is manifested as XML Infoset items in one of the following ways:

(1) a single XML infoset item: e.g. an element or complexType variable or a WSDL message part
(2) a sequence of Character Information Items for simple type data: e.g. used to manifest string (Please note these items may be manifested as a non XML infoset item when needed. e.g. Boolean)

An example of a variable declaration using a message type declared in a WSDL document with the targetNamespace *"http://example.com/orders"*:

```
<variable xmlns:ORD="http://example.com/orders"
        name="orderDetails" messageType="ORD:orderDetails"/>
```

Variables associated with message types can be specified as input or output variables for invoke, receive, and reply activities (see Invoking **Web Service Operations** and **Providing Web Service Operations**). When an invoke operation returns a fault message, this causes a fault in the current scope. The fault variable in the corresponding fault handler is initialized with the fault message received (see **Scopes** and **Fault Handlers**).

Each variable is declared within a scope and is said to belong to that scope. Variables that belong to the global process scope are called global variables. Variables may also belong to other, non-global scopes, and such variables are called local variables. Each variable is visible only in the scope in which it is defined and in all scopes nested within the scope it belongs to. Thus, global variables are visible throughout the process. It is possible to "hide" a variable in an outer scope by declaring a variable with an identical name in an inner scope. These rules are exactly analogous to those in programming languages with lexical scoping of variables.

A global variable is in an uninitialized state at the beginning of a process. A local variable is in an uninitialized state at the start of the scope it belongs to. Note that non-global scopes in general start and complete their behavior more than once in the lifetime

of the process instance they belong to. Variables can be initialized by a variety of means including assignment and receiving a message. Variables can be partially initialized with property assignment or when some but not all parts in the message type of the variable are assigned values.

Values stored in variables can be mutated during the course of process execution. The `validate` activity can be used explicitly to ensure that values of variables are valid against their associated XML data definition, including XML Schema simple type, complex type, element definition and XML definitions of WSDL parts. The `validate` activity has a `variables` attribute, which has a NCNAMES value. The `variables` attribute points to the variables being validated. The syntax of the `validate` activity is:

```
<validate variables="ncnames" />
```

When one or more variables are invalid against their correponding XML definition, a standard fault of "`bpws:invalidVariables`" fault MUST be thrown.

A BPEL implementation MAY provide a mechanism to turn on/off any explicit validation. E.g. `validate` activity.

An BPEL implementation MAY provide a configurable mechanism to enable or disable schema validation of incoming and outgoing messages during the execution of Web Service related activities, e.g., receive, reply, pick, onEvent and invoke activities. The details of configuration mechanism is out of the scope of this specification.

Formatted: (Asian) Chinese (PRC)

Formatted: Font: (Asian) Times New Roman, English (U.S.)

9.2 Usage of Query and Expression Languages

This section models the interaction between Query/Expression languages and the BPEL process from two different perspectives. The first perspective is BPEL's view of the query/expression languages. That view is restricted to what information BPEL will make available for use by the Query/Expression language. The second perspective is the Query/Expression language's view of BPEL, specifically XPath 1.0 and how XPath 1.0's execution context is initialized by BPEL.

9.2.1 Enclosing Elements

In order to describe the view that BPEL provides to Query/Expression languages it is necessary to introduce a new term - Enclosing Element.

Definition of an Enclosing Element of a Query or Expression language: Whenever a query or expression language is used in BPEL (e.g. XPath 1.0), the Enclosing Element is defined as the parent element in the BPEL process definition that contains the Query or Expression. For example, in the following from specification example:

```
<process>
    ...
    <from>$myVar/abc/def</from>
```

```
    ...
</process>
```

The "from" element is the Enclosing Element.

The in-scope namespaces of the enclosing element are the namespaces visible to the Query/Expression language. (Note: XPath 1.0 does not have default namespace concept.)

The links, variables, partnerLinks, fault handlers, compensation handlers, etc. that are visible to a Query/Expression language are defined based on the visibility of those entities to the activity that the enclosing element is contained within. There is no requirement that a Query/Expression language must manifest all the different objects previously described, only that if such objects are accessible within the Query/Expression language then only the objects in scope to the enclosing element's enclosing activity SHOULD be visible from within the Query/Expression language.

Evaluation of a BPEL expression or query will yield one of the following: (here we use XPath 1.0 expressions as examples)

(1) a single XML infoset item: e.g. `$myFooVar/lines/line[2]`
(2) a collection of XML infoset items e.g. `$myFooVar/lines`
(3) a sequence of Character Information Items for simple type data
    e.g. `$myFooVar/lines/line[2]/text()`
    (Please note this sequence of items may be manifested as a non XML infoset item when needed. e.g. boolean)
(4) a variable reference: e.g. `<from>$myFooVar</from> <to>$myBarVar</to>`

9.2.2 Binding BPEL Variables In XPath 1.0

With the exception of Link expressions whose variable access syntax and semantics are described in section 9.2.4, BPEL variables are accessible in XPath expressions in BPEL processes via XPath 1.0 variable bindings. Specifically, all BPEL variables visible from the enclosing element of an XPath 1.0 expression MUST be made available to the XPath 1.0 processor by manifesting the BPEL variable as an XPath variable binding whose name is the same as the BPEL variable's name, except the case of WSDL message variables declared with a messageType, which requires some special handling (discussed below).

BPEL variables declared using an element are manifested as a node-set XPath variable with a single member node. That node is a synthetic DII that contains a single child, the document element, which is the value of the BPEL variable. The XPath variable binding will bind to the document element. For example, given the following schema definition:

```
<xsd:element name="StatusContainer">
   <xsd:complexType>
      <xsd:sequence>
```

```
         <xsd:element name="statusDescription" type="xs:string"
form="qualified" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

And given the following variable declaration:

```
<variable name="AStatus" element="e:StatusContainer" />
```

Then a BPEL XPath expression to access the value of the statusDescription element, assuming the AStatus variable is in scope, would look like:

```
$Astatus/e:statusDescription
```

$AStatus points at the variable's document element, StatusContainer. So to access StatusMessage's child statusDescription it is only necessary to specify the child's element name.

BPEL variables declared using a complex type are manifested as a node-set XPath variable with one member node that contains the anonymous document element that itself contains the actual value of the BPEL complex type variable. The XPath variable binding will bind to the document element. For example, given the following schema definition:

```
<xs:complexType name="AuctionResults">
   <xs:sequence>
      <xs:element name="AuctionResult" maxOccurs="unbounded"
form="qualified">
         <xs:complexType>
            <xs:attribute name="AuctionID" type="xs:int"/>
            <xs:attribute name="Result" type="xs:string"/>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
```

And given the following variable declaration:

```
<variable name="AuctionResults" type="e:AuctionResults" />
```

Then a BPEL XPath expression to access the value of the second AuctionID attribute would look like:

```
$AuctionResults/e:AuctionResult[2]/@AuctionID
```

$AunctionResults points at the variable’s document element, AuctionResult[2] points to the second AuctionResult child of the document element, and @AuctionID points to the AuctionID attribute on the selected AuctionResult element.

WS-BPEL 2.0 messageType variables are manifested in XPath 1.0 as a series of variables, one variable per part in the messageType. Each variable is named by concatenating the message variable's name, the "." character and the local name of the part. The data in a BPEL messageType variable is NOT made available as one single XPath variable to general XPath processing under the default query and expression language binding. For example, if a messageType variable was named "myMessageTypeVar" and it contained two parts, "msgPart1" and "msgPart2" then XPath 1.0 binding that had "myMessageTypeVar" in scope would manifest two XPath 1.0 variables, $myMessageTypeVar.msgPart1 and $myMessageTypeVar.msgPart2. Please note that section 2.3 of [WSDL 1.1] requires that all part names within the same WSDL message definition must be unique.

WSDL message parts are always defined using either an XSD element, an XSD complex type or a XSD simple type. As such the manifestation of these message parts in XPath can be handled in the same manner as specified herein for element, complex type and simple type WS-BPEL 2.0 variables.

Below is a full example of how a WSDL message type is manifested in WS-BPEL 2.0 XPath.

```
<message name="StatusMessage">
   <part name="StatusPart1" element="e:StatusContainer"/>
   <part name="StatusPart2" element="e:StatusContainer"/>
</message>
```

And given the following variable declaration:

```
<variable name="StatusVariable" messageType="e:StatusMessage" />
```

Then a BPEL XPath expression to access the second part's statusDescription element would look like:

```
$StatusVariable.StatusPart2/e:statusDescription
```

Note: It is possible to write XPath 1.0 queries that can simultaneously query across multiple parts of a WSDL message variable by applying a union operator to create one single nodeset. For example:

```
( $StatusVariable.StatusPart1 | $StatusVariable.StatusPart2 )//e:amount
```

BPEL simpleType variables are manifested directly as either an XPath string, Boolean or float object. If the XML schema type of the BPEL simpleType variable is xs:Boolean or any types that are restrictions of xs:Boolean then the BPEL variable will be manifested as an XPath Boolean object. If the XML schema type of the BPEL simpleType variable is xs:float, xs:int, xs:unsignedInt or any restrictions of those types then the BPEL variable will be manifested as an XPath float object. Any other XML schema types will be manifested as an XPath string object.

XPath 1.0 only has a single numeric object, float. Float is a 64 bit IEEE floating point number. However the resolution of the float object is not sufficient to capture the full value of some XML Schema data types. For example, a XSD decimal number must support at least 18 digits where as an XPath float only has to support approximately 16 digits. XSD numeric values that cannot be expressed without loss of accuracy as XPath float objects are instead translated into XPath string objects. XPath string objects can be explicitly translated into XPath float objects using the XPath number() function. Similarly if an XPath string object is used in a situation that calls for an XPath float object the XPath processor will automatically translate the XPath string object to an XPath float object. But implementers should be aware that if the accuracy of the XML schema type is greater than supported by the XPath float object then accuracy will be lost during the translation from string object to float object.

9.2.3 XPath 1.0 perspective and WS-BPEL

Section 1 of the XPath 1.0 specification [XPATH 1.0] defines five points that define the context in which an XPath expression is evaluated. Those points are reproduced below:

- a node (the context node)
- a pair of non-zero positive integers (the context position and the context size)
- a set of variable bindings
- a function library
- the set of namespace declarations in scope for the expression

The following sections define how these contexts are initialized in BPEL for different types of BPEL expression and query language contexts.

9.2.4 Default use of XPath 1.0 for Expression and Query Languages

When XPath 1.0 is used for a Query or Expression Language, except as otherwise specified, the XPath context is initialized as follows:

- Context node = none
- Context position = none ; Context Size = none
- A set of variable bindings = variables visible to the Enclosing Element as defined by the BPEL scope rules
- A function library = core XPath and BPWS function libraries MUST be available and an engine specific function library MAY be available
- Namespace declaration = in-scope namespace declarations from Enclosing Element

It is worth emphasizing that as defined by the XPath 1.0 standard when resolving an XPath the namespace prefixes used inside of the variable (e.g. BPEL variables) are irrelevant. The only prefixes that matter are the in-scope namespaces.

For example, imagine a BPEL variable named “FooVar” of “foo” element type with value:

```
<a:foo xmlns:a="http://example.com"><a:bar >23</a:bar></a:foo>
```

The following XPath would return the value 23:

```
<from xmlns:b="http://example.com">$FooVar/b:bar/text()</from>
```

Notice that in the previous example the bar element is referred to use the 'b' namespace prefix rather than the 'a' namespace prefix that is used inside the actual value.

It is also worth emphasizing that XPath 1.0 explicitly requires that any element or attribute used in an XPath expression that does not have a namespace prefix must be treated as being namespace unqualified. That is, even if there is a default namespace defined on the enclosing element, the default namespace will not be applied.

Using the same value for Foo as provided previously the following would return a bpws:selectionFailure fault (in executable BPEL), because it fails to select any node:

```
<bpws:from xmlns="http://example.com">$FooVar/bar/text()</bpws:from>
```

The values inside of the XPath do not inherit the default namespace of the enclosing element. So the 'bar' element referenced in the XPath does not have any namespace value what so ever and therefore does not match with the bar element in the FooVar variable which has a namespace value of http://example.com.

Allowing BPEL variables to manifest as XPath variable bindings enables BPEL programmers to create powerful XPath expressions involving multiple BPEL variables. For example:

```
<assign>
   <copy>
      <from>$po/lineItem[@prodCode=$myProd]/amt * $exchangeRate</from>
      <to>$convertedPO/lineItem[@prodCode=$myProd]/amt</to>
   </copy>
</assign>
```

When XPath 1.0 is used as an expression or query language in BPEL, with the exception of propertyAlias definitions, there is no context node available. Therefore the legal values of the XPath Expr (http://www.w3.org/TR/xpath#NT-Expr) production must be restricted in order to prevent access to the context node.

Specifically, the "LocationPath" (http://www.w3.org/TR/xpath#NT-LocationPath) production rule of "PathExpr" (http://www.w3.org/TR/xpath#NT-PathExpr) production rule MUST NOT be used when XPath is used as an expression or query language (except in the case of propertyAlias which is covered separately). The previous restrictions on the XPath Expr production for the use of XPath as an expression language MUST be statically enforced.

The result of this restriction is that the "PathExpr" will always start with a "PrimaryExpr" (http://www.w3.org/TR/xpath#NT-PrimaryExpr) for BPEL expression or query language XPaths. It is worth remembering that PrimaryExprs are either variable references,

expressions, literals, numbers or function calls, none of which can access the context node.

A query language is used when an lvalue needs to be returned. In addition to the previously listed restrictions, which except where otherwise indicated MUST apply to XPaths used as a query language, additional restrictions are also needed to ensure that the returned value of a query language expression is an lvalue. Specifically, when XPath is used as a query language the XPath MUST begin with a "VariableReference" and this restriction MUST be statically enforced.

9.2.5 Use of XPath 1.0 for Expression Languages in Join Conditions

When XPath 1.0 is used as an Expression Language for a join condition, the XPath context is initialized as follows:

- Context node = none
- Context position = none ; Context Size = none
- A set of variable bindings = links that the target the activity that the Enclosing Element is contained within
- A function library = the core XPath function library MUST be available, the BPWS function library MUST NOT be available and an engine specific function library MAY be available.
- Namespace declaration = in-scope namespace declarations from Enclosing Element

As explained in section 12.5.1 expressions in join conditions may only access the status of links that target the join condition's enclosing activity. No other data may be made available. To this end the only variable bindings made available to join conditions are ones that access link status. Also note that link status is only available in join conditions which is why links are not bound to XPath variables in any other context.

Link status is made available via XPath variable bindings by manifesting links that target the activity that contains the Enclosing Element as XPath variable bindings of identical name. That is, if there is a link called "ABC" that targets the activity then there must be an XPath variable binding called "ABC". Link variables are manifested as XPath Boolean objects whose value will be set to the Link's value.

Below is an example of a joinCondition inside of a <targets> element:

```
<targets>
   <target linkName="link1"/>
   <target linkName="link2"/>
   <joinCondition>$link1 and $link2</joinCondition>
</targets>
```

9.2.6 Use of XPath 1.0 for Query Languages in propertyAliases

When XPath 1.0 is used as Query Language for a propertyAlias, the XPath context is initialized as follows:

- Context node =
  - When messageType/part attributes are used:
    - If the message part is based on a complex type or an element, the context node will point to node-list containing a single node which is the EII for the referenced part where the EII is defined as specified in section 9.2.2.
    - If the message part is based on a simple type, the context node will point to the XPath object specified for the particular variable type in section 9.2.2.
  - When type attribute is used:
    - If the type is a complex type, the context node will point to node-list containing a single node which is the EII for the referenced part where the EII is defined as specified in section 9.2.2.
    - If the type is a simple type, the context node will point to the XPath object specified for the particular variable type in section 9.2.2.
  - When element attribute is used, the context node will point to node-list containing a single node which is the EII for the referenced part where the EII is defined as specified in section 9.2.2.
- Context position = 1 ; Context Size = 1
- A set of variable bindings = There MUST NOT be any variable bindings available when XPath is used as query language in propertyAlias
- A function library = The core XPath function library MUST be available, the BPWS function library MUST NOT be available and an engine specific function library MAY be available.
- Namespace declaration = in-scope namespace declarations from Enclosing Element (note that in the past enclosing elements were always themselves enclosed within an activity, that is not the case here since propertyAliases are defined in WSDL files. But since there are no variable bindings to worry about this difference should not matter.)

propertyAlias is unique in BPEL in that it has a defined context node. Therefore none of the previously listed restrictions on the syntax of the XPath expression apply here. Any legal XPath expression may be used. It does not matter if an absolute or relative path is used in a propertyAlias as both will resolve to the context node since the context node in this case is the root node.

For example:

```
<propertyAlias propertyName="p:poId"
               messageType="my:POMsg" part="poPart">
    <query> po/id</query>
</propertyAlias>
```

Or

```
<propertyAlias propertyName="p:poId"
      messageType="my:POMsg" part="poPart">
    <query>(po/id + 1)</query>
</propertyAlias>
```

There is no requirement that property aliases return lvalues but it is worth pointing out that an attempt to assign to a property alias that doesn't specify a lvalue will, as defined in section 9.3, fail with a bpws:selectionFailure.

9.4. Assignment

Copying data from one variable to another is a common task within a business process. The `assign` activity can be used to copy data from one variable to another, as well as to construct and insert new data using expressions. The use of expressions is primarily motivated by the need to perform simple computation (such as incrementing sequence numbers) that is required for describing business protocol behavior. Expressions operate on message selections, properties, and literal constants to produce a new value for a variable property or selection. This activity can also be used to copy endpoint references to and from partner links. Finally, it is possible to include extensible data manipulation operations defined as extension elements under namespaces different from the WS-BPEL namespace.

The assign `assign` contains one or more elementary assignments.

```
<assign validate="yes|no"? standard-attributes>
       standard-elements
       (<copy>
          from-spec
          to-spec
        </copy> |
        <extensibleAssign>
          ...assign-element-of-other-namespace...
        </extensibleAssign>) +
</assign>
```

The assign activity allows copying a type-compatible value from the source ("from-spec") to the destination ("to-spec"), using the <copy> element. The `from-spec` MUST be one of the following forms except for the opaque form available in abstract processes:

```
<from variable="ncname" part="ncname"?/>
<from partnerLink="ncname" endpointReference="myRole|partnerRole"/>
<from variable="ncname" property="qname"/>
<from expressionLanguage="anyURI"?>expression</from>
<from> <literal> ... literal value ... </literal> </from>
```

The `to-spec` MUST be one of the following forms:

```
<to variable="ncname" part="ncname"?/>
<to partnerLink="ncname"/>
<to variable="ncname" property="qname"/>
<to queryLanguage="anyURI"?>… query … </to>
```

To-specs MUST identify Lvalues so that assignment is possible. A Lvalue, in the context of XPATH 1.0, is a node-list containing a single node from the value store, e.g. variable, partnerLink, property, etc., identified by the to-spec that is to be replaced by the assignment. If a to-spec does not identify a Lvalue then a bpws:selectionFailure MUST be thrown.

In the first `from-spec` and `to-spec` variants the variable attribute provides the name of a variable. If the type of the variable is a WSDL messge type the optional part attribute MAY be used to provide the name of a part within that variable. When the variable is defined using XML Schema types (simple or complex)or element, the part attribute MUST NOT be used.

The second `from-spec` and `to-spec` variants allow dynamic manipulation of the endpoint references associated with partner links. The value of the partnerLink attribute is the name of a partnerLink declared in the process. In the case of from-specs, the role must also be specified because a process might need to communicate an endpoint reference corresponding to either its own role or the partner's role within the partnerLink. The value “myRole” means that the endpoint reference of the process with respect to that partnerLink is the source, while the value “partnerRole” means that the partner’s endpoint reference for the partnerLink is the source. For the to-spec, the assignment is only possible to the partnerRole, hence there is no need to specify the role. The type of the value used in partnerLink-style from/to-specs is always an endpoint reference (see **Partner Link Types**, **Partner Links**, and **Endpoint References**).

The third `from-spec` and `to-spec` variants allow explicit manipulation of varialbe properties (see Varialbe Properties) occurring in variables. The property value generated by the from-spec is generated in the same manner and with the same faults as the value returned by the getVariableProperty() function. The property forms are especially useful for abstract processes, because they provide a way to clearly define how distinguished data elements in messages are being used.

In the fourth (“expression”) from-spec variant, an expression language, identified by the optional expression language attribute, is used to return a value. In the fourth to-spec (“query”) variant, a query language, identified by the optional query language attribute, is used to return a value. Both from-spec and to-spec allow processes to perform simple computations on properties and variables (for example, increment a sequence number). This computed value MUST be one of the followings:

- a single XML information item other than a character information item (CharII): examples are element information item (EII) and attribute information item (AII)

- a sequence of one or more character information items: this is mapped to a Text Node in the data model of XPath 1.0

In the case of to-spec, the computed value MUST be a lvalue.

Please note that it is possible to use either the first form of from-spec/to-spec or the fourth form of from-spec/to-spec to perform copy on non-message-variables and parts of message variables, as this specification defines how to manifest non-message variables and parts of message variables as XML Infoset information items. However, only the first form of from-spec/to-spec is able to copy an entire message variable including all of its parts. Other from and to-spec forms are only able to refer to a single part in a WSDL message type variable and so cannot copy all of the parts at once.

If the execution of the query expression yields an unhandled query language fault, the WS-BPEL standard fault bpws:subLanguageExecutionFault MUST be thrown.

The fifth `from-spec` variant allows a literal value to be given as the source value to assign to a destination. The type of the literal value MUST be the type of the destination (to-spec). The type of the literal value MAY be optionally indicated inline with the value by using XML Schema's instance type mechanism (`xsi:type`).

The fifth from-spec variant allows a literal value to be given as the source value to assign to a destination. The type of the literal value MUST be the type as the destination (to-spec). The literal value to be assigned is included within a <literal> element in order prevent conflicts with standard extensibility elements under <from>; note that the <literal> element itself does not allow standard extensibility. The type of the literal value MAY be optionally indicated inline with the value by using XML Schema's instance type mechanism (xsi:type).

The fifth from-spec variant also allows an endpoint reference literal value to be assign to a partnerLink directly, when used with the second variant of the to-spec. (see **Partner Link Types**, **Partner Links**, and **Endpoint References**).

In addition to <copy> specifications, other extensibility data manipulation elements MAY be included in an assign activity, inside an <extensibleAssign> element. The extensibility data manipulation elements MUST belong to a namespace different from the WS-BPEL namespace.

An optional `validate` attribute can be used with the `assign` activity. When `validate` is set to "yes", it can be seen as a convenient marco of a combination of `assign` and `validate` activities. With `validate="yes"`, the `assign` activity will validate all the variables being modified by the activity. E.g. variables being referenced in the `to-specs`.

The logic related to the `validate` attribute and assignment logic is considered as one unit of operation. If the "validate" parts of the operation fail, the whole assignment operation is considered as a failure also. When one of the variables is invalid against its

correponding XML definition, a standard fault of "bpws:invalidVariables" fault MUST be thrown. Please note that the default of the validate attribute is "no".

A BPEL implementation MAY provide a mechanism to turn on/off any explicit validation. E.g. validate attribute at assign.

9.4.1. Type Compatibility in Assignment

For an assignment to be valid, the data referred to by the from and to specifications MUST be of compatible types. The following points make this precise:

- The from-spec is a variable of a WSDL message type and the to-spec is a variable of a WSDL message type. In this case both variables MUST be of the same message type, where two message types are said to be equal if their qualified names are the same.
- The from-spec is a variable of a WSDL message type and the to-spec is not, or vice versa. This is not legal because parts of variables, selections of variable parts, or endpoint references cannot be assigned to/from variables of WSDL message types directly.
- In all other cases, the types of the source and destination are XML Schema types or elements, and the constraint is that the source value MUST possess the element or type associated with the destination. Note that this does not require the types associated with the source and destination to be the same. In particular, the source type MAY be a subtype of the destination type. In the case of variables defined by reference to an element, moreover, both the source and the target MUST be the same element.

The semantics of a process in which any of the matching constraints above is violated is undefined.

9.4.2. Assignment Example

The example assumes the following complex type definition in the namespace "http://tempuri.org/bpws/example":

```
<complexType name="tAddress">
        <sequence>
            <element name="number" type="xsd:int"/>
            <element name="street" type="xsd:string"/>
            <element name="city" type="xsd:string"/>
            <element name="phone">
                <complexType>
                    <sequence>
                        <element name="areacode" type="xsd:int"/>
                        <element name="exchange" type="xsd:int"/>
                        <element name="number" type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
```

```
        </sequence>
</complexType>

<element name = “address” type = “tAddress”/>
```

Assume that the following WSDL message definition exists for the same target namespace:

```
<message name="person" xmlns:x="http://tempuri.org/bpws/example">
     <part name="full-name" type="xsd:string"/>
     <part name="address" element="x:address"/>
</message>
```

Also assume the following WS-BPEL variable declarations:

```
<variable name="c1" messageType="x:person"/>
<variable name="c2" messageType="x:person"/>
<variable name="c3" element="x:address"/>
```

The example illustrates copying one variable to another as well as copying a variable part to a variable of compatible element type:

```
<assign>
     <copy>
          <from variable="c1"/>
          <to variable="c2"/>
     </copy>
     <copy>
         <from>$c1.address</from>
         <to variable="c3"/>
     </copy>
</assign>
```

10. Correlation

The information provided so far suggests that the target for messages that are delivered to a business process service is the WSDL port of the recipient service. This is an illusion because, by their very nature, stateful business processes are `instantiated` to act in accordance with the history of an extended interaction. Therefore, messages sent to such processes need to be delivered not only to the correct destination port, but also to the correct `instance` of the business process that provides the port. The infrastructure hosting the process must do this in a generic manner, to avoid burdening every process implementation with the need to implement a custom mechanism for instance routing. Messages, which create a new business process instance, are a special case, as described in **The Lifecycle of a Business Process**.

In the object-oriented world, such stateful interactions are mediated by object references, which intrinsically provide the ability to reach a specific object (instance) with the right state and history for the interaction. This works reasonably well in tightly coupled

implementations where a dependency on the structure of the implementation is normal. In the loosely coupled world of Web Services, the use of such references would create a fragile web of implementation dependencies that would not survive the independent evolution of business process implementation details at each business partner. In this world, the answer is to rely on the business data and communication protocol headers that define the wirelevel contract between partners and to avoid the use of implementation-specific tokens for instance routing whenever possible.

Consider the usual supply-chain situation where a buyer sends a purchase order to a seller. Suppose that the buyer and seller have a stable business relationship and are statically configured to send documents related to the purchasing interaction to the URLs associated with the relevant WSDL service ports. The seller needs to asynchronously return an acknowledgement for the order, and the acknowledgement must be routed to the correct business process instance at the buyer. The obvious and standard mechanism to do this is to carry a business token in the order message (such as a purchase order number) that is copied into the acknowledgement for correlation. The token can be in the message envelope in a header or in the business document (purchase order) itself. In either case, the exact location and type of the token in the relevant messages is fixed and instance independent. Only the value of the token is instance dependent. Therefore, the structure and position of the correlation tokens in each message can be expressed declaratively in the business process description. The WS-BPEL notion of correlation set, described in the following section, provides this feature. The declarative information allows a WS-BPEL-compliant infrastructure to use correlation tokens to provide instance routing automatically.

The declarative specification of correlation relies on declarative properties of messages. A property is simply a "field" within a message identified by a query—by default the query language is XPath 1.0. This is only possible when the type of the message part or binding element is described by using an XML Schema. The use of correlation tokens and endpoint references is restricted to message parts described in this way. To be clear, the actual wire format of such types can still be non-XML, for example, EDI flat files, based on different bindings for port types.

10.1. Message Correlation

During its lifetime, a business process instance typically holds one or more conversations with partners involved in its work. Conversations may be based on sophisticated transport infrastructure that correlates the messages involved in a conversation by using some form of conversation identity and routes them automatically to the correct service instance without the need for any annotation within the business process. However, in many cases correlated conversations involve more than two parties or use lightweight transport infrastructure with correlation tokens embedded directly in the application data being exchanged. In such cases, it is often necessary to provide additional application-level mechanisms to match messages and conversations with the business process instances for which they are intended.

Correlation patterns can become quite complex. The use of a particular set of correlation tokens does not, in general, span the entire interaction between a service instance and a partner (instance), but spans a part of the interaction. Correlated exchanges may nest and overlap, and messages may carry several sets of correlation tokens. For example, a buyer might start a correlated exchange with a seller by sending a purchase order (PO) and using a PO number embedded in the PO document as the correlation token. The PO number is used in the PO acknowledgement by the seller. The seller might later send an invoice that carries the PO number, to correlate it with the PO, and also carries an invoice number so that future payment-related messages need to carry only the invoice number as the correlation token. The invoice message thus carries two separate correlation tokens and participates in two overlapping correlated exchanges.

WS-BPEL addresses correlation scenarios by providing a declarative mechanism to specify correlated groups of operations within a service instance. A set of correlation tokens is defined as a set of properties shared by all messages in the correlated group. Such a set of properties is called a `correlation` set.

Correlation sets are declared within scopes and associated with them in a manner that is analogous to variable declarations. Each correlation set is declared within a scope and is said to belong to that scope. Correlation sets that belong to the global process scope are called global correlation sets. Correlation sets may also belong to other, non-global scopes, and such correlation sets are called local correlation sets. Each correlation set is only visible in the scope in which it is defined and in all scopes nested within the scope it belongs to. Thus, global correlation sets are visible throughout the process. The name of a correlation set MUST be unique amongst the names of all correlation sets defined within the same immediately enclosing scope. This requirement MUST be statically enforced. It is possible to "hide" a correlation set in an outer scope by declaring a correlation set with an identical name in an inner scope.

A global correlation set is in an uninitiated state at the beginning of a process. A local correlation set is in an uninitiated state at the start of the scope it belongs to. Note that non-global scopes in general start and complete their behavior more than once in the lifetime of the process instance they belong to.

Correlation sets resemble late-bound constants rather than variables in their semantics. The binding of a correlation set is triggered by a specially marked message send or receive operation. A correlation set can be initiated only once during the lifetime of the scope it belongs to. Thus, a global correlation set can only be initiated at most once during the lifetime of the process instance. Its value, once initiated, can be thought of as an alias for the identity of the business process instance. A local correlation set is available for binding each time the corresponding scope starts, but once initiated must retain its value until the scope completes.

In multiparty business protocols, each participant process in a correlated message exchange acts either as the initiator or as a follower of the exchange. The initiator process sends the first message (as part of an operation invocation) that starts the conversation,

and therefore defines the values of the properties in the correlation set that tag the conversation. All other participants are followers that bind their correlation sets in the conversation by receiving an incoming message that provides the values of the properties in the correlation set. Both initiator and followers must mark the first activity in their respective groups as the activity that binds the correlation set.

10.2. Defining and Using Correlation Sets

The examples in this section show correlation being used on almost every messaging activity (receive, reply, and invoke). This is because WS-BPEL does not assume the use of any sophisticated conversational transport protocols for messaging. In cases where such protocols are used, the explicit use of correlation in WS-BPEL can be reduced to those activities that establish the conversational connections.

Each correlation set in WS-BPEL is a named group of properties that, taken together, serve to define a way of identifying an application-level conversation within a business protocol instance. A given message can carry multiple correlation sets. After a correlation set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set and occur within the corresponding scope until its completion. This **correlation consistency constraint** MUST be observed in all cases of `initiate` values. The legal values of the `initiate` attribute are: `"yes"`, `"join"`, `"no"`. The default value of the `initiate` attribute is "no".

- When the `initiate` attribute is set to `"yes"`, the related activity MUST attempt to initiate the correlation set.
    - If the correlation set is already initiated and the `initiate` attribute is set to `"yes"`, the semantics is undefined.
- When the `initiate` attribute is set to `"join"`, the related activity MUST attempt to initiate the correlation set, if the correlation set is NOT initiated yet.
    - If the correlation set is already initiated and the `initiate` attribute is set to `"join"`, the correlation consistency constraint MUST be observed. If the constraint is violated, the semantics is undefined.
- When the `initiate` attribute is set to `"no"`, the related activity MUST NOT attempt to initiate the correlation set.
    - If an activity with the `"initiate"` attribute set to `"no"` attempts to use a correlation set that has not been previously initiated, the semantics is undefined.
    - If the correlation set is already initiated and the `initiate` attribute is set to `"no"`, the correlation consistency constraint MUST to be observed. If the constraint is violated, the semantics is undefined.

If multiple correlation sets are used in a message activity, then the consistency constraint MUST be observed for all correlation sets used. For example, the correlation sets used in an inbound message activity (e.g. receive) must all match message for that message to be delivered to the activity in the given process instance. If one of initiated correlation set does NOT match with the message, the semantics is undefined.

As the following examples illustrate, a correlation set is initiated when the activity within which it is used applies the attribute `initiate="yes"` to the set.

```
<correlationSets>?
        <correlationSet name="ncname" properties="qname-list"/>+
</correlationSets>
```

Following is an extended example of correlation. It begins by defining four message properties: `customerID, orderNumber, vendorID` and `invoiceNumber`. All of these properties are defined as part of the `"http://example.com/supplyCorrelation.wsdl"` namespace defined by the document.

```
<definitions name="properties"
        targetNamespace="http://example.com/supplyCorrelation.wsdl"
        xmlns:tns="http://example.com/supplyCorrelation.wsdl"

        xmlns="http://schemas.xmlsoap.org/wsdl/">

        <!-- define correlation properties -->

        <bpws:property name="customerID" type="xsd:string"/>
        <bpws:property name="orderNumber" type="xsd:int"/>
        <bpws:property name="vendorID" type="xsd:string"/>
        <bpws:property name="invoiceNumber" type="xsd:int"/>
</definitions>
```

Note that these properties are global names with known (simple) XMLSchema types. They are abstract in the sense that their occurrence in variables needs to be separately specified (see Variable Properties). The example continues by defining purchase order and invoice messages and by using the concept of aliasing to map the abstract properties to fields within the message data identified by selection.

```
<definitions name="correlatedMessages"
        targetNamespace="http://example.com/supplyMessages.wsdl"
        xmlns:tns="http://example.com/supplyMessages.wsdl"
        xmlns:cor=http://example.com/supplyCorrelation.wsdl
        xmlns:po = "http://example.com/po.xsd"
        xmlns="http://schemas.xmlsoap.org/wsdl/">

<!—define schema types for PO and invoice information -->
<types>
    <xsd:schema targetNamespace = "http://example.com/po.xsd">
        <xsd:complexType name="PurchaseOrder">
              <xsd:element name="CID" type="xsd:string"/>
              <xsd:element name="order" type="xsd:int"/>
                ...
        </xsd:complexType>
        <xsd:complexType name="PurchaseOrderResponse">
              <xsd:element name="CID" type="xsd:string"/>
              <xsd:element name="order" type="xsd:int"/>
              <xsd:element name="VID" type="xsd:string"/>
              <xsd:element name="invNum" type="xsd:int"/>
                ...
```

```
       </xsd:complexType>
       <xsd:complexType name="PurchaseOrderRejectType">
             <xsd:element name="CID" type="xsd:string"/>
             <xsd:element name="order" type="xsd:int"/>
            <xsd:element name="reason" type="xsd:string"/>
              ...
       </xsd:complexType>
       <xsd:complexType name="InvoiceType">
            <xsd:element name="VID" type="xsd:string"/>
            <xsd:element name="invNum" type="xsd:int"/>
       </xsd:complexType>
       <xsd:element name = "PurchaseOrderReject" type=
"po:PurchaseOrderRejectType">
       <xsd:element name = "Invoice" type= "po:invoiceType">
    </xsd:schema>

</types>
<message name="POMessage">
     <part name="PO" type="po:PurchaseOrder"/>
</message>
<message name="POResponse">
     <part name="RSP" type="po:PurchaseOrderResponse"/>
</message>
<message name="POReject">
     <part name="RJCT" element="po:PurchaseOrderReject"/>
</message>
<message name="InvMessage">
    <part name="IVC" element = "po:Invoice"/>
</message>
<bpws:propertyAlias propertyName="cor:customerID"
       messageType="tns:POMessage" part="PO">
       <bpws:query>
       CID
       </bpws:query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="cor:orderNumber"
       messageType="tns:POMessage" part="PO">
       <query>
       Order
       </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="cor:customerID"
       messageType="tns:POResponse" part="RSP">
       <bpws:query>
       CID
       </bpws:query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="cor:orderNumber"
       messageType="tns:POResponse" part="RSP">
       <query>
       Order
       </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="cor:vendorID"
       messageType="tns:POResponse" part="RSP">
       <query>
       VID
```

```
        </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="cor:invoiceNumber"
        messageType="tns:POResponse" part="RSP">
        <query>
        InvNum
        </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="cor:vendorID"
        messageType="tns:InvMessage" part="IVC">
        <query>
        VID
        </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="cor:invoiceNumber"
        messageType="tns:InvMessage" part="IVC">
        <query>
        InvNum
        </query>
</bpws:propertyAlias>
        ...
</definitions>
```

Finally, the portType used is defined, in a separate WSDL document.

```
<definitions name="purchasingPortType"
        targetNamespace="http://example.com/puchasing.wsdl"
        xmlns:smsg="http://example.com/supplyMessages.wsdl"
        xmlns="http://schemas.xmlsoap.org/wsdl/">

        <! - import supplyMessage.wsdl -- >

        <portType name="PurchasingPT">
            <operation name="SyncPurchase">
                <input message="smsg:POMessage"/>
                <output message="smsg:POResponse"/>
                <fault name="tns:RejectPO" message="smsg:POReject"/>
            </operation>
            <operation name="AsyncPurchase">
                <input message="smsg:POMessage"/>
            </operation>
        </portType>
        <portType name="BuyerPT">
            <operation name="AsyncPurchaseResponse">
                <input message="smsg:POResponse"/>
            </operation>
            <operation name="AsyncPurchaseReject">
                <input message="smsg:POReject"/>
            </operation>
        </portType>
</definitions>
```

Both the properties and their mapping to purchase order and invoice messages will be used in the following correlation examples.

```
<correlationSets
        xmlns:cor="http://example.com/supplyCorrelation.wsdl">

        <!-- Order numbers are particular to a customer,
                this set is carried in application data -->
        <correlationSet name="PurchaseOrder"
                properties="cor:customerID cor:orderNumber"/>
        <!-- Invoice numbers are particular to a vendor,
                this set is carried in application data -->
        <correlationSet name="Invoice"
                properties="cor:vendorID cor:invoiceNumber"/>
</correlationSets>
```

Correlation set names are used in invoke, receive, and reply activities (see *Invoking Web Service Operations* and *Providing Web Service Operations*), in the onMessage branches of pick activities, and in the onEvent variant of event handlers (see *Pick* and *Message Events*). These sets are used to indicate which correlation sets (i.e., the corresponding property sets) occur in the messages being sent and received. The `initiate` attribute is used to indicate whether the set is being initiated. When the attribute is set to "yes" the set is initiated with the values of the properties occurring in the message being sent or received. (Please see the beginning of this section for details of of `initiate` attribute.) Finally, in the case of invoke, when the operation invoked is synchronous request/response, a `pattern` attribute is used to indicate whether the correlation applies to the outbound (request) message, the inbound (response) message, or both. These ideas are explained in more detail in the context of the use of correlation in the rest of this example.

A message can carry the tokens of one or more correlation sets. The first example shows an interaction in which a purchase order is received in a one-way inbound request and a confirmation including an invoice is sent in the asynchronous response. The `PurchaseOrder` correlationSet is used in both activities so that the asynchronous response can be correlated to the request at the buyer. The receive activity initiates the `PurchaseOrder` correlationSet. The buyer is therefore the initiator and the receiving business process is a follower for this correlationSet. The `invoke` activity sending the asynchronous response also initiates a new correlationSet Invoice. The business process is the initiator of this correlated exchange and the buyer is a follower. The response message is thus a part of two separate conversations, and forms the bridge between them.

In the following, the prefix `SP:` represents the namespace `"http://example.com/puchasing.wsdl"`.

```
<receive partnerLink="Buyer" portType="SP:PurchasingPT"
        operation="AsyncPurchase"
        variable="PO">

        <correlations>
```

```
            <correlation set="PurchaseOrder" initiate="yes"/>
        </correlations>
</receive>

<invoke partnerLink="Buyer" portType="SP:BuyerPT"
        operation="AsyncPurchaseResponse" inputVariable="POResponse">

        <correlations>
            <correlation set="PurchaseOrder" initiate="no"
pattern="out"/>
            <correlation set="Invoice" initiate="yes" pattern="out"/>
        </correlations>
</invoke>
```

Alternatively, the response might have been a rejection (such as an "out-of-stock" message), which in this case terminates the conversation correlated by the correlationSet `PurchaseOrder` without starting a new one correlated with `Invoice`. Note that the `initiate` attribute is missing. It therefore has the default value of "no".

```
<invoke partnerLink="Buyer" portType="SP:BuyerPT"
        operation="AsyncPurchaseReject" inputVariable="POReject">

        <correlations>
            <correlation set="PurchaseOrder" pattern="out"/>
        </correlations>
</invoke>
```

The use of correlation with synchronous Web Service invocation is illustrated by the alternative synchronous purchasing operation used by an invoke activity used in the buyer's business process.

```
<invoke partnerLink="Seller" portType="SP:PurchasingPT"
        operation="SyncPurchase"
        inputVariable="sendPO"
        outputVariable="getResponse">

        <correlations>
            <correlation set="PurchaseOrder" initiate="yes"
pattern="out"/>
            <correlation set="Invoice" initiate="yes" pattern="in"/>
        </correlations>

        <catch faultName="SP:RejectPO" faultVariable="POReject"
                                       faultMessageType="smsg:POReject">
            <!-- handle the fault -->
        </catch>
</invoke>
```

Note that an `invoke` consists of two messages: an outgoing request message and an incoming reply message. The correlation sets applicable to each message must be separately considered because they can be different. In this case the `PurchaseOrder` correlation applies to the outgoing request that initiates it, while the `Invoice` correlation applies to the incoming reply and is initiated by the reply. Because the `PurchaseOrder`

correlation is initiated by an outgoing message, the buyer is the initiator of that correlation but a follower of the `Invoice` correlation because the values of the correlation properties for Invoice are initiated by the seller in the reply received by the buyer.

11. Basic Activities

11.1. Standard Attributes for Each Activity

Each activity has optional standard attributes: a name and an indicator whether a join fault should be suppressed if it occurs. See **Flow** for a full discussion of the suppressJoinFailure attribute. The name attribute is used to provide machine processable names for activities although currently the BPEL specification only makes programmatic use of the names of scope activities. The name of a named activity MUST be unique amongst all named activities present within the same immediately enclosing scope. This requirement MUST be statically enforced.

```
name="ncname"?
suppressJoinFailure="yes|no"?>
```

When the `suppressJoinFailure` attribute is not specified for an activity, it inherits its value from its closest enclosing activity or from the process if no enclosing activity specifies this attribute.

11.2. Standard Elements for Each Activity

Each WS-BPEL activity has nested standard elements <source> and <target> within the optional containers <sources> and <targets>. The use of these elements is required for establishing synchronization relationships through links (see **Flow**). Each link is defined independently and given a name. The link name is used as value of the linkName attribute of the <source> element. An activity MAY declare itself to be the source of one or more links by including one or more <source> elements. Each <source> element MUST use a distinct link name. Similarly, an activity MAY declare itself to be the target of one or more links by including one or more <target> elements. Each <source> element associated with a given activity MUST use a link name distinct from all other <source> elements at that activity. Each <target> element associated with a given activity MUST use a link name distinct from all other <target> elements at that activity. Each <source> element MAY optionally specify a transition condition that functions as a guard for following this specified link (see Flow). If the transition condition is omitted, it is deemed to be present with the constant value true.

```
<sources>?
    <source linkName="ncname">+
       <transitionCondition expressionLanguage="anyURI"?>?
        ... bool-expr ...
       </transitionCondition>
    </source>
</sources>
<targets>?
```

```
    <joinCondition expressionLanguage="anyURI"?>?
      ... bool-expr ...
    </joinCondition>
    <target linkName="ncname"/>+
</targets>?
```

The value of the <joinCondition> element is a Boolean-valued expression in the expression language indicated by the expressionLanguage attribute, or in the default expression language for this process (see Expressions). If no join condition is specified, the join condition is the logical OR of the link status of all incoming links of this activity (see 12.5.1 Link semantics).

11.3. Invoking Web Service Operations

Web Services provided by partners (see **Partner Link Types, Partner Links, and Endpoint References**) can be used to perform work in a WS-BPEL business process. Invoking an operation on such a service is a basic activity. Recall that such an operation can be a synchronous request/response or an asynchronous one-way operation. WS-BPEL uses the same basic syntax for both with some additional options for the synchronous case.

An asynchronous invocation requires only the input variable of the operation because it does not expect a response as part of the operation (see **Providing Web Service Operations**). A synchronous invocation requires both an input variable and an output variable. Note however that if a WSDL message definition does not contain any parts then the associated variable attribute, inputVariable or outputVariable, MAY be omitted. One or more correlation sets can be specified to correlate the business process instance with a stateful service at the partner's side (see **Correlation**). However, these attributes are both syntactically optional since they are absolutely required only in executable processes. Note however that if a WSDL message definition does not contain any parts then the variable attribute MAY be omitted.

In the case of a synchronous invocation, the operation might return a WSDL fault message. This results in a WS-BPEL fault. Such a fault can be caught locally by the activity, and in this case the specified activity will be performed. If a fault is not caught locally by the activity it is thrown to the scope that encloses the activity (see **Scopes** and **Fault Handlers**).

Note that a WSDL fault is identified in WS-BPEL by a qualified name formed by the target namespace of the corresponding portType and the fault name. This uniform naming mechanism must be followed even though it does not accurately match WSDL's faultnaming model. WSDL 1.1 does not require that fault names be unique within the namespace where the service operation is defined. In WSDL 1.1 it is necessary to specify a portType name, an operation name, and the fault name to uniquely identify a fault. This limits the ability to use fault-handling mechanisms to deal with invocation faults.

BPEL faults are defined exclusively in terms of a fault name and optional fault data. This means, for example, that if a fault is generated from a messaging activity (as opposed to the throw activity or a system fault) no record will be made in the fault of the portType or operation the message activity was using when the fault was received. A consequence of this model is that all faults sharing a common name, defined in the same namespace and sharing the same data type (or lack there of) are indistinguishable in WS-BPEL. Faults of a particular fault name MAY be associated with multiple variable types and the <catch> construct in WS-BPEL facilitates differentiation of faults with different message / variable types of the same fault name. For details regarding the <faultHandlers> and <catch>, please see section "Fault Handlers".

Implementer's Note: WS-BPEL treats faults based on abstract WSDL 1.1 operation definitions, without reference to binding details. Normally, when sending or receiving a fault, a WS-BPEL process only deals with the fault information in the abstract fault message and a WSDL 1.1 binding is required to transform the abstract fault message data to or from specific communication media. In the case of SOAP bindings this would mean providing transformation between abstract fault message data and the sub elements of of the SOAP Fault element, namely the faultcode, faultstring, faultactor and detail elements. However the WSDL 1.1 standard SOAP binding explicitly precludes mapping any information from an abstract fault message to a SOAP Fault other than the contents of the detail element. In other words, there is no standard way to relate the faultcode, faultstring and faultactor sub-elements of a SOAP Fault element to data visible to a WS-BPEL process. This specification does not provide a resolution for this problem.

Finally, an activity can be associated with another activity that acts as its compensation action. This compensation handler can be invoked either explicitly or by the default compensation handler of the enclosing scope (see **Scopes** and **Compensation Handlers**).

Semantically, the specification of local fault and/or compensation handlers is equivalent to the presence of an implicit <scope> activity immediately enclosing the <invoke> activity providing those handlers. The implicit <scope> activity assumes the name of the <invoke> activity it encloses, its suppressJoinFailure attribute, its join condition, as well as its link sources and targets. For example, the following:

```
<invoke name="purchase" suppressJoinFailure="yes"  partnerLink="Seller"
      portType="SP:Purchasing" operation="SyncPurchase" inputVariable="sendPO"
      outputVariable="getResponse">
     <targets>
        <target linkName="linkA"/>
     </targets>
     <sources>
        <source linkName="linkB">
          <transitionCondition>
                ...
          </transitionCondition>
        </source>
```

```
    </sources>
    <compensationHandler>
        <invoke partnerLink="Seller" portType="SP:Purchasing"
            operation="CancelPurchase" inputVariable="getResponse"
            outputVariable="getConfirmation">
    </compensationHandler>
</invoke>
```

is equivalent to:

```
<scope name"purchase" suppressJoinFailure="yes" >
    <targets>
      <target linkName="linkA"/>
    </targets>
    <sources>
      <source linkName="linkB">
        <transitionCondition>
           ...
        </transitionCondition>
       </source>
    </sources>
    <compensationHandler>
     <invoke partnerLink="Seller" portType="SP:Purchasing"
        operation="CancelPurchase" inputVariable="getResponse"
       outputVariable="getConfirmation">
    </compensationHandler>

    <invoke name="purchase" suppressJoinFailure="no" partnerLink="Seller"
        portType="SP:Purchasing" operation="SyncPurchase"
        inputVariable="sendPO" outputVariable="getResponse"/>
</scope>
```

The syntax of the invoke activity is summarized below.

```
<invoke partnerLink="ncname" portType="qname"? operation="ncname"
        inputVariable="ncname"? outputVariable="ncname"?
        standard-attributes>

standard-elements
<correlations>?
     <correlation set="ncname" initiate="yes|no"?
       pattern="in|out|out-in"/>+
</correlations>
<catch faultName="qname"? faultVariable="ncname"?
                        faultMessageType="qname"?
                        faultElement="qname"?>*
        activity
</catch>
<catchAll>?
```

```
        activity
</catchAll>
<compensationHandler>?
      activity
</compensationHandler>
<toPart part="ncname" fromVariable="ncname"/>*
<fromPart part="ncname" toVariable="ncname"/>*
</invoke>
```

See **Correlation** for an explanation of the correlation semantics. The following example shows an invocation with a nested compensation handler. Other examples are shown throughout the specification.

```
<invoke partnerLink="Seller" portType="SP:Purchasing"
        operation="SyncPurchase"
        inputVariable="sendPO"
        outputVariable="getResponse">
     <compensationHandler>
        <invoke partnerLink="Seller" portType="SP:Purchasing"
             operation="CancelPurchase"
            inputVariable="getResponse"
            outputVariable="getConfirmation">
     </compensationHandler>
</invoke>
```

If the WSDL operation used to send and/or receive a message in an invoke activity is defined as a message containing exactly one part which itself is defined using an element then a BPEL variable of the same element type as used to define the part MAY be submitted directly to the invoke activity. The result of submitting a BPEL variable in the previously defined circumstance MUST be the equivalent of declaring an anonymous temporary WSDL message variable based on the associated WSDL message type. In the case of an inputVariable the value of the submitted BPEL variable will be used to set the value of the part in the anonymous temporary WSDL message variable. In the case of an outputVariable the value of the received part in the temporary WSDL message variable will be used to set the value of the submitted BPEL variable.

<toPart> elements provide another short cut that makes it easier to create outgoing multi-part WSDL messages from the contents of BPEL variables. The inputVariable attribute MUST NOT be used on an Invoke activity that also contains a <toPart> element. The <toPart> elements, as a group, act as a single virtual assign where each <toPart> is turned into a copy in the virtual assign. The destination of all the copies is an anonymous temporary WSDL variable, of the type specified by the relevant WSDL operation, which will then be used to send the actual message created by the virtual assign. Each <toPart> is turned into a copy in which data from the variable indicated in the “fromVariable” attribute is copied into the part of the anonymous temporary WSDL variable referenced in the “part” attribute of the <toPart> element.

The virtual assign MUST follow the same semantics and use the same faults as a real assign. When this mechanism is used in an Invoke, it is not required that there be a <toPart> for every part in the WSDL message definition, nor is the order in which parts

are specified relevant. Parts not explicitly represented by <toPart> elements result in empty parts in the target anonymous WSDL variable used by the <invoke> activity.

<fromPart> elements are a similar short cut to <toPart> elements, but in the inverse. <fromPart> elements are used to pull data out of an incoming multi-part WSDL message and place it into individual BPEL variables. The outputVariable attribute MUST NOT be used on an Invoke activity that also contains a <fromPart> element. The <fromPart> elements, as a group, act as a single virtual assign. When a WSDL message is received on an <invoke> activity that uses <fromPart> the message is placed in an anonymous temporary WSDL variable, of the type specified by the relevant WSDL operation. The <fromPart> elements are then gathered together to form a single virtual assign where each <fromPart> is turned into a copy. Each <fromPart> is turned into a copy in which the data at the part of the anonymous temporary WSDL variable referenced in the <fromPart> is copied into the BPEL variable indicated in the “toVariable” attribute. The virtual assign MUST follow the same semantics and use the same faults as a real assign. When this mechanism is used in an Invoke, it is not required that there be a <fromPart> for every part in the WSDL message definition, nor is the order in which parts are specified relevant. Parts not explicitly represented by <fromPart> elements are simply not copied from the anonymous WSDL variable to a BPEL variables.

A single <invoke> can contain any combination of <toPart> elements, <fromPart> elements, outputVariable and inputVariable with the exception, as previously specified, that if <toPart> elements are used then the inputVariable attribute cannot be used and if <fromPart> elements are used then the outVariable attribute cannot be used. The virtual assign created as a consequence of the use of input or output elements occurs as part of the scope of the <invoke> activity and therefore any faults that are thrown can be caught by the invoke’s inline fault handler. Also note that it is legal to use <toPart>/<fromPart> with WSDL messages that only have a single part.

If an invoke activity is used on a partnerLink whose partnerRole endpoint reference is not initialized then a bpws:uninitializedPartnerRole fault MUST be thrown.

11.4. Providing Web Service Operations

A business process provides services to its partners through receive activities and corresponding reply activities. A receive activity specifies the partner link that contains the myRole used to receive messages, the port type (optional) and operation that it expects the partner to invoke. The value of the partnerRole in the partner link is not used when processing a receive activity. In addition, it may specify a variable, using the variable attribute, that is to be used to receive the message data received. However, this attribute is syntactically optional since it is absolutely required only in executable processes. An alternative to the variable attribute are <fromPart> elements. The syntax and semantics of the <fromPart> elements as used on the receive activity are the same as specified in section 11.3 for the invoke activity. This includes the restriction that if <fromPart> elements are used on a receive activity then the variable attribute MUST NOT be used on the same activity.

In addition, receive activities play a role in the lifecycle of a business process. The only way to instantiate a business process in WS-BPEL is to annotate a receive activity with the `createInstance` attribute set to "yes" (see **Pick** for a variant). The default value of this attribute is "no". A "start activity" is a `receive`/`pick` activity that is annotated with a `createInstance="yes"` attribute. NO other "non-start activities" but `scope`, `flow`, `sequence` or `empty` activities may potentially be performed prior to or simultaneously with a "start activity". The logical order of performing activities is determined by static analysis.

It is permissible to have multiple start activities. As specified in section 6.5, the initial start activity must complete execution before any other start activities are allowed to execute. In this case the intent is to express the possibility that any one of a set of required inbound messages can create the process instance because the order in which these messages arrive cannot be predicted. If a process has multiple start activities then all such activities MUST share at least one common correlation set and all correlation sets defined on all the activities MUST be set to "join" (see Correlation). Compliant implementations MUST ensure that only one of the inbound messages that will match to a single process instance actually instantiates the business process (usually the first one to arrive, but this is implementation dependent). The other incoming messages in the concurrent initial set MUST be delivered to the corresponding `receive` activities in the already created instance.

The following example is illegal, since `assign` activity cannot be a start activity:

```
<flow>
    <receive ... createInstance="yes" />
    <assign ... />
</flow>
```

The following example is legal, since the `assign` activity will not be performed prior to or simultaneously with the `receive` activity:

```
<flow>
    <links>
        <link name="RecvToAssign"/>
    </links>
    <sequence>
          <receive ... createInstance="yes">
                <sources> <source linkName="RecvToAssign" />  <sources>
          </receive>
          ...
    </sequence>
    <sequence>
          <assign>
                <targets> <target linkName="RecvToAssign" /> </targets>
                ...
          </assign>
    </sequence>
</flow>
```

If the WSDL operation used to receive or send a message in a request or a reply activity is defined as a message containing exactly one part which itself is defined using an element then a BPEL variable of the same element type as used to define the part MAY be submitted directly to the request/reply activity. The result of submitting a BPEL variable in the previously defined circumstance MUST be the equivalent of declaring an anonymous temporary WSDL message variable based on the associated WSDL message type. In the case of a receive activity, the incoming part's value will be used to set the value of the submitted BPEL variable. In the case of a reply activity the value of the submitted BPEL variable will be used to set the value of the part in the anonymous temporary WSDL message variable that is sent out. In the case of a reply that is sending a fault the same logic applies but using a fault name rather than an operation name. Aletrnatively, it is possible to use <fromPart> elements in a receive activity to indicate how the data from a received message is to be directly copied to BPEL variables from a corresponding anonymous WSDL message variable. Similarly, in the case of a reply activity, <toPart> elements may be used to have data from BPEL variables directly copied into an anonymous WSDL message used by the reply activity.

A business process instance MUST NOT simultaneously enable two or more `receive` activities for the same partnerLink, portType, operation and correlation set(s). Note that `receive` is a blocking activity in the sense that it will not complete until a matching message is received by the process instance. The semantics of a process in which two or more `receive` actions for the same partnerLink, portType, operation and correlation set(s) may be simultaneously enabled is undefined. For the purposes of this constraint, an `onMessage` clause in a `pick` and an onEvent event handler are equivalent to a `receive` (see **Pick** and **Message Events**).

It is worth pointing out that race conditions may occur in business process execution. Messages that target a particular process instance may arrive before the corresponding `receive` activity is started. For example, it is perfectly reasonable to model a process that receives a series of messages in a while loop where all the messages use the same correlation. At run time, the messages will arrive independent of the iterations of the loop. But the fact that the correlation is already initiated should enable the runtime engine and messaging platform to recognize that these messages are correlated to the process instance and handle those messages appropriately. For another example, a process may invoke a remote service, initiate a correlation set for an expected callback message, and the callback message may arrive before the corresponding `receive` activity is started for a variety of reasons. The correlation data in the arriving message should enable the engine to recognize the message is targeted for this process instance and handle it appropriately. Process engines may employ different mechanisms to handle such race conditions. This specification does not mandate any specific mechanism. For the purposes of handling race conditions, an `onMessage` clause in a `pick` and an `onMessage` event handler are equivalent to a receive (see **Pick** and **Message Events**).

```
<receive partnerLink="ncname" portType="qname"? operation="ncname"
        variable="ncname"? createInstance="yes|no"?
         messageExchange="ncname"?
```

```
        standard-attributes>

      standard-elements
      <correlations>?
        <correlation set="ncname" initiate="yes|no"?/>+
      </correlations>
      <fromPart part="ncname" toVariable="ncname"/>*
</receive>
```

A `reply` activity is used to send a response to a request previously accepted through a `receive` activity. Such responses are only meaningful for synchronous interactions. An asynchronous response is always sent by invoking the corresponding one-way operation on the partner link. A `reply` activity may specify a variable that contains the message data to be sent in reply. However, this attribute is syntactically optional since it is absolutely required only in executable processes. Note however that if a WSDL message definition does not contain any parts then the variable attribute MAY be omitted. An alternative to the variable attribute are <toPart> elements. The syntax and semantics of the <toPart> elements as used on the reply activity are the same as specified in section 11.3 for the invoke activity. This includes the restriction that if <toPart> elements are used on a receive activity then the variable attribute MUST NOT be used on the same activity.

The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner link for a particular portType, operation and correlation set(s) MUST NOT be outstanding simultaneously. The semantics of a process in which this constraint is violated is undefined. For the purposes of this constraint, an `onMessage` clause in a `pick` is equivalent to a `receive` (see **Pick**). Moreover, a reply activity must always be preceded by a `receive` activity for the same partner link, portType and (request/response) operation, such that no reply has been sent for that `receive` activity. The semantics of a process in which this constraint is violated is undefined.

```
<reply partnerLink="ncname" portType="qname"? operation="ncname"
        variable="ncname"? faultName="qname"?
        messageExchange="ncname"?
        standard-attributes>

      standard-elements
      <correlations>?
        <correlation set="ncname" initiate="yes|no"?/>+
      </correlations>
      <toPart part="ncname" fromVariable="ncname"/>*
</reply>
```

Note that the <reply> activity corresponding to a given request has two potential forms. If the response to the request is normal, the `faultName` attribute is not used and the `variable` attribute, when present, will indicate a variable of the normal response message type. If, on the other hand, the response indicates a fault, the `faultName` attribute is used and the `variable` attribute, when present, will indicate a variable of the

message type for the corresponding fault. Observe that WS-BPEL treats faults based on abstract WSDL 1.1 operation definitions, without reference to binding details. This limits te ability of a WS-BPEL process to determine the information transmitted when faults are returned over a SOAP binding. We refer to the Implementer's Note in Section 11.3 for additional details.

The optional messageExchange attribute is used to associate a <reply> activity with an inbound message activity, such as, <receive>, <onMessage> and <onEvent>, when there are multiple simultaneously incomplete inbound message activities which requires a reply message to complete.

A reply activity is associated with a inbound message activity based on the tuple - partnerLink, operation, and messageExchange. The value defined in messageExchange is scoped to the combination of partnerLink and operation. That is, it is legal to use the same messageExchange value in multiple simultaneously incomplete receive activities so long as the combination of partnerLink and operation on the receives are all different from each other. An incomplete inbound message activity describes the state of a BPEL process from the point that a request/response receive activity starts execution until its associated reply begins execution.

If there should ever be multiple simultaneous incomplete inbound message activities which have the same partnerLink, operation and messageExchange tuple then the `bpws:conflictingRequest` fault MUST be thrown within the BPEL process on the conflicting inbound message activities subsequent to the execution of the successful incomplete receive.

If a reply activity cannot be associated with an incomplete receive activity by matching the tuples and this error situation is not caught during static analysis, then `bpws:missingRequest` fault MUST be thrown within the BPEL process on the reply activity. Because conflicting requests should have been rejected at the time inbound message activity is executed, there cannot be more than one corresponding inbound message activity at the time <reply> is executed.

If the messageExchange attribute is not specified on a receive then its value is taken to be empty. For matching purposes two empty messageExchange values with the same partnerLink and operation values are said to match. Empty value does not match with other non-empty values.

11.5. Updating Variable Contents

Variable update occurs through the assignment activity, which is described in **Assignment**.

11.6. Signaling Faults

The `throw` activity can be used when a business process needs to signal an internal fault explicitly. Every fault is required to have a globally unique QName. The `throw` activity is required to provide such a name for the fault and can optionally provide a variable of data that provides further information about the fault. A fault handler can use such data to analyze and handle the fault and also to populate any fault messages that need to be sent to other services.

WS-BPEL does not require fault names to be defined prior to their use in a throw element. An application or process-specific fault name can be directly used by using an appropriate QName as the value of the faultName attribute and providing a variable with the fault data if required. This provides a very lightweight mechanism to introduce application-specific faults. The throw activity MAY be used to throw faults defined by this specification.

```
<throw faultName="qname" faultVariable="ncname"? standard-attributes>
       standard-elements
</throw>
```

A simple example of a throw activity that does not provide a variable of fault data is:

```
<throw xmlns:FLT="http://example.com/faults"
faultName="FLT:OutOfStock"/>
```

11.7. Waiting

The `wait` activity allows a business process to specify a delay for a certain period of time or until a certain deadline is reached (see **Expressions** for the grammar of duration expressions and deadline expressions).

```
<wait standard-attributes>
       standard-elements
       ( <for expressionLanguage="anyURI"?>duration-expr</for> |
         <until expressionLanguage="anyURI"?>deadline-expr</until> )

</wait>
```

A typical use of this activity is to invoke an operation at a certain time (in this case a constant, but more typically an expression dependent on process state):

```
<sequence>
     <wait><until>'2002-12-24T18:00+01:00'</until></wait>
     <invoke partnerLink="CallServer" portType="AutomaticPhoneCall"
        operation="TextToSpeech"
        inputVariable="seasonalGreeting">
     </invoke>
</sequence>
```

11.8. Doing Nothing

There is often a need to use an activity that does nothing, for example when a fault needs to be caught and suppressed. The `empty` activity is used for this purpose. The syntax is obvious and minimal.

```
<empty standard-attributes>
        standard-elements
</empty>
```

11.9 Extension Activity

New activities can be introduced for use in BPEL by placing them inside of the `extensionActivity` element. The contents of an `extensionActivity` element MUST be a single element that MUST make available BPEL's standard-attributes and standard-elements. If the element contained within the `extensionActivity` element is not recognized by the BPEL processor and is not subject to a `mustUnderstand="yes"` requirement from an extension declaration then the unknown activity MUST be treated as if it were an empty activity. In all cases however any standard-attributes or standard-elements used on the contained activity MUST be treated as defined by this specification.

```
<extensionActivity>
    <???? standard-attributes>
       standard-elements
    </????>
</extensionActivity>
```

12. Structured Activities

Structured activities prescribe the order in which a collection of activities take place. They describe how a business process is created by composing the basic activities it performs into structures that express the control patterns, data flow, handling of faults and external events, and coordination of message exchanges between process instances involved in a business protocol.

The structured activities of WS-BPEL include:

- Ordinary sequential control between activities is provided by `sequence`, `if`, and `while`.
- Concurrency and synchronization between activities is provided by `flow`.
- Nondeterministic choice based on external events is provided by `pick`.

The set of structured activities in WS-BPEL is not intended to be the minimal required set. There are cases where one activity can replace another. For example, the sequence activity, used to structure sequential processing, may be emulated by a properly configured flow with additional links. The purpose in providing what are, strictly

speaking, redundant activities is to make it easier for BPEL process designers to both read and write BPEL processes using familiar, even if functionally redundant, activity constructs.

Structured activities can be used recursively in the usual way. A key point to understand is that structured activities can be nested and combined in arbitrary ways. This provides a somewhat unusual but very attractive free blending of the graph-like and program-like control regimes that have traditionally been seen as alternatives rather than orthogonal composable features. A simple example of such blended usage is found in the **Initial Example**.

It is important to emphasize that the word `activity` is used throughout the following to include both basic and structured activities.

12.1. Sequence

A `sequence` activity contains one or more activities that are performed sequentially, in the order in which they are listed within the <sequence> element, that is, in lexical order. The `sequence` activity completes when the final activity in the sequence has completed.

```
<sequence standard-attributes>
       standard-elements
       activity+
</sequence>
```

Example:

```
<sequence>
        <flow>
            ...
        </flow>
        <scope>
             ...
        </scope>
        <pick>
            ...
        </pick>
</sequence>
```

12.2. If

The `if` structured activity supports conditional behavior in a pattern that occurs quite often. The activity consists of an ordered list of one or more conditional branches defined by `if, elseif` elements, followed optionally by an optional `else` branch. The `if and elseif` branches of the `if activity` are considered in the order in which they appear. The first branch whose condition holds true is taken and provides the activity performed for the if activity. If no branch with a condition is taken, then the `else` branch is taken if specified. If the `else` branch is not explicitly specified, then an `else` branch with an

`empty` activity is deemed to be present. The `if` activity is complete when the activity of the selected branch completes.

```
<if standard-attributes>
     standard-elements
     <condition expressionLanguage="anyURI"?>
          ... bool-expr ...
     </condition>
     <then>
          activity
     </then>
     <elseif>*
        <condition expressionLanguage="anyURI"?>
          ... bool-expr ...
        </condition>
        activity
     </elseif>
     <else>?
        activity
     </else>
</if>
```

Example:

```
<if xmlns:inventory="http://supply-chain.org/inventory"
        xmlns:FLT="http://example.com/faults">
     <condition>
        bpws:getVariableProperty('stockResult','level') > 100
     </condition>
     <then>
        <flow>
            <!-- perform fulfillment work -->
        </flow>
     </then>
     <elseif>
       <condition>
        bpws:getVariableProperty('stockResult','level') >= 0
       </condition>
        <throw faultName="FLT:OutOfStock"
             variable="RestockEstimate"/>
     </elseif>
     <else>
        <throw faultName="FLT:ItemDiscontinued"/>
    </else>
</if>
```

12.3. While

The `while` activity supports repeated performance of a specified iterative activity. The iterative activity is performed as long as the given Boolean while `condition` holds true at the beginning of each iteration.

```
<while standard-attributes>
       standard-elements
       <condition expressionLanguage="anyURI"?>
           ... bool-expr ...
       </condition>
       activity
</while>
```

Example:

```
      ...
<variable name="orderDetails" type="xsd:integer"/>
     ...
<while>
  <condition>
       $orderDetails > 100
  </condition>
      <scope>
            ...
      </scope>
</while>
```

12.4 RepeatUntil

The `repeatUntil` activity supports repeated performance of a specified iterative activity. The iterative activity will continue to be performed so long as after it executes the given Boolean `<repeatUntil>` condition holds true.

```
<repeatUntil standard-attributes>
       standard-elements
       activity
       <condition expressionLanguage="anyURI"?>
          ... bool-expr ...
       </condition>
</repeatUntil>
```

12.5. Pick

The `pick` activity awaits the occurrence of one of a set of events and then performs the activity associated with the event that occurred. The occurrence of the events is often mutually exclusive (the process will either receive an acceptance message or a rejection message, but not both). If more than one of the events occurs, then the selection of the activity to perform depends on which event occurred first. If the events occur almost simultaneously, there is a race and the choice of activity to be performed is dependent on both timing and implementation.

The form of `pick` is a set of branches of the form event/activity, and exactly one of the branches will be selected based on the occurrence of the event associated with it before any others. Note that after the pick activity has accepted an event for handling, the other events are no longer accepted by that `pick`. The possible events are the arrival of some

message in the form of the invocation of an inbound one-way or request/response operation, or an "alarm" based on a timer (in the sense of an alarm clock).

A special form of `pick` is used when the creation of an instance of the business process could occur as a result of receiving one of a set of possible messages. In this case, the `pick` itself has a `createInstance` attribute with a value of `yes` (the default value of the attribute is no). In such a case, the events in the `pick` must all be inbound messages and each of those is equivalent to a `receive` with the attribute `"createInstance=yes"`. No alarms are permitted for this special case.

Each pick activity MUST include at least one onMessage event.

The semantics of the onMessage event are identical to a receive activity regarding the optional nature of the variable attribute, the handling of race conditions, the single element-based part message short cut, the constraint regarding simultaneous enablement of conflicting receive actions and the optional use of <fromPart> elements. For the last case, recall that the semantics of a process in which two or more `receive` actions for the same partner link, portType, operation and correlation set(s) may be simultaneously enabled is undefined (see **Providing Web Service Operations**). Enablement of each `onMessage` handler is equivalent to enablement of the corresponding `receive` activity for the purposes of this constraint. The optional messageExchange attribute is used to associate an <onMessage> activity with a <reply> activity. (For details, see **Providing Web Service Operations**).

```
<pick createInstance="yes|no"? standard-attributes>

      standard-elements

   <onMessage partnerLink="ncname" portType="qname"?
        operation="ncname" variable="ncname"?
        messageExchange="ncname"? >+
     <correlations>?
        <correlation set="ncname" initiate="yes|no"?/>+
     </correlations>
     <fromPart part="ncname" toVariable="ncname"/>*
     activity
   </onMessage>
   <onAlarm>*
          ( <for expressionLanguage="anyURI"?>duration-expr</for> |
            <until expressionLanguage="anyURI"?>deadline-expr</until> )

      activity
   </onAlarm>
</pick>
```

The `pick` activity completes when one of the branches is triggered by the occurrence of its associated event and the corresponding activity completes. The following example shows a typical usage of pick. Such a pick activity can occur in a loop that is accepting line items for a large order, but a completion action is enabled as an alternative event.

```
<pick>
      <onMessage partnerLink="buyer"
        portType="orderEntry"
        operation="inputLineItem"
        variable="lineItem">
        <!-- activity to add line item to order -->
     </onMessage>
     <onMessage partnerLink="buyer"
        portType="orderEntry"
        operation="orderComplete"
        variable="completionDetail">
        <!-- activity to perform order completion -->
     </onMessage>
<!-- set an alarm to go after 3 days and 10 hours -->
     <onAlarm>
        <for>'P3DT10H'</for>
        <!-- handle timeout for order completion -->
     </onAlarm>
</pick>
```

12.6. Flow

The `flow` construct provides concurrency and synchronization. The grammar for `flow` is:

```
<flow standard-attributes>
        standard-elements
       <links>?
         <link name="ncname">+
       </links>
       activity+
</flow>
```

The **standard attributes** and **standard elements** for activities nested within a `flow` are especially significant because the standard attributes and elements primarily exist to provide `flow-related` semantics to activities.

The most fundamental semantic effect of grouping a set of activities in a `flow` is to enable concurrency. A `flow` completes when all of the activities in the `flow` have completed. Completion of an activity in a flow includes the possibility that it will be skipped if its enabling condition turns out to be false (see **Dead-Path-Elimination**). Thus the simplest use of `flow` is equivalent to a nested concurrency construct. In the following example, the two invoke activities are enabled to start concurrently as soon as the `flow` is started. The completion of the `flow` occurs after both the seller and the shipper respond (assuming the invoke operations were synchronous request/response). The bank is invoked only after the flow completes.

```
<sequence>
     <flow>
        <invoke partnerLink="Seller" .../>
        <invoke partnerLink="Shipper" .../>
    </flow>
```

```
    <invoke partnerLink="Bank" .../>
</sequence>
```

More generally, a flow activity creates a set of concurrent activities directly nested within it. It further enables expression of synchronization dependencies between activities that are nested directly or indirectly within it. The `link` construct is used to express these synchronization dependencies. A `link` has a name and all the links of a flow activity MUST be defined separately within the flow activity. A link's name MUST be unique amongst all link names defined within the same immediately enclosing flow. This requirement MUST be statically enforced.

The standard `source` and `target` elements of an activity are used to link two activities. The source of the link MUST specify a `source` element specifying the link's name and the target of the link MUST specify a `target` element specifying the link's name. The source activity MAY also specify a transition condition by including a <transitionCondition> element under the <source> element. If the transition condition is omitted, it is deemed to be present with a value of `"true"`. Targets MAY also specify a joinCondition by including the <joinCondition> element. This is described further in 12.5.1. Every link declared within a flow activity MUST have exactly one activity within the flow as its source and exactly one activity within the flow as its target. Moreover, at most one link may be used to connect two activities; that is, two different links MUST NOT share the same source and target activities. Finally, the source and target of a link MAY be nested arbitrarily deeply within the (structured) activities that are directly nested within the flow, except for the boundary-crossing restrictions.

The following example shows that links can cross the boundaries of structured activities. There is a link named "CtoD" that starts at activity C in sequence Y and ends at activity D, which is directly nested in the enclosing flow. The example further illustrates that sequence X must be performed prior to sequence Y because X is the source of the link named "XtoY" that is targeted at sequence Y.

```
<flow>
        <links>
            <link name="XtoY"/>
            <link name="CtoD"/>
        </links>
        <sequence name="X">
            <sources>
               <source linkName="XtoY"/>
            </sources>
            <invoke name="A" .../>
            <invoke name="B" .../>
        </sequence>
        <sequence name"Y">
            <targets>
                <target linkName="XtoY"/>
            </targets>
            <receive name="C" ...>
                <sources>
                    <source linkName="CtoD"/>
```

```
                </sources>
            </receive>
            <invoke name="E" .../>
        </sequence>
        <invoke partnerLink="D" ...>
            <targets>
              <target linkName="CtoD"/>
            </targets>
        </invoke>
</flow>
```

A link is said to cross the boundary of a syntactic construct if the source or target activity for the link is nested within the construct while the link is declared outside the construct. Note that it is possible for a link to cross the boundary of a syntactic construct even in those cases where both the source and the target activities are nested within the same construct (so long as the link is declared outside that construct).A link MUST NOT cross the boundary of a while or forEach activity, an isolated scope, an event handler or a compensation handler (see **Scopes** for the specification of event, fault and compensation handlers). In addition, a link that crosses a fault-handler boundary MUST be outbound, that is, it MUST have its source activity within the fault handler and its target activity within a scope that encloses the scope associated with the fault handler. Finally, a link MUST NOT create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity, where an activity A logically precedes an activity B if the initiation of B semantically requires the completion of A. Therefore, directed graphs created by links are always acyclic.

12.6.1. Link Semantics

In the rest of this section, the links for which activity A is the source will be referred to as A's `outgoing` links, and the links for which activity A is the target will be referred to as A's `incoming` links. If activity X is the target of a link that has activity Y as the source, X has a `synchronization dependency` on Y.

Every activity that is the target of a link has an implicit or explicit “join condition” associated with it. This applies even when an activity has exactly one incoming link. Explicit join conditions are provided by the <joinCondition> element under the <targets> element. If the explicit join condition is missing, the implicit condition requires the status of `at least` one incoming link to be positive (see below for an explanation of link status). A join condition is a Boolean expression (see **Expressions**). The expression for a join condition for an activity MUST be constructed using only Boolean operators and the control links variable values for the incoming links at the activity.

Without considering links, the semantics of business processes, scopes, and structured activities determine when a given activity is ready to start. For example, the second activity in a sequence is ready to start as soon as the first activity completes. An activity that defines the behavior of a branch in an if is ready to start if and when that branch is chosen. Similarly, an activity nested directly within a flow is ready to start as soon as the flow itself starts, because flow is fundamentally a concurrency construct.

If an activity that is ready to start in this sense has incoming links, then it does not start until the status of all its incoming links has been determined and the (implicit or explicit) join condition associated with the activity has been evaluated.

The link status is a tri-state flag associated with each declared link. This flag may be in the following three states: "positive", "negative", or "unset". Each time a certain flow activity is activated, the link status of all the links declared in that activity is "unset", that is the lifetime of the status of a link is exactly the lifetime of the flow activity within which it is declared.

The precise semantics of link status evaluation are as follows:

When activity A completes, the following steps are performed to determine the effect of the synchronization links on other activities:

- Determine the status of all outgoing links for A. The status will be either `positive` or `negative`. To determine the status for each link its `transitionCondition` is evaluated. Note that the evaluation is carried out with the actual values of the variables referenced in the transition condition expression. If some of the variables are modified in a concurrent behavior path, the result of the transition condition evaluation may depend nondeterministically on the timing of behavior among concurrent activities. If the value is `true` the status is positive, otherwise it is negative.

  NOTE: The transition condition is evaluated after the activity has completed. If an error occurs while evaluating the transition condition, that error does not affect the completion status of the activity and is handled by the source activity's enclosing scope. If the target of the link is outside the source activity's enclosing scope then the status of the link is negative (see the **Dead-Path-Elimination** section below). If the target is within the enclosing scope the status is irrelevant since the scope has faulted. It is important to keep in mind that in the case of source activities that are themselves scopes, completion does not necessarily imply successful completion. A scope may suffer an internal fault and yet complete (unsuccessfully) if there is a corresponding fault handler associated with the scope and that fault handler completes without throwing a fault. In the case of a link L with a scope S as its source activity, a fault resulting from an error in evaluating the transition condition for L would be propagated to the enclosing scope for S.

- For each activity B that has a synchronization dependency on A, check whether:
  - B is ready to start (except for its dependency on incoming links) in the sense described above.
  - The status of all incoming links for B has been determined.
- If both these conditions are true, then evaluate the join condition for B. If the join condition evaluates to false, a standard `bpws:joinFailure` fault is thrown, otherwise activity B is started.

If, during the performance of structured activity S, the semantics of S dictate that activity X nested within S will not be performed as part of the behavior of S, then the status of all outgoing links from X is set to negative. In order to avoid violating control dependencies, this step is performed only after all of X's incoming links and all incoming links of any activity upon which X has a control dependency, have been determined. An example for where this rule applies is an activity within a branch that is not taken in an if activity, or activities that were not completed in a scope in which processing was halted due to a fault, including a bpws:joinFailure (see **Scopes** and **Compensation Handlers**).
It is illegal for a link to have an activity as a target if the source activity of the link is an ancestor of the target activity of the link. This requirement MUST be statically enforced.

Note that `onEvent` event handlers are permitted to have several simultaneously active instances. A private copy of all process data and control behavior defined within an event handler is provided to each instance of an event handler. This includes the behavior of links defined within an event handler. Each instance of the event handler must independently evaluate the status of the link as needed.

Note that in general multiple target activities will be enabled based on the completion of an activity with multiple outgoing links; because of this, such an activity is often called a fork activity.

When a flow activity is nested within another flow activity, the inner flow activity may define a link with the same name as in the enclosing flow activity. When this happens, a source or target reference to such link from an activity matches the innermost link visible to the activity and hides all other links with the same name. Consider the following example:

```
  <flow name="F1">
       <links>
           <link name="L1"/> <!-- L1 is defined here and ... -->
       </links>
       <sequence name="S1">
           <flow name="F2">
              <links>
                  <link name="L1"/> <!-- ... here -->
              </links>
              <sequence name="S2">
                  <receive name="R">
                    <sources>
                      <source linkName="L1"/> <!—This matches F2.L1
and not F1.L1 -->
                    </sources>
                  </receive>
                  <invoke name="I" .../>
               </sequence>
              ...
           </flow>
          ...
       </sequence>
       ...
```

</flow>A link with the name “L1” is defined in the flow “F1” as well in its nested flow “F2”. The source reference to link with the name L1 from the receive activity R, matches the link L1 defined in F2 as it is the inner most link with that name visible to the activity.

12.6.2. Dead-Path-Elimination (DPE)

In cases where the control flow is largely defined by networks of links, the normal interpretation of a false join condition for activity A is that A should not be performed, rather than that a fault has occurred. Moreover, there is a need to propagate the consequences of this decision by assigning a negative status to the outgoing links for A. WS-BPEL makes it easy to express these semantics by using an attribute `suppressJoinFailure` on an activity. A value of "yes" for this attribute has the effect of suppressing the `bpws:joinFailure` fault for the activity and all nested activities, except where the effect is overridden by using the `suppressJoinFailure` attribute with a value of "no" in a nested activity. Suppressing the bpws:joinFailure is equivalent to surrounding each affected activity with a <scope> activity that defines an "empty" fault handler for the bpws:joinFailure fault. For example, the following:

```
<process suppressJoinFailure="yes" ..>
   <invoke name="invokeA" ... />
</process>
```

is equivelent to:

```
<process...>
   <scope name="invokeA">
       <faultHandlers>
           <catch faultName="bpws:joinFailure">
                <empty/>
            </catch>
       </faultHandlers>
       <invoke name="invokeA" ... />
   </scope>
</process>
```

The default handler behavior is an `empty` activity, that is, the handler suppresses the fault and does nothing about it. However, because the activity with the join condition was not performed, its outgoing links are automatically assigned a negative status according to the rules of **Link Semantics**. Thus within an activity with the value of the `suppressJoinFailure` attribute set to "yes", the semantics of a join condition that evaluates to false are to skip the associated activity and to set the status of all outgoing links from that activity to negative. This is called dead-pathelimination because in a graph-like interpretation of networks of links with transition conditions, these semantics have the effect of propagating negative link status transitively along entire paths formed by consecutive links until a join condition is reached that evaluates to true.

Note that the name of the scope created to suppress the bpws:joinFailure fault that immediately encloses an activity with a join condition is exactly the same as the name of the activity itself. In case this is an invoke activity (see Invoking Web Service Operations)

with an inlined fault or compensation handler, join failure suppression is applied to the implied scope of the <invoke> activity; i.e. effectively two scopes are implied: an outer scope containing an empty fault handler for the bpws:joinFailure fault and an inner scope containing the links, fault handlers, and compensation handlers of the original <invoke> activity. Identical rules apply to <scope> activities: a <scope> with join failure suppression enabled implies an *additional* surrounding scope..

The default value of the `suppressJoinFailure` attribute of the global process element is "no". This is to avoid unexpected behavior in simple use cases where complex graphs are not involved and links without transition conditions are used for synchronization. The designers of such use cases are likely to be naive about link semantics and are likely to be surprised by the consequences of a default interpretation that suppresses a well-defined fault. For example, consider the interpretation of the **Initial Example** with the `suppressJoinFailure` attribute set to `"yes"`. Suppose further that the invocations of the `shippingProvider` are enclosed in a scope that provides a fault handler (see **Scopes** and **Fault Handlers**). If one of these invocations were to fault, the status of the outgoing link from the invocation would be negative, and the (implicit) join condition at the target of the link would be false, but the resulting `bpws:joinFailure` would be implicitly suppressed and the target activity would be silently skipped within the sequence instead of causing the expected fault.

If universal suppression of the `bpws:joinFailure` is desired, it is easy to achieve by using the `suppressJoinFailure` attribute with a value of `"yes"` in the overall `process` element at the root of the business process definition.

12.6.3. Flow Graph Example

In the following example, the activities with the names `getBuyerInformation`, `getSellerInformation`, `settleTrade`, `confirmBuyer`, and `confirmSeller` are nodes of a graph defined through the flow activity. The following links are defined:

- The link named `buyToSettle` starts at `getBuyerInformation` (specified through the corresponding `source` element nested in `getBuyerInformation`) and ends at `settleTrade` (specified through the corresponding target element nested in `settleTrade`).
- The link named `sellToSettle` starts at `getSellerInformation` and ends at `settleTrade`.
- The link named `toBuyConfirm` starts at `settleTrade` and ends at `confirmBuyer`.
- The link named `toSellConfirm` starts at `settleTrade` and ends at `confirmSeller`.

Based on the graph structure defined by the flow, the activities `getBuyerInformation` and `getSellerInformation` can run concurrently. The `settleTrade` activity is not performed before both of these activities are completed. After `settleTrade` completes the two activities, `confirmBuyer` and `confirmSeller` are performed concurrently again.

```
<flow suppressJoinFailure="yes">
        <links>
             <link name="buyToSettle"/>
             <link name="sellToSettle"/>
             <link name="toBuyConfirm"/>
             <link name="toSellConfirm"/>
        </links>
        <receive name="getBuyerInformation">
              <sources>
                   <source linkName="buyToSettle"/>
              </sources>
        </receive>
        <receive name="getSellerInformation">
             <sources>
                <source linkName="sellToSettle"/>
             </sources>
        </receive>
        <invoke name="settleTrade"
              <targets>
                <joinCondition>
                  $buyToSettle and $sellToSettle
                </joinCondition>
                 <target linkName="buyToSettle"/>
                 <target linkName="sellToSettle"/>
              </targets>
             <sources>
                  <source linkName="toBuyConfirm"/>
                  <source linkName="toSellConfirm"/>
             </sources>
        </invoke>
        <reply name="confirmBuyer">
            <targets>
                <target linkName="toBuyConfirm"/>
            </targets>

        </reply>
        <reply name="confirmSeller">
            <targets>
                <target linkName="toSellConfirm"/>
            </targets>

        </reply>
</flow>
```

12.6.4. Links and Structured Activities

Links can cross the boundaries of structured activities. When this happens, care must be taken to ensure the intended behavior of the business process. The following example illustrates the behavior when links target activities within structured constructs.

The following flow is intended to perform the sequence of activities A, B, and C. Activity B has a synchronization dependency on the two activities X and Y outside of the sequence, that is, B is a target of links from X and Y. The join condition at B is missing, and therefore implicitly assumed to be the default, which is the disjunction of the status

of the links targeted to B. The condition is therefore true if at least one of the incoming links has a positive status. In this case that condition reduces to the Boolean condition `P(X,B) OR P(Y,B)` based on the transition conditions on the links.

In the flow, the sequence S and the two receive activities X and Y are all concurrently enabled to start when the flow starts. Within S, after activity A is completed, B cannot start until the status of its incoming links from X and Y is determined and the implicit join condition is evaluated. When activities X and Y complete, the join condition for B is evaluated.

Suppose that the expression `P(X,B) OR P(Y,B)` evaluates to false. In this case, the standard fault `bpws:joinFailure` will be thrown, because the environmental attribute `suppressJoinFailure` is set to "no". Thus the behavior of the flow is interrupted and neither B nor C will be performed.

If, on the other hand, the environmental attribute `suppressJoinFailure` is set to "yes", then B will be skipped but C will be performed because the `bpws:joinFailure` will be suppressed by the implicit scope associated with B.

```
<flow suppressJoinFailure="no">
       <links>
           <link name="XtoB"/>
           <link name="YtoB"/>
       </links>
       <sequence name="S">
           <receive name="A" ...>
               ...
           </receive>
           <receive name="B" ...>
               <targets>
                    <target linkName="XtoB"/>
                    <target linkName="YtoB"/>
               </targets>

               ...
          </receive>
          <receive name="C" ...>
               ...
          </receive>
       </sequence>
       <receive name="X" ...>
         <sources>
             <source linkName="XtoB">
               <transitionCondition>P(X,B)</transitionCondition>
             </source>
         </sources>      ...
       </receive>
       <receive name="Y" ...>
          <sources>
              <source linkName="YtoB">
                <transitionCondition>P(Y,B)</transitionCondition>
```

```
              </source>
           </sources>
              ...
        </receive>
</flow>
```

Finally, assume that the preceding flow is slightly rewritten by linking A, B, and C through links (with transition conditions with constant truth-value of "true") instead of putting them into a sequence. Now, B and thus C will always be performed. Because the join condition is a disjunction and the transition condition of link AtoB is the constant "true", the join condition will always evaluate to "true", independent from the values of P(X,B) and P(Y,B).

```
<flow suppressJoinFailure="no">
        <links>
            <link name="AtoB"/>
            <link name="BtoC"/>
            <link name="XtoB"/>
           <link name="YtoB"/>
        </links>
        <receive name="A">
            <sources>
                <source linkName="AtoB"/>
            </sources>
        </receive>
        <receive name="B">
           <targets>
                <target linkName="AtoB"/>
               <target linkName="XtoB"/>
               <target linkName="YtoB"/>
            </targets>
            <sources>
               <source linkName="BtoC"/>
            </sources>
        </receive>
        <receive name="C">
            <targets>
                <target linkName="BtoC"/>
            </targets>
        </receive>
        <receive name="X">
            <sources>
                <source linkName="XtoB">
                  <transitionCondition>P(X,B)</transitionCondition>
                </source>
            </sources>
        </receive>
        <receive name="Y">
            <sources>
                <source linkName="YtoB">
                  <transitionCondition>P(Y,B)</transitionCondition>
                 </source>
            </sources>
        </receive>
</flow>
```

12.7 ForEach

```
<forEach counterName="ncname" parallel="yes|no" standard-attributes>
     standard-elements
     <startCounterValue expressionLanguage="anyURI">
        ...
     </startCounterValue>
     <finalCounterValue expressionLanguage="anyURI">
        ...
     </finalCounterValue>
     activity
</foreach>
```

The foreach activity is an iterator that will repeat its contained scope activity exactly N+1 times where N equals the <finalCounterValue> minus the <startCounterValue>.

When the foreach activity is started the expressions in <startCounterValue> and <finalCounterValue> are evaluated for the first and only time. That is, once the two values are returned they remain constant for the lifespan of the activity. Both expressions MUST each return a TII (meaning they contain at least one character) that can be validated as a xs:unsignedint, if any of these restrictions are violated then the bpws:forEachCounterError fault MUST be thrown. If the startCounterValue is less than or equal to the finalCounterValue, then no iteration will be performed.

If parallel="no" then this is a serial foreach where the enclosed scope activity will be repeated N+1 times. During each repetition a variable of type xs:unsignedint will be created with the name specified in the counterName attribute in an implicit scope contained within the forEach activity; this implicit scope in turn contains the activities specified within the forEach element. The counter variable is local to the implicit scope and although its value can be changed during an iteration that value will be lost at the end of each iteratoin, and will not affect the value that the next iteration's counter will be set to. The first iteration of the scope will see the counter variable set to the startCounterValue. The next iteration will cause the counter variable to be set to the startCounterValue plus one. Each subsequent iteration will increment the previously initialized counter value by one until the final iteration where the counter will be set to the finalCounterValue.

If parallel="yes" then this is a parallel foreach where the N+1 instances of the enclosed activity will occur in parallel. In essence an implicit flow is dynamically created with N+1 copies of the foreach's implicit scope activity as children. Each copy of the implicit scope activity will have the same counter variable defined in the same manner as specified for serial foreach with the exception that each counter variable will be uniquely initialized to one of the integer values starting with startCounterValue and covering all integer values, incremented by one, up to and including finalCounterValue.

13. Scopes

The behavior context for each activity is provided by a `scope`. A `scope` can provide fault handlers, event handlers, a compensation handler, data variables, partner links, and correlation sets.

All scope elements are syntactically optional and some have default semantics when omitted. The syntax and semantics of scopes are explained in detail below.

```
<scope isolated="yes|no" standard-attributes>
        standard-elements
        <partnerLinks>?
            ...
        </partnerLinks>
        <variables>?
            ...
        </variables>
        <correlationSets>?
            ...
        </correlationSets>
        <faultHandlers>?
            ...
        </faultHandlers>
        <compensationHandler>?
            ...
        </compensationHandler>
        <terminationHandler>?
            ...
        </terminationHandler>
        <eventHandlers>?
            ...
        </eventHandlers>
        activity
</scope>
```

All handlers on a scope are lexically subordinate to the scope and so can access all variables, partnerLinks and correlation sets defined on the scope and its linear ancestors subject to any restrictions unique to the handler type specified elsewhere in this document.

Each scope has a primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities within it to arbitrary depth. The scope is shared by all the nested activities.

When a process/scope is entered scope initialization occurs. Scope initialization consists of instantiating the scope's fault handlers, termination handlers, partner links, correlation sets and variables, including variable initializations. Any partner links defined in the scope MUST be created before variables defined in the same scope whose initialization logic refers to those partner links. Scope initialization is an all or nothing behavior, either it all occurs successfully or a bpws:scopeInitializationFailure fault is thrown which MUST be caught by the parent scope of the failed scope. In the case of a failure at the process level the entire process is treated as faulted. Once scope initialization completes the main activity of the scope and the scope's event handlers are instantiated in parallel with each other. An exception to the previous rule applies to scopes that contain a

process's initial start activity. An initial start activity is the start activity that caused a particular process instance to be instantiated. If a scope contains an initial start activity then the start activity MUST complete before the event handlers are instantiated.

In the following example, the scope has a primary `flow` activity, which contains two concurrent invoke activities. Either of the invoke activities can receive one or more types of fault responses. The fault handlers for the scope are shared by both invoke activities and can be used to catch the faults caused by the possible fault responses.

```
<scope>
        <faultHandlers>?
            ...
        </faultHandlers>
        <flow>
           <invoke partnerLink="Seller" portType="Sell:Purchasing"
                operation="SyncPurchase"
                inputVariable="sendPO"
                outputVariable="getResponse"/>
           <invoke partnerLink="Shipper"
                portType="Ship:TransportOrders"
                operation="OrderShipment"
                inputVariable="sendShipOrder"
                outputVariable="shipAck"/>
        </flow>
</scope>
```

13.1. Data Handling and Partner Links

A scope can declare variables and parter links that live only within the scope. For further information see the chapter about **Data Handling** and **Partner Links** respectively.

13.2. Error Handling in Business Processes

Business processes are often of long duration and use asynchronous messages for communication. They also manipulate sensitive business data in back-end databases and line-of-business applications. Error handling in this environment is both difficult and business critical. The use of ACID transactions is usually limited to local updates because of trust issues and because locks and isolation cannot be maintained for the long periods during which technical and business errors and fault conditions can occur in a business process instance. As a result, the overall business transaction can fail or be cancelled after many ACID transactions have been committed during its progress, and the partial work done must be undone as best as possible. Error handling in business processes therefore relies heavily on the well-known concept of `compensation`, that is, application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned. There is a long history of work in this area regarding the use of Sagas [Sagas] and open nested transactions [Trends]. WS-BPEL provides a variant of such a compensation protocol by providing the ability for

flexible control of the reversal. WS-BPEL achieves this by providing the ability to define fault handling and compensation in an application-specific manner, resulting in a feature called Long-Running (Business) Transactions (LRTs).

It is important to understand that the notion of LRT described here is meant to be used purely within a platform-specific implementation. There is no prescribed requirement that the business process be distributed or span multiple vendors and platforms. Additionally, it is important to understand that the notion of LRT described here is purely local and occurs within a single business process instance. There is no distributed coordination regarding an agreed-upon outcome among multiple-participant services. The achievement of distributed agreement is an orthogonal problem outside the scope of WS-BPEL.

As an example of an LRT, consider the planning and fulfillment of a travel itinerary. This can be viewed as an LRT in which individual service reservations can use nested transactions within the scope of the overall LRT. If the itinerary is cancelled, the reservation transactions must be compensated for by cancellation transactions, and the corresponding payment transactions must be compensated accordingly. For ACID transactions in databases the transaction coordinator(s) and the resources that they control know all of the uncommitted updates and the order in which they must be reversed, and they are in full control of such reversal. In the case of business transactions, the compensation behavior is itself a part of the business logic and protocol, and must be explicitly specified. For example, there might be penalties or fees applied for cancellation of an airline reservation depending on the class of ticket and the timing. If a payroll advance has been given to pay for the travel, the reservation must be successfully cancelled before the payroll advance for it can be reversed in the form of a payroll deduction. This means the compensation actions might need to run in the same order as the original transactions, which is not the standard or default in most transaction systems. Using activity scopes as the definition of logical units of work, the LRT feature of WS-BPEL addresses these requirements.

13.3. Compensation Handlers

Scopes can delineate a part of the behavior that is meant to be reversible in an applicationdefined way by a compensation handler. Scopes with compensation and fault handlers can be nested without constraint to arbitrary depth.

13.3.1. Defining a Compensation Handler

A compensation handler is simply a wrapper for a compensation activity as shown below.

```
<compensationHandler>?
       activity
</compensationHandler>
```

As explained in **Invoking Web Service Operations**, there is a special shortcut for the invoke activity to inline a compensation handler rather than explicitly using an immediately enclosing scope. For example:

```
<invoke partnerLink="Seller" portType="SP:Purchasing"
        operation="SyncPurchase"
        inputVariable="sendPO"
        outputVariable="getResponse">
     <correlations>
        <correlation set="PurchaseOrder" initiate="yes"
             pattern="out"/>
    </correlations>
    <compensationHandler>
        <invoke partnerLink="Seller" portType="SP:Purchasing"
           operation="CancelPurchase"
           inputVariable="getResponse"
          outputVariable="getConfirmation">
             <correlations>
                  <correlation set="PurchaseOrder" pattern="out"/>
              </correlations>
        </invoke>
     </compensationHandler>
</invoke>
```

In this example, the original invoke activity makes a purchase and in case that purchase needs to be compensated, the `compensationHandler` invokes a cancellation operation at the same port of the same partnerLink, using the response to the purchase request as the input.

In standard syntax (without the invoke shortcut) this example would be equivalently expressed as follows:

```
<scope>
   <compensationHandler>
      <invoke partnerLink="Seller" portType="SP:Purchasing"
        operation="CancelPurchase"
        inputVariable="getResponse"
        outputVariable="getConfirmation">
        <correlations>
              <correlation set="PurchaseOrder" pattern="out"/>
        </correlations>
      </invoke>
   </compensationHandler>
   <invoke partnerLink="Seller" portType="SP:Purchasing"
        operation="SyncPurchase"
        inputVariable="sendPO"
        outputVariable="getResponse">
        <correlations>
              <correlation set="PurchaseOrder" initiate="yes"
                pattern="out"/>
        </correlations>
   </invoke>
</scope>
```

Note that the variable `getResponse` is not local to the scope to which the compensation handler is attached and can be reused later for other purposes before compensation for this scope is invoked. The current state of non-local variables is available in compensation handlers as explained more fully below. But assuming the compensation

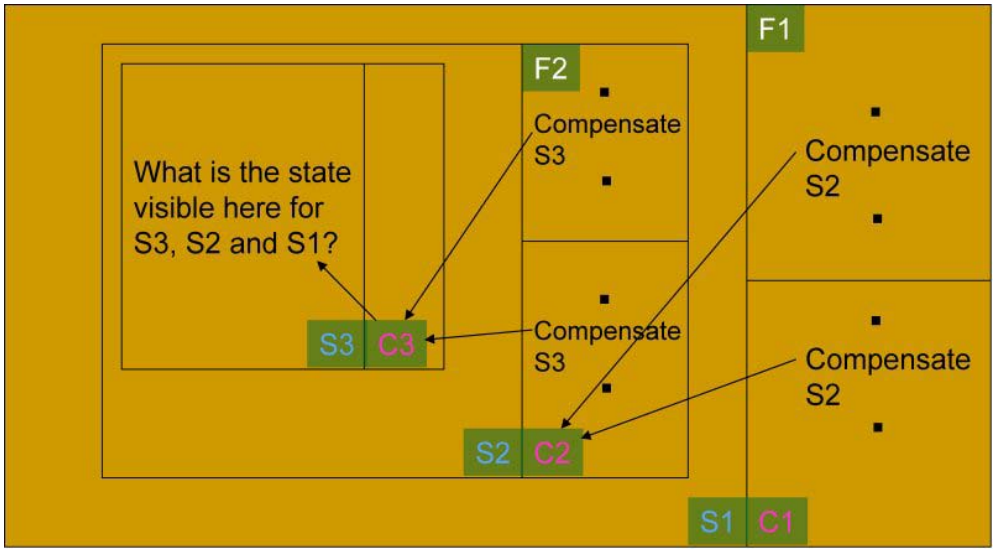
handler needs the specific response to the `invoke` operation that is being reversed, that response would most conveniently be stored in a variable local to the scope, e.g., by making getResponse local to the scope.

13.3.2. Process State Usage in Compensation Handlers

Compensation handlers always use the current state of the process, specifically the state of variables declared in their associated scope and all enclosing scopes. The variables include partnerLinks at the process scope. Compensation handlers are able to both get and set the values of all such variables. Other parts of the process will see the changes made to shared variables by compensation handlers, and conversely, compensation handlers will see changes made to shared variables by other parts of the process, including situations where a compensation handler runs concurrently with other parts of the process. Compensation handlers will need to use isolated scopes when they touch state in enclosing scopes to avoid interference if concurrency is expected.

The current state of the process consists of the current local state of all scopes that have been started. This includes scopes that have completed successfully but for which the associated compensation handler has not been invoked. For successfully completed uncompensated scopes their current local state is the state as it was at the time of completion. Such scopes are in suspended animation because their compensation handlers are still available and therefore their execution may continue in compensation mode. Note that a scope may have been executed several times in a loop, and the current state of the process includes the state of each successfully completed (and uncompensated) iteration through the scope. We refer to the preserved state of a successfully completed uncompensated scope as a *scope snapshot*.

The behavior of a compensation handler can be thought of as an optional continuation of the behavior of the associated scope and as such its usage of variables is similar to the usage that occurred in the body of the scope itself, including update actions. This includes variables in both the local scope and all enclosing scopes. For the variables in the local scope, the compensation handler starts with the scope snapshot. Note that the compensation handler may itself have been called from an enclosing compensation handler. It will then share the continuation of the state of the enclosing scope that its caller is using. In the picture below showing three nested scopes S1, S2 and S3, and their compensation handlers C1, C2, C3, and failure handlers F1 and F2, we may have an error handling call stack F1->C2->C3. In that case C3 will share the state of S2 as it is being seen and used by C2, and the current state of the uncompleted scope S1.



13.3.3. Invoking a Compensation Handler

The compensation handler can be invoked by using the `compensate` activity, which names the scope for which the compensation is to be performed, that is, the scope whose compensation handler is to be invoked. A compensation handler for a scope is available for invocation only when the scope completes normally. The first attempt to invoke a compensation handler that has never been installed is equivalent to the `empty` activity (it is a no-op) – this ensures that fault handlers do not have to rely on state to determine which nested scopes have completed successfully. Any subsequent attempts to invoke the never installed compensation handler will also result in a no-op (as does any attempt to invoke an installed compensation more than once)

Note that in case an invoke activity has a compensation handler defined inline, the name of the activity is the name of the scope to be used in the `compensate` activity.

```
<compensate scope="ncname"? standard-attributes>
        standard-elements
</compensate>
```

The ability to explicitly invoke the `compensate` activity is the underpinning of the application-controlled error-handling framework of WS-BPEL. This activity can be used only in the following parts of a business process:

- In a fault handler of the scope that immediately encloses the scope for which compensation is to be performed.
- In the compensation handler of the scope that immediately encloses the scope for which compensation is to be performed.

Example:

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

If a scope being compensated by name was nested in a loop, the instances of the compensation handlers in the successive iterations are invoked in reverse order.

All scopes and activities directly nested in a scope MUST be uniquely named. If the value of the scope attribute specified on a compensate activity does not resolve to a unique scope or activity name in the same scope as the compensate activity, the BPEL definition MUST be rejected from processing. This requirement MUST be statically enforced.

If the explicit compensation handler for a scope is absent, the default compensation handler for the scope is invoked.

The `<compensate/>` form, in which the scope name is omitted in a `compensate` activity, causes the default compensation behavior for the current scope to be invoked explicitly. This is useful when an enclosing fault or compensation handler needs to perform additional work, such as updating variables or sending external notifications, in addition to performing default compensation for the associated scope. Note that the `<compensate/>` activity in a fault or compensation handler attached to scope S causes the default-order invocation of compensation handlers for completed scopes directly nested within S. The use of this activity can be mixed with any other user-specified behavior except the explicit invocation of `<compensate scope="Sx"/>` for scope Sx nested directly within S. Explicit invocation of compensation for such a scope nested within S disables the availability of default-order compensation, as expected.

13.4. Fault Handlers

Fault handling in a business process can be thought of as a mode switch from the normal processing in a scope. Fault handling in WS-BPEL is always treated as "reverse work" in that its sole aim is to undo the partial and unsuccessful work of a scope in which a fault has occurred. The completion of the activity of a fault handler, even when it does not rethrow the fault handled, is never considered successful completion of the attached scope and compensation is never enabled for a scope that has had an associated fault handler invoked.

The optional fault handlers attached to a scope provide a way to define a set of custom fault-handling activities, syntactically defined as `catch` activities. Each `catch` activity is defined to intercept a specific kind of fault, defined by a globally unique fault QName and a variable for the data associated with the fault. If the fault name is missing, then the catch will intercept all faults with the right type of fault data. The fault variable is specified using the faultVariable attribute in a catch handler. The variable is deemed to be declared by virtue of being used as the value of this attribute and is local to the fault handler. It is not visible or usable outside the fault handler in which it is declared. The fault variable is optional because a fault might not have additional data associated with it.

A fault response to an invoke activity is one source of faults, with name and data aspects based on the definition of the fault in the WSDL operation (please refer to the Implementer's Note in Section 11.3 for important details regarding access to fault information when faults are transmitted using SOAP). A programmatic `throw` activity is another source, again with explicitly given name and data. The core concepts and executable pattern extensions of WS-BPEL define several standard faults with their names and data, and there might be other platform-specific faults such as communication failures that can occur in a business process instance. A `catchAll` clause can be added to catch any fault not caught by a more specific fault handler.

```
<faultHandlers>?
      <!-- there must be at least one fault handler or default -->
     <catch faultName="qname"? faultVariable="ncname"?
                               faultMessageType="qname"?
                               faultElement="qname"?>*
        activity
     </catch>
     <catchAll>?
        activity
    </catchAll>
</faultHandlers>
```

Data thrown with a fault can be a WSDL message type or a XML schema element. Each catch which specifies a faultName can only catch a fault of a single type. This one-to-one relationship is necessary in order to guarantee proper typing. If the data to be caught is a WSDL message type then the faultMessageType attribute MUST be used to specify the message type's qname. If the data to be caught is a XML element definition then the faultElement attribute MUST be used to specify the element definition's qname.

The faultName may reflect a purely internal custom fault in a process, or the faultName may be missing. In such cases, the faultVariable, which is local to the fault handler and declared by its occurrence in the catch element, will not have a defined type. To avoid this possibility faultVariable MUST only be used if either the faultMessageType or faultElement attributes, but not both, accompany it. faultMessageType and faultElement MUST NOT be used unless accompanied by faultVariable.

Because of the flexibility allowed in expressing the faults that a catch activity can handle, it is possible for a fault to match more than one fault handler.

In the case of faults thrown without associated data the fault will be caught as follows:

1. If there is a catch activity with a matching faultName value that does not specify a faultVariable attribute then the fault is passed to the identified catch activity.
2. Otherwise if there is a catchAll handler then the fault is passed to the catchAll handler.
3. Otherwise, the fault will be handled by the default fault handler.

In the case of faults thrown with associated data the fault will be caught as follows:

1. If there is a catch activity with a matching faultName value that has a faultVariable whose type matches the type of the fault data then the fault is passed to the identified catch activity.
2. Otherwise if the fault data is a WSDL message type where the message contains a single part defined by an element and there exists a catch activity with a matching faultName value that has a faultVariable whose type matches the type of the element used to define the part then the fault is passed to the identified catch activity with the faultVariable initialized to the value in the single part's element.
3. Otherwise if there is a catch activity with a matching faultName value that does not specify a faultVariable or faultMessageType value then the fault is passed to the identified catch activity. Note that in this case the fault value will not be available from within the fault handler but will be available to the <rethrow> activity.
4. Otherwise if there is a catch activity without a faultName attribute that has a faultVariable whose type matches the type of the fault data then the fault is passed to the identified catch activity.
5. Otherwise if the fault data is a WSDL message type where the message contains a single part defined by an element and there exists a catch activity without a faultName attribute that has a faultVariable whose type matches the type of the element used to define the part then the fault is passed to the identified catch activity with the faultVariable initialized to the value in the single part's element.
6. Otherwise if there is a catchAll handler then the fault is passed to the catchAll handler.
7. Otherwise, the fault will be handled by the default fault handler.

If the fault occurs in (or is rethrown to) the global process scope, and there is no matching fault handler for the fault at the global level, the process terminates abnormally, as though an exit activity had been performed. See Implicit Fault and Compensation Handlers for a more complete description of the default fault and compensation handling behavior.

Consider the following example:

```
<faulthandlers>
        <catch faultName="x:foo">
            <empty/>
        </catch>
        <catch faultVariable="bar" faultMessageType="tns:barType">
            <empty/>
        </catch>
        <catch faultName="x:foo" faultVariable="bar"
                                 faultMessageType="tns:barType">
            <empty/>
        </catch>
        <catchAll>
            <empty/>
        </catchAll>
</faulthandlers>
```

Assume that a fault named ”x:foo” is thrown. The first `catch` will be selected if the fault carries no fault data. If there is fault data associated with the fault, the third `catch` will be selected if and only if the type of the fault’s data matches the type of variable “bar”, otherwise the default `catchall` handler will be selected. Finally, a fault with a fault variable whose type matches the type of “bar” and whose name is not “x:foo” will be processed by the second catch. All other faults will be processed by the default `catchall` handler.

It is sometimes necessary to rethrow a caught fault. However the <throw> activity requires a fault name and an optional fault value. In the case of a <catchAll> handler or a custom fault handler that does not specify a faultName the fault name will not be accessible for use in the <throw> activity. In the case that a fault with associated data was caught by a custom fault handler that only specified the fault name, the fault value will not be accessible for use in the <throw> activity. Hence all fault handlers are allowed to rethrow the original fault with a <rethrow> activity that is defined to be an empty element. In essence <rethrow> can be considered a macro for a <throw> that rethrows the fault that was originally caught by the immediately enclosing fault handler. For example if a fault handler should modify the fault data and then call <rethrow> the original fault data would be rethrown and not the modified fault data. Similarly if a fault is caught using the shortcut that allows message type faults with one part defined using an element to be caught by fault handlers looking for the same element type, then a <rethrow> would rethrow the original message type data.

An example is shown below:

```
<faulthandlers>
        <catch faultName="x:foo">
            <empty/>
        </catch>
        <catch faultVariable="bar" faultMessageType="y:myMsgType">
            ….
            <rethrow/> <!- rethow of original fault -->
        </catch>
        <catchAll>
            ….
            <rethrow/> <!- rethow of original fault -->
        </catchAll>
</faulthandlers>
```

Although the use of compensation can be a key aspect of the behavior of fault handlers, each handler performs an arbitrary activity, which can even be `<empty/>`. When a fault handler is present, it is in charge of handling the fault. It might rethrow the same fault or a different one, or it might handle the fault by performing cleanup and allowing normal processing to continue in the enclosing scope.

A scope in which a fault occurred is considered to have ended abnormally, whether or not the fault was caught and handled without rethrow by a fault handler. A compensation handler is never installed for a scope in which a fault occurred.

When a fault handler for scope S handles a fault that occurred in S without rethrowing, links that have S as the source will be subject to regular evaluation of status after the fault has been handled, because processing in the enclosing scope is meant to be continued.

As explained in **Invoking Web Service Operations**, there is a special shortcut for the invoke activity to inline fault handlers rather than explicitly using an immediately enclosing scope. For example:

```
<invoke partnerLink="Seller"
        portType="SP:Purchasing"
        operation="SyncPurchase"
        inputVariable="sendPO"
        outputVariable="getResponse">
     <catch faultName="SP:POFault" faultVariable="POFault"
                                faultMessageType="lns:orderFaultType">
        <!-- handle the fault -->
     </catch>
</invoke>
```

In this example, the original `invoke` makes a purchase and a fault handler is inlined to handle the case where the purchase request results in a fault response. In standard syntax (without the `invoke` shortcut), this example would be equivalently expressed as follows:

```
<scope>
     <faultHandlers>
        <catch faultName="SP:POFault" faultVariable="POFault">
                                  faultMessageType="lns:orderFaultType">
             <!-- handle the fault -->
        </catch>
    </faultHandlers>
    <invoke partnerLink="Seller"
        portType="SP:Purchasing"
        operation="SyncPurchase"
        inputVariable="sendPO"
        outputVariable="getResponse">
    </invoke>
</scope>
```

The compensation handler for scope C becomes available for invocation by the fault and compensation handlers for its immediately enclosing scope exactly when scope C completes normally. A fault handler for scope C is available for invocation exactly when C has commenced but has not been completed. If the scope faults before completion, then the appropriate fault handler gets control and all other fault handlers are uninstalled. It is never possible to run more than one fault handler for the same scope under any circumstances.

Note that availability also applies to **Implicit Fault and Compensation Handlers**.

The behavior of a fault handler for scope C begins by implicitly terminating all activities that are currently active and directly enclosed within C (see **Semantics of Activity Termination**). The termination of these activities occurs before the specific behavior of a fault handler is started. This also applies to the implicit fault handlers described below. The activity of a fault handler is deemed to occur in the scope to which the fault handler is attached.

13.4.1. Implicit Fault and Compensation Handlers

Because the visibility of scope names and therefore of compensation handlers is limited to the next enclosing scope, the ability to compensate a scope would be lost if the enclosing scope did not have a compensation handler or was missing a fault handler for some fault. Because many faults are not programmatic or the result of operation invocation, it is not reasonable to expect an explicit handler for every fault in every scope. WS-BPEL therefore provides default compensation and fault handlers when these are missing. Whenever a fault handler (for any fault) or the compensation handler is missing for any given scope, they are implicitly created with the following behavior:

`Fault handler:`

- Run all *available* compensation handlers for all directly and indirectly enclosed scopes in any order consistent with the rules for default compensation order defined below. Note that a compensation handler is available for a scope exactly when that scope has completed successfully.
- Rethrow the fault to the next enclosing scope.

`Compensation handler:`

- Run all *available* compensation handlers for all directly and indirectly enclosed scopes in any order consistent with the rules for default compensation order defined below. Note that a compensation handler is available for a scope exactly when that scope has completed successfully.

13.4.2 Default Compensation Order

There are two rules for default compensation order that address different aspects of the order relation. Note that they are additive, i.e., they MUST both be obeyed in every case in performing default compensation.

Informally, Rule 1 states that default compensation must respect the forward order of execution for the scopes being compensated, but only in so far as that order is mandated by the process definition. In cases where concurrency is permitted as a result of the use of the <flow> construct, and not otherwise constrained by links, any actual temporal order during execution is not a part of the constraint defined by the first rule. More formally, we state the rule based on a precise notion of control dependency.

Definition: **Control Dependency**. If an activity A must complete before activity B begins, as a result of the existence of a control path from A to B in the process definition, then we say that B has a *control dependency* on A. Note that control dependencies may occur due to control links in a `<flow>` as well as due to constructs like `<sequence>`. Control flow due to explicit `<throw>` is not considered a control dependency.

Rule 1: Consider scopes A and B such that B has a control dependency on A. Assuming both A and B completed successfully and both must be compensated as part of default compensation behavior, the compensation handler of B must run to completion before the compensation handler of A is started.

This rule permits scopes that executed concurrently on the forward path to also be compensated concurrently in the reverse path. Of course, if one follows the strict reverse order of completion, then that necessarily respects control dependencies and is also consistent with this rule. The rule imposes a constraint on the order in which compensation handlers run during default compensation, and is not meant to be fully prescriptive about the exact order and concurrency.

Informally, the second rule is needed as a result of the fact that all scopes are not isolated. It is syntactically possible for two scopes to have links crossing from activities within one to activities within the other, and moreover such links may cross in both directions. If both such scopes were isolated, this would be illegal because the semantics of links crossing isolated scope boundaries would imply that such bidirectional links constitute a cycle. The simple way to state the intent of Rule 2 is that we should treat all scopes as if they were isolated, but only for purposes of cycle detection regarding links crossing scope boundaries. This allows us to apply Rule 1 to any pair of scopes to decide unambiguously if there is a control dependency between them, and if so, in which direction. Formally, we need three definitions to state the rule precisely.

Definition: **Peer-Scopes**. Two scopes S1 and S2 are said to be *peer scopes* if they are both directly nested within the same parent scope (including process scope).

Definition: **Scope-controlled set**. An activity A is within the *scope-controlled set* of activities of scope S if either A is S itself, or A is nested within S, at any depth.

Definition: **Peer-Scope Dependency**. If S1 and S2 are peer scopes then we say that S2 has a direct peer-scope dependency on S1 if there is an activity B within the scope-controlled set of S2 and an activity A within the scope-controlled set of S1 such that B has a control dependency on A. The *peer-scope dependency* relation is the transitive closure of the direct peer-scope dependency relation.

Rule 2: The peer-scope dependency relation MUST NOT include cycles. In other words, WS-BPEL forbids a process in which there are peer scopes S1 and S2 such that S1 has a peer-scope dependency on S2 and S2 has a peer-scope dependency on S1.

One of the side effects of Rule 2 is to permit a depth-first traversal of the lexical scope tree for default compensation, respecting the control dependency relation among peer scopes as dictated by Rule 1. Thus, we no longer have any difficulty in defining default compensation of any scope, since depth-first order implies that such compensation is only dependent on the compensation of its nested scopes. A related point is also worth pointing out. The default compensation order mandated by the rules here is consistent with strict reverse order of completion, *but not in depth-first order*. Strict reverse order of completion applied to compensation of all scopes may require interleaving of nested compensations across peer scopes.

13.4.3. Compensation handlers and Isolated Scopes

Given that compensation handlers may run concurrently with other activities including other compensation handlers, it is necessary to allow compensation handlers to use isolation scope semantics. Compensation handlers do not run within the isolation domain of their associated scopes, but fault handlers do. This creates difficulties in the isolation semantics of compensation handlers for scopes nested inside an isolated scope. Such handlers cannot use isolated scopes themselves because isolated scopes cannot be nested. However, their isolation environment is uncertain because they may be invoked from either a fault handler within the isolation domain of their enclosing scope or within the compensation handler of the same enclosing scope which is not in that isolation domain.

In order to ensure consistency of behavior, WS-BPEL mandates that the compensation handler of an isolated scope will itself have isolated behavior implicitly, although it will create a separate isolation domain from that of its associated scope.

13.4.4. Semantics of Activity Termination

As stated above, the behavior of a fault handler for scope C begins by implicitly terminating all activities directly enclosed within C that are currently active. The following paragraphs define what this means for all WS-BPEL activity types.

The `assign` activities are sufficiently short-lived that they are allowed to complete rather than being interrupted when termination is forced. The evaluation of expressions when already started is also allowed to complete. Each `wait, receive, reply` and `invoke` activity is interrupted and terminated prematurely. When a synchronous `invoke` activity (corresponding to a request/reply operation) is interrupted and terminated prematurely, the response (if received) for such a terminated activity is silently discarded. The notion of termination does not apply to `empty, exit`, and `throw`.

All structured activity behavior is interrupted. The iteration of `while` is interrupted and termination is applied to the loop body activity. If an <if> activity has selected a branch, then the termination is applied to the activity of the selected branch. The same applies to pick. If either of these activities has not yet selected a branch, then the `if activity` and the `pick` activity are terminated immediately. The `sequence` and `flow` constructs are

terminated by terminating their behavior and applying termination to all nested activities currently active within them.

Scopes provide the ability to control the semantics of forced termination to some degree. When the activity being terminated is in fact a scope, the forced termination of a scope begins by terminating all activities directly enclosed within its associated scope that are currently active. Following this, the custom termination handler for the scope, if present, is run. If the custom termination handler is missing, the default termination handler performs compensation of all successfully completed nested scopes in the same order as in the case of a default fault handler.

Forced termination for a scope applies only if the scope is in normal processing mode. If the scope has already experienced an internal fault and invoked a fault handler, then the termination handler is uninstalled, and the forced termination has no effect. The already active fault handler is allowed to complete.

The termination handler for a scope is permitted to use the same range of activities as a fault handler, including the <compensate/> activity. However, a termination handler cannot throw any fault. Even if an uncaught fault occurs during its behavior, it is not rethrown to the next enclosing scope. This is because the enclosing scope has already either faulted or is in the process of being terminated, which is what is causing the forced termination of the nested scope.

Forced termination of nested scopes occurs in innermost-first order as a result of the rule (stated above) that the termination handler is run after terminating all activities (including scope activities) directly enclosed within its associated scope that are currently active.

13.4.5. Handling Faults That Occur Inside Fault and Compensation Handlers

Compensation handlers are always invoked directly or indirectly as part of the processing of some fault handler E. The behavior of a compensation handler invoked by E can cause a fault to be thrown. Such a fault, if uncaught by scopes within the chain of compensation handlers invoked by E, is treated as being a fault within E.

If a fault occurs in a fault handler E for a scope C, the fault can be caught through the use of a scope `within` E. If the fault is not caught by a scope within E, it is immediately thrown to the parent scope of C and the behavior of E terminates prematurely. In effect, no distinction is made between faults that E rethrows deliberately and faults that occur as undesired faults in E.

13.4.6. Handling WS-BPEL Standard Faults

If the value of the exitOnStandardFault attribute on a <scope> is set to “yes”, then the process MUST exit immediately as if an <exit> activity has been reached, when any WS-BPEL standard fault other than bpws:joinFailure is enountered. If the value of this

attribute is set to “no”, then the process can handle a WS-BPEL standard fault using a fault handler. The default value for this attribute is “no”. When this attribute is not specified on a <scope> it inherits its value from its enclosing <scope> or <process>.

13.5. Event Handlers

The whole process as well as each scope can be associated with a set of event handlers that are invoked concurrently if the corresponding event occurs. The actions taken within an event handler can be any type of activity, such as sequence or flow, but invocation of compensation handlers using the <compensate/> activity is not permitted. As stated earlier, the <compensate/> activity can only be used in fault and compensation handlers. There are two types of events. First, events can be incoming messages that correspond to a request/response or one-way operation in WSDL. For instance, a status query is likely to be a request/response operation, whereas a cancellation may be a oneway operation. Second, events can be alarms, that go off after user-set times. The grammar for the set of event handlers associated with a scope or process is

```
<eventHandlers>?
<!-- Note: There must be at least one onEvent or onAlarm handler. -->
       <onEvent partnerLink="ncname" portType="qname"?
         operation="ncname" messageType="qname" variable="ncname"
         messageExchange="ncname"? >*
             <correlationSets>?
                   <correlationSet name="ncname"
                      properties="qname-list"/>+
             </correlationSets>
             <correlations>?
               <correlation set="ncname"
                        initiate="yes|join|no"?/>+
             </correlations>
             <fromPart part="ncname" toVariable="ncname"/>*
            activity
       </onEvent>
       <onAlarm>*
         ( <for expressionLanguage="anyURI"?>duration-expr</for> |
           <until expressionLanguage="anyURI"?>deadline-expr</until> )
           <repeatEvery expressionLanguage="anyURI"?>
             duration-expr
           </repeatEvery>?
           activity
       </onAlarm>
</eventHandlers>
```

The portType attribute on <onEvent> is optional. If the portType attribute is included for readability, the value of the portType attribute MUST match the portType value implied by the combination of the partnerLink specified and the associated role. All instances of onEvent MUST use either the messageType or element attribute but not both.

It is important to emphasize that event handlers are considered a part of the normal behavior of the scope, unlike fault and compensation handlers.

13.5.1. Message Events

The onEvent tag indicates that the specified event waits for a message to arrive. The interpretation of this tag and its attributes is very similar to a receive activity. The partnerLink attribute defines the partner link that contains the myRole endpoint reference on which the request is expected to arrive; the <partnerLink> must be defined in the <partnerLinks> section. As with <receive> the partnerRole endpoint reference is ignored for purposes of executing the receive semantics of an event handler. The portType and operation attributes define the appropriate port type and operation that is invoked by the partner in order to cause the event.

The variable attribute identifies a variable local to the eventHandler that will contain the message received from the partner. The messageType attribute specifies the type of the variable by referencing a message type definition using its QName. The type of the variable (as specified by the messageType attribute) must be the same as the type of the input message defined by operation referenced by the operation attribute. Optionally the messageType attribute may be omitted and instead the element attribute substituted if the message to be received has a single part and that part is defined with an element type that matches the element type referenced by the element attribute. The variable and messageType/element attribute constitute the declaration of a variable of that name and type within an implicit scope associated with the event handler. If an element attribute is used then the binding of the incoming message to the variable declared in the onEvent handler occurs as specified for the receive activity in section 11.4.

An alternative to the variable attribute are `<toPart>` elements. The syntax and semantics of the `<toPart>` elements as used on the `onEvent` tag are the same as specified in section 11.3 for the `invoke` activity. This includes the restriction that if `<toPart>` elements are used on an `onEvent` tag then the `variable` attribute MUST NOT be used on the same tag.

Upon receipt of the input message the event handler assigns the input message to the variable before proceeding to perform the event handler activity. Since the variable is declared within a scope associated with the event handler, each instance of the event handler (whether executed serially or concurrently relative to other instances) contains a private copy of the variable, which is not shared with other instances.

Note that the operation specified in the onEvent handler may be either an asynchronous (oneway) or a synchronous (request/response) operation. In the latter case the event handler is expected to use a reply activity to send the response. The usage and interpretation of correlation is exactly the same as for receive activities with the addition that it is possible to declare correlation sets inline to an onEvent handler instance. These inline correlation sets can then be used as part of onEvent's receive. In effect something like a virtual scope is declared around the onEvent handler which contains a local variable declaration for the variable used to receive the incoming message, a local correlation set declaration for any correlation sets declared inline to the onEvent handler and a sequence with a receive, reflecting the receive parameters of the onEvent handler, and finally the actual activity contained within the onEvent handler declaration. However

this virtual scope is fundamentally different than a real scope in that it has neither fault nor compensation handlers, default or otherwise, it also has no name. The scope exists, such as it can be said to exist at all given that it is only intended as a model to understand the onEvent handler's variable behavior and not as an actual entity, only to provide lexical scoping for the local message variable and inline correlation sets. For example:

```
<scope name="S1">
    <compensationHandler>
       <sequence>
          <compensate scope="S2"/>
       </sequence>
    </compensationHandler>
    <eventHandlers>
       <onEvent partnerLink=”travelAgency” portType=”ns:agent”
          operation=”travelUpdate” messageType=”ns:travelStatsUpdate”
          variable=”travelUpdate”>
          <correlationSets>
              <correlationSet name=”updateCode”
                 properties=”ns:updateCode”/>
          </correlationSets>
          <correlations>
             <correlation set=”travelCode” initialize=”no”/>
             <correlation set=”updateCode” initialize=”yes”/>
          </correlations>
       <scope name="S2">
          ...
       </scope>
    </eventHandlers>
    ...
</scope>
```

In this example a process is managing travel reservations for a customer and needs to handle reservation updates from the travel booking system. The onEvent handler is used to receive the update messages which are correlated using the travelCode property, which is defined and initialized elsewhere in the process. However sometimes the event handler needs to contact the travel booking system to follow up on an update message. To do that the outgoing message needs both the value in the travelCode property but also the value in an update code included in the travel update message. This is where the updateCode correlation set, declared locally to the onEvent handler comes in. When the update message is received the updateCode correlation set is initialized and its value made available only to the onEvent handler instance. Note that the compensation handler on scope S1 directly invokes the compensation handler on scope S2 defined inside the onEvent handler. This invocation is legal because the 'notional' virtual scope described above to handle lexical scoping issues for the onEvent handler does not exist for purposes of determining which scopes are considered a child of the parent scope S1. Therefore, from the perspective of the compensation handler scope S2 is a child of scope S1. Note however that if S2's compensation handler was invoked the variable used to receive the message for the onEvent handler as well as any in-line correlation sets would be visible as the 'notional' virtual scope does have existence for the purpose of defining the lexical scope for the onEvent handler's variables.

The semantics of the onEvent event is identical to a receive activity regarding the optional nature of the variable attribute, the handling of race conditions, and the constraint regarding simultaneous enablement of conflicting receive actions. For the last case, recall that the semantics of a process in which two or more `receive` actions for the same partner link, portType, operation and correlation set(s) may be simultaneously enabled is undefined (see **Providing Web Service Operations**). Enablement of each onEvent event handler is equivalent to enablement of the corresponding `receive` activity for the purposes of this constraint.

The optional messageExchange attribute is used to associate an <onEvent> activity with a <reply> activity. (For details, see **Providing Web Service Operations**).

As specified in the grammar above, event handlers for message events are not permitted to carry the createInstance attribute. A business process instance cannot be created by a message event. This is because the event handler cannot be enabled until the instance is created.

When the message constituting an event arrives, the activity specified in the corresponding handler is carried out. The key point to understand is that the business process is enabled to receive such messages concurrently with the normal activity of the scope to which the event handler is attached. This allows such events to occur (or not occur) at arbitrary times and an arbitrary number of times while the corresponding scope (which may be the entire business process instance) is active.

The following example shows the usage of an event handler to support the termination of a process instance through an external message. Alternatively, the event handler could throw a fault to cause the ongoing work to be undone and compensated.

```
<process name="orderCar">
        ...
        <eventHandlers>
            <onEvent partnerLink="buyer"
               portType="ns:car"
               operation="cancel"
               messageType="ns:cancelOrder"
               variable="cancelDetails">
             <exit/>
            </onEvent>
             ...
        </eventHandlers>
        ...
</process>
```

In this example, if the buyer invokes the cancel operation on the port type car, the exit activity is carried out, which results in immediate termination of the process instance without the ongoing work being undone and compensated. And this event is attached to the global process scope and is therefore available during the lifetime of the entire business process instance.

13.5.2. Alarm events

The `onAlarm` tag marks a timeout event. The `for` expression specifies the duration after which the event will be signaled. The clock for the duration starts at the point in time when the associated scope starts. The alternative `until` expression specifies the specific point in time when the alarm will be fired. Only one of these two expressions may occur in any `onAlarm` event. The third alternate `repeatEvery` expression also specifies a duration. When the `repeatEvery` expression is specified, the alarm will be fired repeatedly once each time the duration period expires, while the associate scope is active. The `repeatEvery` expression may be specified on its own or with either the `for` or the `until` expression. If the `repeatEvery` expression is specified alone, the clock for the very first duration starts at the point in time when the associated scope starts. If the `repeatEvery` is expression is specified with either the `for` or the `until` expression, the first alarm is not fired until the time specified in the `for` or `until` expression expires; thereafter it is fired repeatedly at the interval specified by the `repeatEvery` expression.

13.5.3. Enablement of Events

The event handlers associated with a scope are enabled when the associated scope starts .

If the event handler is associated with the global process scope, the event handler is enabled as soon as the process instance is created. The process instance is created when the first `receive` activity that provides for the creation of a process instance (indicated via the `createInstance` attribute set to "yes") has received and processed the corresponding message. This allows the alarm time for a global alarm event to be specified using the data provided within the message that creates a process instance, as shown in the following example:

```
<wsdl:definitions
        targetNamespace="http://www.example.com/wsdl/exmple"
        ...>
        <wsdl:message name="orderDetails">
            <part name="processDuration" type="xsd:duration"/>
        </wsdl:message>
</wsdl:definitions>
```

The message type above is used in

```
<process name="orderCar"
        xmlns:def="http://www.example.com/wsdl/example" ...>
            ...
        <eventHandlers>
            <onAlarm>
              <for>$orderDetails.processDuration</for>
              ...
            </onAlarm>
            ...
        </eventHandlers>
        ...
```

```
        <variable name="orderDetails" messageType="def:orderDetails"/>
        </variable>
        ...
        <receive name="getOrder"
                partnerLink="buyer"
                portType="car"
                operation="order"
                variable="orderDetails"
                createInstance="yes"/>
        ....
</process>
```

The `onAlarm` tag specifies a timer event that is fired when the duration specified in the `processDuration` field in the `orderDetails` variable is exceeded. The value of the field is provided via the `getOrder` activity that receives message containing the order details and causes the creation of a process instance for that order.

13.5.4. Processing of Events

13.5.4.1. ALARM EVENTS

The counting of time for an alarm event with a duration starts when the enclosing event handler is activated. An alarm event goes off when the specified time or duration has been reached. Except for the repeatEvery alarm, an alarm event is carried out at most once while the corresponding scope is active; the event is disabled for the rest of the activity of the corresponding scope after it has occurred and the specified processing has been carried out. The repeatEvery alarm event is created repeatedly each time the duration expires, while the associated scope is active.

13.5.4.2. MESSAGE EVENTS

A message event occurs when the appropriate message is received on the specified partner link using the specified port type and operation. When such an event occurs, the corresponding activity is carried out. However, the event remains enabled, even for concurrent use. Thus a particular message event can occur multiple times while the corresponding scope is active. See below for concurrency considerations.

13.5.5. Disablement of Events

All event handlers associated with a scope are disabled when the normal processing of the scope is complete. The already dispatched event handlers are allowed to complete. The completion of the scope as a whole is delayed until all active event handlers have completed.

13.5.6. Fault Handling Considerations

As we stated above, event handlers are considered a part of the normal processing of the scope, i.e., active event handlers are concurrent activities within the scope. Faults within event handlers are therefore faults within the associated scope. Moreover, if a fault occurs

within a scope, the behavior of the fault handler begins by implicitly terminating all activities directly enclosed within the scope that are currently active. This includes the activities within currently active event handlers.

13.5.7. Concurrency Considerations

Multiple `onEvent` and `onAlarm` events can occur concurrently and they are treated as concurrent activities even if they are request/response events representing the same partner link, port type, operation and correlation sets. The constraint that there can be at most one outstanding synchronous request on a given partner link at a given port type and operation applies here as well (see **Providing Web Service Operations**). Concurrent invocation of event handlers necessarily relies heavily on the use of isolated scoping to ensure consistent access to shared variables.

Note that, unlike `onAlarm` event handlers, individual `onEvent` event handlers are permitted to have several simultaneously active instances. A private copy of all process data and control behavior defined within an event handler is provided to each instance of an event handler. This includes the behavior of links defined within an event handler. Each instance of the event handler must independently evaluate the status of the link as needed.

13.6. Isolated Scopes

When the `isolated` attribute is set to "yes", the scope provides concurrency control in governing access to shared variables. Such a scope is called a `isolated scope`. Isolated scopes must not be nested. A scope marked with `isolated="yes"` must not contain other isolated scopes, but may contain scopes that are not marked as isolated. In the latter case, access to shared variables from within such enclosed scopes is controlled by the enclosing isolated scope.

Suppose two concurrent isolated scopes, S1 and S2, access a common set of variables (external to them) for read or write operations. The semantics of isolated scopes ensure that the results of their behavior would be no different if all conflicting activities (read/write and write/write activities) on any shared variable were conceptually reordered in such a way that either all activities within S1 are completed before those in S2 or vice versa. The actual mechanisms used to ensure this are implementation dependent.

Note that isolation of variable access cannot lead to internal deadlock in a BPEL process instance. The reason being that, conceptually, an isolated scope is not started until it can gain sufficiently exclusive access to all the non-local variables it needs.

The use of error handling features in an isolated scope is governed by the following rules:

- The fault handlers for an isolated scope share the isolation domain of the associated scope, that is, in case a fault occurs in an isolated scope, the behavior of the fault handler is considered part of the isolated behavior (in commonly used

implementation terms, locks are not released when making the transition to the fault handler). This is because the repair of the fault needs a shared isolation environment to provide predictable behavior.
- The compensation handler for an isolated scope does not share the isolation domain of the associated scope.

For an isolated scope with a compensation handler, the creation of the scope snapshot for compensation is part of the isolated behavior. In other words, it is always possible to reorder behavior steps as if the scope had sufficiently exclusive access to the shared variables all the way to completion, including the creation of the scope snapshot. Note that the isolation semantics does *not* extend to the compensation handler. It is useful to note that the semantics of isolated scopes are very similar to the standard isolation level "serializable" used in database transactions.

Finally, if an isolated scope has an event handler associated with it, no scope within the body of the event handler can be isolated. If on the other hand a non-isolated scope has an event handler associated with it, a scope in the body of the event handler can be isolated.

14. Extensions for Executable Processes

In this section we define the essential extensions required for the use of WS-BPEL to define executable processes. The extensions are grouped by the core concepts to which they apply.

[Editor Note: The old section “14.1 Expressions” has been merged into section “9.1/9.3 Expressions”.]

14.1. Variables

These extensions apply to the **Variables** feature of WS-BPEL.

An attempt during process execution to use a variable or in the case of a message type variable a part of a variable before it is initialized MUST result in the standard `bpws:uninitializedVariable` fault.

14.2. Assignment

The from-spec and to-spec MUST yield a node-set that contains exactly one node. If the from-spec or to-spec selects zero nodes or more than one node during execution, then the standard fault bpws:selectionFailure MUST be thrown by a compliant implementation.

If any of the matching constraints defined in the section Type Compatibility in Assignment is violated during execution, the standard fault bpws:mismatchedAssignmentFailure MUST be thrown by a compliant implementation.

The assign activity is treated as if it were atomic. This means that the assign activity MUST be executed as if, for the duration of its execution, it was the only activity in the process being executed. The mechanisms used to implement the previous requirement are implementation dependent.If there is any fault during the execution of an assignment activity, the destination variables are left unchanged as they were at the start of the activity. This applies regardless of the number of assignment elements within the overall assignment activity.

14.3. Correlation

After a `correlation` set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set and occur within the corresponding scope until its completion. This **correlation consistency constraint** must be observed in all cases of `initiate` values. The legal values of the `initiate` attribute are: `"yes"`, `"join"`, `"no"`, where `"no"` is the default value.

- When the `initiate` attribute is set to `"yes"`, the related activity MUST attempt to initiate the correlation set.
    - If the correlation set is already initiated and the `initiate` attribute is set to `"yes"`, the standard fault `bpws:correlationViolation` MUST be thrown by a compliant implementation.
- When the `initiate` attribute is set to `"join"`, the related activity MUST attempt to initiate the correlation set, if the correlation set is NOT initiated yet.
    - If the correlation set is already initiated and the `initiate` attribute is set to `"join"`, the correlation consistency constraint MUST be observed. If the constraint is violated, the standard fault `bpws:correlationViolation` MUST be thrown by a compliant implementation.
- When the `initiate` attribute is set to `"no"`, the related activity MUST NOT attempt to initiate the correlation set.
    - If an activity with the `initiate` attribute set to `"no"` attempts to use a correlation set that has not been previously initiated, the standard fault `bpws:correlationViolation` MUST be thrown by a compliant implementation.
    - If the correlation set is already initiated and the `initiate` attribute is set to `"no"`, the correlation consistency constraint MUST to be observed. If the constraint is violated, the standard fault `bpws:correlationViolation` MUST be thrown by a compliant implementation.

If multiple correlation sets are used in a message activity, then the consistency constraint MUST be observed for all correlation sets used. If one of initiated correlation set does

NOT match with the message, the standard fault `bpws:correlationViolation` MUST be thrown by a compliant implementation.

14.4. Web Service Operations

The first extension defines a standard fault for the case where multiple conflicting receive activities create ambiguity about message delivery.

If during the execution of a business process instance, two or more receive activities for the same partner link, portType, operation and correlation set(s) are in fact simultaneously enabled, then the standard fault `bpws:conflictingReceive` MUST be thrown by a compliant implementation.

The second extension defines a standard fault for the case where multiple outstanding synchronous requests create an ambiguity about response correlation.

If more than one outstanding synchronous request on a specific partner link for a particular portType, operation and correlation set(s) is outstanding simultaneously during the execution of a business process instance, then the standard fault `bpws:conflictingRequest` MUST be thrown by a compliant implementation. Note that this is semantically different from the `bpws:conflictingReceive`, because it is possible to create the `conflictingRequest` by consecutively receiving the same request on a specific partner link for a particular portType, operation and correlation set(s). If a `reply` activity is being carried out during the execution of a business process instance and no synchronous request is outstanding for the specified partnerLink, portType, operation and correlation set(s), then the standard fault `bpws:invalidReply` MUST be thrown by a compliant implementation.

The third extension specifies that the `inputVariable` attribute for invoke and the `variable` attribute for `receive` and `reply` activities are not optional in executable processes. In addition, the `outputVariable` attribute is not optional for `invoke`when the operation concerned is a request/response operation.

The fourth extension concerns incompleted inbound requests. A receive activity for an inbound request/response operation is said to be *open* if that activity has been performed and no corresponding reply activity has been performed. If the process instance reaches the end of its behavior, and one or more receive activities remain open, then the standard fault `bpws:missingReply` MUST be thrown by a compliant implementation.

14.5. Terminating a Service Instance

The `exit` activity can be used to immediately terminate the behavior of a business process instance within which the `exit` activity is performed. All currently running

activities MUST be terminated as soon as possible without any termination handling, fault handling, or compensation behavior.

```
<exit standard-attributes>
       standard-elements
</exit>
```

Deleted: 14.6. Compensation¶
If an installed compensation handler is invoked more than once during the execution of a process instance, a compliant implementation MUST throw the standard `bpws:repeatedCompensation` fault. ¶

14.7. Event Handlers

This extension explains the relationship of `onEvent` event handlers to the standard fault extension in **Web Service Operations** for multiple conflicting receive activities create ambiguity about message delivery Enablement of an `onEvent` event handler is equivalent to enablement of a receive activity for the semantics of the occurrence of the `bpws:conflictingReceiveFault` fault (see **Providing Web Service Operations**).

The `variable` (inputVariable) attribute of `onEvent` handlers is not optional. In addition, the `outputVariable` attribute is not optional for `invoke` when the operation concerned is a request/response operation.

15. Extensions for Business Protocols

There are two extensions for the business protocol usage pattern.

15.1. Variables

This extension clarifies the rules regarding variable initialization in abstract processes. Unlike executable processes, variables in abstract processes do not need to be fully initialized before being used since some computation is left implicit in abstract processes. However, since message properties are meant to represent "transparent," i.e., protocol relevant data, WS-BPEL requires that all message properties in a message must be initialized before the message can be used, for example before the variable of the message is used as the `inputVariable` in a Web Service operation invocation.

In many cases, the level of abstraction appropriate in abstract processes makes it unnecessary to use message variables in web service interaction activities, when the intent is to simply constrain the sequencing of such activities, and the actual message data is not relevant. To simplify these common cases it is permissible, in abstract processes, to omit the variable reference attributes from the <invoke/>, <receive/>, and <reply/> activities. The meaning of such an omission must be stated clearly. If no variable is specified for an incoming message, then the abstract process may not refer subsequently to the message or its properties (if any). If the variable reference is omitted for an outgoing message, then any properties of the message are considered to have been initialized through opaque assignment, as described in the following section.

When variable references are omitted, correlation set references may be interpreted as follows:

- For an incoming message which initializes a correlation set (initiator case), the correlation set is deemed to be initialized.
- For an outgoing message which initializes a correlation set (initiator case), the correlation tokens (which are message properties) are initialized through implicit opaque assignment as described above.
- For an outgoing message which references but does not initialize a correlation set (follower case), the proper initialization of the message properties is implicit. In this case, the already initialized correlation set itself provides the token values for the outgoing message.

Note that it is not possible to mix the variable-using and variable-less web service interaction styles freely. If a correlation set is initialized by rule 1 or 2 above, then outgoing messages in the same correlated exchange must also refrain from referencing a message variable. This restriction applies because it is not possible to initialize the properties of the outgoing messages from the correlation set alone.

15.2. Assignment

This extension adds a special form of assignment to abstract processes to permit the modeling of the non-deterministic effects of private computation on external protocol behavior.

Abstract processes add a sixth `from-spec` variant to allow an opaque value to be assigned based on non-deterministic choice, yielding the form:

```
<from opaque="yes">
```

The value of this form in the interpretation of assignment is chosen nondeterministically from the XSD value space of the target. It can only be used in assignments where the "tospec" refers to a variable property. Two distinct use cases exist for opaque assignment. If the value space of the target is suitably constrained, then opaque assignment is a useful way to describe behavioral alternatives where the mechanism for choosing the alternative is private or otherwise external to the process specification. For this use case, the XSD type of the target property must be one of the following:

- `xsd:boolean`
- A type derived from `xsd:string` and restricted by enumeration
- A type derived from any XSD integral numeric type restricted by either enumeration or a combination of minExclusive or minInclusive and maxExclusive or maxInclusive

A second use cases exists for target properties which don’t meet these requirements. When the target’s value space is not constrained, it is useful to think of opaque assignment as providing a unique identifier. Semantically, each opaque assignment of this form should be considered to generate a unique value similar to a GUID. This style

of opaque assignment is most useful to model the initialization of properties used for correlation.

A process that uses assignment of opaque values is clearly not executable in the normal sense. However, it is feasible to emulate possible execution traces using assignment of random values of the correct type.

16. Examples

16.1. Shipping Service

This example presents the use of a WS-BPEL abstract process to describe a rudimentary shipping service. This service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The shipping service offers two types of shipment: shipments where the items are held and shipped together and shipment where the items are shipped piecemeal until all of the order is accounted for.

16.1.1. Service Description

The context for the shipping service is a two-party interaction between a customer and the service. This is modeled in the following `partnerLinkType` definition:

```
<plnk:partnerLinkType name="shippingLT">
        <plnk:role name="shippingService"
                   portType ="shippingServicePT"/>

        <plnk:role name="shippingServiceCustomer">
                   portType ="shippingServiceCustomerPT"/>

</plnk:partnerLinkType>
```

The corresponding message and portType definitions are as follows:

```
<wsdl:definitions
        targetNameSpace="http://ship.org/wsdl/shipping"
        xmlns:ship= ...>

        <! -- import ship schema -- >
        <message name="shippingRequestMsg">
              <part name="shipOrder" type="ship:shipOrder"/>
        </message>
        <message name="shippingNoticeMsg">
             <part name="shipNotice" type="ship:shipNotice"/>
        </message>
        <portType name="shippingServicePT">
           <operation name="shippingRequest">
                <input message="shippingRequestMsg"/>
            </operation>
        </portType>
        <portType name="shippingServiceCustomerPT">
```

```
            <operation name="shippingNotice">
               <input message="shippingNoticeMsg"/>
            </operation>
        </portType>
</wsdl:definitions>
```

16.1.2. Message Properties

The properties relevant to the service behavior are:

- The ship order ID that is used to correlate the ship notice(s) with the ship order (`shipOrderID`)
- Whether the order is to be shipped complete or not (`shipComplete`)
- The total number of items in the order (`itemsTotal`)
- The number of items referred to in a ship notice so that, when partial shipments are acceptable, we can use this, along with `itemsTotal`, to track the overall fulfillment of the shipment (`itemsCount`)

Here are the definitions for the properties and their aliases:

```
<wsdl:definitions
        targetNamespace="http://example.com/shipProps/"
        xmlns:sns="http://ship.org/wsdl/shipping"
        xmlns:ship = "http://example.com/ship.xsd">

        <!-- types used in abstract processes are required to be finite
domains.
        The itemCountType is restricted by range -->
        <wsdl:types>
             <xsd:schema targetNamespace ="http://example.com/ship.xsd">
                <xsd:simpleType name="itemCountType">
                     <xsd:restriction base="xsd:int">
                        <xsd:minInclusive value="1"/>
                        <xsd:maxInclusive value="50"/>
                    </xsd:restriction>
                </xsd:simpleType>
             </xsd:schema>
        </wsdl:types>

        <bpws:property name="shipOrderID" type="xsd:int"/>
        <bpws:property name="shipComplete" type="xsd:boolean"/>
        <bpws:property name="itemsTotal" type="ship:itemCountType"/>
        <bpws:property name="itemsCount" type="ship:itemCountType"/>
        <bpws:property name="numItemsShipped"
type="ship:itemCountType"/>

<bpws:propertyAlias propertyName="tns:shipOrderID"
        messageType="sns:shippingRequestMsg"
        part="shipOrder">
        <query>
         ShipOrderRequestHeader/shipOrderID
         </query>
</bpws:propertyAlias>
```

```
<bpws:propertyAlias propertyName="tns:shipOrderID"
        messageType="sns:shippingNoticeMsg"
        part="shipNotice">
        <query>
         ShipNoticeHeader/shipOrderID
         </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="tns:shipComplete"
        messageType="sns:shippingRequestMsg"
        part="shipOrder">
        <query>
         ShipOrderRequestHeader/shipComplete
         </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="tns:itemsTotal"
        messageType="sns:shippingRequestMsg"
        part="shipOrder">
        <query>
         ShipOrderRequestHeader/itemsTotal
         </query>
</bpws:propertyAlias>
<bpws:propertyAlias propertyName="tns:itemsCount"
        messageType="sns:shippingNoticeMsg"
        part="shipNotice">
        <query>
         ShipNoticeHeader/itemsCount
         </query>
</bpws:propertyAlias>
</wsdl:definitions>
```

16.1.3. Process

Next is the process definition. For brevity, the abstract process definition does not include, for example, the handling of error conditions (business or otherwise) that a complete description would account for. The rough outline of the process is as follows:

```
receive shipOrder
if
        condition shipComplete
             send shipNotice
        else
             itemsShipped := 0
             while itemsShipped < itemsTotal
                itemsCount := opaque // non-deterministic assignment
                                     // corresponding e.g. to
                                     // internal interaction with
                                     // back-end system
                send shipNotice
                itemsShipped = itemsShipped + itemsCount
```

And here is the more complete version:

```
<process name="shippingService"
        targetNamespace="http://acme.com/shipping"
```

```
       xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
       xmlns:sns="http://ship.org/wsdl/shipping"
       xmlns:props="http://example.com/shipProps/"
       abstractProcess="yes">

<partnerLinks>
     <partnerLink name="customer"
       partnerLinkType="sns:shippingLT"
       partnerRole="shippingServiceCustomer"
       myRole="shippingService"/>
</partnerLinks>

<variables>
    <variable name="shipRequest"
       messageType="sns:shippingRequestMsg"/>
    <variable name="shipNotice"
       messageType="sns:shippingNoticeMsg"/>
    <variable name="itemsShipped"
       type="props:itemCountType"/>
</variables>

<correlationSets>
    <correlationSet name="shipOrder"
       properties="props:shipOrderID"/>
</correlationSets>

<sequence>
     <receive partnerLink="customer"
       portType="sns:shippingServicePT"
       operation="shippingRequest"
       variable="shipRequest">
       <correlations>
          <correlation set="shipOrder" initiate="yes"/>
       </correlations>
    </receive>
    <if>

      <condition>
bpws:getVariableProperty('shipRequest','props:shipComplete')
      </condition>
        <then>
          <sequence>
               <assign>
                  <copy>
                     <from variable="shipRequest"
property="props:itemsCount"/>
                     <to variable="shipNotice"
property="props:itemsCount"/>
                  </copy>
               </assign>
               <invoke partnerLink="customer"
                       portType="sns:shippingServiceCustomerPT"
                       operation="shippingNotice"
                       inputVariable="shipNotice">
                    <correlations>
                       <correlation set="shipOrder" pattern="out"/>
                    </correlations>
```

```
               </invoke>
            </sequence>
         </then>
         <else>
            <sequence>
                 <assign>
                    <copy>
                         <from>0</from>
                         <to>$itemsShipped</to>
                   </copy>
                 </assign>
                 <while>
                    <condition>
                         $itemsShipped &lt;
         bpws:getVariableProperty('shipRequest','props:itemsTotal')
                    </condition>
                         <sequence>
                            <assign>
                                 <copy>
                                   <from opaque="yes"/>
                                   <to variable="shipNotice"
property="props:itemsCount"/>
                                 </copy>
                         </assign>
                         <invoke partnerLink="customer"
                                 portType="sns:shippingServiceCustomerPT"
                                 operation="shippingNotice"
                                 inputVariable="shipNotice">
                                 <correlations>
                                    <correlation set="shipOrder"
pattern="out"/>
                                 </correlations>
                         </invoke>
                         <assign>
                             <copy>
                                 <from>
                                 $itemsShipped
                                 +
                                 bpws:getVariableProperty('shipNotice',
                                 'props:itemsCount')
                                 </from>
                                 <to>$itemsShipped</to>
                            </copy>
                         </assign>
                    </sequence>
                 </while>
             </sequence>
           </else>
         </if>
</sequence>
</process>
```

16.2. Loan Approval

This example considers a simple loan approval Web Service that provides a port where customers can send their requests for loans. Customers of the service send their loan requests, including personal information and amount being requested. Using this information, the loan service runs a simple process that results in either a "loan approved" message or a "loan rejected" message. The approval decision can be reached in two different ways, depending on the amount requested and the risk associated with the requester. For low amounts (less than $10,000) and low-risk individuals, approval is automatic. For high amounts or medium and high-risk individuals, each credit request needs to be studied in greater detail. Thus, to process each request, the loan service uses the functionality provided by two other services. In the streamlined processing available for lowamount loans, a "risk assessment" service is used to obtain a quick evaluation of the risk associated with the requesting individual. A full-fledged "loan approval" service (possibly requiring direct involvement of a loan expert) is used to obtain in-depth assessments of requests when the streamlined approval process does not apply.

16.2.1. Service Description

The WSDL portType supported by this service is shown below ("loanServicePT" portType). It is assumed that an independent "loan.org" consortium has provided definitions of the loan service portType as well as the risk assessment and in-depth loan approval service, so all the required WSDL definitions appear in the same WSDL document. In particular, the portTypes for the Web Services providing the risk assessment and approval functions, and all the required partner link types that relate to the use of these portTypes, are also defined there.

```
<definitions
        targetNamespace="http://loans.org/wsdl/loan-approval"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:lns=http://loans.org/wsdl/loan-approval
        xmlns:ens = "http://loans.org/xsd/error-messages" >

        <! -- import schemas -- >
        <message name="creditInformationMessage">
             <part name="firstName" type="xsd:string"/>
             <part name="name" type="xsd:string"/>
             <part name="amount" type="xsd:integer"/>
        </message>

        <message name="approvalMessage">
             <part name="accept" type="xsd:string"/>
        </message>

        <message name="riskAssessmentMessage">
            <part name="level" type="xsd:string"/>
        </message>
        <message name="errorMessage">
             <part name="errorCode" element ="ens:integer"/>
        </message>

        <portType name="loanServicePT">
            <operation name="request">
```

```
            <input message="lns:creditInformationMessage"/>
            <output message="lns:approvalMessage"/>
            <fault name="unableToHandleRequest"
                message="lns:errorMessage"/>
        </operation>
    </portType>

    <portType name="riskAssessmentPT">
        <operation name="check">
            <input message="lns:creditInformationMessage"/>
            <output message="lns:riskAssessmentMessage"/>
            <fault name="loanProcessFault"
                message="lns:errorMessage"/>
        </operation>
    </portType>

    <portType name="loanApprovalPT">
       <operation name="approve">
            <input message="lns:creditInformationMessage"/>
            <output message="lns:approvalMessage"/>
            <fault name="loanProcessFault"
                message="lns:errorMessage"/>
       </operation>
    </portType>

    <plnk:partnerLinkType name="loanPartnerLinkType">
        <plnk:role name="loanService"
                   portType="lns:loanServicePT"/>

    </plnk:partnerLinkType>

    <plnk:partnerLinkType name="loanApprovalLinkType">
       <plnk:role name="approver"
                  portType="lns:loanApprovalPT"/>

    </plnk:partnerLinkType>

    <plnk:partnerLinkType name="riskAssessmentLinkType">
       <plnk:role name="assessor"
                   portType="lns:riskAssessmentPT"/>

    </plnk:partnerLinkType>
</definitions>
```

16.2.2. Process

In the business process defined below, the interaction with the customer is represented by the initial <receive> and the matching <reply> activities. The use of risk assessment and loan approval services is represented by <invoke> elements. All these activities are contained within a <flow>, and their (potentially concurrent) behavior is staged according to the dependencies expressed by corresponding <link> elements. Note that the transition conditions attached to the <source> elements of the links determine which links get activated. Dead path elimination is enabled by the value "yes" taken by the "suppressJoinFailure" attribute on the <process> element. This implies that as certain

links are set false the consequences of this decision can be propagated and the excecution of certain activities can be skipped.

Because the operations invoked can return a fault of type "loanProcessFault", a fault handler is provided. When a fault occurs, control is transferred to the fault handler, where a <reply> element is used to return a fault response of type "unableToHandleRequest" to the loan `requester`.

```
<process name="loanApprovalProcess"
        targetNamespace="http://acme.com/loanprocessing"
        xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
        xmlns:lns="http://loans.org/wsdl/loan-approval"
        suppressJoinFailure="yes">

        <partnerLinks>
           <partnerLink name="customer"
                partnerLinkType="lns:loanPartnerLinkType"
                myRole="loanService"/>
          <partnerLink name="approver"
                partnerLinkType="lns:loanApprovalLinkType"
                partnerRole="approver"/>
          <partnerLink name="assessor"
                partnerLinkType="lns:riskAssessmentLinkType"
                partnerRole="assessor"/>
        </partnerLinks>
        <variables>
            <variable name="request"
                messageType="lns:creditInformationMessage"/>
            <variable name="risk"
                messageType="lns:riskAssessmentMessage"/>
           <variable name="approval"
                messageType="lns:approvalMessage"/>
            <variable name="error"
                messageType="lns:errorMessage"/>
        </variables>

        <faultHandlers>
            <catch faultName="lns:loanProcessFault"
                   faultVariable="error"
                   faultMessageType="lns:errorMessage">
                   <reply partnerLink="customer"
                       portType="lns:loanServicePT"
                       operation="request"
                       variable="error"
                       faultName="unableToHandleRequest"/>
               </catch>
        </faultHandlers>

        <flow>
            <links>
                <link name="receive-to-assess"/>
                <link name="receive-to-approval"/>
                <link name="approval-to-reply"/>
                <link name="assess-to-setMessage"/>
                <link name="setMessage-to-reply"/>
```

```
        <link name="assess-to-approval"/>
    </links>

    <receive partnerLink="customer"
        portType="lns:loanServicePT"
        operation="request"
        variable="request" createInstance="yes">
        <sources>
            <source linkName="receive-to-assess">
                <transitionCondition>
                $request.amount &lt; 10000
                </transitionCondition>
            </source>
            <source linkName="receive-to-approval">
               <transitionCondition>
                 $request.amount >=10000
               </transitionCondition>
            </source>
        </sources>

    </receive>

    <invoke partnerLink="assessor"
        portType="lns:riskAssessmentPT"
                operation="check"
                inputVariable="request"
                outputVariable="risk">
        <targets>
            <target linkName="receive-to-assess"/>
        </targets>
        <sources>
             <source linkName="assess-to-setMessage">
                <transitionCondition>
                   $risk.level='low'
                </transitionCondition>
            </source>
             <source linkName="assess-to-approval">
                <transitionCondition>
                   $risk.level!='low'
                </transitionCondition>
            </source>
        </sources>
    </invoke>

    <assign>
        <targets>
             <target linkName="assess-to-setMessage"/>
        </targets>
        <sources>
            <source linkName="setMessage-to-reply"/>>
        </sources>

        <copy>
            <from><expression>'yes'</expression></from>
            <to variable="approval" part="accept"/>
        </copy>
    </assign>
```

```
            <invoke partnerLink="approver"
                   portType="lns:loanApprovalPT"
                   operation="approve"
                   inputVariable="request"
                   outputVariable="approval">
                <targets>
                    <target linkName="receive-to-approval"/>
                    <target linkName="assess-to-approval"/>
                </targets>
                <sources>
                    <source linkName="approval-to-reply" />
                </sources>
            </invoke>

            <reply partnerLink="customer"
                   portType="lns:loanServicePT"
                   operation="request"
                  variable="approval">
                <targets>
                    <target linkName="setMessage-to-reply"/>
                    <target linkName="approval-to-reply"/>
                </targets>
            </reply>
        </flow>
</process>
```

16.3. Multiple Start Activities

A process can have multiple activities that create a process instance. An example of this situation is a (simplified) business process run by an auction house. The purpose of the business process is to collect information from the buyer and the seller of a particular auction, report the appropriate auction results to some auction registration service, and then send the registration result back to the seller and the buyer. Thus the business process starts with two activities, one for receiving the seller information and one for receiving the buyer information. Because a particular auction is uniquely identified by an auction ID, the seller and the buyer need to provide this information when sending in their data. The sequence in which the seller and buyer requests arrive at the auction house is random. Thus, when such a request comes in, it needs to be checked whether a business process instance exists already or not. If not, a business process instance is created. After both requests have been received, the auction registration service is invoked. Because the invocation is done asynchronously, the auction house passes the auction ID to the auction registration service. The auction registration service returns this auction ID in its answer so that the auction house can locate the proper business process instance. Because there are many buyers and sellers, each of them needs to provide their endpoint references, so that the auction service can respond properly. In addition, the auction house needs to provide its own endpoint reference to the auction registration service so that the auction registration service can send the response back to the auction house.

16.3.1. Service Description

The auction service offers two port types, called sellerPT and buyerPT, with appropriate operations for accepting the data provided by the seller and the buyer. Because the processing time of the business process is lengthy, the auction service responds to the seller and buyer through appropriate port types, sellerAnswerPT and buyerAnswerPT. These portTypes are properly combined into two partner link types, one for the seller called sellerAuctionHouseLT and one for the buyer called buyerAuctionHouseLT.

The auction service needs two port types, called auctionRegistrationPT and auctionRegistrationAnswerPT, that provide for the invocation of the auction registration service. The port types are part of the appropriate partner link type auctionHouseAuctionRegistrationServiceLT.

```
<definitions
       targetNamespace="http://www.auction.com/wsdl/auctionService"
       xmlns:tns="http://www.auction.com/wsdl/auctionService"
       xmlns="http://schemas.xmlsoap.org/wsdl/">

<!-- Messages for communication with the seller -->

       <message name="sellerData">
            <part name="creditCardNumber" type="xsd:string"/>
            <part name="shippingCosts" type="xsd:integer"/>
            <part name="auctionId" type="xsd:integer"/>
            <part name="endpointReference" type="bpws:ServiceRefType"/>
       </message>
       <message name="sellerAnswerData">
            <part name="thankYouText" type="xsd:string"/>
       </message>

<!-- Messages for communication with the buyer -->

       <message name="buyerData">
            <part name="creditCardNumber" type="xsd:string"/>
           <part name="phoneNumber" type="xsd:string"/>
           <part name="ID" type="xsd:integer"/>
           <part name="endpointReference" type="bpws:ServiceRefType"/>
       </message>
       <message name="buyerAnswerData">
            <part name="thankYouText" type="xsd:string"/>
       </message>

<!-- Messages for communication with the auction registration service -
->

       <message name="auctionData">
           <part name="auctionId" type="xsd:integer"/>
           <part name="amount" type="xsd:integer"/>
       </message>
       <message name="auctionAnswerData">
           <part name="registrationId" type="xsd:integer"/>
           <part name="auctionId" type="xsd:integer"/>
           <part name="auctionHouseEndpointReference"
              type="bpws:ServiceRefType"/>
       </message>
```

```
<!-- Port types for interacting with the seller -->

       <portType name="sellerPT">
           <operation name="submit">
              <input message="tns:sellerData"/>
           </operation>
       </portType>
       <portType name="sellerAnswerPT">
           <operation name="answer">
              <input message="tns:sellerAnswerData"/>
           </operation>
       </portType>

<!-- Port types for interacting with the buyer -->

       <portType name="buyerPT">
            <operation name="submit">
              <input message="tns:buyerData"/>
            </operation>
       </portType>

       <portType name="buyerAnswerPT">
           <operation name="answer">
              <input message="tns:buyerAnswerData"/>
           </operation>
       </portType>

<!-- Port types for interacting with the auction registration service -
->

       <portType name="auctionRegistrationPT">
           <operation name="process">
              <input message="tns:auctionData"/>
           </operation>
       </portType>

       <portType name="auctionRegistrationAnswerPT">
           <operation name="answer">
              <input message="tns:auctionAnswerData"/>
           </operation>
       </portType>

<!-- Context type used for locating business process via auction Id -->
       <bpws:property name="auctionId"
              type="xsd:string"/>
       <bpws:propertyAlias propertyName="tns:auctionId"
              messageType="tns:sellerData"
              part="auctionId"/>
       <bpws:propertyAlias propertyName="tns:auctionId"
              messageType="tns:buyerData"
              part="ID"/>
       <bpws:propertyAlias propertyName="tns:auctionId"
              messageType="tns:auctionData"
              part="auctionId"/>
       <bpws:propertyAlias propertyName="tns:auctionId"
              messageType="tns:auctionAnswerData"
```

```
               part="auctionId"/>

<!-- Partner link type for seller/auctionHouse -->

        <plnk:partnerLinkType name="tns:sellerAuctionHouseLT">
               <plnk:role name="auctionHouse"
                          portType="tns:sellerPT"/>

               <plnk:role name="seller"
                           portType="tns:sellerAnswerPT"/>

        </plnk:partnerLinkType>

<!-- Partner link type for buyer/auctionHouse -->

        <plnk:partnerLinkType name="buyerAuctionHouseLT">
               <plnk:role name="auctionHouse"
                      portType="tns:buyerPT"/>

               <plnk:role name="buyer"
                           portType="tns:buyerAnswerPT"/>

        </plnk:partnerLinkType>

<!-- Partner link type for auction house/auction
registration service -->

        <plnk:partnerLinkType
name="auctionHouseAuctionRegistrationServiceLT">
               <plnk:role name="auctionRegistrationService"
                          portType="tns:auctionRegistrationPT"/>

               <plnk:role name="auctionHouse"
                     portType="tns:auctionRegistrationAnswerPT"/>

        </plnk:partnerLinkType>

</definitions>
```

16.3.2. Process

The WS-BPEL definition for the business process offered by the auction house follows:

```
<process name="auctionService"
        targetNamespace="http://www.auction.com"
        isolated="no"
        xmlns:as="http://www.auction.com/wsdl/auctionService"
        xmlns:addr="http://www.some.org/addressing"
        xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">

<!-- Partners -->

        <partnerLinks>
             <partnerLink name="seller"
               partnerLinkType="as:sellerAuctionHouseLT"
               myRole="auctionHouse" partnerRole="seller"/>
```

```
            <partnerLink name="buyer"
               partnerLinkType="as:buyerAuctionHouseLT"
               myRole="auctionHouse" partnerRole="buyer"/>
            <partnerLink name="auctionRegistrationService"
        partnerLinkType=
               "as:auctionHouseAuctionRegistrationServiceLT"
               myRole="auctionHouse"
               partnerRole="auctionRegistrationService"/>
        </partnerLinks>

<!-- Variables -->

        <variables>
               <variable name="sellerData"
messageType="as:sellerData"/>
               <variable name="sellerAnswerData"
messageType="as:sellerAnswerData"/>
               <variable name="buyerData" messageType="as:buyerData"/>
               <variable name="buyerAnswerData"
messageType="as:buyerAnswerData"/>
               <variable name="auctionData"
                      messageType="as:auctionData"/>
               <variable name="auctionAnswerData"
                      messageType="as:auctionAnswerData"/>
        </variables>

<!-- Correlation set for correlating buyer and seller request
as well as auction house and auction registration service exchange -->

        <correlationSets>
            <correlationSet name="auctionIdentification"
                 properties="as:auctionId"/>
        </correlationSets>

<!-- Structure of the business process -->

        <sequence>

<!-- Process buyer and seller request concurrently
Either one can create a process instance -->
            <flow>

<!-- Process seller request -->
               <receive name="acceptSellerInformation"
                      partnerLink="seller"
                      portType="as:sellerPT"
                      operation="provide"
                      variable="sellerData"
                      createInstance="yes">
                   <correlations>
                      <correlation set="auctionIdentification"
                          initiate="join"/>
                   </correlations>
               </receive>

<!-- Process buyer request -->
```

```
               <receive name="acceptBuyerInformation"
                      partnerLink="buyer"
                      portType="as:buyerPT"
                      operation="provide"
                      variable="buyerData"
                      createInstance="yes">
                    <correlations>
                      <correlation set="auctionIdentification"
                          initiate="join"/>
                    </correlations>
               </receive>
       </flow>

<!-- Invoke auction registration service by setting the target endpoint
reference  and setting my own endpoint reference for call back
and receiving the answer Correlation of request and answer is
via auction Id -->

       <assign>
           <copy>
               <from>
                 <literal>
                  <service-ref>
                   <addr:EndpointReference>
                          <addr:Address>xs:anyURI</addr:Address>

<addr:ServiceName>as:RegistrationService</addr:ServiceName>
                   </addr:EndpointReference>
                  </service-ref>
                 </literal>
               </from>
               <to partnerLink="auctionRegistrationService"/>
            </copy>
       </assign>

       <assign>
              <copy>
                  <from partnerLink="auctionRegistrationService"
                      endpointReference="myRole"/>
                 <to>$auctionData.auctionHouseServiceRef</to>
              </copy>
       </assign>

       <invoke name="registerAuctionResults"
              partnerLink="auctionRegistrationService"
              portType="as:auctionRegistrationPT"
              operation="process"
              inputVariable="auctionData">
            <correlations>
              <correlation set="auctionIdentification"/>
            </correlations>
       </invoke>

       <receive name="receiveAuctionRegistrationInformation"
              partnerLink="auctionRegistrationService"
              portType="as:auctionRegistrationAnswerPT"
              operation="answer"
```

```
              variable="auctionAnswerData">
            <correlations>
             <correlation set="auctionIdentification"/>
            </correlations>
        </receive>

<!-- Send responses back to seller and buyer -->

        <flow>

<!-- Process seller response by setting the seller to
the endpoint reference provided by the seller
and invoking the response -->

            <sequence>
                <assign>
                    <copy>
                        <from>$sellerDataendpointReference</from>
                        <to partnerLink="seller"/>
                    </copy>
                </assign>

                <invoke name="respondToSeller"
                        partnerLink="seller"
                        portType="as:sellerAnswerPT"
                        operation="answer"
                        inputVariable="sellerAnswerData"/>
           </sequence>

<!-- Process buyer response by setting the buyer to
the endpoint reference provided by the buyer
and invoking the response -->

             <sequence>
                <assign>
                    <copy>
                        <from>$buyerData.endpointReference</from>
                        <to partnerLink="buyer"/>
                    </copy>
                </assign>

                <invoke name="respondToBuyer"
                     partnerLink="buyer"
                     portType="as:buyerAnswerPT"
                     operation="answer"
                     inputVariable="buyerAnswerData"/>
             </sequence>
        </flow>
    </sequence>
</process>
```

17. Security Considerations

Because messages can be modified or forged, it is strongly RECOMMENDED that business process implementations use WS-Security to ensure messages have not been

modified or forged while in transit or while residing at destinations. Similarly, invalid or expired messages could be re-used or message headers not specifically associated with the specific message could be referenced. Consequently, when using WS-Security, signatures MUST include the semantically significant headers and the message body (as well as any other relevant data) so that they cannot be independently separated and re-used.

Messaging protocols used to communicate among business processes are subject to various forms of replay attacks. In addition to the mechanisms listed above, messages SHOULD include a message timestamp (as described in WS-Security) within the signature. Recipients can use the timestamp information to cache the most recent messages for a business process and detect duplicate transmissions and prevent potential replay attacks.

It should also be noted that business process implementations are subject to various forms of denial-of-service attacks. Implementers of business process execution systems compliant with this specification should take this into account.

A. Standard Faults

The following list specifies the standard faults defined within the WS-BPEL specification. All these faults are named within the WS-BPEL namespace standard prefix `bpws`: corresponding to URI "http://schemas.xmlsoap.org/ws/2004/03/business-process/".

Table A.1. Standard Faults

Fault name	Reason
selectionFailure	Thrown when a selection operation performed either in a function such as bpws:getVariableProperty, or in an assignment, encounters an error.
conflictingReceive	Thrown when more than one receive activity or equivalent (currently, onMessage branch in a pick activity) are enabled simultaneously for the same partner link, port type, operation and correlation set(s).
conflictingRequest	Thrown when more than one synchronous inbound request on the same partner link for a particular port type, operation and correlation set(s) are active.
mismatchedAssignmentFailure	Thrown when incompatible types are encountered in an assign activity.
joinFailure	Thrown when the join condition of an activity evaluates to false.
correlationViolation	Thrown when the contents of the messages that are processed in an invoke, receive, or reply activity do not

Fault name	Reason
	match specified correlation information.
uninitializedVariable	Thrown when there is an attempt to access the value of an uninitialized variable or in the case of a message type variable one of its uninitialized parts.
invalidReply	Thrown when a reply is sent on a partner link, portType and operation for which the corresponding receive with the same correlation has not been carried out.
missingReply	Thrown when a receive has been executed, and the process instance reaches the end of its execution without a corresponding reply having been executed.
missingRequest	Thrown when a reply activity cannot be associated with an incomplete receive activity by matching the partner link, operation and messageExchange tuple.
subLanguageExecutionFault	Thrown when the execution of an expression results in an unhandled expression / query language execution fault.
unsupportedReference	Thrown when a BPEL implementation fails to interpret the combination of the "reference-scheme" attribute and the content element OR just the content element alone
invalidVariables	Thrown when any XML validation (implicit or explicit: e.g. `<validate>` or `<assign validate="yes">`) fails.
uninitializedPartnerRole	Thrown when an Invoke activity is used on a partnerLink whose partnerRole endpoint reference is not initialized.
bpws:scopeInitializationFailure	Thrown if there is any problem creating any of the objects defined as part of scope initialization. This fault is always caught by the parent scope of the faulted scope.

Deleted: repeatedCompensation ... [1]

B. Attributes and Defaults

The following list specifies the defaults for all standard attributes at the process and activity level. The table does not include activity-specific attributes (such as `partnerLink` in an invoke activity).

Table B.1. Attributes and Defaults

Parameter	Default
queryLanguage	urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0

Parameter	Default
expressionLanguage	urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0
suppressJoinFailure	no — but for process element only. When this attribute is not specified for an activity, it inherits its value from its closest enclosing activity, or from the process if no enclosing activity specifies this attribute.
isolated	no
abstractProcess	no
initiate	no
pattern	No default
createInstance	no
joinCondition	Disjunction of the status of the incoming links
transitionCondition	true

C. XSD Schemas

WS-BPEL Schema

[Editor Note: Sept 01, 2004: Before the finalization of the specification, if there are any discrepancies (caused by manual mistakes) between the XSD Schemas Appendix of the specification and standalone XSD files, the standalone XSD files would be the primary source of truth. We copy XML Schemas and their changes into this appendix mainly for the convenience of reviewers.**]**

[Editor Note: Jun 23, 2005: To make editing of XSD easier and less error-prone for inconsistencies, we temporarily remove the XSD from this document. The master copy will be maintained in the CVS, until we are ready to release the next draft.**]**

```
<?xml version="1.0" encoding="UTF-8"?>
```

Partner Link Type Schema

```
<?xml version='1.0' encoding="UTF-8"?>
```

Message Properties Schema

```
<?xml version='1.0' encoding="UTF-8"?>
```

D. 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 implementors 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.

OASIS has been notified of intellectual property rights claimed in regard to some or all of the contents of this specification. For more information consult the online list of claimed rights.

E. Intellectual Property Rights

For information on wether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the WS-BPEL Technical Committee web page (http://www.oasis-open.org/committees/wsbpel)

F. Revision History

Revision 01.000	10 June 2003	First Draft. Produced from May 05 2003 version of BPEL 1.1	Prasad Yendluri
Revision 01.001	16 Oct 2003	First Draft uploaded to CVS system	Kevin Liu
Revision 01.002	16 Oct 2003	Notes to editor for to do items	Kevin Liu
Revision 01.003	18 Oct 2003	Applied Issue-27 Resolution (impact of error on evaluation condition on a link)	Prasad Yendluri
Revision 01.004	21 Oct 2003	Applied Issue-36 Resolution (Change onMessage Event Handler to onEvent)	Prasad Yendluri
Revision 01.005	22 Oct 2003	Created this revision history section	Prasad Yendluri
Revision 01.006	22 October 2003	Applied Issue-23 Resolution (clarify that redundancy of structured activities is desirable)	Yaron Y. Goland
Revision 01.007	27 October 2003	Added the words "properly configured" to the change made in Revision 006 per Satish's request.	Yaron Y. Goland
Revision 01.008	2 Nov 2003	Incorporated resolution of Issue 3 (current state influence in compensation handlers)	Satish R. Thatte
Revision 01.009	2 Nov 2003	Incorporated resolutions of issues 41 and 45 and corrected a number of XML errors	
Revision 01.010	10 November 2003	Applied resolution to issue 32 (link semantics in event handlers)	Sid Askary

Revision 01.011	17 November 2003	Applied resolution of issues 7, 39, 46	Francisco Curbera
Revision 01.012	23 Nov 2003	Revisited all previous changes and corrected them for clarity and completeness	Satish R. Thatte
Revision 01.013	01 February 2004	Applied resolution of issues 72, 73, 74	Ben Bloch
Revision 01.014	09 February 2004	Applied resolution of issues 67, 68, 79	Satish R. Thatte
Revision 01.015	17 February 2004	Applied resolution of issues 13, 85. Added example to Section 14.3	Francisco Curbera
Revision 01.016	19 February 2004	Fixed Section 12.5 sentence about <joinCondition> and added BP/use case comment in Section 3	Ben Bloch
Revision 01.017	20 February 2004	Applied resolution of issue 70. Removed editor's notes for "Editors" section	Kevin Liu
Revision 01.018	27 February 2004	Applied resolution of issue 50. Added a new standard fault `missingReply`	Neelakantan Kartha
Revision 01.019	09 March 2004	Miscellaneous revisions of prior edits	Satish R. Thatte
Revision 01.020	12 March 2004	Applied resolutions 13 and 68 to schema, corrected xsd definitions for <import> element	Francisco Curbera
Revision 01.021	14 March 2004	Applied resolutions for issues 80 and 95 (rethrow in <catchAll> etc. and corresponding schema changes to add the <rethrow> definition)	Prasad Yendluri
Revision 01.022	24 March 2004	Added namespace prefix list to end of section 2; added editor notes for to do items	Kevin Liu
Revision 01.023	6 April 2004	Incorporated 104 and fixed examples and pseudo syntax to reflect resolution of issue 68	Satish R. Thatte
Revision 01.024	07 April 2004	Corrected the namespace for bpws and plnk to be in sync with the latest schema and updated all examples not	Prasad Yendluri

		to declare the namespace explicitly so that when we update the namespace the examples are not impacted.	
Revision 01.025	07 April 2004	Fixed the default namespace values that were missed in the last pass.	Prasad Yendluri
Revision 01.026	07 April 2004	(1) Copy and paste the updated and corrected schema back to the main spec doc (2) Applied resolution for Issue 69 from Maciej Szefle in Section 12.5 and 12.5.1	Alex Yiu
Revision 01.027	23 April 2004	Applied resolution of issue 62. Checked that issue 45 had already been applied.	Francisco Curbera
Revision 01.028	05 May 2004	Picked up the new OASIS Logo	Prasad Yendluri
Revision 01.029	06 May 2004	Applied resolution of issue101	Prasad Yendluri
Revision 01.030	06 May 2004	Inserted the lost pictures (with proper hyperlinks) back into the document.	Prasad Yendluri
Revision 01.031	06 May 2004	Applied resolution of issue117	Prasad Yendluri
Revision 01.032	11 May 2004	Incorporated resolution for issue 33	Kevin Liu
Revision 01.033	02 June 2004	Revised wording for resolution of issue 33	Kevin Liu
Revision 01.034	20 June 2004	Applied resolution of issues 25 and 53	Satish Thatte
Revision 01.035	28 June 2004	Applied resolution: issue 94 in Section 6.2 and schema; issue 75 in Section 6.2, 7.2, 13, 13.1 and schema; issue 37 in Section 6.2, 10.2, 14.4 and schema; issue 114 in Section 10.2 and 14.4	Alex Yiu
Revision 01.036	15 July 2004	Applied resolution of issue 43. Changed schema in the doc and pseudo-schema fragments scattered	Prasad Yendluri

		thruought the specification.	
Revision 01.037	15 July 2004	Applied resolution of issue 44.	Prasad Yendluri
Revision 01.038	19 July 2004	Applied resolution of issue 34	Alex Yiu
Revision 01.039	28 July 2004	Applied resolution of issue 128, plus made all examples WS-I BP compliant	Kevin Liu
Revision 01.040	01 Sept 2004	Re-sychronize XSD Appendix with XSD files from CVS	Alex Yiu
Revision 01.041	01 Sept 2004	Applying Issue 146: Making tVariable Extensible Applying Issue 149: Adding formal documentation support to BPEL	Alex Yiu
Revision 01.042	01 Sept 2004	Fixed a Typo in XSD	Alex Yiu
Revision 01.043	01 Sept 2004	A bunch of misc cleanup for Issue 13	Alex Yiu
Revision 01.044	08 Sept 2004	Created the first clean copy and performed misc clean-up that surfaced after clean-up, including messed formatting, TOC links, Appendix ordering and numbering and messed formatting.	Prasad Yendluri
Revision 01.045	20 Oct 2004	Fixing joinCondition syntax typo in Section 6.2 Applied Issue 134 to Section 9.1.4 Applied Issue 123 to Section 11.4 and other sections Fixing schema problems discovered by Dieter Koenig	Alex Yiu
Revision 01.046	21 Oct 2004	Applying some non-normative changes in XSD suggested by Paul Brown Applying changes in XSD for “tRole” for Issue 129 (Other spec text changes will be done by Kevin Liu)	Alex Yiu
Revision 01.047	28 Oct 2004	Applied to section 6.2 the resolution for issue 89.(note the resolution proposal was friendly amended by the	Kevin Liu

		editors group) Appled resolution for issue 129: portType is now an attribute of <role>	
Revision 01.048	17 Nov 2004	Applied resolutions for issues 98, 135 and 176. For 176 the section content was deleted but the section number was left in, since the headers are not using auto numbering yet	Satish Thatte
Revision 01.049	18 Nov 2004	Applied resolution for Issue 10. Noted that the level 3 headers are using the Header 4 style and decided against making a heroic effort to fix all the headers.	Satish Thatte
Revision 01.050	22 Nov 2004	Applied resolution for issues 172 and 173.	Assaf Arkin
Revision 01.051	23 Nov 2004	Applied resolutions for issues 137, 166, 175 & 178	Yaron Y. Goland
Revision 01.052	29 Nov 2004	Revised edits for issue 137	Yaron Y. Goland
Revision 01.053	30 Nov 2004	Applied resolutions for issues 152 and 165	Alex Yiu
Revision 01.054	01 Dec 2004	Revised sections 9.1 and 14.1 per the discussion on the editors list on resolution for issue 137.	Prasad Yendluri
Revision 01.055	11 Jan 2005	Applied resolution for issue 145, 155, and 171.	Kevin Liu
Revision 01.056	26 Jan 2005	Applied resolution for issue 162.	Prasad Yendluri
Revision 01.057	27 Jan 2005	Applied resolution for issue 167.	Prasad Yendluri
Revision 01.058	30 Jan 2005	Applied resolution for issue 170.	Francisco Curbera
Revision 01.059	26 Feb 2005	Applied resolution for issue 130.	Francisco Curbera
Revision 01.060	27 Feb 2005	Applied resolutions for issues 12.1, 186.	Francisco Curbera
Revision	27 Feb 2005	Reflect the spec update date for	Prasad

01.061		publication.	Yendluri
Revision 01.062	2 April 2005	Applied resolution for issues 93 and 111.1	Francisco Curbera
Revision 01.063	24 April 2005	Applied resolution of issue 112	Francisco Curbera
Revision 01.064	01 May 2005	Synchronizing XSD changes into Appendix C for issue 112 and fixing some XSD syntax typo for issue 112	Alex Yiu
Revision 01.065	02 May 2005	Fixing misc typo in the spec: (a) Moving terminationHandler from process level to scope level (b) removing compensationHandler from the process level (c) renaming an attribute of "scope" from "variableAccessSerializable" to "isolated" (d) removing "enableInstanceCompensation" attribute from XSD	Alex Yiu
Revision 01.066	07 May 2005	Applied issues 29, 81, 160	Assaf Arkin
Revision 01.067	07 May 2005	Applied issues 29, 81, 160	Assaf Arkin
Revision 01.068	20 May 2005	Updated Appendix H	Prasad Yendluri
Revision 01.069	21 June 2005	Accepted all changes todate to create a clean TC approved Draft	Prasad Yendluri
Revision 01.070	23 June 2005	XSD are removed temporarily from this doc for ease of editing and will be maintained in the CVS tree	Alex Yiu
Revision 01.071	28 June 2005	Apply Issues 182, 148, 140 (with renaming <until> activity to <repeatUntil> activity to avoid conflict in <onAlarm>)	Alex Yiu
Revision 01.072	08 July 2005	Applied issues 198, 181, 92.2, 86	Assaf Arkin
Revision 01.073	13 July 2005	Changes related to resolution of issue 183	Prasad Yendluri

Revision 01.074	13 July 2005	Changes related to resolution of issue 187	Prasad Yendluri
Revision 01.075	13 July 2005	Changes related to resolution of issue 194	Prasad Yendluri
Revision 01.076	26 July 2005	Apply Issues 102, 103, 199 and 203. Apply Issue 196 also – pure XSD changes	Alex Yiu
Revision 01.077	14 August 2005	Apply issues 138, 139, 139.1, 147, 150, 154, 119, 126	Francisco Curbera
Revision 01.078	20 August 2005	Applied 202, 206, 209,214,132 (??)	Assaf Arkin
Revision 01.079	23 August 2005	Changes related to resolution of Issue 190. Note: Alex needs to add the @exitOnStandardFault to WS-BPEL schema (impacts tProcess and tScope).	Prasad Yendluri
Revision 01.080	23 August 2005	Changes related to resolution of Issue 163.	Prasad Yendluri
Revision 01.081	23 August 2005	Changes related to resolution of Issue 136. Note: Alex needs to apply the schema changes to change <swith> to <if>	Prasad Yendluri
Revision 01.082	30 August 2005	Pure XSD Changes: (a) rectify the definition of "forEach" (issue 147) according to the issue 204 by adding "scope" (note: we still need to apply the rest of issue 204.) (b) apply XSD changes for Issues 190 and 136 Applied changes (both spec text and XSD changes) for Issue 213, Issue 192, remaining parts of Issue 11.1 and Issue111 Made some correction of Issue 145 (spec text and XSD changes)	Alex Yiu
Revision 01.083	01 September 2005	Changes related to resolution of Issue 210.	Prasad Yendluri
Revision 01.084	01 September 2005	Changes related to resolution of Issue 200.	Prasad Yendluri

Revision 01.084	21 September 2005	Created clean Committee Draft based on approval on the 09/21/05 TC call. This accepts and removes all change tracking in the draft created for the 9/13-09/15 Redmond F2F meeting.	Prasad Yendluri

G. References (Normative)

[XMLSpec] *XML Specification*, W3C Recommendation.

[SOAP 1.1] *Simple Object Access Protocol (SOAP) 1.1*, W3C Note

[WSDL 1.1] *Web Services Definition Language (WSDL) 1.1*, W3C Note

[UDDI] *Universal Description, Discovery and Integration*, Industry Initiative

[XLANG] *Web Services for Business Process Design*

[WSFL] *Web Service Flow Language 1.0*

[XML Schema Part 1] *XML Schema Part 1: Structures*, W3C Recommendation

[XML Schema Part 2] *XML Schema Part 2: Datatypes*, W3C Recommendation

[XPATH 1.0] *XML Path Language (XPath) Version 1.0*, W3C Recommendation

[Sagas] *Sagas* H. Garcia-Molina and K. Salem, Proc. ACM SIGMOD (1987).

[Trends] *Trends in systems aspects of database management," I.L. Traiger, Proc. 2nd Intl.* , Conf. on Databases (ICOD-2), Wiley and Sons 1983.

[RFC 2119] *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*, S. Bradner. IETF (Internet Engineering Task Force), 1997.

[RFC 2396] *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*, T. Berners-Lee, R. Fielding, L. Masinter, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

[BPEL4WS 1.0] *Business Process Execution Language for Web Services Version 1.0*, BEA, IBM and Microsoft, August 2002.

[WS-I Basic Profile] *WS-I Basic Profile*, Web Services Interoperability Organization

[Infoset] *XML Information Set (Second Edition)*, W3C Recommendation

H. Committee Members (Non-Normative)

The following individuals were members of the committee during the formulation of this document:

- Lars Abrell
- Ashish Agarwal, Adobe
- Kristofer Agren, Pakalert
- Ravi Akireddy
- Hedy Alban
- Randall Anderson, Macgregor
- Tony Andrews, Microsoft
- Assaf Arkin, Intalio
- Sid Askary, Nuperus
- Kent Below, IBM
- Dave Bettin, Attachmate
- Terry Bjornsen, Booz Allen Hamilton
- Mike Blevins, BEA
- Ben Bloch, Systinet
- Alex Boisvert, Intalio, Inc.
- Subhra Bose, Reuters America
- Allen Brookes, Roguewave
- George Brown, Intel
- Steve Brown, Metastorm
- Justin Brunt, WfMC
- David Burdett, CommerceOne
- Laurence Cable
- Ryan Cairnsi, OpenStorm Software
- Arun Candadai
- Greg Carter, Metastorm
- Brian Carroll, Serena
- Andy Chan, FileNet
- Sarat Chand, PeopleSoft
- Chia Chao, IONA
- Martin Chapman, Oracle
- Dave Chappell, Sonic Software
- Jamie Clark, OASIS
- Doron Cohen, BMC Software
- Ugo Corda, SeeBeyond
- Robin Cover, OASIS
- Fred Cummins, EDS
- Francisco Curbera, IBM
- Dennis Curry, EDS
- Sanjay Dalal, BEA
- Khoi Dang, FileNet
- Michael DeBellis, Fujitsu

- Linda DeMichiel, Sun Microsystems
- Jens Doerpmund, Hewlett-Packard
- Wolfgang Dostal, IBM
- Bernd Eckenfels, Seeburger, AG
- Rami Elron, BMC Software
- John Evdemon, Microsoft (TC-Chair)
- Charls Fenton, Sterling Commerce
- B.J. Fesq
- Rabih Filfili, Microsoft
- Layna Fischer, BPMI.org
- Layna Fischer, WfMC
- Tony Fletcher, Choreology
- Bill Flood, Sybase
- Daniel Foody, Actional
- Steven Forgey, SeeBeyond
- Andrew Francis, McGill
- Franz Fritz, SAP
- Yuzo Fujishima, NEC Corporation
- Shoichi Fukuda, Fujitsu
- Peter Furniss, Choreology
- Jean-Luc Giraud, Axway software
- Yaron Goland, BEA
- Alastair Green, Choreology
- Ajay Gummadi
- Michael Hafner, SAP
- Tommy Hansen, FileNet
- Kevin Hein, Teamplate
- Scott Hinkelman, IBM
- Patrick Hogan, Sino Technologies
- Chunbo Huang, BEA
- Yin-Leng Husband, Hewlett-Packard
- David Ingham, Arjuna Technologies
- Ricardo Jimenez-Peris, UPM
- Ram Jeyaraman, Sun Microsystems
- Diane Jordan, IBM (TC-Chair)
- Bala Kamallakharan, Cap Gemini Ernst and Young
- Neelakantan Kartha, Sterling Commerce
- Nicholas Kassem, Sun Microsystems
- Richard Katz
- Nickolas Kavantzas, Oracle
- Michael Keay
- Christopher Keller, Active-Endpoints
- Debra Kellington, Convergys
- Rania Khalaf, IBM
- Edwin Khodabakchian, Collaxa
- Sun-Ho Kim

- Yoav Kirsch, Business Layers
- Doug Knowels, Novell
- Dieter Koenig, IBM
- Melanie Kudela, Uniform Code Council
- Pradeep Kumar, MRO Software
- Sanjeev Kumar
- Chris Kurt, Microsoft
- Kelvin Lawrence, IBM
- Philip Lee, BPMI.org
- Frank Leymann, IBM
- Yanming Li, France Telecom
- Paul Lipton, Computer Associates
- Mark Little, Arjuna Technologies
- Melton Littlepage, Booz Allen Hamilton
- Canyang Kevin Liu, SAP
- Brian Lorenz, Sybase
- Hop Luu, FileNet
- Art Machado, PeopleSoft
- Balinder Malhi, Microsoft
- Sumeet Malhotra, Unisys
- Rajesh Manglani, Uniform Code Council
- Bernard Manouvrier, Axway software
- Mike Marin, FileNet
- Axel Martens, IBM
- Fabienne Marquardt, IBM
- Monica Martin, Sun Microsystems
- Tod Matola, Sterling Commerce
- carol mcdonald, Sun Microsystems
- Bimal Mehta, Microsoft
- Vinkesh Mehta, Deloitte
- Glenn Mi, Collaxa
- Jeff Mischkinsky, Oracle
- JP Morgenthal, Software AG
- Simon Moser, IBM
- Tim Moses, Entrust
- Kenji Nagahashi, Fujitsu
- Trevor Naidoo, IDS Scheer
- Goran Olsson, Oracle
- Srinivas Padmanabhuni, Infosys
- John Parkinson, Cap Gemini Ernst and Young
- James Pasley, Capeclear
- Sanjay Patil, IONA
- Mark Potts, Talking Blocks
- Rajesh Pradhan, Iopsis Software
- Lalitha Prakash, Bahwancybertek
- Matthew Pryor, BPMI.org

- Ken Pugsley, PeopleSoft
- Andrew Pugsley, Hewlett-Packard
- Roshan Punnoose
- Jon Pyke, WfMC
- Harvey Reed, Mitre
- Sundari Revanur, IONA
- Greg Ritzinger, Novell
- Anthony Roby, Accenture
- Jorge Rodriguez, IBM
- Dieter Roller, IBM
- Darran Rolls, Waveset Technologies
- Muthu Ramadoss, Bahwancybertek
- Phil Rossomando, Unisys
- Steve Ross-Talbot, Enigmatec
- Michael Rowley, BEA
- Frank Ryan, Active-Endpoints
- Waqar Sadiq, EDS
- Yuji Sakata
- Nobuyuki Sambuichi, Hitachi
- Krishna Sankar, Cisco Systems
- Bob Schmidt, Microsoft
- Marc-Thomas Schmidt, IBM
- Yoko Seki, Hitachi
- Pinaki Shah, E2Open
- Vishwanath Shenoy, Infosys
- Frank Siebenlist, Argonne National Laboratory
- Parijat Sinha, Convergys
- Dale Skeen, Vitria
- Howard Smith, BPMI.org
- Gavenraj Sodhi, Business Layers
- Rajaraman Sowmya
- Vinay Srinivasaiah, SeeBeyond
- Venkat Srinivasan, SeeBeyond
- Nallan Sriraman, Reuters America
- Sally St. Amand
- Donald Steiner, WebV2
- Karl-Heinz Sternemann, BizTalk Competence Center
- Nikola Stojanovic
- Goutham Sukumar, Microsoft
- Keith Swenson, Fujitsu
- Maciej Szefler, FiveSight Technologies
- Pal Takacsi-Nagy, BEA
- Ran Tamir, BMC Software
- Bob Taylor, Hewlett-Packard
- Sazi Temel, BEA
- Ron Ten-Hove, Sun Microsystems

- Aniruddha Thakur, Oracle
- Satish Thatte, Microsoft
- Darrel Thomas, EDS
- Steve Towers, BPM Group
- Derick Townsend, OpenStorm Software
- Ivana Trickovic, SAP
- Kenwood Tsai, Documentum
- William Vambenepe, Hewlett-Packard
- Danny van der Rijn, Tibco
- Gloria Vargas, Reuters America
- Ganesh Vednere, Cap Gemini Ernst and Young
- Claus von Riegen, SAP
- Jim Webber, Arjuna Technologies
- David Webber
- Michael Weiner, IBM
- Pete Wenzel, SeeBeyond
- Stuart Wheater,Arjuna Technologies
- Stephen White
- Rob Williams, Concurrence
- Scott Woodgate, Microsoft
- John Wunder, Imco
- Prasad Yendluri, webMethods
- Alex Yiu, Oracle
- Eric Yuan, Booz Allen Hamilton
- Hadrian Zbarcea, Iona
- Sinisa Zimek, SAP
- Michael zur Muehlen, WfMC

repeatedCompensation	Thrown when an installed compensation handler is invoked more than once.