Web Services Business Process Execution Language Version 2.0

OASIS Standard

11 April 2007

Specification URIs:
This Version:
http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.doc
http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

Previous Version:
http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html
http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.doc
http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf

Latest Version:
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.doc
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

Technical Committee:
OASIS Web Services Business Process Execution Language (WSBPEL) TC

Chair(s):
Diane Jordan, IBM
John Evdemon, Microsoft

Editor(s):
Alexandre Alves, BEA
Assaf Arkin, Intalio
Sid Askary, Individual
Charlton Barreto, Adobe Systems
Ben Bloch, Systinet
Francisco Curbera, IBM
Mark Ford, Active Endpoints, Inc.
Yaron Goland, BEA
Alejandro Guízar, JBoss, Inc.
Neelakantan Kartha, Sterling Commerce
Canyang Kevin Liu, SAP
Rania Khalaf, IBM
Dieter König, IBM
Mike Marin, IBM, formerly FileNet Corporation
Vinkesh Mehta, Deloitte
Satish Thatte, Microsoft
Danny van der Rijn, TIBCO Software
Prasad Yendluri, webMethods
Alex Yiu, Oracle

Related work:

- See Section 3.

Declared XML Namespace(s):

http://docs.oasis-open.org/wsbpel/2.0/process/abstract
http://docs.oasis-open.org/wsbpel/2.0/process/executable
http://docs.oasis-open.org/wsbpel/2.0/plnktype
http://docs.oasis-open.org/wsbpel/2.0/serviceref
http://docs.oasis-open.org/wsbpel/2.0/varprop

Abstract:

This document defines a language for specifying business process behavior based on Web Services. This language 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. Abstract business processes are partially specified processes that are not intended to be executed. An Abstract Process may hide some of the required concrete operational details. Abstract Processes serve a descriptive role, with more than one possible use case, including observable behavior and process template. WS-BPEL is meant to be used to model the behavior of both Executable and Abstract Processes.

WS-BPEL provides a language for the specification of Executable and Abstract business processes. 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 document was last revised or approved by the Web Services Business Process Execution Language (WSBPEL) TC on the above date. The level of approval is also listed above. Check the current location noted above for possible later revisions of this document. This document is updated periodically on no particular schedule.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/wsbpel.

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 Technical Committee web page (http://www.oasis-open.org/committees/wsbpel/ipr.php.

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/wsbpel.

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.

Table of Contents

1. Introduction

The goal of the Web Services effort is to achieve 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 [SOAP 1.1], Web Services Description Language (WSDL) [WSDL 1.1], and Universal Description, Discovery, and Integration (UDDI) [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, platform independent 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 request-response or uncorrelated one-way interactions.

Models for business interactions typically assume sequences of peer-to-peer message exchanges, both request-response and one-way, 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. An Abstract Process may be used to describe observable message exchange behavior of each of the parties involved, 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 observable behavior. Observable behavior must clearly be described in a platform independent manner and captures behavioral aspects that may have cross enterprise business significance.

The following concepts for describing business processes should be considered:

- Business processes include data-dependent behavior. For example, a supply-chain process 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 processes 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 processes frequently require cross partner coordination of the outcome (success or failure) of units of work at various levels of granularity.

The basic concepts of WS-BPEL can be applied in one of two ways, Abstract or Executable.

A WS-BPEL Abstract Process is a partially specified process that is not intended to be executed and that must be explicitly declared as 'abstract'. Whereas Executable Processes are fully specified and thus can be executed, an Abstract Process may hide some of the required concrete operational details expressed by an Executable artifact.

All the constructs of Executables Processes are made available to Abstract Processes; consequently, Executable and Abstract WS-BPEL Processes share the same expressive power. In addition to the features available in Executable Processes, Abstract Processes provide two mechanisms for hiding operational details: (1) the use of explicit opaque tokens and (2) omission. Although a particular Abstract Process definition might contain complete information that would render it Executable, its Abstract status states that any concrete realizations of it are permitted to perform additional processing steps that are not relevant to the audience to which it has been given.

Abstract Processes serve a descriptive role, with more than one use case. One such use case might be to describe the observable behavior of some or all of the services offered by an Executable Process. Another use case would be to define a process template that embodies domain-specific best practices. Such a process template would capture essential process logic in a manner compatible with a design-time representation, while excluding execution details to be completed when mapping to an Executable Process.

Regardless of the specific use case and purpose, all Abstract Processes share a common syntactic base. They have different requirements for the level of opacity and restrictions on which parts of a process definition may be omitted or hidden. Tailored uses of Abstract Processes have different effects on the consistency constraints and on the semantics of that process. Some of these required constraints are not enforceable by the XML Schema.

A common base specifies the features that define the syntactic universe of Abstract Processes. Given this common base, a usage profile provides the necessary specializations and semantics based on Executable WS-BPEL for a particular use of an Abstract Process.

As mentioned above it is possible to use WS-BPEL to define an Executable Business Process. While a WS-BPEL Abstract Process definition is not required to be fully specified, 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.

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 Abstract Processes as process or role templates while maintaining the intent and structure of the observable behavior. This applies even where private implementation aspects use platform dependent functionality.

This is a key feature 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 this specification, the description of Abstract Business Processes is presented after Executable. We clearly differentiate concepts required for Abstract Business Process description from the concepts for Executable in the section 13. WS-BPEL Abstract Processes.

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 is called a partnerLink. 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. Moreover, WS-BPEL introduces a mechanism to define how individual or composite activities within a unit of work are to be compensated in cases where exceptions occur or a partner requests reversal.

WS-BPEL utilizes several XML specifications: WSDL 1.1, XML Schema 1.0, XPath 1.0 and XSLT 1.0. WSDL messages and XML Schema type definitions provide the data model used by WS-BPEL processes. XPath and XSLT provide 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.

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. The description of the deployment of a WS-BPEL process is out of scope for this specification.

2. Notational Conventions

The upper case 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.
- The name of user defined extension activity is indicated by *anyElementQName*.

Syntax specifications are highlighted as follows:

```
<variables>
   <variable name="BPELVariableName"
             messageType="QName"?
             type="QName"?
             element="QName"?>+
      from-spec?
   </variable>
</variables>
```

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.

The examples and other explanatory material in this document are not fully specified unless otherwise noted. For instance, some examples import WSDL definitions that are not specified in this document.

Examples are highlighted as follows:

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

XSD Schemas are provided as a definition of grammars [XML Schema Part 1]. Where there is disagreement between the separate XML schema files, the XML schemas in the appendices, any pseudo-schema in the descriptive text, and the normative descriptive text, the normative descriptive text will take precedence over the separate XML Schema files. The separate XML Schema files take precedence over any pseudo-schema and over any XML schema included in the appendices. The WS-BPEL XML Schemas offer supplementary normative XML syntax details, such as details regarding extensibility of a WS-BPEL process definition, as long as those XML syntax details do not violate explicit normative descriptive text.

XML Schemas only enforce a subset of constraints described in the normative descriptive text. Hence, a WS-BPEL artifact, such as a process definition, can be valid according to the XML Schemas only but not valid according to the normative descriptive text.

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, non-normative 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/"
- **vprop** - "http://docs.oasis-open.org/wsbpel/2.0/varprop"
- **sref** - "http://docs.oasis-open.org/wsbpel/2.0/serviceref"
- **plnk** – "http://docs.oasis-open.org/wsbpel/2.0/plnktype"
- **bpel** – "http://docs.oasis-open.org/wsbpel/2.0/process/executable"
- **abstract** – "http://docs.oasis-open.org/wsbpel/2.0/process/abstract"

3. Relationship with Other Specifications

WS-BPEL refers to the following XML-based specifications: WSDL 1.1, XML Schema 1.0, XPath 1.0, XSLT 1.0 and Infoset. All WS-BPEL implementations SHOULD be configurable such that they can participate in Basic Profile 1.1 [WS-I Basic Profile] conforming interactions. A WS-BPEL implementation MAY allow the Basic Profile 1.1 configuration to be disabled, even for scenarios encompassed by the Basic Profile 1.1.

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 exposed 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 is used to describe the message exchanges followed by the business process of a specific role in the interaction.

The definition of a WS-BPEL business process follows the WSDL model of separation between the abstract message contents used by the business process and deployment information (messages and port type 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 (port types and operations); no references are made to the actual services used by a process instance. WS-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.

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 is not feasible.

- Fault naming with its restriction, as discussed later in this document (see section 10.3. Invoking Web Service Operations – Invoke)
- [SA00002] Overloaded operation names in WSDL port types. Regardless of whether the WS-I Basic Profile configuration is enabled, a WS-BPEL processor MUST reject any WSDL port type 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.
- [SA00001] Port types that contain solicit-response or notification operations as defined in the WSDL 1.1 specification. Regardless of whether the WS-I Basic Profile configuration is enabled, a WS-BPEL processor MUST reject a WS-BPEL that refers to such port types.

At the time this specification was completed, various Web Service standards work, such as WSDL 2.0 and WS-Addressing, were ongoing and not ready for consideration for WS-BPEL 2.0. Future versions of WS-BPEL may provide support for these standards.

It should be noted that the examples provided in this specification adopt the Schema at location "http://schemas.xmlsoap.org/wsdl/2004-08-24.xsd" for the namespace URI http://schemas.xmlsoap.org/wsdl/ [WSDL 1.1]. This XML Schema incorporates fixes for known errors, and is the XML Schema selected by the [WS-I Basic Profile 1.1 Errata] (October 25, 2005).

4. Static Analysis of a Business Process

WS-BPEL takes it as a general principle that conformant implementations MUST perform basic static analysis listed in Appendix B to detect and reject process definitions that fail any of those static analysis checks. Please note that such analysis might in some cases prevent the use of processes that would not, in fact, create situations with errors, either in specific uses or in any use. For example, a WS-BPEL implementation will reject a process with `<invoke>` activity referring to an undefined variable, where the `<invoke>` activity may not be actually reached during execution of the process.

A WS-BPEL implementation MAY perform extra static analysis checking beyond the basic static analysis required by this specification to signal warnings or even reject process definitions. Such an implementation SHOULD be configurable to disable these non-specified static analysis checks.

5. Defining a Business Process

5.1. 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 Figure 1: Purchase Order Process Outline. 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 paths 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 paths. 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 concurrent paths are completed, invoice processing can proceed and the invoice is sent to the customer.

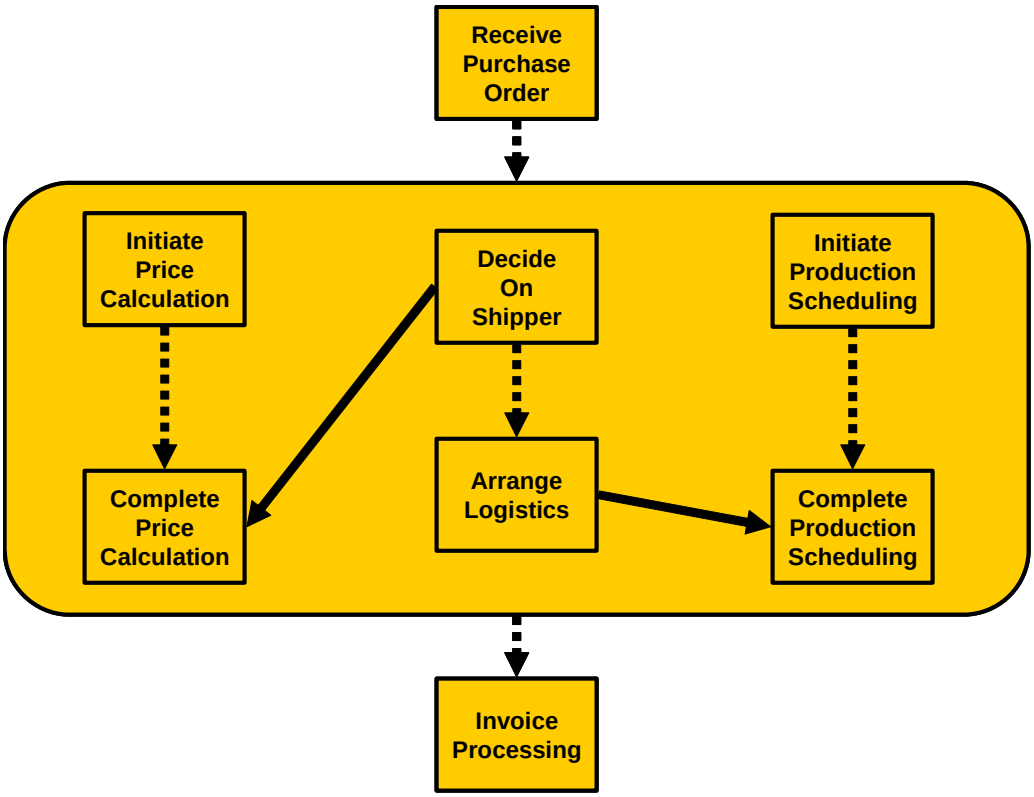
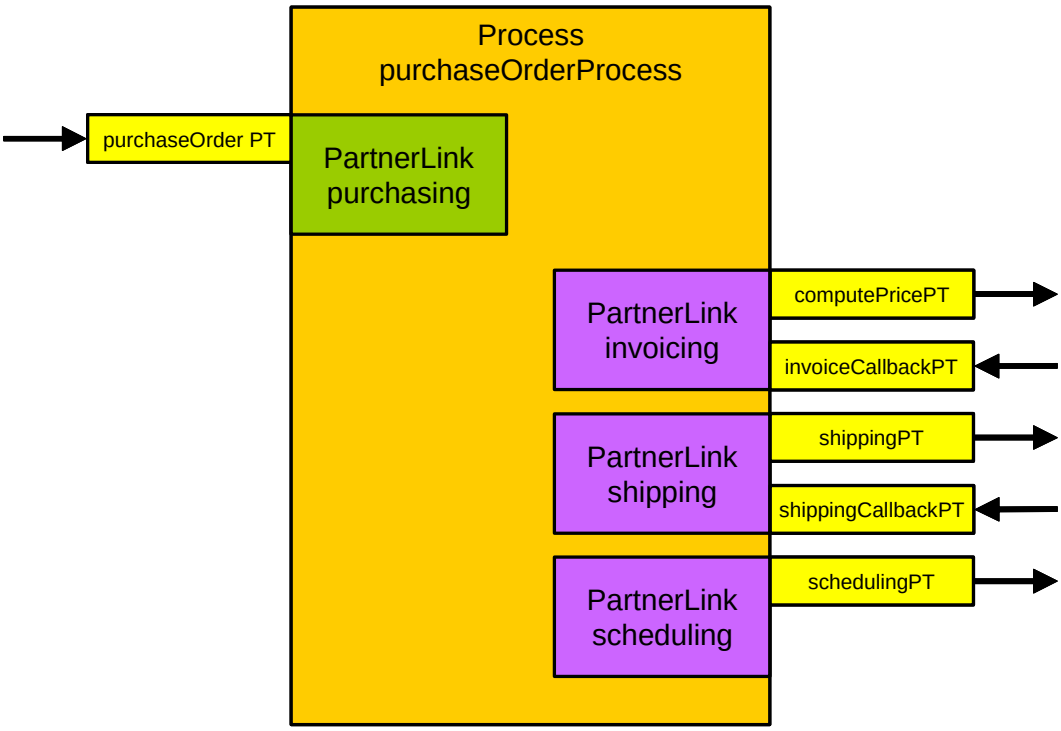


Figure 1: Purchase Order Process - Outline

The WSDL port type 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 port types 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 by referencing only the port types 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 `<partnerLinkType>`s 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 section 6. Partner Link Types, Partner Links, and Endpoint References). `<PartnerLinkType>`s 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 `<partnerLinkType>` defines up to two "role" names, and lists the port types that each role must support for the interaction to be carried out successfully. In this example, two `<partnerLinkType>`s, "purchasingLT" and "schedulingLT", list a single role because, in the corresponding service interactions, one of the parties provides all the invoked operations: The "purchasingLT" `<partnerLinkType>` 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" `<partnerLinkType>` represents the interaction between the purchase order service and the scheduling service, in which only operations of the latter are invoked. The two other

<partnerLinkType>s, "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 notifications to be sent ("invoiceCallbackPT" and "shippingCallbackPT" port types).




```
<wsdl:definitions
   targetNamespace="http://manufacturing.org/wsdl/purchase"
   xmlns:sns="http://manufacturing.org/xsd/purchase"
   xmlns:pos="http://manufacturing.org/wsdl/purchase"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

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

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

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

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

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

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

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

</wsdl:definitions>
```

The business process for the order service is defined next. There are four major sections in this process definition. Note that the example provides a simple case. In order to complete it, additional elements may be needed such as `<correlationSets>`.

- The `<partnerLinks>` section defines the different parties that interact with the business process in the course of processing the order. The four `<partnerLink>` definitions shown here correspond to the sender of the order (customer), as well as the providers of price (invoicing provider), shipment (shipping provider), and manufacturing scheduling services (scheduling provider). Each `<partnerLink>` is characterized by a `partnerLinkType` and either one or two role names. 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 port types that the purchase order process and the partner need to implement.
- 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 between message exchanges.
- 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 port type and fault are defined, and the NCName of the fault.

- 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.

```
<process name="purchaseOrderProcess"
   targetNamespace="http://example.com/ws-bp/purchase"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   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="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"
         createInstance="yes">
         <documentation>Receive Purchase Order</documentation>
      </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">
               <documentation>Decide On Shipper</documentation>
               <sources>
                  <source linkName="ship-to-invoice" />
               </sources>
            </invoke>
            <receive partnerLink="shipping"
               portType="lns:shippingCallbackPT"
               operation="sendSchedule" variable="shippingSchedule">
               <documentation>Arrange Logistics</documentation>
               <sources>
                  <source linkName="ship-to-scheduling" />
               </sources>
            </receive>
         </sequence>
         <sequence>
            <invoke partnerLink="invoicing"
               portType="lns:computePricePT"
               operation="initiatePriceCalculation"
               inputVariable="PO">
               <documentation>
                  Initial Price Calculation
               </documentation>
            </invoke>
            <invoke partnerLink="invoicing"
               portType="lns:computePricePT"
               operation="sendShippingPrice"
               inputVariable="shippingInfo">
               <documentation>
                  Complete Price Calculation
               </documentation>
               <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">
               <documentation>
                  Initiate Production Scheduling
               </documentation>
            </invoke>
            <invoke partnerLink="scheduling"
               portType="lns:schedulingPT"
               operation="sendShippingSchedule"
               inputVariable="shippingSchedule">
               <documentation>
                  Complete Production Scheduling
               </documentation>
               <targets>
                  <target linkName="ship-to-scheduling" />
               </targets>
            </invoke>
         </sequence>
      </flow>
      <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
         operation="sendPurchaseOrder" variable="Invoice">
         <documentation>Invoice Processing</documentation>
      </reply>
   </sequence>

</process>
```

5.2. 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:

```
<process name="NCName" targetNamespace="anyURI"
   queryLanguage="anyURI"?
   expressionLanguage="anyURI"?
   suppressJoinFailure="yes|no"?
   exitOnStandardFault="yes|no"?
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

   <extensions>?
      <extension namespace="anyURI" mustUnderstand="yes|no" />+
   </extensions>

   <import namespace="anyURI"?
      location="anyURI"?
      importType="anyURI" />*

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

```
      initializePartnerRole="yes|no"?>+
   </partnerLink>
</partnerLinks>

<messageExchanges>?
   <messageExchange name="NCName" />+
</messageExchanges>

<variables>?
   <variable name="BPELVariableName"
      messageType="QName"?
      type="QName"?
      element="QName"?>+
      from-spec?
   </variable>
</variables>

<correlationSets>?
   <correlationSet name="NCName" properties="QName-list" />+
</correlationSets>

<faultHandlers>?
   <!-- Note: There must be at least one faultHandler -->
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )? >*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
</faultHandlers>

<eventHandlers>?
   <!-- Note: There must be at least one onEvent or onAlarm. -->
   <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName" )?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
      <correlations>?
         <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      <scope ...>...</scope>
   </onEvent>
   <onAlarm>*
      <!-- Note: There must be at least one expression. -->
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr</until>
      )?
      <repeatEvery expressionLanguage="anyURI"?>
         duration-expr
```

```
         </repeatEvery>?
         <scope ...>...</scope>
      </onAlarm>
   </eventHandlers>
   activity
</process>
```

The top-level attributes are as follows:

- `queryLanguage`. This attribute specifies the query language used in the process for selection of nodes in assignment. 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.
- `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.
- `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`. 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 `bpel:joinFailure` is encountered[1]. 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>`.

  [SA00003] 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 by static analysis and MUST be rejected by a conformant implementation.

- The syntax of Abstract Process has its own distinct target namespace. Additional top-level attributes are defined for Abstract Processes.

The value of the `queryLanguage` and `expressionLanguage` attributes on the `<process>` element are global defaults and can be overridden on specific constructs, such as `<condition>` of a `<while>` activity, as defined later in this specification. In addition, the `queryLanguage` attribute is also available for use in defining WS-BPEL `<vprop:propertyAlias>`es in WSDL. WS-BPEL processors MUST:

[1] bpel:joinFailure does not represent a modeling error and hence it is excluded from other standard faults in this case.

- statically determine which languages are referenced by `queryLanguage` or `expressionLanguage` attributes either in the WS-BPEL process definition itself or in any WS-BPEL property definitions in associated WSDLs and
- [SA00004] if any referenced language is unsupported by the WS-BPEL processor then the processor MUST reject the submitted WS-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 the previous sections. Detailed description of `<documention>` is provided in the next section 5.3. Language Extensibility.

Each business process has one main activity.

A WS-BPEL activity can be any of the following:

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

The syntax of each of these elements is described in the following paragraphs.

The `<receive>` activity allows the business process to wait for a matching message to arrive. The `<receive>` activity completes when the message arrives. The `portType` attribute on the `<receive>` activity is optional. [SA00005] 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.

```
<receive partnerLink="NCName"
   portType="QName"?
```

```
   operation="NCName"
   variable="BPELVariableName"?
   createInstance="yes|no"?
   messageExchange="NCName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
   </correlations>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</receive>
```

The <reply> activity allows the business process to send a message in reply to a message that was received by an inbound message activity (IMA), that is, <receive>, <onMessage>, or <onEvent>. The combination of an IMA and a <reply> forms a request-response operation on a 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 IMA.

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

The <invoke> activity allows the business process to invoke a one-way or request-response operation on a portType offered by a partner. In the request-response case, the invoke activity completes when the response is received. 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).

```
<invoke partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
```

```
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"?
         pattern="request|response|request-response"? />+
   </correlations>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      faultMessageType="QName"?
      faultElement="QName"?>*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
   <compensationHandler>?
      activity
   </compensationHandler>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</invoke>
```

The <assign> activity is 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.

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

The <validate> activity is 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.

```
<validate variables="BPELVariableNames" standard-attributes>
   standard-elements
</validate>
```

The <throw> activity is used to generate a fault from inside the business process.

```
<throw faultName="QName"
   faultVariable="BPELVariableName"?
   standard-attributes>
```

```
   standard-elements
</throw>
```

The <wait> activity is used to wait for a given time period or until a certain point in time has been reached. Exactly one of the expiration criteria MUST be specified.

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

The <empty> activity is a "no-op" in a business process. This is useful for synchronization of concurrent activities, for instance.

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

The <sequence> activity is used to define a collection of activities to be performed sequentially in lexical order.

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

The <if> activity is used to select exactly one activity for execution from a set of choices.

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

The <while> activity is used to define that the child activity is to be repeated as long as the specified <condition> is true.

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

The <repeatUntil> activity is used to define that the child activity is to be repeated until the specified <condition> becomes true. The <condition> is tested after the child activity completes. The <repeatUntil> activity is used to execute the child activity at least once.

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

The <forEach> activity iterates its child scope activity exactly N+1 times where N equals the <finalCounterValue> minus the <startCounterValue>. If parallel="yes" then this is a parallel <forEach> where the N+1 instances of the enclosed <scope> activity SHOULD occur in parallel. In essence an implicit flow is dynamically created with N+1 copies of the <forEach>'s <scope> activity as children. A <completionCondition> may be used within the <forEach> to allow the <forEach> activity to complete without executing or finishing all the branches specified.

```
<forEach counterName="BPELVariableName" parallel="yes|no"
   standard-attributes>
   standard-elements
   <startCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </startCounterValue>
   <finalCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </finalCounterValue>
   <completionCondition>?
      <branches expressionLanguage="anyURI"?
         successfulBranchesOnly="yes|no"?>?
         unsigned-integer-expression
      </branches>
   </completionCondition>
   <scope ...>...</scope>
</forEach>
```

The <pick> activity is used to wait for one of several possible messages to arrive or for a time-out to occur. When one of these triggers occurs, the associated child activity is performed. When the child activity completes then the <pick> activity completes.

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. The optional messageExchange attribute is used to associate a <reply> activity with a <onMessage> event.

```
<pick createInstance="yes|no"? standard-attributes>
   standard-elements
   <onMessage partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      variable="BPELVariableName"?
```

```
      messageExchange="NCName"?>+
      <correlations>?
         <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      activity
   </onMessage>
   <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr</until>
      )
      activity
   </onAlarm>
</pick>
```

The <flow> activity is used to specify one or more activities to be performed concurrently. <links> can be used within a <flow> to define explicit control dependencies between nested child activities.

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

The <scope> activity is used to define a nested activity with its own associated <partnerLinks>, <messageExchanges>, <variables>, <correlationSets>, <faultHandlers>, <compensationHandler>, <terminationHandler>, and <eventHandlers>.

```
<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
   standard-attributes>
   standard-elements
   <partnerLinks>?
      ... see above under <process> for syntax ...
   </partnerLinks>
   <messageExchanges>?
      ... see above under <process> for syntax ...
   </messageExchanges>
   <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>?
      ...
   </compensationHandler>
```

```
   <terminationHandler>?
      ...
   </terminationHandler>
   <eventHandlers>?
      ... see above under <process> for syntax ...
   </eventHandlers>
   activity
</scope>
```

The <compensateScope> activity is used to start compensation on a specified inner scope that has already completed successfully. [SA00007] This activity MUST only be used from within a fault handler, another compensation handler, or a termination handler.

```
<compensateScope target="NCName" standard-attributes>
   standard-elements
</compensateScope>
```

The <compensate> activity is used to start compensation on all inner scopes that have already completed successfully, in default order. [SA00008] This activity MUST only be used from within a fault handler, another compensation handler, or a termination handler.

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

The <exit> activity is used to immediately end a business process instance within which the <exit> activity is contained.

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

The <rethrow> activity is used to rethrow the fault that was originally caught by the immediately enclosing fault handler. [SA00006] The <rethrow> activity MUST only be used within a fault handler (i.e. <catch> and <catchAll> elements). This syntactic constraint MUST be statically enforced.

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

The <extensionActivity> element is used to extend WS-BPEL by introducing a new activity type. The contents of an <extensionActivity> element MUST be a single element that MUST make available WS-BPEL's standard-attributes and standard-elements.

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

The "*standard-attributes*" referenced above are:

```
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.

The "*standard-elements*" referenced above are:

```
<targets>?
   <joinCondition expressionLanguage="anyURI"?>?
      bool-expr
   </joinCondition>
   <target linkName="NCName" />+
</targets>
<sources>?
   <source linkName="NCName">+
      <transitionCondition expressionLanguage="anyURI"?>?
         bool-expr
      </transitionCondition>
   </source>
</sources>
```

5.3. Language Extensibility

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.

Extensions are either mandatory or optional (see section 14. Extension Declarations). [SA00009] In the case of mandatory extensions not supported by a WS-BPEL implementation, the process definition MUST be rejected. Optional extensions not supported by a WS-BPEL implementation MUST be ignored.

In addition, WS-BPEL provides two explicit extension constructs: `<extensionAssignOperation>` and `<extensionActivity>`. Specific rules for these constructs are described in sections 8.4. Assignment and 10.9. Adding new Activity Types – ExtensionActivity.

Extensions MUST NOT contradict the semantics of any element or attribute defined by the WS-BPEL specification.

Extensions are allowed in WS-BPEL constructs used in WSDL definitions, such as `<partnerLinkType>`, `<role>`, `<vprop:property>` and `<vprop:propertyAlias>`. The same syntax pattern and semantic rules for extensions of WS-BPEL constructs are applied to these extensions as well. For the WSDL definitions transitively referenced by a WS-BPEL process, extension declaration directives of this WS-BPEL process are applied to all extensions used in WS-BPEL constructs in these WSDL definitions (see section 14. Extension Declarations).

The optional <documentation> construct is applicable to any WS-BPEL extensible construct. Typically, the contents of <documentation> are for human targeted annotation. 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 extensibility mechanisms.

5.4. 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.

```
<import namespace="anyURI"?
   location="anyURI"?
   importType="anyURI" />*
```

The <import> element is used within a WS-BPEL process to declare a dependency on external XML Schema or WSDL definitions. Any number of <import> elements may appear as children of the <process> element. Each <import> element contains one mandatory and two optional attributes.

- namespace. The namespace attribute specifies an absolute URI that identifies the imported definitions. This attribute is optional. An import element without a namespace attribute indicates that external definitions are in use which are not namespace qualified. [SA00011] If a namespace is specified then the imported definitions MUST be in that namespace. [SA00012] If no namespace is specified then the imported definitions MUST NOT contain a targetNamespace specification. If either of these rules are not met then the process definition MUST be rejected by a conforming WS-BPEL implementation. The namespace http://www.w3.org/2001/XMLSchema is imported implicitly. Note, however, that there is no implicit XML Namespace prefix defined for http://www.w3.org/2001/XMLSchema.
- location. The location attribute contains a URI indicating the location of a document that contains relevant definitions. The location URI may be a relative URI, following the usual rules for resolution of the URI base (XML Base and RFC 2396). The location attribute is optional. An <import> element without a location attribute indicates that external definitions are used by the process but makes no statement about where those definitions may be found. The location attribute is a hint and a WS-BPEL processor is not required to retrieve the document being imported from the specified location.
- importType. The mandatory importType attribute identifies the type of document being imported by providing an absolute URI that identifies the encoding language used in the document. [SA00013] The value of the importType attribute 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. Other importType URI values MAY be used here.

Observe that according to these rules, it is permissible to have an <import> element without namespace and location attributes, and only containing an importType attribute. Such an <import> element indicates that external definitions of the indicated type are in use which are not namespace qualified, and makes no statement about where those definitions may be found.

[SA00010] A WS-BPEL process definition MUST import all XML Schema and WSDL definitions it uses. This includes all XML Schema type and element definitions, all WSDL port types and message types as well as <vprop:property> and <vprop:propertyAlias> definitions used by the process. [SA00053], [SA00054] A WS-BPEL processor MUST verify that all message parts referenced by a <vprop:propertyAlias>, <from>, <to>, <fromPart>, and <toPart> are found in their respective WSDL message definitions. In order to support the use of definitions from namespaces spanning multiple documents, a WS-BPEL process MAY include more than one import declaration for the same namespace and importType, provided that those declarations include different location values. <import> elements are conceptually unordered. [SA00014] A WS-BPEL process definition MUST be rejected if the imported documents contain conflicting definitions of a component used by the importing process definition (as could be caused, for example, when the XSD redefinition mechanism is used).

Schema definitions defined in the types section of a WSDL document which is imported by a WS-BPEL process definition are considered to be effectively imported themselves and are available to the process for the purpose of defining XML Schema variables. However, documents (or namespaces) imported by an imported document (or namespace) MUST NOT be transitively imported by the WS-BPEL processor. In particular, this means that if an external item is used by a WS-BPEL process, then a document (or namespace) that defines that item MUST be directly imported by the process; observe however that this requirement does not limit the ability of the imported document itself to import other documents or namespaces. The following example clarifies some of the issues related to the lack of transitivity of imports.

Assume a document D1 defines a type called d1:Type. However, d1:Type's definition could depend on another type called d2:Type which is defined in document D2. D1 could include an import for D2 thus making d2:Type's definition available for use within the definition of d1:Type. If a WS-BPEL process refers to d1:Type it must import document D1. By importing D1 the WS-BPEL process can legally refer to d1:Type. But the WS-BPEL process could not refer to d2:Type even though D1 imports D2. This is because transitivity of import is not supported by WS-BPEL. Note, however, that D1 can still import D2 and d1:Type can still use d2:Type in its definition. In order to allow the WS-BPEL process to refer to d2:Type it would be necessary for the WS-BPEL process to directly import document D2.

5.5. The Lifecycle of an Executable Business Process

As noted in the introduction, the interaction model that is directly supported by WSDL is essentially a stateless client-server model of request-response or uncorrelated one-way 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 sections 10.4. Providing Web Service Operations – Receive and Reply and 11.5. Selective Event Processing – Pick).

A start activity is a `<receive>` or a `<pick>` activity annotated with a `createInstance="yes"` attribute. [SA00015] Each executable business process MUST contain at least one start activity (see section 10.4. Providing Web Service Operations – Receive and Reply for more details on start activities).

If more than one start activity exists in a process and these start activities contain `<correlations>` then all such activities MUST share at least one common `<correlation>` (see the example in section 9.2. Declaring and Using Correlation Sets).

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

A business process instance ends either normally or abnormally. The process ends normally when the main activity and all event handler instances of the process complete without propagating any fault. The process ends abnormally if either:

- a process level (explicit or default) fault handler completes without propagating any fault or
- the execution of a process level fault handler itself faults (the effect of this particular case is similar to an `<exit>` activity) or
- the process instance is explicitly ended by an `<exit>` activity (see section 10.10. Immediately Ending a Process – Exit).

5.6. Revisiting the Initial Example

In the `purchaseOrderProcess` example in section 5.1. Initial Example, the structure of the main activity of the process 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).

The processing taking place inside the <flow> element consists of three concurrent <sequence> activities. 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 <link>s 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").

Data is shared between different activities through shared variables, for example, the two <variable>s "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 fault 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.

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

An important use case for WS-BPEL is describing cross enterprise business interactions in which the business processes of each enterprise interact through Web Service interfaces. Therefore, WS-BPEL provides the ability to model the required relationships between partner processes. 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 one-way operations rather than on request-response operations. The notion of `<partnerLinks>` is used to directly model peer-to-peer conversational partner relationships. `<partnerLinks>` define the shape of a relationship with a partner by defining the `portTypes` 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 `<sref:service-ref>`, to represent the data required to describe a partner service endpoint.

Introduction of service reference container `<sref:service-ref>` avoids inventing a private WS-BPEL mechanism for web service endpoint references. It also provides pluggability of different versions of service referencing or endpoint addressing schemes being used within WS-BPEL.

6.1. Partner Link Types

A `<partnerLinkType>` 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. Each `<role>` specifies exactly one WSDL `portType`. The following example illustrates the basic syntax of a `<partnerLinkType>` declaration:

```
<plnk:partnerLinkType name="BuyerSellerLink">
   <plnk:role name="Buyer" portType="buy:BuyerPortType" />
   <plnk:role name="Seller" portType="sell:SellerPortType" />
</plnk:partnerLinkType>
```

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. This allows reuse of the WSDL target namespace specification and its import mechanism to import `portType` definitions. The `<partnerLinkType>` definition can be a separate artifact independent of either service's WSDL document. Alternatively, the `<partnerLinkType>` definition can be placed within the WSDL document defining the `portTypes` from which the different roles are defined.

The syntax for defining a `<partnerLinkType>` is:

```
<wsdl:definitions name="NCName" targetNamespace="anyURI" ...>
   ...
   <plnk:partnerLinkType name="NCName">
      <plnk:role name="NCName" portType="QName" />
      <plnk:role name="NCName" portType="QName" />?
   </plnk:partnerLinkType>
   ...
</wsdl:definitions>
```

This defines a <partnerLinkType> in the namespace indicated by the value of the targetNamespace attribute of the WSDL document element. The portTypes identified within <role>s are referenced by using QNames according to the rules in WSDL specifications.

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

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

6.2. Partner Links

The services with which a business process interacts are modeled as partner links in WS-BPEL. Each <partnerLink> is characterized by a partnerLinkType. More than one <partnerLink> 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.

```
<partnerLinks>
   <partnerLink name="NCName"
      partnerLinkType="QName"
      myRole="NCName"?
      partnerRole="NCName"?
      initializePartnerRole="yes|no"? />+
</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 <partnerLink>s for simultaneous requests of the same kind (see section 10.3. Invoking Web Service Operations – Invoke and 10.4. Providing Web Service Operations – Receive and Reply).

Within a <partnerLink>, 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. When a partnerLinkType has only one role, one of these attributes is omitted as appropriate. [SA00016] Note that a <partnerLink> MUST specify the myRole, or the partnerRole, or both. This syntactic constraint MUST be statically enforced.

The <partnerLink> declarations specify the relationships that a WS-BPEL process will employ in its behavior. In order to utilize operations via a <partnerLink>, the binding and communication data, including endpoint references (EPR), for the <partnerLink> must be

available (see also section 10.3. Invoking Web Service Operations – Invoke). The relevant information about a <partnerLink> can be set as part of business process deployment. This is outside the scope of the WS-BPEL specification. Partner link types establish a relationship between WSDL port types of two Web services. The purpose of partner link types is to keep this relationship clear within the process, and make processes with more than one partner easier to understand. No other syntactic or semantic relationships are implied by partner link types in this specification. It is also possible to bind partner links dynamically. WS-BPEL provides the mechanisms to do so via assignment of endpoint references (see section 8.4. Assignment). Since the partners are likely to be stateful, the service endpoint information may need to be extended with instance-specific information.

The initializePartnerRole attribute specifies whether the WS-BPEL processor is required to initialize a <partnerLink>'s partnerRole value. The attribute has no affect on the partnerRole's value after its initialization. [SA00017] The initializePartnerRole attribute MUST NOT be used on a partner link that does not have a partner role; this restriction MUST be statically enforced. If the initializePartnerRole attribute is set to "yes" then the WS-BPEL processor MUST initialize the EPR of the partnerRole before that EPR is first utilized by the WS-BPEL process. An example would be when an EPR is used in an <invoke> activity. If the initializePartnerRole attribute is set to "no" then the WS-BPEL processor MUST NOT initialize the EPR of the partnerRole before that EPR is first utilized by the WS-BPEL process. If the initializePartnerRole attribute is omitted then the partner role MAY be initialized by a WS-BPEL processor.

When initializePartnerRole is set to “yes”, the EPR value used in partnerRole initialization is typically specified as a part of WS-BPEL process deployment or execution environment configuration. Hence, the initializePartnerRole attribute may be used as a part of process deployment contract.

A <partnerLink> can be declared within a <process> or <scope> element. [SA00018] The name of a <partnerLink> MUST be unique among the names of all partner links defined within the same immediately enclosing scope. This requirement MUST be statically enforced. Access to a <partnerLink> follows common lexical scoping rules. The lifecycle of a <partnerLink> is the 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 of whether it is declared on the <process> or <scope> element level.

6.3. Endpoint References

WSDL makes an important distinction between port types and ports. Port types 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 port types, 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.

Endpoint references associated with `partnerRole` and `myRole` of `<partnerLink>`s are manifested as service reference containers (`<sref:service-ref>`). This container is used as an envelope to wrap the actual endpoint reference value. The design pattern here is similar to those of expression language, also known as open-content models, for example:

```
<sref:service-ref reference-scheme="http://example.org">
   <foo:barEPR xmlns:foo="http://example.org">...</foo:barEPR>
</sref:service-ref>
```

The `<sref:service-ref>` has an optional attribute called `reference-scheme` to denote the URI of the reference interpretation scheme of service endpoint, which is the child element of `<sref:service-ref>`.

The URI of `reference-scheme` and the namespace URI of the child element of `<sref:service-ref>` will not necessarily be the same. The optional `reference-scheme` attribute SHOULD be used when the child element of the `<sref:service-ref>` is ambiguous by itself. This optional attribute supplies further information to disambiguate the usage of the content. For example, if `wsdl:service` is used as the endpoint reference, different treatments of the `wsdl:service` element may occur.

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

If the attribute is specified, the URI SHOULD be used as the reference scheme of service endpoint and the content element within the wrapper is treated accordingly.

When a WS-BPEL implementation fails to interpret the combination of the `reference-scheme` attribute and the content element or just the content element alone, a standard fault "unsupportedReference" MUST be thrown.

The `<sref:service-ref>` element is not always exposed to WS-BPEL process definitions. For example, it is not exposed in an assignment from the endpoint reference of `myRole` of partnerLink-A to that of `partnerRole` of partnerLink-B. On the contrary, it is exposed in an assignment from a messageType or element based variable through expression or from a literal `<sref:service-ref>`.

7. Variable Properties

7.1. Motivation

7.1.1 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 `<correlationSets>` (see section 9.2. Declaring and Using Correlation Sets). 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 Abstract and Executable Business Processes.

7.1.2 Motivation for Variable Properties

Message properties are an instance of a more generic mechanism, `<variable>` properties. All variables in WS-BPEL can have properties defined on them. Properties are useful on non-message variables as a way to isolate the WS-BPEL process's logic from the details of a particular variable's definition. Using properties a WS-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 rest of the WS-BPEL process definition that manipulates that variable can remain unchanged.

7.2. Defining Properties

A `<vprop:property>` definition creates a unique name for a WS-BPEL process definition and associates it with an XML Schema type. The intent is to create a `name` that has semantic significance beyond the type itself. For example, a sequence number can be an integer, but the integer type does not convey this significance, whereas a named sequence-number property does. Properties can refer to any parts of a variable.

A typical use for a `<vprop:property>` in WS-BPEL is to name a token for correlation of service instances with messages. 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 name is given to this use of the type by defining a <vprop:property>, as in the following example:

```
<wsdl:definitions name="properties"
   targetNamespace="http://example.com/properties.wsdl"
   xmlns:tns="http://example.com/properties.wsdl"
   xmlns:txtyp="http://example.com/taxTypes.xsd"
   xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <!-- import schema taxTypes.xsd -->

   <!-- define a correlation property -->
   <vprop:property name="taxpayerNumber" type="txtyp:SSN" />
   ...

</wsdl:definitions>
```

In correlation, the property name must have process-wide 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 significance. It is likely that they will be mapped to multiple messages, and therefore they need to be named as in the case of correlation properties.

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 on which the property is available 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. The target namespace and other useful aspects of WSDL are available to them.

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

```
<wsdl:definitions name="NCName">
   <vprop:property name="NCName" type="QName"? element="QName"? />
   ...
</wsdl:definitions>
```

[SA00019] 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.

7.3 Defining Property Aliases

The notion of aliasing is introduced to map a 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 assignments. As an example, consider the following WSDL message definition:

```
<wsdl:definitions name="messages"
```

```
   targetNamespace="http://example.com/taxMessages.wsdl"
   xmlns:txtyp="http://example.com/taxTypes.xsd"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <!-- define a WSDL application message -->
   <wsdl:message name="taxpayerInfoMsg">
      <wsdl:part name="identification"
         element="txtyp:taxPayerInfoElem" />
   </wsdl:message>
   ...

</wsdl:definitions>
```

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

```
<wsdl: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" ...>

   <!-- define a correlation property -->
   <vprop:property name="taxpayerNumber" type="txtyp:SSN" />
   ...

   <vprop:propertyAlias propertyName="tns:taxpayerNumber"
      messageType="txmsg:taxpayerInfoMsg" part="identification">
      <vprop:query>txtyp:socialsecnumber</vprop:query>
   </vprop:propertyAlias>

   <vprop:propertyAlias propertyName="tns:taxpayerNumber"
      element="txtyp:taxPayerInfoElem">
      <vprop:query>txtyp:socialsecnumber</vprop:query>
   </vprop:propertyAlias>

</wsdl:definitions>
```

The first `<vprop:propertyAlias>` defines a named property `tns:taxpayerNumber` as an alias for a location in the identification part of the message type `txmsg:taxpayerInfoMsg`.

The second `<vprop:propertyAlias>` provides a second definition for the same 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 `<vprop:propertyAlias>` definition is:

```
<wsdl:definitions name="NCName" ...>

   <vprop:propertyAlias propertyName="QName"
```

```
      messageType="QName"?
      part="NCName"?
      type="QName"?
      element="QName"?>
      <vprop:query queryLanguage="anyURI"?>?
         queryContent
      </vprop:query>
   </vprop:propertyAlias>
   ...

</wsdl:definitions>
```

The interpretation of the `messageType` and `part` attributes, as well as the `<query>` element is the same as in the corresponding from-spec in copy assignments (see section 8.4. Assignment). The one exception is that the default value of the `queryLanguage` attribute for the `<query>` element within a `<vprop:propertyAlias>` is `urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0`.

[SA00020] A `<vprop:propertyAlias>` element MUST use one of the three following combinations of attributes:

- `messageType` and `part`,
- `type` or
- `element`.

If a `<vprop:propertyAlias>` is defined with the `messageType/part` combination then the property MUST be available on all WS-BPEL variables where the `messageType` QName of the `variable` declaration is identical to that of the `<vprop:propertyAlias>`. The `part` attribute and `<query>` element are applied against the WS-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.

If a `<vprop:propertyAlias>` is defined with a `type` attribute then the property MUST be available on all WS-BPEL variables where the type QName of the variable declaration is identical to that of the `<vprop:propertyAlias>`. The query is applied against the WS-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 WS-BPEL variables defined using a type.

If a `<vprop:propertyAlias>` is defined with an `element` attribute then the property MUST be available on all WS-BPEL variables where the element QName of the variable declaration is identical to that of the `<vprop:propertyAlias>`. The query is applied against the WS-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 WS-BPEL variables defined using an element definition.

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

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

and

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

have the same output (see section 8.4. Assignment for details).

[SA00022] A WS-BPEL process definition MUST NOT be accepted for processing if it defines two or more property aliases for the same property name and WS-BPEL variable type. For example, it is not legal to define two property aliases for the property `tns:taxpayerNumber` and the `messageType` `txmsg:taxpayerInfoMsg`. The same logic would prohibit having two property aliases on the same `element` QName and property name value or two property aliases on the same `type` QName and property name value.

[SA00021] Static analysis MUST detect property usages where property aliases for the associated variable's type are not found in any WSDL definitions directly imported by the WS-BPEL process. As described in 8. Data Handling and 9. Correlation, property usages in WS-BPEL include `<correlationSets>`, `getVariableProperty` functions as well as assign activity copy `<from>` and `<to>` property formats.

8. Data Handling

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. The maintenance of the state of a business process requires the use of variables. 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 allows for the incorporation of future standards.

Both Executable and Abstract Processes are permitted to use the full power of data selection and assignment. Executable Processes are not permitted to use opaque expressions, while Abstract Processes are permitted to use them to hide behavior. Detailed differences are specified in the following sections.

8.1. Variables

Variables provide the means for holding messages that constitute a part of 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.

WS-BPEL uses three kinds of variable declarations: WSDL message type, XML Schema type (simple or complex), and XML Schema element. The syntax of the `<variables>` declaration is:

```
<variables>
   <variable name="BPELVariableName"
      messageType="QName"?
      type="QName"?
      element="QName"?>+
      from-spec?
   </variable>
</variables>
```

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" />
```

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 to which it belongs. Thus, global variables are visible throughout the process. It is possible to hide a variable declared

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.

[SA00023] The `name` of a `<variable>` MUST be unique among the names of all variables defined within the same immediately enclosing scope. This requirement MUST be statically enforced. [SA00024] 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 WS-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.

In this specification, the type `BPELVariableName` is used to describe the `name` of a `<variable>`. It is derived from the XML Schema `NCName` as described below. The type `BPELVariableNames` is used to describe a list of variable names.

```
<xsd:simpleType name="BPELVariableName">
   <xsd:restriction base="xsd:NCName">
      <xsd:pattern value="[^\.]+" />
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="BPELVariableNames">
   <xsd:restriction>
      <xsd:simpleType>
         <xsd:list itemType="tns:BPELVariableName" />
      </xsd:simpleType>
      <xsd:minLength value="1" />
   </xsd:restriction>
</xsd:simpleType>
```

Variable access follows common lexical scoping rules. A variable resolves to the nearest enclosing scope, regardless of the type of the variable, except as described in 12.7. Event Handlers. 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 the `bpel:getVariableProperty` function (as defined below).

[SA00025] 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 WS-BPEL element variable consists of a Document Information Item (DII) that contains exactly one child, an Element Information Item (EII) which is referenced by the document element property. The EII is the value of the element variable.

If a WS-BPEL implementation chooses to manifest a simple type variable as an XML infoset**,** the infoset SHOULD consist of a DII that contains exactly one child, which is an EII referenced by the document element property. The properties of the document element, specifically the namespace name and local name properties, are undefined by this specification. An implementation MUST specify a namespace name/local name value. However the children of the document element MUST exclusively consist of a series of Character Information Items (CIIs) that represent the simple type value. A WS-BPEL implementation MAY choose to map simple type variables to non-XML-infoset data-models defined in the expression/query language being used (e.g. Boolean in XPath 1.0).

The infoset for a complex type variable consists of a DII that contains exactly one child, which is an EII referenced by the document element property. The properties of the document element, specifically the namespace name and local name properties, are undefined by this specification. An implementation MUST specify a namespace name/local name value. However the children of the document element MUST exclusively consist of the complex type values assigned to the variable.

In order to simplify data access, WSDL parts of WSDL message variables are manifested in WS-BPEL as infosets, one infoset per WSDL message part. WS-BPEL engines MUST use 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 those specified in step 2.
- 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 processor and are not specified by this document. The handling of this EII is similar to how WS-BPEL 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 submits and retrieves XML variable values using infoset definitions. WS-BPEL processors are not required to implement an infoset model. Regardless of how the variable binding is handled, the end result SHOULD duplicate the behavior defined using the infoset model above. For example, a WS-BPEL implementation may choose to bind a simple type WS-BPEL variable of type xsd: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. For example, if a `xsd:string` variable is manifested as a string object, a `xsd:string` message part MUST be manifested as a string object also. For detailed definition of manifestation of WS-BPEL variables in XPath 1.0, see section 8.2.2 Binding WS-BPEL Variables In XPath 1.0.

In summary, a WS-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 complex type variable or a WSDL message part
(2) a sequence of CIIs for simple type data: e.g. used to manifest a string (these items may be manifested as a non XML infoset item when needed, e.g. Boolean)

A variable can optionally be initialized by using an in-line from-spec. From-spec is defined in section 8.4. 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 `<assign>` activities 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.

[SA00026] Variable initialization logic contained in scopes that contain or whose children contain a start activity MUST only use idempotent functions in the from-spec. The use of idempotent functions allows for all the values for such variables to be pre-computed and re-used on each process instance.

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 receipt of 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.

An attempt during process execution to read 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 `bpel:uninitializedVariable` fault. This includes the `<invoke>` and `<reply>` activity, where the presence of an uninitialized part also results in the standard fault `bpel:uninitializedVariable`.

Variable Validation

Values stored in variables can be mutated during the course of process execution. The `<validate>` activity can be used 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, listing the variables to validate. The attribute accepts one or more variable names (BPELVariableName), separated by whitespaces. The syntax of the `validate` activity is:

```
<validate variables="BPELVariableNames" standard-attributes>
   standard-elements
</validate>
```

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

A WS-BPEL implementation MAY provide a mechanism to turn on/off any explicit validation, for example, the `<validate>` activity.

A WS-BPEL implementation MAY validate incoming and outgoing messages during the execution of message related activities, e.g., `<receive>`, `<reply>`, `<pick>`, `<onEvent>` and `<invoke>` activities. If such Schema validation is enabled and messages are invalid, "`bpel:invalidVariables`" fault SHOULD be thrown during those message activities.

8.2 Usage of Query and Expression Languages

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

WS-BPEL provides an extensible mechanism for the language used in queries and expressions. The languages are specified by the `queryLanguage` and `expressionLanguage` attributes of the `process` element. WS-BPEL constructs that require or allow queries or expressions provide the ability to override the default query/expression language for individual queries/expressions. WS-BPEL implementations MUST support the use of [XPath 1.0] as the query and expression language. XPath 1.0 is indicated by the default value of the `queryLanguage` and `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.

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

8.2.1 Enclosing Elements

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

Definition (Enclosing Element). An *Enclosing Element* is defined as the parent element in the WS-BPEL process definition that contains the Query or Expression. In the following example, the `<from>` element is the Enclosing Element.

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

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, etc. that are visible to a Query/Expression language are defined based on the entities' visibility to the activity that the Enclosing Element is contained

within. Query/Expression languages need not manifest all the different objects. Only the objects in scope to the Enclosing Element's enclosing activity SHOULD be visible from within the Query/Expression language.

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

- a single XML infoset item: e.g. `$myFooVar/lines/line[2]`
- a collection of XML infoset items e.g. `$myFooVar/lines/*`
- a sequence of CIIs for simple type data
  e.g. `$myFooVar/lines/line[2]/text()`
  (Please note this sequence of items may be manifested as a non XML infoset item when needed. e.g. as a Boolean)
- a variable reference: e.g. `<from>$myFooVar</from>`

8.2.2 Binding WS-BPEL Variables In XPath 1.0

With the exception of link expressions whose variable access syntax and semantics are described in section 8.2.4 Default use of XPath 1.0 for Expression Languages, WS-BPEL variables are accessible in XPath expressions via XPath variable bindings. Specifically, all WS-BPEL variables visible from the Enclosing Element of an XPath expression MUST be made available to the XPath processor by manifesting the WS-BPEL variable as an XPath variable binding whose name is the same as the WS-BPEL variable's name, except in the case of variables declared with a WSDL messageType, which requires some special handling (discussed below).

WS-BPEL variables declared using an element MUST be 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 WS-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="xsd:string"
            form="qualified" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

And given the following variable declaration:

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

Then a WS-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 `StatusContainer`'s child `statusDescription` it is only necessary to specify the child's element name.

WS-BPEL variables declared using a complex type MUST be manifested as a node-set XPath variable with one member node containing the anonymous document element that contains the actual value of the WS-BPEL complex type variable. The XPath variable binds to the document element. For example, given the following Schema definition:

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

And given the following variable declaration:

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

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

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

`$Results` 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 `messageType` variables MUST be manifested in XPath 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 name of the part. The data in a WS-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 the XPath binding that had "`myMessageTypeVar`" in scope would manifest two XPath variables, `$myMessageTypeVar.msgPart1` and `$myMessageTypeVar.msgPart2`.

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 variables.

Below is a full example of how a WSDL message type is manifested in WS-BPEL 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 WS-BPEL XPath expression to access the second part's statusDescription element would look like:

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

It is possible to write XPath 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:statusDescription
```

WS-BPEL simple type variables MUST be manifested directly as either an XPath string, Boolean or float object. If the XML Schema type of the WS-BPEL simple type variable is `xsd:boolean` or any types that are restrictions of `xsd:boolean` then the WS-BPEL variable MUST be manifested as an XPath Boolean object. If the XML Schema type of the WS-BPEL simple type variable is `xsd:float, xsd:int, xsd:unsignedInt` or any restrictions of those types then the WS-BPEL variable MUST be manifested as an XPath float object. Any other XML Schema types MUST be manifested as an XPath string object.

The precision of the float object in XPath 1.0 is not sufficient to capture the full value of some XML Schema data types, such as xsd:decimal. XSD numeric values that cannot be expressed without loss of accuracy as XPath float objects MUST be translated into XPath string objects by a WS-BPEL processor.

8.2.3 XPath 1.0 Perspective and WS-BPEL

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 WS-BPEL for different types of WS-BPEL Expression and Query Language contexts.

8.2.4 Default use of XPath 1.0 for Expression Languages

When XPath 1.0 is used for an Expression Language, except as specified in the sections 8.2.5 Use of XPath 1.0 for Expression Languages in Join Conditions, 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 WS-BPEL scope rules
A function library	WS-BPEL and core XPath 1.0 functions MUST be available and processor-specific functions 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. WS-BPEL variables) are irrelevant. The only prefixes that matter are the in-scope namespaces.

For example, imagine a WS-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 bpel:selectionFailure fault (in Executable WS-BPEL), because it fails to select any node in the context of `<copy>` operation:

```
<from xmlns="http://example.com">$FooVar/bar/text()</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 WS-BPEL variables to manifest as XPath variable bindings enables WS-BPEL programmers to create powerful XPath expressions involving multiple WS-BPEL variables. For example:

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

[SA00027] When XPath 1.0 is used as an expression language in WS-BPEL 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 language. 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 WS-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.

Extra restrictions are applied to XPath usage as an expression language within to-spec (see section 8.4. Assignment).

8.2.5 Use of XPath 1.0 for Expression Languages in Join Conditions

When XPath 1.0 is used as an Expression Language in 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 target the activity that the Enclosing Element is contained within
A function library	Core XPath functions MUST be available, [SA00028] WS-BPEL functions MUST NOT be available, and processor-specific functions MAY be available.
Namespace declaration	In-scope namespace declarations from Enclosing Element

As explained in section 11.5.1 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.

<link> status is obtained via XPath variable bindings, manifesting <link>s that target the activity containing 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>
```

8.2.6 Use of XPath 1.0 for Query Languages in Copy Operations and Property Aliases

When XPath 1.0 is used as Query Language in the first variant of from-spec and to-spec in <copy> assignments (also known as variable variant) or a <vprop:propertyAlias>, the XPath context is initialized as follows:

	Variable variant from-spec or to-spec	<vprop:propertyAlias>
Context node	See below	See below
Context position	1	1
Context size	1	1
A set of variable bindings	Variables visible to the Enclosing Element as defined by the WS-BPEL scope rules	There MUST NOT be any variable bindings available when XPath is used as the query language in a <vprop:propertyAlias>
A function library	WS-BPEL and core XPath 1.0 functions MUST be available and processor-specific functions MAY be available	Core XPath functions MUST be available, [SA00029] WS-BPEL functions MUST NOT be available, and processor-specific functions MAY be available.
Namespace declaration	In-scope namespace declarations from Enclosing Element	In-scope namespace declarations from Enclosing Element (note that the Enclosing Element is in a

		<vprop:propertyAlias> defined in a WSDL definition)

The context node is determined as follows:

- When the from-spec or to-spec references a messageType variable or the <vprop:propertyAlias>'s messageType/part attributes are used:
  - If the message part is based on a complex type or an element, the context node MUST point to a node-list containing a single node which is the EII for the referenced part specified in section 8.2.2 Binding WS-BPEL Variables In XPath 1.0.
  - If the message part is based on a simple type, the context node MUST point to the XPath object specified in section 8.2.2 Binding WS-BPEL Variables In XPath 1.0.
- When the from-spec or to-spec references an XML Schema type variable or the <vprop:propertyAlias>'s type attribute is used:
  - If the type is a complex type, the context node MUST point to a node-list containing a single node which is the EII for the referenced part specified in section 8.2.2 Binding WS-BPEL Variables In XPath 1.0.
  - If the type is a simple type, the context node MUST point to the XPath object specified in section 8.2.2 Binding WS-BPEL Variables In XPath 1.0.
- When the from-spec or to-spec references an XML Schema element variable or the <vprop:propertyAlias>'s element attribute is used, the context node MUST point to a node-list containing a single node which is the EII for the referenced part specified in section 8.2.2 Binding WS-BPEL Variables In XPath 1.0.

None of the previously listed restrictions on the syntax of the XPath expression apply to a <query> in from-spec/to-spec and <vprop:propertyAlias> because it has a defined context node. Any legal XPath expression may be used. An absolute or relative path can be used in a <vprop:propertyAlias> as both resolve to the context node which is the root node.

This example shows a <vprop:propertyAlias> using a relative XPath query. It returns an lvalue:

```
<vprop:propertyAlias propertyName="p:price"
   messageType="my:POMsg"
   part="poPart">
   <vprop:query>price</vprop:query>
</vprop:propertyAlias>
```

In contrast, this example shows a <vprop:propertyAlias> using an absolute XPath query. It does not return an lvalue:

```
<vprop:propertyAlias propertyName="p:goldCustomerPrice"
   messageType="my:POMsg"
   part="poPart">
   <vprop:query>(/p:po/price * 0.9)</vprop:query>
</vprop:propertyAlias>
```

There is no requirement that <query> return lvalues. When the <query> used in a variable variant to-spec or the <query> of <vprop:propertyAlias> used in a property variant to-spec does not return an lvalue, an attempt to assign to such a to-spec MUST fail with a

bpel:selectionFailure (as defined in section 8.4. Assignment). Multiple nodes may be selected with this <vprop:propertyAlias> feature. However, those selections may be then filtered in the rest of expression and result in one node returned.

8.3. Expressions

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

- Boolean expressions (transition, join, while, and if conditions)
- Deadline expressions (until expression of <onAlarm> and <wait>)
- Duration expressions (for expression of <onAlarm> and <wait>, <repeatEvery> expression of <onAlarm>)
- Unsigned Integer expressions (<startCounterValue>, <finalCounterValue>, and <branches> in <forEach>)
- General expressions (<assign>)

When the above first four types of expressions are being used, the corresponding expressions SHOULD return values which are valid according to the corresponding XML Schema types:

- Boolean expressions should return valid values of xsd:boolean
- Deadline expressions should return valid values of xsd:date and xsd:dateTime
- Duration expressions should return valid values of xsd:duration
- Unsigned Integer expressions should return valid values of xsd:unsignedInt

Otherwise, a bpel:invalidExpressionValue fault SHOULD be thrown. Implicit data conversion or casting MAY be applied when computing returned values from expressions, based on the data model or type conversion semantics established in the underlying expression language.

The following values conversion and validity checking semantics MUST be applied when WS-BPEL's default binding to XPath 1.0 is used as the expression language:

- For WS-BPEL Boolean expressions, XPath's boolean(object) function is used to convert the expression result into a Boolean value if needed.
- For WS-BPEL Deadline expressions, XPath's string(object) function is used to convert the expression result into a string value if needed. The string value MUST be valid values of xsd:date and xsd:dateTime. Otherwise, a bpel:invalidExpressionValue fault MUST be thrown.
- For WS-BPEL Duration expressions, XPath's string(object) function is used to convert the expression result into a string value if needed. The string value MUST be valid values of xsd:duration. Otherwise, a bpel:invalidExpressionValue fault MUST be thrown.
- For WS-BPEL Unsigned Integer expressions, XPath's number(object) function is used to convert the expression result into a numeric value if needed. The numeric value MUST be valid values of xsd:unsignedInt (i.e. neither negative or NaN and it must be an integer value). Otherwise, a bpel:invalidExpressionValue fault MUST be thrown.

The following XPath extension functions are defined by WS-BPEL and MUST be supported by a WS-BPEL implementation:

- `getVariableProperty`, described below
- `doXslTransform`, described in section 8.4. Assignment

These extensions are defined in the standard WS-BPEL namespace (see section 5.3. Language Extensibility for an overall discussion of WS-BPEL Language Extensibility) .

Any qualified names used within XPath expressions MUST be resolved by using namespace declarations currently in scope in the WS-BPEL document at the location of the expression. Null prefixes MUST be handled as specified in [XSLT 1.0] section 2.4 (i.e., a null prefix means that the empty namespace is used).

The function signature of `bpel:getVariableProperty` is:

```
object bpel:getVariableProperty(string, string)
```

This function extracts property values from variables. The first argument names the source variable for the data and the second is the QName of the property to select from that variable (see section 7. Variable Properties). [SA00031] The second argument MUST be a string literal conforming to the definition of QName in section 3. Relationship with Other Specifications, and these constraints MUST be enforced by static analysis.

The return value of this function is calculated by applying the appropriate `<vprop:propertyAlias>` for the requested property to the current value of the submitted variable.

[SA00030] The arguments to `bpel:getVariableProperty` MUST be given as quoted strings. The previous requirement MUST be statically enforced. It is therefore illegal to pass into a WS-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 `bpel:getVariableProperty("varA","b:propB")` meets the previous requirement while `bpel:getVariableProperty( $varA, string(bpel:getVariableProperty("varB", "b:propB") )` does not. Note that the previous requirement institutes a restriction which does not exist in the XPath standard.

8.3.1. Boolean Expressions

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

8.3.2. Deadline 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.

8.3.3. Duration 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 lack of XML Schema awareness applies here as well.

8.3.4. Unsigned Integer Expressions

These are expressions that conform to the XPath 1.0 *Expr* production where the evaluation results in number object values that are of the XML Schema type *unsignedInt*.

8.3.5. 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).

8.4. Assignment

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). Expressions operate on variables, properties, and literal constants to produce a new value. The `<assign>` activity can also be used to copy endpoint references to and from partnerLinks. It is also possible to include extensible data manipulation operations defined as extension elements under namespaces different from the WS-BPEL namespace. If the element contained within the extensionAssignOperation element is not recognized by the WS-BPEL processor and is not subject to a mustUnderstand="yes" requirement from an extension declaration then the extensionAssignOperation operation MUST be ignored. (See section 14 Extension Declarations).

Finally, it is possible to include extensible data manipulation operations defined as extension elements under namespaces different from the WS-BPEL namespace (see section 5.3. Language Extensibility).

The `<assign>` activity contains one or more elementary operations.

```
<assign validate="yes|no"? standard-attributes>
   standard-elements
   (
   <copy keepSrcElementName="yes|no"? ignoreMissingFromData="yes|no"?>
      from-spec to-spec
   </copy>
   |
```

```
   <extensionAssignOperation>
      assign-element-of-other-namespace
   </extensionAssignOperation>
   )+
</assign>
```

The <assign> activity copies a type-compatible value from the source ("from-spec") to the destination ("to-spec"), using the <copy> element. [SA00032] Except in Abstract Processes, the from-spec MUST be one of the following variants:

```
<from variable="BPELVariableName" part="NCName"?>
   <query queryLanguage="anyURI"?>?
      queryContent
   </query>
</from>
<from partnerLink="NCName" endpointReference="myRole|partnerRole" />
<from variable="BPELVariableName" property="QName" />
<from expressionLanguage="anyURI"?>expression</from>
<from><literal>literal value</literal></from>
<from/>
```

In Abstract Processes, the from-spec MUST be either one of the above or the opaque variant described in section 13.1.3. Hiding Syntactic Elements

The to-spec MUST be one of the following variants:

```
<to variable="BPELVariableName" part="NCName"?>
   <query queryLanguage="anyURI"?>?
      queryContent
   </query>
</to>
<to partnerLink="NCName" />
<to variable="BPELVariableName" property="QName" />
<to expressionLanguage="anyURI"?>expression</to>
<to/>
```

A to-spec MUST return an lvalue. If a to-spec does not return an lvalue then a `bpel:selectionFailure` MUST be thrown. An lvalue, in the context of XPath, is a node-list containing a single node from a `variable` or a `partnerLink` identified by the to-spec. The restrictions listed in 8.2.4 Default use of XPath 1.0 for Expression Languages MUST apply to XPath used as a query language. [SA00033] In addition, the XPath query MUST begin with an XPath VariableReference. This restriction MUST be statically enforced.

Variable variant: 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 messageType the optional `part` attribute may be used to provide the name of a part within that variable. [SA00034] When the variable is defined using XML Schema types (simple or complex) or element, the `part` attribute MUST NOT be used. An optional <query> element may be used to select a value from the source or target variable or message part. The computed value of the query MUST be one of the following:

- a single XML information item other than a CII, for example, EII and AII

- a sequence of zero or more CIIs: this is mapped to a Text Node or a string in the XPath 1.0 data model

PartnerLink variant: the second from-spec and to-spec variants allow manipulation of the endpoint references associated with partnerLinks. The value of the `partnerLink` attribute is the name of a `partnerLink` that is in scope. In the case of from-specs, the role MUST be specified. 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. [SA00035] [SA00036] If the value "`myRole`" or "`partnerRole`" is used, the corresponding `<partnerLink>` declaration MUST specify the corresponding `myRole` or `partnerRole` attribute. This restriction MUST be statically enforced. For the to-spec, the assignment is only possible to the `partnerRole`, hence there is no need to specify the role. [SA00037] Therefore, the to-spec can only refer to a `<partnerLink>` of which the declaration specifies the `partnerRole` attribute. This restriction MUST be statically enforced. The type of the value referenced by partnerLink-style from/to-specs is always a `<sref:service-ref>` element (see section 6. Partner Link Types, Partner Links, and Endpoint References).

An attempt during process execution to read a partner link before its `partnerRole` EPR is initialized MUST result in the `bpel:uninitializedPartnerRole` standard fault. Partner roles of partner links are read when they are referenced in an `<invoke>` or the `<from>` part of a `<copy>` in an `<assign>` activity.

Property variant: the third from-spec and to-spec variants allow data manipulation using properties (see section 7. Variable Properties). The `property` value generated by the from-spec is generated in the same manner as the value returned by the `bpel:getVariableProperty()` function. The property variants provide a way to clearly define how distinguished data elements in messages are being used.

Expression variant: in the fourth from-spec variant, an expression language, identified by the optional `expressionLanguage` attribute, is used to return a value. In the fourth to-spec variant, an expression language, identified by the optional `expressionLanguage` attribute, is used to select a value. This computed value of the expression MUST be one of the followings:

- a single XML information item other than a CII, for example, EII and AII
- a sequence of zero or more CIIs: this is mapped to a Text Node or a string in the XPath 1.0 data model

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-spec and to-spec variants 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.

Literal variant: the fifth from-spec variant allows a literal value to be given as the source value to assign to a destination. The literal value to be assigned is included within a `<literal>` element in order to prevent conflicts with standard extensibility elements under `<from>`. 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 returns values as if it were a from-spec that selects the children of the `<literal>` element in the WS-BPEL source code. [SA00038] The return value MUST be a single EII or Text Information Item (TII) only. This constraint MUST be enforced during static analysis.(see section 8.4.1. Selection Result of Copy Operations for the definition of TIIs). The XML parsing context of the `<literal>` element in the source code, such as XML Namespace, is carried into the parsing of the children within the `<literal>` element. An empty `<literal/>` element returns an empty TII. Here are some examples for illustration:

```
<assign>
   <copy>
      <from>
         <literal xmlns:foo="http://example.com">
            <foo:bar />
         </literal>
      </from>
      <to variable="myFooBarElemVar" />
   </copy>
   <copy>
      <from>
         <literal>
            <![CDATA[<foo:bar/>]]>
         </literal>
      </from>
      <to variable="myStringVar" />
   </copy>
   <copy>
      <from>
         <literal />
      </from>
      <to variable="myStringVar" />
   </copy>
</assign>
```

The first `<copy>` above copies a `<foo:bar/>` element with a “`foo`” prefix associated to “`http://example.com`” namespace into “`myFooBarElemVar`”. The second `<copy>` copies a string whose value is “`<foo:bar/>`” into “`myStringVar`”. The last `<copy>` copies an empty string into “`myStringVar`”.

The literal from-spec variant also allows a literal `<sref:service-ref>` value to be assigned to a partnerLink, when used with the partnerLink variant of the to-spec.

Empty variant: The sixth from-spec variant and fifth to-spec variant are included to explicitly show that from-spec and to-spec are extensible. Note that if these variants are not extended, or the extensions are not understood, they MUST behave as if they were an expression variant returning zero nodes.

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

Attributes of Assign and Copy

The optional keepSrcElementName attribute of the <copy> construct is used to specify whether the element name of the destination (as selected by the to-spec) will be replaced by the element name of the source (as selected by the from-spec) during the copy operation (see section 8.4.2. Replacement Logic of Copy Operations).

The optional ignoreMissingFromData attribute of the <copy> construct is used to specify whether a bpel:selectionFailure standard fault is suppressed as specified in section 8.4.1. Selection Result of Copy Operations. The default value of the ignoreMissingFromData is "no".

The optional validate attribute can be used with the <assign> activity. Its default value is "no". When validate is set to "yes", the <assign> activity validates all the variables being modified by the activity. A WS-BPEL implementation MAY provide a mechanism to turn on/off any explicit validation. E.g. validate attribute at assign.

If the "validate" part of the <assign> activity fails, that is, one of the variables is invalid against its corresponding XML definition, a standard fault bpel:invalidVariables MUST be thrown.

If there is any fault during the execution of an assignment activity the destination variables MUST be left unchanged, as they were at the start of the activity (as if the assign activity were atomic). This applies regardless of the number of assignment elements within the overall assignment activity.

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 copy mechanism as described thus far, when combined with the default XPath 1.0 expression language, cannot perform complex XML transformations. To address this restriction in a portable fashion, a WS-BPEL processor MUST support the bpel:doXslTransform() XPath 1.0 extension function. The function signature of bpel:doXslTransform is:

```
object bpel:doXslTransform(string, node-set, (string, object)*)
```

where:

- The first parameter is an XPath string providing a URI naming the style sheet to be used by the WS-BPEL processor. [SA00039] This MUST take the form of a string literal. The purpose of this constraint is to allow implementations to statically analyze the process (and named style sheets) for variable dependencies; it MUST be enforced by static analysis.

- The second parameter is an XPath node set providing the source document for the transformation to be performed by the WS-BPEL processor. This set MUST contain a single EII (i.e. an element node in XPath 1.0 data model). If it does not, the WS-BPEL processor MUST throw a `bpel:xsltInvalidSource` fault. The single EII as specified by this parameter MUST be treated as the single child of the root node of the source tree for XSLT processing.
- The optional parameters that follow MUST appear in pairs. Each pair is defined as:
    - an XPath string parameter providing the qualified name of an XSLT parameter
    - an XPath object parameter providing the value for the named XSLT parameter. It can be an XPath *Expr*.

  [SA00040] The WS-BPEL processor MUST enforce the pairing of these parameters by static analysis (i.e., an odd number of parameters must cause a static analysis error).

- The function MUST return the result of the transformation. The result is one of the following infoset items, depending on the XSLT output method employed by the selected style sheet:
    - A single TII (an XPath 1.0 text node), created by the XSLT "text" or "html" output methods, or
    - A single EII (an XPath element node that is the single child of the root of the result tree), which is created by the XSLT "xml" output method.

The WS-BPEL processor MUST execute the `bpel:doXslTransform` function such that all of the following apply:

- The first parameter, naming the style sheet to be used, MUST be used to find the style sheet corresponding to the given URI. This is accomplished in an implementation-dependent fashion. If the style sheet corresponding to the given URI cannot be found, the WS-BPEL processor MUST throw a `bpel:xsltStylesheetNotFound` fault.
- The processor MUST perform an XSLT 1.0 transformation, as described in section 5.1 (Processing Model) of the XSLT 1.0 specification, using the named style sheet as primary sheet, the provided source EII as the source document, and the result tree as the result of the transformation.
- XSLT global parameters ([XSLT 1.0], section 11.4 of the XSLT 1.0 specification) are used to pass additional values from the WS-BPEL process to the XSLT processor. The optional parameters for `doXslTransform` function come in the form of name-value pair in the argument list, as described above. They are used to identify the XSLT global parameters by QName, and to supply values for the named global parameters. [SA00041] The global parameter names MUST be string literals conforming to the definition of QName in section 3 of [Namespaces in XML], and these constraints MUST be enforced by static analysis. The WS-BPEL processor MUST pass the given global parameter names and values to the XSLT processor.
- If any XSLT processing faults occur during the transformation, then a `bpel:subLanguageExecutionFault` MUST be thrown.

Since XSLT is a side effect-free language, execution of the transformation cannot by definition cause any changes to WS-BPEL variables referred to in the style sheet.

The first XPath function parameter, which names the style sheet, has similar semantics as the location attribute of an `<import>` element. Style sheets associated with a process (through its `doXslTransform` invocations) SHOULD be considered part of the process definition, like WSDL definitions and XML Schemas referenced by an `<import>` element.

bpel:doXslTransform Examples

The following examples show complex document transformation and iterative document construction.

Complex document transformation. A common pattern in WS-BPEL processes involves receiving an XML document from one service, converting it to a different Schema to form a new request message, and sending the new request to another service. Such documentation conversion can be accomplished using XSLT via the `bpel:doXslTransform` function.

```
<variables>
   <variable name="A" element="foo:AElement" />
   <variable name="B" element="bar:BElement" />
</variables>
...
<sequence>
   <invoke ... inputVariable="..." outputVariable="A" />
   <assign>
      <copy>
         <from>
            bpel:doXslTransform("urn:stylesheets:A2B.xsl", $A)
         </from>
         <to variable="B" />
      </copy>
   </assign>
   <invoke ... inputVariable="B" ... />
</sequence>
```

In the sequence, a service is invoked, and the result (`foo:AElement`) copied to variable `A`. The `<assign>` activity is used to transform the contents of variable `A` to `bar:BElement`, and copy the result of that transformation to variable `B`. Variable `B` is used to invoke another service.

The style sheet `A2B.xsl` would contain the XSL rules for converting documents of Schema `foo:AElement` to Schema `bar:BElement`.

Iterative document construction. Suppose that a document is constructed by repeatedly calling a service, and accumulating the result in an XML variable. The loop might look something like this:

```
<variables>
   <variable name="PO" element="foo:POElement" />
   <variable name="OutVar" element="foo:ItemElement" />
</variables>

<!-- ... PO is initialized ... -->

<!-- Iteratively add more items to PO until complete -->
```

```
<while>
   <condition>...</condition>
   <sequence>
      <!-- Fetch next chunk into OutVar -->
      <invoke ... inputVariable="..." outputVariable="OutVar" />
      <assign>
         <copy>
            <from>
               bpel:doXslTransform( "urn:stylesheets:AddToPO.xsl",
                                    $PO, "NewItem", $OutVar)
            </from>
            <to variable="PO" />
         </copy>
      </assign>
   </sequence>
</while>
```

The optional parameters given in the `doXslTransform` call specify that the XSLT parameter named "`NewItem`" is set with the value of the WS-BPEL variable `OutVar`. To allow the XSLT style sheet access to this value, it contains a global (top-level) parameter with a name matching that given in the third parameter of the function call shown above.

```
<xsl:transform version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" ...>
   <!-- NewItem variable set by WS-BPEL process;
        defaults to empty item -->
   <xsl:param name="NewItem">
      <foo:itemElement />
   </xsl:param>
   ...
</xsl:transform>
```

The style sheet contains a template that appends the value of global parameter `NewItem` (the value of `OutVar` from the process instance)to the existing list of items in the PO variable.

```
<xsl:template match="foo:itemElement">  <!-- line 1 -->
   <xsl:copy-of select="." />           <!-- line 2 -->
   <xsl:if test="position()=last()">    <!-- line 3 -->
      <xsl:copy-of select="$NewItem" /> <!-- line 4 -->
   </xsl:if>                            <!-- line 5 -->
</xsl:template>                         <!-- line 6 -->
```

This template copies all existing items in the source document (lines 1 & 2) and appends the contents of the XSLT parameter `NewItem` to the list of items (lines 3 & 4). It tests to see if the current node is at the end of the item list (line 3) and copies the result-tree fragment from the XSLT parameter `NewItem` to follow the last item (line 4).

If PO has a value of:

```
<foo:poElement>
   <foo:itemElement>item 1</foo:itemElement>
</foo:poElement>
```

at the beginning of an iteration of the <while> loop and the <invoke> activity returns a value of <foo:itemElement>item 2</foo:itemElement>, evaluation of the <from> expression will result in a value of:

```
<foo:poElement>
   <foo:itemElement>item 1</foo:itemElement>
   <foo:itemElement>item 2</foo:itemElement>
</foo:poElement>
```

which, when the <copy> operation completes, becomes the new value of the PO variable.

8.4.1. Selection Result of Copy Operations

The selection result of the from-spec or to-spec used within a <copy> operation MUST be one of the following three Information Items: Element Information Item (EII), Attribute Information Item (AII), or Text Information Item (TII). EII and AII are defined in [Infoset], while TII is defined in this specification to bridge the gap between the XML Infoset Model and other common XML data models, such as XPath 1.0.

A Text Information Item (TII) is a sequence of zero or more Character Information Items, according to document order; as such, a TII is not manifested in and of itself directly in XML serialization. When mapped to the XPath 1.0 model, it generalizes a string object (which has zero or more characters) and text node (which has one or more characters). A TII lvalue MUST NOT be empty. A TII rvalue MAY be mapped to a text node, a string/Boolean/Number object in XPath 1.0, while a TII lvalue MUST be mapped to a text node.

If the selection result of a from-spec or a to-spec belongs to Information Items other than EII, AII or TII, a bpel:selectionFailure fault MUST be thrown. If any of the unsupported Information Items are contained within the selection result, they MUST be preserved; the only restriction is that they MUST NOT be directly selected by the from-spec or the to-spec as the top-level item.

The <copy> operation is a one-to-one replacement operation. If the optional ignoreMissingFromData attribute has the value of "yes" and the from-spec returns zero XML information items then the <copy> MUST be a "no-op"; no bpel:selectionFailure is thrown. In this case, the to-spec MUST not be evaluated. A bpel:selectionFailure MUST still be thrown in the following cases, even if the ignoreMissingFromData attribute has the value of "yes":

1. the from-spec selects multiple XML information items
2. the from-spec selects one XML information item and the to-spec does not select exactly one XML information item

If the ignoreMissingFromData attribute has the value of "no" this requires that both the from-spec and to-spec MUST select exactly one of the three information items described above. If the from-spec or to-spec do not select exactly one information item during execution, then the standard fault bpel:selectionFailure MUST be thrown. The following table illustrates the behavior of the ignoreMissingFromData attribute in the <copy> operation:

returned nodes		ignoreMissingFromData	
from	**to**	**“no”**	**“yes”**
0	0	selectionFailure	no-op
0	1	selectionFailure	no-op
0	N	selectionFailure	no-op
1	0	selectionFailure	selectionFailure
1	1	copy	copy
1	N	selectionFailure	selectionFailure
N	0	selectionFailure	selectionFailure
N	1	selectionFailure	selectionFailure
N	N	selectionFailure	selectionFailure

ignoreMissingFromData Logic Table

Literal values (the literal variant of from-spec) MUST contain either a single TII or a single EII as its top-level value. When the rvalue of a from-spec is an AII, the to-spec is set to a TII constructed from the normalized value property of the AII as specified in section 8.4.2. Replacement Logic of Copy Operations.

When using the partnerLink variants of from-spec and to-spec with a non-partnerLink variant of the respective from-spec and to-spec in a `<copy>` operation, the `partnerLink` variants should be treated as if they produce an rvalue and lvalue of an EII whose [local name] is “service-ref” and [namespace name] is "http://docs.oasis-open.org/wsbpel/2.0/serviceref".

8.4.2. Replacement Logic of Copy Operations

This section provides rules for replacing data referenced by the to-spec in a `<copy>` operation. Detailed examples are provided in Appendix Appendix D. Examples of Replacement Logic.

Replacement Logic for WSDL Message Variables

When the from-spec and to-spec of a `<copy>` operation both select WSDL message variables, the value of the from-spec message variable MUST be copied, becoming the value of the to-spec message variable. If the from-spec message variable is completely uninitialized then the standard `bpel:uninitializedVariable` fault is thrown. If the from-spec message variable is partially initialized then any uninitialized parts of the from-spec variable result in the same parts of the to-spec variable becoming uninitialized. The original message parts of the to-spec message variable will not be available after the `<copy>` operation.

Replacement Table for XML Data Item

When the from-spec (Source) and to-spec (Destination) select one of three Information Items types, a WS-BPEL processor MUST use the following replacement rules identified in the table below:

Source\Destination	EII	AII	TII
EII	RE	RC	RC
AII	RC	RC	RC
TII	RC	RC	RC

Replacement Logic Table

- RE (Replace-Element-properties):
  - Replace the element at the destination with a copy of the entire element at the source, including [children] and [attribute] properties.

    An optional `keepSrcElementName` attribute is provided to further refine the behavior. [SA00042] It is only applicable when the results of both from-spec and to-spec are EIIs, and MUST NOT be explicitly set in other cases. A WS-BPEL processor MAY enforce this checking through static analysis of the expression/query language. If a violation is detected during runtime, a bpel:selectionFailure fault MUST be thrown.
    - When the `keepSrcElementName` attribute is set to "`no`", the name (i.e. [namespace name] and [local name] properties) of the original destination element is used as the name of the resulting element. This is the default value.
    - When the `keepSrcElementName` attribute is set to "`yes`", the source element name is used as the name of the resulting destination element.

    When the `keepSrcElementName` attribute is set to "`yes`" and the destination element is the Document EII of an element-based variable or an element-based part of a WSDL message-type-based variable, a WS-BPEL processor MUST make sure the name of the source element belongs to the substitutionGroup of the destination element used in the element variable declaration or WSDL part definition. The substitutionGroup relation is determined by XML Schemas known to the WS-BPEL processor. [SA00094] A WS-BPEL processor MAY enforce this checking through static analysis of the expression/query language. If a violation is detected during runtime, a `bpel:mismatchedAssignmentFailure` fault MUST be thrown.

- RC (Replace-Content):
  - To obtain the source content:
    - Once the information item is returned from the source, a TII will be computed based upon it. This source content TII is based on a series of CIIs, generally based on the document order (unless a sorting specification is present in the underlying expression or query), taken from the returned information item. The CIIs are copied, concatenated together, and the resulting value is assigned to the TII. This is analogous to the XPath 1.0 `string()` function.

        - If the source is an EII with an `xsi:nil="true"`, a `selectionFailure` fault MUST be thrown. This check is performed during EII-to-AII or EII-to-TII copy.
    - To replace the destination content:
        - If the destination is an EII, all [children] properties (if any) are removed and the source content TII is added as the child of the EII.
        - If the destination is an AII, the value of AII is replaced with the TII from the source. The value MUST be normalized, in accordance with the XML 1.0 Recommendation (section 3.3.3 Attribute Value Normalization: http://www.w3.org/TR/1998/REC-xml-19980210#AVNormalize).
        - If the destination is a TII, the TII in the destination is replaced with the TII from the source.
- In addition, the following rules apply:
    - Information items referenced by the to-spec MUST be an lvalue. In the XPath 1.0 data model, a TII lvalue MUST be a text node.
    - A `bpel:mismatchedAssignmentFailure` fault MUST be thrown when the to-spec selects a TII as an lvalue, which does NOT belong to a WS-BPEL variable of an XSD string type (or a type derived from XSD string), and one of the following is computed as an rvalue from the from-spec:
        - a TII which has zero CIIs
        - an AII which has an empty string as its [normalized value]
        - an EII which has zero CIIs as its descendants, that is, its [children] and nested [children]. Note that applying XPath 1.0 string() function to this kind of EII would yield an empty string.
    - Attribute values are not text nodes in XPath 1.0. Attribute nodes have a string value that corresponds to the XML normalized attribute value, which is a TII.

Using <copy> to initialize variables

When the destination selected by the to-spec in a `<copy>` operation is un-initialized, which is either an entire WS-BPEL variable or a message part, that destination MUST first be initialized before executing the replacement rules defined above, as if the following has been applied:

- For complex type and simple type variables or message parts, initialize to a skeleton structure composed of a DII and an anonymous Document Element EII.
- For element based variables or message parts, initialize to a skeleton structure composed of a DII and an Document Element EII with the name matching the element name used in variable declaration.

This initialization behavior is an integral part of an atomic `<assign>` activity.

Handling Non-XML Infoset Data Objects in <copy>

Simple type variables and values MAY be allowed to manifest as non-XML infoset data objects, such as `boolean`, `string`, or `float`, as defined in XPath 1.0. Also expressions may return non-XML infoset data objects, for example:

```
   <from>number($order/amt) * 0.8</from>
```

To consistently apply the above replacement rules, such non-XML infoset data are handled as TIIs. This is achieved through converting data to strings, as TII resembles a string object. More specifically, when the XPath 1.0 data model is used in WS-BPEL, "string(object)" (http://www.w3.org/TR/1999/REC-xpath-19991116#function-string) coercion MUST be used to convert boolean or number objects to strings. A WS-BPEL processor MAY skip the actual conversion if the result of <copy> remains the same.

XML Namespace Preservation

In the `<copy>` operation, the [in-scope namespaces] properties from the source (similar to other XML infoset item properties) MUST be preserved in the result at the destination. A WS-BPEL processor may use a namespace-aware XML infrastructure to maintain the XML namespace consistency.

In some XML Schema designs, QName may be used for attribute or element values. When a TII or an AII containing a QName value is selected via a Schema-unaware expression/query language, its data model will fail to capture the namespace property of the QName value. Therefore, the XML namespace may be lost. Note that XPath 1.0 is Schema unaware.

For example, where the value of `attrX` is a QName (`"myPrefix:somename"`) and the value of `"foo:bar3"` is another QName (`"myPrefix:somename2"`). When `"foo:bar2/@attrX"` is copied as the source with XPath 1.0 data model, the namespace declaration for `"myPrefix"` might be missing in the destination.

```
<foo:bar1 xmlns:myPrefix="http://example.org"
   xmlns:foo="http://example.com ">
   <foo:bar2 attrX="myPrefix:somename" />
   <foo:bar3>myPrefix:somename2</foo:bar3>
</foo:bar1>
```

8.4.3. Type Compatibility in Copy Operations

[SA00043] For a copy operation to be valid, the data referred to by the from-spec and the to-spec MUST be of compatible types.

The following situations are considered type incompatible:

- the selection results of `both` the from-spec and the to-spec are variables of a WSDL message type, and the two variables are not of the same WSDL message type (two WSDL message types are the same if their QNames are equal).
- the selection result of the from-spec is a variable of a WSDL message type and that of the to-spec is not, or vice versa (parts of variables, selections of variable parts, or endpoint references cannot be assigned to/from variables of WSDL message types directly).
- the selection result of the from-spec is an EII, that of the to-spec is a Document EII of an element-based variable or an element-based part of a WSDL message-type-based variable, the `keepSrcElementName` attribute is set to “`yes`” and the name of the source element

does not belong to the substitutionGroup of the destination (see section 8.4.2. Replacement Logic of Copy Operations).

If any incompatible types are detected during assignment, the standard fault `bpel:mismatchedAssignmentFailure` MUST be thrown.

8.4.4. Assignment Example

Assume the following complex type definition in the namespace "http://example.org/bpel/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://example.org/bpel/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>
```

9. 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. Messages which create a new business process instance, are a special case, as described in 5.5. The Lifecycle of an Executable 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 set 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 wire-level contract between partners; and to avoid the use of implementation-specific tokens for instance routing whenever possible.

Consider a supply-chain situation where a buyer sends a purchase order to a seller. Suppose the buyer and seller have a stable business relationship and are statically configured to send documents related to purchasing interactions to the URLs associated with the relevant WSDL service ports. The seller needs to 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 purchase order message (such as a purchase order number) that is copied into the acknowledgement message 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 a correlation set, described below, provides this feature. The declarative information allows infrastructure which conforms to WS-BPEL to use correlation tokens to provide instance routing automatically.

The declaration of correlation relies on declarative properties of messages. A property is simply a "field" within a message identified by a query. This is only possible when the message structure is well-defined (for example, described using an XML Schema). The use of correlation tokens is restricted to message parts described in this way. The actual wire format of such messages can be non-XML, for example, EDI flat files, based on different bindings for port types.

9.1. 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 process instance without the need to specify any correlation information within the business process. However, in many cases 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 process instance and a partner, 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) message and using a PO number embedded in the message as the correlation token. The PO number is used in the acknowledgement message by the seller. The seller might later send an invoice message that carries the PO number, to correlate it with the original 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 message exchanges.

WS-BPEL addresses correlation scenarios by providing a declarative mechanism to specify correlated groups of operations within a process 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.

```
<correlationSets>?
   <correlationSet name="NCName" properties="QName-list" />+
</correlationSets>
```

A <correlationSet> can be declared within a process or scope element in a manner that is analogous to a variable declaration. [SA00044] The name of a <correlationSet> MUST be unique among the names of all <correlationSet> defined within the same immediately enclosing scope. This requirement MUST be statically enforced. Access to a <correlationSet> follows common lexical scoping rules.

A process' <correlationSet> is in an uninitiated state at the beginning of a process. A scope's <correlationSet> is in an uninitiated state at the start of the scope to which it belongs. Note that scopes may start and complete their behavior more than once in the lifetime of the process instance if they are contained in repeatable constructs or event handlers. In this case, the <correlationSet> initiation semantics applies to each instance of the scope.

A <correlationSet> resembles a late-bound constant rather than a variable. The binding of values to a <correlationSet> is triggered by a specially marked send or receive message operation. A <correlationSet> can be initiated only once during the lifetime of the scope to which it belongs. Once initiated, the <correlationSet> MUST retain its values, regardless of any variable updates. Thus, a process' <correlationSet> can be initiated at most once during the lifetime of the process instance. Its values, once initiated, can be thought of as an identity of

the business process instance. A scope's <correlationSet> instance is available for binding each time the corresponding scope starts.

In multiparty business conversations, each participant process in a correlated message exchange acts either as the originator or as a follower of the exchange. The originator 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 <correlationSet> that tag the conversation. All other participants are followers that bind their <correlationSet>'s in the conversation by receiving an incoming message that provides the values of the properties in the <correlationSet>. Both originator and followers mark the first activity in their respective groups as the activity that initiates the <correlationSet>.

9.2. Declaring and Using Correlation Sets

Correlation can be used on every messaging activity (<receive>, <reply>, <onMessage>, <onEvent>, and <invoke>). 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. These protocol mechanisms MAY be used implicitly with or without any explicit use of correlation.

[SA00045] Properties used in a <correlationSet> MUST be defined using XML Schema simple types. This restriction MUST be statically enforced. Each <correlationSet> is a named group of properties that, taken together, serve to identify a conversation. A given message can carry information that matches or initiates one or more correlation sets.

The correlation set specifications are used in <invoke>, <receive>, and <reply> activities (see sections 10.3. Invoking Web Service Operations and 10.4. Providing Web Service Operations – Receive and Reply); in the <onMessage> branches of <pick> activities, and in the <onEvent> variant of <eventHandlers> (see sections 11.5. Pick and 12.5.1. Message Events). These <correlation> specifications identify the correlation sets by name and are used to indicate which correlation sets (i.e., the corresponding property sets) occur in the messages being sent and received. The initiate attribute on a <correlation> specification is used to indicate whether the correlation set is being initiated.

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, the standard fault bpel:correlationViolation MUST be thrown.
- When the initiate attribute is set to "join", the related activity MUST attempt to initiate the correlation set, if the correlation set is not yet initiated.

    - If the correlation set is already initiated and the correlation consistency constraint is violated, the standard fault `bpel:correlationViolation` MUST be thrown.
- When the `initiate` attribute is set to `"no"` or is not explicitly set, the related activity MUST NOT attempt to initiate the correlation set.
    - If the correlation set has not been previously initiated, the standard fault `bpel:correlationViolation` MUST be thrown.
    - If the correlation set is already initiated and the correlation consistency constraint is violated, the standard fault `bpel:correlationViolation` MUST be thrown.

The bullets above describe the correlation set *Initiation Constraint*. If multiple correlation sets are used in an outbound message activity (e.g., `<invoke>`), both *initiation constraint* and *consistency constraint*s MUST be observed for all correlation sets used. If multiple correlation sets are used in an inbound message activity (IMA) (e.g. `<receive>`), then the *initiation constraint* MUST be observed for all correlation sets used. If any one of the correlation sets does not follow the constraints above, the standard fault `bpel:correlationViolation` MUST be thrown.

When multiple correlation sets are used in an IMA with `initiate="no"`, a message MUST match all such correlation sets for that message to be delivered to the activity in the given process instance. When correlation set in a message does not match an already initiated correlation set in the process instance or if the correlation set is not initiated, the message MUST not be delivered to an IMA. Therefore, the correlation set consistency constraint checking is not applicable for IMA.

If an inbound Web service request message arrives and both (1) no running process instance can be identified by a message correlation set mechanism and (2) all inbound message activities referencing the Web service operation have the `createInstance` attribute set to "no" are true then this scenario is out of scope of this specification because there is no process instance that would be able to handle it.

When a `bpel:correlationViolation` is thrown by an `<invoke>` activity because of a violation on the response of a request/response operation, the response MUST be received before the `bpel:correlationViolation` is thrown. In all other cases of `bpel:correlationViolation`, the message that causes the fault MUST NOT be sent or received.

Observe that in order to retrieve correlation values from a message, a processor MUST find a matching `<vprop:propertyAlias>` and apply it to the message. A `<vprop:propertyAlias>` is considered matching with a message if:

1. the messageType attribute value used in `<vprop:propertyAlias>` definition matches the QName of the WSDL message type associated with the message;

or

2. the message is associated with a WSDL message type where the message contains a single part defined by an element and the element attribute value used in

<vprop:propertyAlias> definition matches the QName of the element used to define the WSDL part.

This matching <vprop:propertyAlias> constraint MUST be statically enforced. If both a messageType and element based <vprop:propertyAlias> match the message, then the messageType based <vprop:propertyAlias> MUST take priority. A type based <vprop:propertyAlias> is never considered for retrieving correlation values. These matching rules apply only to retrieving correlation values and have no effect on selecting a <vprop:propertyAlias> for use in a from-spec, to-spec, or bpel:getVariableProperty.

In the case in which the application of the <vprop:propertyAlias> results in a response that contains anything other than exactly one information item and/or a collection of Character Information Items then a bpel:selectionFailure fault MUST be thrown.

In the case of <invoke>, when the operation invoked is a request/response operation, a pattern attribute on the <correlation> specification is used to indicate whether the correlation applies to the outbound message ("request"), the inbound message ("response"), or both ("request-response"). [SA00046] The pattern attribute used in <invoke> is required for request-response operations, and disallowed when a one-way operation is invoked. Any violation of this rule MUST be detected during static analysis. In the case of <invoke>, when the operation invoked is an one-way operation, or in the case of <reply>, the usage of correlation sets with initiate="no" is for message validation purposes only. With this, a business process can ensure that the message to be sent carries the expected correlation tokens.

```
<correlations>
   <correlation set="NCName"
      initiate="yes|join|no"?
      pattern="request|response|request-response"? />+
</correlations>
```

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" namespace defined by the document:

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

   <!-- define correlation properties -->
   <vprop:property name="customerID" type="xsd:string" />
   <vprop:property name="orderNumber" type="xsd:int" />
   <vprop:property name="vendorID" type="xsd:string" />
   <vprop:property name="invoiceNumber" type="xsd:int" />

</wsdl:definitions>
```

These properties are names with XML Schema simple types. They are abstract in the sense that their occurrence in variables needs to be separately specified (see section 7. 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.

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

   <wsdl:import namespace="http://example.com/supplyCorrelation"
      location="..." />

   <!-- define schema types for PO and invoice information -->
   <wsdl: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>
   </wsdl:types>

   <wsdl:message name="POMessage">
      <wsdl:part name="PO" type="po:PurchaseOrder" />
   </wsdl:message>
   <wsdl:message name="POResponse">
      <wsdl:part name="RSP" type="po:PurchaseOrderResponse" />
   </wsdl:message>
   <wsdl:message name="POReject">
      <wsdl:part name="RJCT" element="po:PurchaseOrderReject" />
   </wsdl:message>
   <wsdl:message name="InvMessage">
      <wsdl:part name="IVC" element=“po:Invoice " />
   </wsdl:message>

   <vprop:propertyAlias propertyName="cor:customerID"
      messageType="tns:POMessage" part="PO">
```

```
      <vprop:query>CID</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:orderNumber"
      messageType="tns:POMessage" part="PO">
      <vprop:query>Order</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:customerID"
      messageType="tns:POResponse" part="RSP">
      <vprop:query>CID</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:orderNumber"
      messageType="tns:POResponse" part="RSP">
      <vprop:query>Order</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:vendorID"
      messageType="tns:POResponse" part="RSP">
      <vprop:query>VID</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:invoiceNumber"
      messageType="tns:POResponse" part="RSP">
      <vprop:query>InvNum</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:vendorID"
      messageType="tns:InvMessage" part="IVC">
      <vprop:query>VID</vprop:query>
   </vprop:propertyAlias>
   <vprop:propertyAlias propertyName="cor:invoiceNumber"
      messageType="tns:InvMessage" part="IVC">
      <vprop:query>InvNum</vprop:query>
   </vprop:propertyAlias>
   ...
</wsdl:definitions>
```

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

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

   <wsdl:import namespace="http://example.com/supplyMessages"
      location="..." />

   <wsdl:portType name="PurchasingPT">
      <wsdl:operation name="Purchase">
         <wsdl:input message="smsg:POMessage" />
         <wsdl:output message="smsg:POResponse" />
         <wsdl:fault name="tns:RejectPO" message="smsg:POReject" />
      </wsdl:operation>
      <wsdl:operation name="PurchaseRequest">
         <wsdl:input message="smsg:POMessage" />
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:portType name="BuyerPT">
      <wsdl:operation name="PurchaseResponse">
         <wsdl:input message="smsg:POResponse" />
      </wsdl:operation>
      <wsdl:operation name="PurchaseReject">
```

```
         <wsdl:input message="smsg:POReject" />
      </wsdl:operation>
   </wsdl:portType>

</wsdl: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">

   <!-- 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>
```

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 one-way response. The `PurchaseOrder` `<correlationSet>` is used in both activities so that the one-way response is validated against the correlation set to correlate with the request at the buyer. The `<receive>` activity initiates the `PurchaseOrder` `<correlationSet>`. The buyer is therefore the leader and the receiving business process is a follower for this `<correlationSet>`. The `<invoke>` activity sending the one-way response also initiates a new `<correlationSet>` called `Invoice`. The business process is the leader 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/purchasing"`.

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

   <correlations>
      <correlation set="PurchaseOrder" initiate="yes" />
   </correlations>
</receive>
...
<invoke partnerLink="Buyer" portType="SP:BuyerPT"
   operation="PurchaseResponse" inputVariable="POResponse">

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

Alternatively, the response might have been a rejection (such as an "out-of-stock" message), which in this case the conversation correlated by the `<correlationSet>` `PurchaseOrder` does not trigger a new conversation correlated with `Invoice`. The `pattern` attribute is not used, since the operation is one-way.

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

   <correlations>
      <correlation set="PurchaseOrder" initiate="no" />
   </correlations>
</invoke>
```

From the perspective of the buyer's business process, the correlation sets are defined in an one-way invoke activity used for sending the purchase order and in a pick activity used for receiving the purchase order response or rejection message, respectively.

```
<invoke partnerLink="Seller" portType="SP:PurchasingPT"
   operation="PurchaseRequest" variable="PO">

   <correlations>
      <correlation set="PurchaseOrder" initiate="yes" />
   </correlations>
</invoke>
...
<pick>
   <onMessage partnerLink="Seller" portType="SP:BuyerPT"
      operation="PurchaseResponse" variable="POResponse">
      <correlations>
         <correlation set="PurchaseOrder" initiate="no" />
         <correlation set="Invoice" initiate="yes" />
      </correlations>
      ...
      <!-- handle the response message -->
   </onMessage>

   <onMessage partnerLink="Seller" portType="SP:BuyerPT"
      operation="PurchaseReject" variable="POReject">
      <correlations>
         <correlation set="PurchaseOrder" initiate="no" />
      </correlations>
      ...
      <!-- handle the reject message -->
   </onMessage>
</pick>
```

Alternatively, if the request-response purchasing operation is used in the buyer's business process, the correlation sets are specified for the request and response messages of the invoke activity, respectively. The PO rejection from the seller is sent via a fault message.

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

   <correlations>
```

```
      <correlation set="PurchaseOrder" initiate="yes"
         pattern="request" />
      <correlation set="Invoice" initiate="yes" pattern="response" />
   </correlations>

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

An `<invoke>` can consist of two messages: an outgoing request message and an incoming reply message. The `<correlationSet>`s 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 leader of that correlation. However, the buyer is a follower of the `Invoice` correlation because the values of the correlation properties for `Invoice` are initiated by the reply message of the seller received by the buyer.

10. Basic Activities

WS-BPEL activities perform the process logic. Activities are divided into 2 classes: basic and structured. Basic activities are those which describe elemental steps of the process behavior. Structured activities encode control-flow logic, and therefore can contain other basic and/or structured activities recursively. Structured activities are described in section 11. Structured Activities.

10.1. Standard Attributes for All Activities

Each activity has two optional standard attributes: the `name` of the activity and `suppressJoinFailure` (see section 5.2. The Structure of a Business Process for the definition) indicating whether a join fault should be suppressed if it occurs. WS-BPEL language extensibility allows for other namespace-qualified attributes to be added. The `name` attribute is used to provide machine-processable names for activities. WS-BPEL only makes programmatic use of the names of scope activities. See section 12.4.3. Invoking a Compensation Handler for uniqueness constraints of the name attribute. For a full discussion of the `suppressJoinFailure` attribute, see section 11.6. Parallel and Control Dependencies Processing – Flow.

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

10.2. Standard Elements for All Activities

Each activity has optional containers `<sources>` and `<targets>`, which contain standard elements `<source>` and `<target>` respectively. WS-BPEL language extensibility allows these to be extended by adding namespace-qualified elements. These, source and target, elements are used to establish synchronization relationships through links (see section 11.6. Parallel and Control Dependencies Processing – Flow).

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

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

10.3. Invoking Web Service Operations – Invoke

The `<invoke>` activity is used to call Web Services offered by service providers (see section 6. Partner Link Types, Partner Links, and Endpoint References). The typical use is invoking an

operation on a service, which is considered a basic activity. The <invoke> activity can enclose other activities, inlined in compensation handler and fault handlers, as detailed below. Operations can be request-response or one-way operations, corresponding to WSDL 1.1 operation definitions. WS-BPEL uses the same basic syntax for both, with some additional options for the request-response case.

The syntax of the <invoke> activity is summarized below.

```
<invoke partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
   <correlations>?
      <correlation set="NCName" initiate="yes|join|no"?
         pattern="request|response|request-response"? />+
   </correlations>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      faultMessageType="QName"?
      faultElement="QName"?>*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
   <compensationHandler>?
      activity
   </compensationHandler>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</invoke>
```

One-way invocation requires only the inputVariable (or its equivalent <toPart> elements) since a response is not expected as part of the operation (see section 10.4. Providing Web Service Operations – Receive and Reply). Request-response invocation requires both an inputVariable (or its equivalent <toPart> elements) and an outputVariable (or its equivalent <fromPart> elements). If a WSDL message definition does not contain any parts, then the associated attributes, inputVariable or outputVariable, MAY be omitted, [SA00047] and the <fromParts> or <toParts> construct MUST be omitted. Zero or more correlationSets can be specified to correlate the business process instance with a stateful service at the partner's side (see section 9. Correlation).

If an <invoke> activity is used on a partnerLink whose partnerRole EPR is not initialized then a bpel:uninitializedPartnerRole fault MUST be thrown.

In the case of a request-response invocation, the operation might return a WSDL fault message. This results in a fault identified in WS-BPEL by a QName formed by the target namespace of the corresponding port type and the fault name. To ensure consistent fault identification, this uniform naming mechanism MUST be followed even though it does not match the WSDL's fault-naming model. WSDL 1.1 does not require fault names to be unique within the namespace where the service operation is defined. Therefore, in WSDL 1.1 it is necessary to specify a port type name, an operation name, and the fault name to uniquely identify a fault. Using WSDL 1.1's scheme would limit the ability to use fault-identification and handling mechanisms to deal with invocation faults. In WSDL it is possible to define an operation that declares more than one fault using the same data type. Certain WSDL bindings do not provide enough information for the WS-BPEL processor to determine which fault was intended. In this case, the WS-BPEL processor MUST select the fault that:

- Matches the transmitted data and
- Occurs first in lexical order in the operation definition.

A result of this requirement is that a process, which uses the `<catch>` construct based on `faultName` and deals with such an operation definition, may have different behavior when deployed against different bindings.

Faults in WS-BPEL are defined only 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 (see section 10.6. Signaling Internal Faults) or a system fault), there is no need to keep track of the port type or operation the message activity was using when the fault was received. In consequence, all faults sharing a common name, defined in the same namespace and sharing the same data type (or lack thereof) are indistinguishable in WS-BPEL. Faults of a particular name may be associated with multiple variable types. The `<catch>` construct in WS-BPEL facilitates differentiation of faults with the same name, but with different message or variable types. For details regarding fault handling and `<catch>`, see section 12.5. Fault Handlers.

An `<invoke>` activity can be associated with another activity that acts as its compensation action. Thus, a `<compensationHandler>` can be invoked either explicitly, or by the default `<compensationHandler>` of the enclosing scope (see sections 12. Scopes and 12.3. Error Handling in Business Processes).

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

```
<invoke name="purchase"
   suppressJoinFailure="yes"
   partnerLink="Seller"
   portType="SP:Purchasing"
   operation="Purchase"
   inputVariable="sendPO"
   outputVariable="getResponse">
   <targets>
```

```
      <target linkName="linkA" />
   </targets>
   <sources>
      <source linkName="linkB" />
   </sources>
   <catch faultName="SP:rejectPO">...</catch>
   <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" />
   </sources>
   <faultHandlers>
      <catch faultName="SP:rejectPO">...</catch>
   </faultHandlers>
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation" />
   </compensationHandler>

   <invoke name="purchase"
      partnerLink="Seller"
      portType="SP:Purchasing"
      operation="Purchase"
      inputVariable="sendPO"
      outputVariable="getResponse" />
</scope>
```

In this example, the call to the `Purchase` operation can be compensated, if necessary, by a call to the `CancelPurchase` operation (see section 12.4. Compensation Handlers for details).

[SA00048] When the optional `inputVariable` and `outputVariable` attributes are being used in an `<invoke>` activity, the variables referenced by `inputVariable` and `outputVariable` MUST be messageType variables whose QName matches the QName of the input and output message type used in the operation, respectively, except as follows: if the WSDL operation used in an `<invoke>` activity uses a message containing exactly one part which itself is defined using an element, then a variable of the same element type as used to define the part MAY be referenced by the `inputVariable` and `outputVariable` attributes respectively. The result of using a 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. The copying of the element data between the anonymous temporary WSDL message variable and the element variable acts as a single virtual <assign> with one <copy> operation whose keepSrcElementName attribute is set to "yes". The virtual <assign> MUST follow the same semantics and use the same faults as a real <assign>. In the case of an inputVariable, the value of the variable referenced by the attribute 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 variable referenced by the attribute.

10.3.1. Mapping WSDL Message Parts

The <toParts> element provides an alternative to explicitly creating multi-part WSDL messages from the contents of WS-BPEL variables. By using the <toParts> element, an anonymous temporary WSDL variable is declared based on the type specified by the relevant WSDL operation's input message. The <toPart> elements, as a group, act as the single virtual <assign>, with each <toPart> acting as a <copy>. At most one <toPart> exists for each part in the WSDL message definition. Each <copy> operation copies data from the variable indicated in the fromVariable attribute into the part of the anonymous temporary WSDL variable referenced in the part attribute of the <toPart> element (see section 8.4. Assignment). If the <copy> operation is copying an element variable to an element part then the keepSrcElementName option for the operation is set to "yes". The virtual <assign> MUST follow the same semantics and use the same faults as a real <assign>. [SA00050] When <toParts> is present, it is required to have a <toPart> for every part in the WSDL message definition; the order in which parts are specified is irrelevant. Parts not explicitly represented by <toPart> elements would result in uninitialized parts in the target anonymous WSDL variable used by the <invoke> or <reply> activity. Such processes with missing <toPart> elements MUST be rejected during static analysis. [SA00051] The inputVariable attribute MUST NOT be used on an <invoke> activity that contains <toPart> elements.

The <fromPart> element is similar to the <toPart> element. The <fromPart> element is used to retrieve data from an incoming multi-part WSDL message and place it into individual WS-BPEL variables. When a WSDL message is received on an <invoke> activity that uses <fromPart> elements, the message is placed in an anonymous temporary WSDL variable of the type specified by the relevant WSDL operation's output message. The <fromPart> elements, as a group, act as a single virtual <assign>, with each <fromPart> acting as a <copy>. Each <copy> operation copies the data at the part of the anonymous temporary WSDL variable referenced in the part attribute of the <fromPart> into the variable indicated in the toVariable attribute. If the <copy> operation is copying an element part to an element variable then the keepSrcElementName option for the operation is set to "yes". The virtual <assign> MUST follow the same semantics and generate the same faults as a real <assign> (see section 8.4. Assignment). When a <fromPart> is present in an <invoke>, it is not required to have 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 not copied from the anonymous WSDL variable to the variable. [SA00052] The outputVariable attribute MUST NOT be used on an <invoke> activity that contains a <fromParts> element.

The choice to use the inputVariable form instead of the <toParts> form, or vice versa, creates no restriction on which outputVariable or <fromParts> form is used. Similarly, the choice to use the outputVariable form instead of the <fromParts> form, or vice versa, creates no restriction on which inputVariable or <toParts> form is used.

The virtual <assign> created as a consequence of the <fromPart> or <toPart> elements occurs as part of the scope of the <invoke> activity and therefore any fault that is thrown are caught by an <invoke>'s inline fault handler when defined. The <toPart> or <fromPart> elements MAY be used with WSDL messages that only have a single part.

See section 9. Correlation for an explanation of the correlation semantics.

10.4. Providing Web Service Operations – Receive and Reply

A business process provides services to its partners through inbound message activities (IMA - <receive>, <pick> and <onEvent>) and corresponding <reply> activities. This section describes the details of <receive> and <reply> activities (see sections 11.5. Selective Event Processing – Pick and 12.7.1. Message Events for <onEvent>).

A <receive> activity specifies the partnerLink that contains the myRole used to receive messages, the portType (optional) and operation that it expects the partner to invoke. The value of the partnerRole in the partnerLink is not used when processing a <receive> activity. In addition, <receive> specifies a variable, using the variable attribute, that is to be used to receive the message data. An alternative to the variable attribute is the use of <fromPart> elements. The syntax and semantics of the <fromPart> elements as used on the <receive> activity are the same as specified for the <invoke> activity in section 10.3.1. Mapping WSDL Message Parts. [SA00055] Including the restriction that if <fromPart> elements are used on a <receive> activity then the variable attribute MUST NOT be used on the same activity. If a WSDL message definition does not contain any parts, then the associated variable attribute MAY be omitted, [SA00047] and the <fromParts> construct MUST be omitted. The syntax of the <receive> activity is summarized below:

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

The <receive> activity plays 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 (or a <pick> activity) with the createInstance attribute set to "yes" (see section 11.5. Selective Event Processing – Pick for a variant). The default value of this attribute is "no". A *start activity* is a <receive> or <pick> activity that is annotated with a createInstance="yes" attribute, or an <extensionActivity> child element. In order for the <extensionActivity> child element to qualify as a start activity, it MUST exhibit the behavior of receiving an inbound message. [SA00056] Non-start activities except <scope>, <flow>, <sequence> or <extensionActivity> activities MUST have a control dependency on a start activity (see section 12.5.2. Default Compensation Order for the definition of a control dependency). If an <extensionActivity> does not have a control dependency on a start activity then the <extensionActivity> child element MUST be a structured activity containing the start activity. This structured activity MUST be consistent with the WS-BPEL process instantiation model, that is, it MUST not be a repeatable activity. If an <extensionActivity> child element is itself a start activity or contains a start activity then the namespace of the <extensionActivity> child element MUST be declared with mustUnderstand="yes". For other semantic constraints, see section 5.3. Language Extensibility. The logical order of performing activities is determined by static analysis. For an explanation of the messageExchange attribute, see the <reply> activity description in section 10.4.1. Message Exchanges.

It is permissible to have multiple start activities. An *initial start activity* is the start activity that caused a particular process instance to be instantiated. As specified in section 12. Scopes, the initial start activity MUST complete execution before any other start activities are allowed to execute. This allows any inbound message used in start activities to create the process instance since the order in which these messages arrive is unpredictable. [SA00057] If a process has multiple start activities with correlation sets then all such activities MUST share at least one common correlation set and all common correlation sets defined on all the activities MUST have the value of the initiate attribute be set to "join" (see section 9. Correlation). Conforming implementations MUST ensure that only one of the inbound messages that match a single process instance actually instantiate the business process. (It will usually be the first one to arrive, but this is implementation dependent) 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 not allowed, since the <assign> activity is not a start activity:

```
<flow>
   <!-- this example is illegal -->
   <receive ... createInstance="yes" />
   <assign ... />
</flow>
```

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

```
<flow>
   <links>
      <link name="RecvToAssign" />
   </links>
   <receive ... createInstance="yes">
```

```
      <sources>
         <source linkName="RecvToAssign" />
      </sources>
   </receive>
   <assign>
      <targets>
         <target linkName="RecvToAssign" />
      </targets>
      ...
   </assign>
</flow>
```

[SA00058] In a <receive> or <reply> activity, the variable referenced by the variable attribute MUST be a messageType variable whose QName matches the QName of the input (for <receive>) or output (for <reply>) message type used in the operation, except as follows: if the WSDL operation uses a message containing exactly one part which itself is defined using an element, then a WS-BPEL variable of the same element type as used to define the part MAY be referenced by the variable attribute of the <receive> or <reply> activity. The result of using a WS-BPEL variable in the previously defined circumstance MUST be equivalent to declaring an anonymous temporary WSDL message variable based on the associated WSDL message type. The copying of the element data between the anonymous temporary WSDL message variable and the element variable acts as a single virtual <assign> with one <copy> operation whose keepSrcElementName attribute is set to "yes". The virtual <assign> MUST follow the same semantics and use the same faults as a real <assign>. In the case of a <receive> activity, the incoming part's value will be used to set the value of the variable referenced by the variable attribute. In the case of a <reply> activity the value of the variable referenced by the variable attribute 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> sending a fault, the same logic applies.

The <fromParts> element in a <receive> activity is used as an alternative to indicate that the data from a received message is to be directly copied to WS-BPEL variables from a corresponding anonymous WSDL message variable. Similarly, the <toParts> element is used as an alternative to have data from WS-BPEL variables directly copied into an anonymous WSDL message used by the <reply> activity (see section 10.3.1. Mapping WSDL Message Parts for rules on the use of these two elements).

A <receive> is a blocking activity in that it will not complete until a matching message is received by the process instance. A business process instance MUST NOT simultaneously enable two or more <receive> activities for the same partnerLink, portType, operation and correlationSet(s) (including WS-BPEL processor-specific correlation). If during the execution of a business process instance, two or more receive activity instances for the same partnerLink, operation and correlationSet(s) are simultaneously enabled, then the standard fault bpel:conflictingReceive MUST be thrown (note bpel:conflictingReceive differs from bpel:conflictingRequest, see section 10.4.1. Message Exchanges). There may be receive activity instances on an operation where the partnerLink and correlationSet(s) are different, yet indistinguishable to a WS-BPEL processor at runtime. In these cases, a WS-BPEL processor SHOULD throw a bpel:conflictingReceive fault. If a business process instance simultaneously enables two or more IMAs for the same partnerLink, portType, operation but different correlationSet(s), and the correlations of multiple of these activities match an

incoming request message, then the `bpel:ambiguousReceive` standard fault MUST be thrown by all IMAs whose correlation set(s) match the incoming message. For the purpose of these constraints, an `<onMessage>` clause in a `<pick>` and an `<onEvent>` event handler are equivalent to a `<receive>` (see sections 11.5. Selective Event Processing – Pick and 12.7.1. Message Events).

Race conditions may occur in a business process execution. Messages that target a particular process instance may arrive before the corresponding `<receive>` activity is started. For example, consider a process that receives a series of messages in a loop where all the messages use the same correlation. At runtime, the messages will arrive independent of the iterations of the loop. The fact that the correlation is already initiated, however, should enable the runtime engine and messaging platform to recognize that these messages are correlated to the process instance, and handle those messages appropriately. Another example is a process that may invoke a remote service then initiate a correlation set for an expected callback message. For a variety of reasons, the callback message may arrive before the corresponding `<receive>` activity is started. The correlation data in the arriving message should enable the engine to recognize that the message is targeted for this process instance. Process engines MAY employ different mechanisms to handle such race conditions. This specification does not mandate any specific mechanism. Details of message delivery mechanisms are outside of the scope of this specification. However, a WS-BPEL processor should deliver messages to the process instance according to the quality of service of the underlying message delivery and transport mechanisms. For the purposes of handling race conditions, an `<onMessage>` clause in a `<pick>` and an `<onEvent>` event handler are equivalent to a receive (see sections 11.5. Selective Event Processing – Pick and 12.7.1. Message Events).

The `<reply>` activity is used to send a response to a request previously accepted through an inbound message activity such as the `<receive>` activity. These responses are only meaningful for request-response interactions. A one-way "response" can be sent by invoking the corresponding one-way operation on the partnerLink. A `<reply>` activity may specify a `variable` attribute that references the variable that contains the message data to be sent. If a WSDL message definition does not contain any parts, then the associated `variable` attribute MAY be omitted, [SA00047] and the `<toParts>` construct MUST be omitted. The syntax and semantics of the `<toPart>` elements as used on the `<reply>` activity are the same as specified in section 10.3.1. Mapping WSDL Message Parts for the `<invoke>` activity, [SA00059] including the restriction that if `<toPart>` elements are used on a `<reply>` activity then the `variable` attribute MUST NOT be used on the same activity.

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

```
</reply>
```

The <reply> activity has two potential forms. First, in a normal response, the faultName attribute is not used and the variable attribute (or its equivalent <toPart> elements), when present, will indicate a variable with the response message. Second, when the response indicates a fault, the faultName attribute is used and the variable attribute (or its equivalent <toPart> elements), when present, will indicate a variable for the corresponding fault. The faultName attribute SHOULD refer to a fault defined in the operation used in the <reply> activity and the variable SHOULD match the message type associated with the referenced fault as well (note: the matching semantics here refer to points #1 and #2 in <catch> related matching rules in section 12.5. Fault Handlers). WS-BPEL treats faults based on abstract WSDL 1.1 operation definitions. This limits the ability of a WS-BPEL process to determine the information transmitted when faults are returned over a SOAP binding (see section 10.3. Invoking Web Service Operations – Invoke).

10.4.1. Message Exchanges

The optional messageExchange attribute is used to disambiguate the relationship between inbound message activities (IMA) and <reply> activities. The explicit use of messageExchange is needed only where the execution can result in multiple IMA-<reply> pairs (e.g. <receive>-<reply> pair) on the same partnerLink and operation being executed simultaneously. [SA00060] In these cases, the process definition MUST explicitly mark the pairing-up relationship.

A <reply> activity is associated with an IMA, such as, <receive>, <onMessage> and <onEvent> based on the tuple partnerLink, operation, and messageExchange. [SA00061] The name used in the optional messageExchange attribute MUST resolve to a messageExchange declared in a scope (where the process is considered the root scope) which encloses the <reply> activity and its corresponding IMA. This resolution follows the same scoping rules as correlation set resolution.

An open IMA describes the state of a Web Service operation from the point that a request-response IMA starts execution until an associated <reply> activity completes successfully. If a <reply> activity faults, the IMA is still open and another <reply> activity MAY be attempted, for example from a fault handler. It is illegal to have multiple simultaneous open IMAs, with the same partnerLink, operation and messageExchange tuple. A WS-BPEL processor MUST throw a bpel:conflictingRequest fault when a conflicting IMA begins execution. It is legal to use the same messageExchange in multiple simultaneously open IMAs as long as the combination of partnerLink and operation on the IMAs are all different from each other. Note that bpel:conflictingRequest is semantically different from bpel:conflictingReceive, because it is possible to create the conflictingRequest by consecutively receiving the same request on a specific partnerLink, operation and messageExchange tuple, while conflictingReceive fault is not triggered (see section 10.4. Providing Web Service Operations – Receive and Reply above for conflictingReceive semantics).

If a <reply> activity cannot be associated with an open IMA by matching the tuple partnerLink, operation, and messageExchange then a WS-BPEL processor MUST throw a

bpel:missingRequest fault on the <reply> activity. Since conflicting requests are rejected at the time the IMA begins execution there cannot be more than one corresponding IMA at the time a <reply> activity is executed.

When the primary activity and the event handlers of a <scope> complete then all Web service interactions dependent on partner links or message exchanges declared inside of the <scope> need to be completed. An open IMA using a partner link or message exchange declared in a completing or completed <scope> is termed as an orphaned IMA. Detection of orphaned IMAs will cause a bpel:missingReply fault to be thrown. Orphaned IMAs are defined and discussed in further detail in section 12.2. Message Exchange Handling. Accordingly, if a process instance completes with one or more open IMAs then a bpel:missingReply fault MUST be thrown as well.

If the messageExchange attribute is not specified on an IMA or <reply> then the activity's messageExchange is automatically associated with a default messageExchange with no name. Default messageExchange's are implicitly declared by the <process> and the immediate child scopes of <onEvent> and the parallel form of <forEach>. Other occurrences of <scope> activities do not provide a default messageExchange. Default messageExchange instances, just like non-default messageExchange elements, are created each time the scope declaring the default messageExchange is executed. For example each time an <onEvent> is executed (i.e. when a new message arrives for processing) it creates a new default messageExchange instance associated with each <onEvent> instance. This allows a request-response <onEvent> event handler to receive messages in parallel without faulting or explicitly specifying a messageExchange. Similarly it allows the use of <receive>-<reply> or <onMessage>-<reply> pairs in the parallel form of <forEach> without the need to explicitly specify a messageExchange.

10.5. Updating Variables and Partner Links – Assign

Variable update occurs through the <assign> activity, which is described in section 8.4. Assignment.

10.6. Signaling Internal Faults – Throw

The <throw> activity is used when a business process needs to signal an internal fault explicitly. A fault MUST be identified with a QName (see section 10.3. Invoking Web Service Operations). The <throw> activity provides the name for the fault, and can optionally provide data with further information about the fault. A fault handler can use such data to handle the fault and 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> activity. This provides a lightweight mechanism to introduce business-process faults. A fault name defined in a business process, a WSDL definition or a WS-BPEL standard fault 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.

```
<throw faultName="QName" faultVariable="BPELVariableName"?
```

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

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

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

10.7. Delayed Execution – Wait

The <wait> activity specifies a delay for a certain period of time or until a certain deadline is reached (see section 8.3. Expressions for the grammar of duration expressions and deadline expressions). If the specified duration value in <for> is zero or negative, or a specified deadline in <until> has already been reached or passed, then the <wait> activity completes immediately.

```
<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 example 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" />
</sequence>
```

10.8. Doing Nothing – Empty

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. Another use of <empty> is to provide a synchronization point in a <flow>.

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

10.9. Adding new Activity Types – ExtensionActivity

A WS-BPEL process definition can include new activities, which are not defined by this specification, by placing them inside the <extensionActivity> element. These activities are known as extension activities. The contents of an <extensionActivity> element MUST be a

single element qualified with a namespace different from WS-BPEL namespace. That single element MUST make available WS-BPEL's `standard-attributes` and `standard-elements`. If the element contained within the `<extensionActivity>` element is not recognized by the WS-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 that has the `standard-attributes` and `standard-elements` of the unrecognized element; all its other attributes and child elements are ignored. The `standard-attributes` and `standard-elements` MUST be treated as defined by this specification, whether the extension is understood or not.

Static analysis is performed by a WS-BPEL processor *after* it ignores the non-standard-attributes and non-standard-elements of an unrecognized extension activity not subject to `mustUnderstand="yes"`. It may detect violations of some WS-BPEL required semantics. For example:

- At least one start activity MUST be present – if an `<extensionActivity>` has a nested start activity, then a requirement could be broken if non-standard child constructs of the `<extensionActivity>` are ignored.
- Links MUST have exactly one source and target – if an `<extensionActivity>` has a nested activity that is the source or target of a link that crosses the `<extensionActivity>` boundary, then a requirement would be broken if non-standard child constructs of the `<extensionActivity>` are ignored.

An `<extensionActivity>` MAY be also a structured activity, that means it contains other activities. If an `<extensionActivity>` allows a nested activity, its corresponding extension declaration SHOULD be subject to `mustUnderstand="yes"`.

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

10.10. Immediately Ending a Process – Exit

The `<exit>` activity is used to immediately end the business process instance. All currently running activities MUST be ended immediately without involving any termination handling, fault handling, or compensation behavior.

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

10.11. Propagating Faults – Rethrow

The `<rethrow>` activity is used in fault handlers to rethrow the fault they caught, i.e. the fault name and, where present, the fault data of the original fault. It can be used only within a fault handler (`<catch>` and `<catchAll>`). Modifications to the fault data MUST be ignored by `<rethrow>`. For example, if the logic in a fault handler modifies 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 (see section 12.5. Fault Handlers).

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

11. Structured Activities

Structured activities prescribe the order in which a collection of activities is executed. They describe how a business process is created; by composing the basic activities (see section 10. Basic Activities) it performs into structures that express the control patterns, handling of faults and external events, and coordination of message exchanges between process instances involved in a business protocol.

WS-BPEL defines structured activities for various control-flow patterns:

- Sequential control between activities is provided by `<sequence>`, `<if>`, `<while>`, `<repeatUntil>`, and the serial variant of `<forEach>`.
- Concurrency and synchronization between activities is provided by `<flow>` and the parallel variant of `<forEach>`.
- Deferred choice controlled by external and internal events is provided by `<pick>`.

The set of structured activities in WS-BPEL is not intended to be minimal. There are cases where the semantics of one activity can be represented using another activity. For example, sequential processing may be modeled using either the `<sequence>` activity, or by a `<flow>` with properly defined links.

Structured activities can be nested and combined in arbitrary ways. This provides a blending of graph-structured and block-structured modeling styles that have traditionally been seen as alternatives rather than orthogonal composable features. A simple example of such blended usage is found in section 5.1. Initial Example.

The word *activity* is used throughout the following to include both basic and structured activities.

11.1. Sequential Processing – Sequence

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

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

Example:

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

11.2. Conditional Behavior – If

The <if> activity provides conditional behavior. The activity consists of an ordered list of one or more conditional branches defined by the <if> and optional <elseif> elements, followed by an optional <else> element. The <if> and <elseif> branches are considered in the order in which they appear. The first branch whose <condition> holds true is taken, and its contained activity is performed. If no branch with a condition is taken, then the <else> branch is taken if present. The <if> activity is complete when the contained activity of the selected branch completes, or immediately when no <condition> evaluates to true and no <else> branch is specified.

```
<if standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>bool-expr</condition>
   activity
   <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>
      bpel:getVariableProperty('stockResult','inventory:level') > 100
   </condition>
   <flow>
      <!-- perform fulfillment work -->
   </flow>
   <elseif>
      <condition>
         bpel:getVariableProperty('stockResult','inventory:level') >= 0
      </condition>
      <throw faultName="FLT:OutOfStock" variable="RestockEstimate" />
   </elseif>
   <else>
      <throw faultName="FLT:ItemDiscontinued" />
   </else>
</if>
```

11.3. Repetitive Execution – While

The <while> activity provides for repeated execution of a contained activity. The contained activity is executed as long as the Boolean <condition> evaluates to true at the beginning of each iteration.

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

```
   activity
</while>
```

Example:

```
<while>
   <condition>$orderDetails > 100</condition>
   <scope>...</scope>
</while>
```

11.4. Repetitive Execution – RepeatUntil

The <repeatUntil> activity provides for repeated execution of a contained activity. The contained activity is executed until the given Boolean <condition> becomes true. The condition is tested after each execution of the body of the loop. In contrast to the <while> activity, the <repeatUntil> loop executes the contained activity at least once.

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

11.5. Selective Event Processing – Pick

The <pick> activity waits for the occurrence of exactly one event from a set of events, then executes the activity associated with that event. After an event has been selected, the other events are no longer accepted by that <pick>. If a race condition occurs between multiple events, the choice of the event is implementation dependent (see the race condition description in section 10.4. Providing Web Service Operations – Receive and Reply).

The <pick> activity is comprised of a set of branches, each containing an event-activity pair. The <pick> activity completes when the selected activity completes. The <pick> activity's events come in two forms:

- The <onMessage> is similar to a <receive> activity, in that it waits for the receipt of an inbound message.
- The <onAlarm> corresponds to a timer-based alarm. If the specified duration value in <for> is zero or negative, or a specified deadline in <until> has already been reached or passed, then the <onAlarm> event is executed immediately. Again, the handling of race conditions is implementation dependent.

Each pick activity MUST include at least one <onMessage>.

A special form of <pick> is used when a new instance of a business process is to be created upon the receipt of an <onMessage> event. This form of <pick> has a createInstance attribute with a value of yes (the default value of the attribute is no). [SA00062] In such a case, the events in the <pick> MUST all be <onMessage> events. This requirement MUST be statically enforced.

[SA00063] The semantics of the <onMessage> event are identical to a <receive> activity regarding the optional nature of the variable attribute or <fromPart> elements (see also [SA00047]), the handling of race conditions, the handling of correlation sets, the single element-based part message short cut and the constraint regarding simultaneous enablement of conflicting receive actions. For the last case, if two or more receive actions for the same partnerLink, portType, operation and correlationSet(s) are simultaneously enabled during execution, then the standard fault bpel:conflictingReceive MUST be thrown (see section 10.4. Providing Web Service Operations – Receive and Reply). Enablement of an <onMessage> event 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> construct with a <reply> activity (for details, see section 10.4.1. Message Exchanges).

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

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

The following example shows a typical usage of <pick>. The <pick> activity occurs in a loop that is accepting line items for a large order An order completion timeout is enabled by the <onAlarm> 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 off
        3 days and 10 hours after the last order line -->
   <onAlarm>
      <for>'P3DT10H'</for>
      <!-- handle timeout for order completion -->
   </onAlarm>
</pick>
```

11.6. Parallel and Control Dependencies Processing – Flow

The <flow> activity provides concurrency and synchronization. The syntax for <flow> is:

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

A fundamental semantic effect of grouping a set of activities in a <flow> is to enable concurrency. A <flow> completes when all of the activities enclosed by the <flow> have completed. If its enabling condition evaluates to false then an activity is skipped and also considered completed (see section 11.6.3. Dead-Path-Elimination).

In the following example, the two <invoke> activities are enabled to start concurrently when the <flow> starts. Assuming the <invoke> operations are request-response operations, the completion of the <flow> occurs after both the seller and the shipper respond. The “transferMoney” activity is executed after the <flow> completes.

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

A <flow> activity creates a set of concurrent activities directly nested within it. It enables synchronization dependencies between activities that are nested within it to any depth. The <link> construct is used to express these synchronization dependencies. Declaration of <link>'s are enclosed by a <flow> activity. [SA00064] A <link> has a mandatory name attribute, which MUST be unique among all <link> name's defined within the same immediately enclosing <flow>. This requirement MUST be statically enforced.

11.6.1. Flow-related Standard Attributes and Elements

The standard-attributes and standard-elements for activities nested within a <flow> are significant because the standard attributes and elements exist to provide link semantics to the activities. Each WS-BPEL activity has the optional containers <sources> and <targets>, which contain collections of <source> and <target> elements respectively. These elements are used to establish synchronization relationships through a <link>.

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

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

[SA00065] The value of the linkName attribute of the <source> or <target> MUST be the name of a <link> declared in an enclosing <flow> activity. [SA00068] An activity can declare itself to be the source of one or more links by including one or more <source> elements. Each <source> element associated with a given activity MUST use a linkName distinct from all other <source> elements of that activity. Similarly, [SA00069] an activity can declare itself to be the target of one or more links by including one or more <target> elements. Each <target> element associated with an activity MUST use a linkName distinct from all other <target> elements of that activity. [SA00067] Two different links MUST NOT share the same source and target activities; that is, at most one link may be used to connect two activities. [SA00066] 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. The source and target of a link can be nested arbitrarily deeply within structured activities nested in the <flow>, except for the boundary-crossing restrictions described below. All of the requirements specified in this paragraph MUST be statically enforced.

The <targets>, as a whole, can specify an optional <joinCondition>. The value of the <joinCondition> element is a Boolean expression in the expression language indicated by the expressionLanguage attribute, or in the default expression language for this process (see section 8.3. Expressions). If no <joinCondition> is specified, the <joinCondition> is the disjunction (i.e. a logical OR operation) of the link status of all incoming links of this activity.

Each <source> element can specify an optional <transitionCondition> as a guard for following the specified link. If the <transitionCondition> is omitted, it is assumed to evaluate to true.

One of the optional standard-attributes on every activity, suppressJoinFailure, is related to links. This attribute indicates whether a join fault (bpel:joinFailure) should be suppressed if it occurs (see section 11.6.3. Dead-Path-Elimination). When the suppressJoinFailure

attribute is not specified for an activity, it inherits its value from its closest enclosing construct (i.e. activity or the process itself).

The semantics of <joinCondition>, <transitionCondition>, and suppressJoinFailure are discussed below in section 11.6.2. Link Semantics.

Consider a link whose source is nested inside a syntactic construct, at any level, and the link is not declared inside that construct at any level. We say such a link is *leaving* that construct. Also consider a link whose target is nested inside a syntactic construct at any level, but the link is not declared inside that construct at any level. We say that such a link is *entering* that construct. A link which either enters or leaves a construct is said to *cross the boundary* of the construct. When both the source and target activities for the link are nested within the construct X, while the link is declared outside the construct X, the link is said to both enter and leave the construct.

The following example shows links crossing the boundaries of structured activities. The <link> named CtoD starts at activity C in <sequence> Y and ends at activity D, which is directly enclosed by the <flow> activity. 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. The link XtoY crosses the boundaries of both <sequence> X and <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 name="D" ...>
      <targets>
         <target linkName="CtoD" />
      </targets>
   </invoke>
</flow>
```

A link used within a repeatable construct (<while>, <repeatUntil>, <forEach>, <eventHandlers>) or a <compensationHandler> MUST be declared in a <flow> that is itself nested inside the repeatable construct or <compensationHandler>. [SA00070] A link MUST

NOT cross the boundary of a repeatable construct or the <compensationHandler> element. [SA00071] A link that crosses a <catch>, <catchAll> or <terminationHandler> element boundary MUST be outbound only, that is, it MUST have its source activity within the <faultHandlers> or <terminationHandler>, and its target activity outside of the scope associated with the handler (see section 12. Scopes for the specification of the <eventHandlers>, <faultHandlers>, <terminationHandler>, and <compensationHandler>) .

[SA00072] A <link> declared in a <flow> MUST NOT create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity. This implies that such directed graphs are always acyclic. Activity A is said to *logically precede* activity B if the initiation of B semantically requires the completion of A. In particular, a link MUST NOT have an activity as a target if the source activity encloses the target activity or vice versa. These requirements MUST be statically enforced.

To illustrate the above, the following example shows an invalid use of links, because it violates the restriction that a link must not have a target activity enclosed in the source activity:

```
<sequence>
   <sources>
      <source linkName="L1">
   </sources>
   ...
   <invoke ...>
      <targets>
         <target linkName="L1" />
      </targets>
   </invoke>
   ...
</sequence>
```

11.6.2. 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, we say that 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 just 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 true (see below for an explanation of link status). A join condition is a Boolean expression (see section 8.3.1. Boolean Expressions). [SA00073] The expression for a join condition MUST be constructed using only Boolean operators and the activity's incoming links' status values.

Ignoring links, the semantics of the 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. The activity contained in a

branch of an <if> is ready to start when that branch is selected. Similarly, an activity nested directly within a <flow> is ready to start when the <flow> itself starts.

If an activity that is ready to start in this sense has incoming links, then it MUST NOT start until the status of all its incoming links has been determined and the, implicit or explicit, join condition has been evaluated. In order to avoid violating control dependencies, evaluation of the join condition is performed only after the status of all incoming links has been determined.

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

The semantics of link status evaluation are described in the following paragraphs.

When activity A completes without propagating any fault, the following steps MUST be performed to determine the effect of the links on other activities:

- Determine the status of all outgoing links for A. The status will be either true or false. To determine the status for each link its <transitionCondition> is evaluated. If some of the variables referenced by the <transitionCondition> are modified in a concurrent path, the result of the transition condition evaluation may depend non-deterministically on the timing of behavior among concurrent activities.
- 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. Note that if the incoming link is leaving an isolated scope, then the final status of the link cannot be known until the isolated scope has completed (see section 12.8. Isolated Scopes).

  If both of the above conditions are true, then evaluate the <joinCondition> for B, if it evaluates to true, activity B is started. Otherwise a standard bpel:joinFailure fault MUST be thrown, unless the value of suppressJoinFailure is yes in which case bpel:joinFailure is not thrown (see section 11.6.3. Dead-Path-Elimination).

When an activity has multiple outgoing links, the order in which the status of the links and the associated transition conditions are evaluated is defined to be sequential, according to the order the links are declared in the <source> element.

The associated source activity MUST complete before the <transitionCondition> of a link is evaluated. In the case of source activities that are themselves <scope>'s, successful completion is not required. That is, 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. If an error occurs while evaluating the <transitionCondition>, 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 `false`. There is no difference in the status of the link that faults on transition condition evaluation and one whose transition condition has not been evaluated. If the target is within the enclosing scope the status is irrelevant since the scope has faulted (see section 11.6.3. Dead-Path-Elimination below). In the case of a link `L` with a `<scope>` `X` 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 `<scope> X`.

If an error occurs while evaluating the transition condition of one of an activity's outgoing links, then all remaining outgoing links with targets within the source activity's enclosing scope MUST NOT have their transition conditions evaluated and remain in the unset state. However, if the target of a remaining outgoing link is outside the source activity's enclosing scope then the status of the link MUST be set to false.

If, during the performance of structured activity `A`, the semantics of `A` dictate that activity `B` nested within `A` will not be performed as part of the execution of `A`, then the status of all outgoing links from `B` MUST be set to `false`. However, in order to avoid violating control dependencies, this rule MUST only be applied after the status of all of `B`'s incoming links, as well as all incoming links of any activity, upon which `B` has a control dependency, has been determined. An example of where this rule applies is that of an activity within an `<if>` activity's branch whose `<condition>` is `false`. Another example is seen in activities that were not completed because of a faulted `<scope>` (see sections 12. Scopes and 12.4. Compensation Handlers). The rule on control dependencies also holds for links which are outgoing from `<faultHandlers>` and `<terminationHandler>`'s: If it is determined that one of these handlers will not run, then the status of all outgoing links are set to `false`.

In the following example, the `toSkipped` link creates a control dependency from the `<receive>` activity to the `<empty>` activity in the `<if>`. The `fromSkipped` link creates a dependency from the `<empty>` activity to the `<reply>` activity. These two links create a transitive dependency from the `<receive>` activity to the `<reply>` activity. Even though the `<if>` condition evaluates to `false`, thus skipping the `<empty>` activity, the transitive dependency is retained, and therefore the status of `fromSkipped` is not set to `false` until after the status of `toSkipped` is known.

```
<flow>
   <links>
      <link name="toSkipped" />
      <link name="fromSkipped" />
   </links>

   <receive ...>
      <sources>
         <source linkName="toSkipped" />
      </sources>
      ...
   </receive>

   <if>
      <condition>
         ... <!-- evaluates to false -->
      </condition>
```

```
      <empty name="skipped">
         <targets>
            <target linkName="toSkipped">
         </targets>
         <sources>
            <source linkName="fromSkipped">
         </sources>
      </empty>
   </if>

   <reply ...>
      <targets>
         <target linkName="fromSkipped" />
      </targets>
   </reply>
</flow>
```

The <onEvent> and <onAlarm> handlers, as well as parallel <forEach> activities can have simultaneously active instances. Data and resources declared within the child scopes of these constructs, including links, MUST be processed independently in each instance.

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. A source or target reference to such a <link> from an activity matches the innermost <link> visible to the activity.

11.6.3. Dead-Path-Elimination

When the control flow is defined by links and the value of the suppressJoinFailure attribute is yes, the interpretation of a join condition for activity A that evaluates to false is that A MUST NOT be executed. In this case, the fault bpel:joinFailure MUST NOT be generated. The value of this attribute is inherited by all nested activities, except where overridden by another suppressJoinFailure attribute setting.

When a target activity is not performed due to the value of the <joinCondition> (implicit or explicit) being false, its outgoing links MUST be assigned a false status according to the rules of section 11.6.2. Link Semantics. This has the effect of propagating false link status transitively along entire paths formed by successive links until a join condition is reached that evaluates to true. This approach is called *Dead-Path Elimination* (DPE).

The default value of the suppressJoinFailure attribute of the <process> element is no. This avoids suppressing a well-defined fault by a default setting. Consider the interpretation of the example in section 5.1. Initial Example with the suppressJoinFailure attribute set to yes. Suppose further that the invocations of the shipping provider are enclosed in a scope that provides a fault handler (see sections 12. Scopes and 12.5. Fault Handlers). If one of these invocations were to fault, the status of the outgoing link from the invocation would be false, and the (implicit) <joinCondition> at the target of the link would be false, but the resulting bpel: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 `bpel:joinFailure` fault is desired, it can be achieved by setting the `suppressJoinFailure` attribute to `yes` in the `<process>` element.

11.6.4. Flow Graph Example

In the following example, the activities with the names `receiveBuyerInformation`, `receiveSellerInformation`, `settleTrade`, `confirmBuyer`, and `confirmSeller` are nodes of a graph defined within a `<flow>` activity.

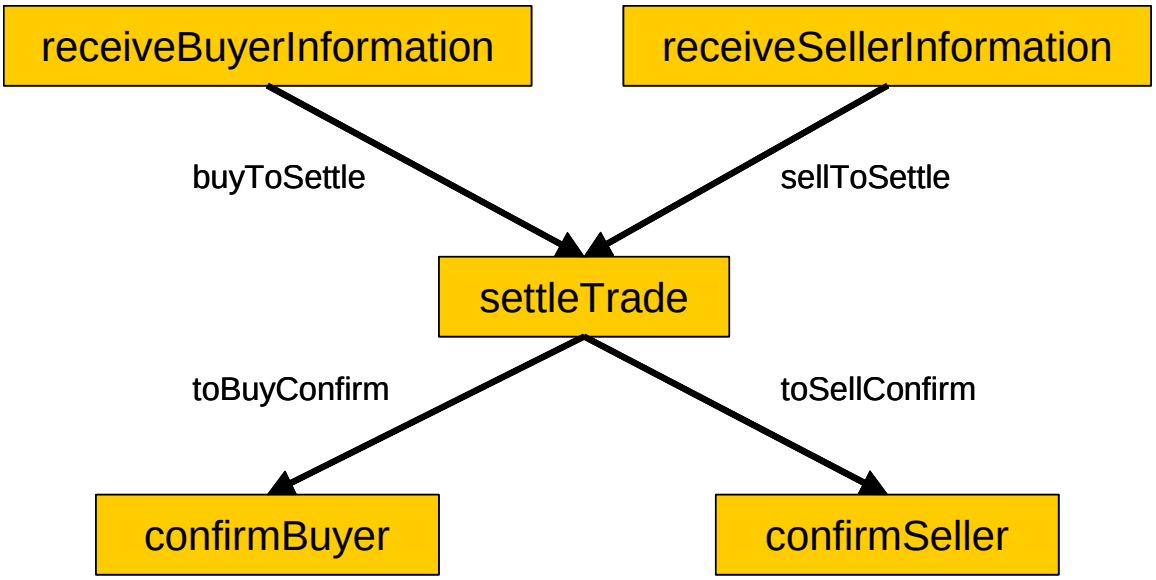


Figure 2: Flow Graph

The following `<link>`'s are defined as:

- `buyToSettle` starts at `receiveBuyerInformation` (specified in the corresponding `<source>` element nested in `receiveBuyerInformation`) and ends at `settleTrade` (specified in the corresponding `<target>` element nested in `settleTrade`).
- `sellToSettle` starts at `receiveSellerInformation` and ends at `settleTrade`.
- `toBuyConfirm` starts at `settleTrade` and ends at `confirmBuyer`.
- `toSellConfirm` starts at `settleTrade` and ends at `confirmSeller`.

Based on the graph structure defined by the `<flow>`, the activities `receiveBuyerInformation` and `receiveSellerInformation` can run concurrently. The `settleTrade` activity is performed only after 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="receiveBuyerInformation" ...>
      <sources>
         <source linkName="buyToSettle" />
      </sources>
   </receive>
   <receive name="receiveSellerInformation" ...>
      <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>
```

11.6.5. Links and Structured Activities

Links can cross the boundaries of structured activities (see section 11.6.1. Flow-related Standard Attributes and Elements). The following example illustrates the behavior when links target activities within structured constructs.

The `<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 `<joinCondition>` at `B` is not specified, and so the disjunction (i.e. a logical OR) of the links targeted to B will be used. The condition is `true` if at least one of the incoming links has a `true` status. In this case, that condition reduces to the Boolean condition `(P:funcXB() or P:funcYB())`.

In the `<flow>`, the `<sequence>` named `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 both transition conditions `P:funcXB()` and `P:funcYB()` evaluate to `false`, then the standard fault `bpel:joinFailure` will be thrown, because the attribute `suppressJoinFailure`

of the enclosing <flow> activity is set to no. Thus the behavior of the <flow> is interrupted and neither B nor C will be performed.

If the attribute suppressJoinFailure of the enclosing <flow> activity is set to yes, then B will be skipped but C will be executed because the bpel:joinFailure will be suppressed.

```
<flow suppressJoinFailure="no">
   <links>
      <link name="XtoB" />
      <link name="YtoB" />
   </links>
   <receive name="X" ...>
      <sources>
         <source linkName="XtoB">
            <transitionCondition>P:funcXB()</transitionCondition>
         </source>
      </sources>
      ...
   </receive>
   <receive name="Y" ...>
      <sources>
         <source linkName="YtoB">
            <transitionCondition>P:funcYB()</transitionCondition>
         </source>
      </sources>
      ...
   </receive>
   <sequence name="S">
      <receive name="A" ...>...</receive>
      <receive name="B" ...>
         <targets>
            <target linkName="XtoB" />
            <target linkName="YtoB" />
         </targets>
      </receive>
      <receive name="C" ... />
   </sequence>
</flow>
```

Finally, assume that the preceding <flow> is slightly rewritten by linking A, B, and C through links (with default <transitionCondition> elements with constant value of true), instead of putting them into a <sequence>. Since the default join condition is a disjunction and the <transitionCondition> of link AtoB is the constant true, the join condition will always evaluate to true, independent from the values of P:funcXB() and P:funcYB(). Now, B and subsequently C will always be executed.

```
<flow suppressJoinFailure="no">
   <links>
      <link name="XtoB" />
      <link name="YtoB" />
      <link name="AtoB" />
      <link name="BtoC" />
   </links>
   <receive name="X">
      <sources>
```

```
         <source linkName="XtoB">
            <transitionCondition>P:funcXB()</transitionCondition>
         </source>
      </sources>
   </receive>
   <receive name="Y">
      <sources>
         <source linkName="YtoB">
            <transitionCondition>P:funcYB()</transitionCondition>
         </source>
      </sources>
   </receive>
   <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>
</flow>
```

11.7. Processing Multiple Branches – ForEach

The `<forEach>` activity will execute its contained `<scope>` activity exactly `N+1` times where `N` equals the `<finalCounterValue>` minus the `<startCounterValue>`.

```
<forEach counterName="BPELVariableName" parallel="yes|no"
   standard-attributes>
   standard-elements
   <startCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </startCounterValue>
   <finalCounterValue expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </finalCounterValue>
   <completionCondition>?
      <branches expressionLanguage="anyURI"?
         successfulBranchesOnly="yes|no"?>?
         unsigned-integer-expression
      </branches>
   </completionCondition>
   <scope ...>...</scope>
</forEach>
```

When the <forEach> activity is started, the expressions in <startCounterValue> and <finalCounterValue> are evaluated. Once the two values are returned they remain constant for the lifespan of the activity. [SA00074] Both expressions MUST return a TII (meaning they contain at least one character) that can be validated as a xsd:unsignedInt. If these expressions do not return valid values, a bpel:invalidExpressionValue fault will be thrown (see section 8.3. Expressions). If the <startCounterValue> is greater than the <finalCounterValue>, then the child <scope> activity MUST NOT be performed and the <forEach> activity is complete.

The child activity of a <forEach> MUST be a <scope> activity. The <forEach> construct introduces an implicit counter variable, and also introduces dynamic parallelism (i.e. having parallel branches of which number is not known ahead of time). The <scope> activity provides a well-defined scope snapshot semantic and a way to name the dynamic parallel work for compensation purposes (see scope snapshot description in section 12.4.2. Process State Usage in Compensation Handlers).

If the value of the parallel attribute is no then the activity is a serial <forEach>. The enclosed <scope> activity MUST be executed N+1 times, each instance starting only after the previous repetition is complete. If premature termination occurs such as due to a fault, or the completion condition evaluates to true, then this N+1 requirement does not apply. During each repetition, a variable of type xsd:unsignedInt is implicitly declared in the <forEach> activity's child <scope>. This implicit variable has the name specified in the counterName attribute. The first iteration of the scope will see the counter variable initialized to the <startCounterValue>. The next iteration will cause the counter variable to be initialized to the <startCounterValue> plus one. Each subsequent iteration will increment the previously initialized counter variable value by one until the final iteration where the counter will be set to the <finalCounterValue>. The counter variable is local to the enclosed <scope> and although its value can be changed during an iteration, that value will be lost at the end of each iteration. Therefore, the counter variable value will not affect the value of the next iteration's counter.

If the value of the parallel attribute is yes then the activity is a parallel <forEach>. The enclosed <scope> activity MUST be concurrently executed N+1 times. In essence an implicit <flow> is dynamically created with N+1 copies of the <forEach>'s enclosed <scope> activity as children. Each copy of the <scope> activity will have the same counter variable declared in the same manner as specified for serial <forEach>. Each instance's counter variable MUST be uniquely initialized in parallel to one of the integer values starting with <startCounterValue> up to and including <finalCounterValue>, as a part of <scope> instantiation.

[SA00076] If a variable of the same name as the value of the counterName attribute is declared explicitly in the enclosed scope, it would be considered a case of duplicate variable declaration and MUST be reported as an error during static analysis.

The <forEach> activity without a <completionCondition> completes when all of its child <scope>'s have completed. The <completionCondition> element is optionally specified to prevent some of the children from executing (in the serial case), or to force early termination of some of the children (in the parallel case).

The <branches> element represents an unsigned-integer expression (see section 8.3.4. Unsigned Integer Expressions) used to define a completion condition of the "at least N out of M" form. The actual value B of the expression is calculated once, at the beginning of the <forEach> activity. It will not change as the result of the <forEach> activity's execution. At the end of execution of each directly enclosed <scope> activity, the number of completed children is compared to B, the value of the <branches> expression. If at least B children have completed, the <completionCondition> is triggered: No further children will be started, and currently running children will be terminated (see section 12.6 Termination Handlers). Note that enforcing the semantic of "exactly N out of M" in parallel <forEach> would involve a race condition, and is therefore not specified.

When the completion condition B is calculated, if its value is larger than the number of directly enclosed activities N+1, then the standard bpel:invalidBranchCondition fault MUST be thrown. [SA00075] Both B and N+1 may be constant expressions, and in such cases, static analysis SHOULD reject processes where it can be detected that B is greater than N+1.

The <branches> element has an optional successfulBranchesOnly attribute with the default value of no. If the value of successfulBranchesOnly is no, all <scope>'s which have completed (successfully or unsuccessfully) MUST be counted. If successfulBranchesOnly is yes, only <scope>'s which have completed successfully MUST be counted.

The <completionCondition> is evaluated each time a directly enclosed <scope> activity completes. If the <completionCondition> evaluates to true, the <forEach> activity completes:

- When the <completionCondition> is fulfilled for a parallel <forEach> activity, all still running directly enclosed <scope> activities MUST be terminated (see section 12.6 Termination Handlers).
- When the <completionCondition> is fulfilled for a serial <forEach> activity, further child <scope>'s MUST NOT be instantiated, and the <forEach> activity completes.

If upon completion of a directly enclosed <scope> activity, it can be determined that the <completionCondition> can never be true, the standard bpel:completionConditionFailure fault MUST be thrown.

When a <completionCondition> does not have any sub-elements or attributes understood by the WS-BPEL processor, it MUST be treated as if the <completionCondition> does not exist.

12. Scopes

A <scope> provides the context which influences the execution behavior of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler. Contexts provided by <scope> activities can be nested hierarchically, while the "root" context is provided by the <process> construct (see also sections 8.1. Variables, 12.4. Compensation Handlers and 12.5. Fault Handlers).

The <process> and <scope> elements share syntax constructs, which have the same semantics. However, they do have the following differences:

- The <process> construct is not an activity; hence, standard attributes and elements are not applicable to the <process> construct
- A compensation handler and a termination handler can not be attached to the <process> construct
- The isolated attribute is not applicable to the <process> construct (see section 12.8. Isolated Scopes)

Each <scope> has a required primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities to arbitrary depth. All other syntactic constructs of a <scope> activity are optional, and some of them have default semantics. The context provided by a <scope> is shared by all its nested activities.

The syntax for scope is:

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

```
      ...
   </terminationHandler>
   activity
</scope>
```

All handlers on a <scope> are lexically subordinate to the <scope> and can access all variables, partner links, message exchanges and correlation sets defined on the <scope> and its linear ancestors. This is subject to any restrictions, unique to the handler type, specified elsewhere in this document.

A <scope> can declare variables, partner links, message exchanges and correlation sets that are visible only within the <scope>. For further information, see sections 6.2. Partner Links, 8.1. Variables, 9. Correlation and 10.4. Providing Web Service Operations – Receive and Reply , respectively.

12.1. Scope Initialization

Scope initialization occurs when a <process> or <scope> is entered. Scope initialization consists of instantiating and initializing the scope's variables and partner links; instantiating the correlation sets; and installing fault handlers, termination handler and event handlers. Any partner links defined in the <scope> MUST be set 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 bpel:scopeInitializationFailure fault MUST be thrown to 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 primary activity of the <scope> is executed and the event handlers are installed in parallel with each other. An exception to the previous rule applies to <scope>'s that contain a process' 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 installed.

In the following example, the <scope> has a primary <flow> activity, which contains two concurrent <invoke> activities. Either of the <invoke> activities can receive fault responses. The <faultHandlers> 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="Purchase"
         inputVariable="sendPO"
         outputVariable="getResponse" />
      <invoke partnerLink="Shipper"
         portType="Ship:TransportOrders"
         operation="OrderShipment"
         inputVariable="sendShipOrder"
         outputVariable="shipAck" />
   </flow>
```

```
</scope>
```

12.2. Message Exchange Handling

When the primary activity and the event handlers of a <scope> complete then all Web service interactions dependent on partner links or message exchanges declared inside of the <scope> need to be completed. An *orphaned IMA* occurs when an IMA using a partner link or message exchange, declared in the completing <scope> or its descendants, remains open. In this case, the standard fault bpel:missingReply MUST be thrown. The definition of orphaned IMA situations and how they can be detected are:

- If the contained primary activity and the event handlers of the scope have completed without any unhandled fault then a check for orphaned IMA's MUST be made. If one or more orphaned IMA's are detected then a bpel:missingReply fault is thrown to the completing <scope> itself. When the bpel:missingReply fault is thrown, all the orphaned IMA's are encompassed by the fault and are no longer considered orphaned.
- If a fault handler has completed without any unhandled fault then a check for orphaned IMA's MUST be made. If any orphaned IMA is detected then a new bpel:missingReply is thrown to the parent scope (similar to throwing or rethrowing other faults from a fault handler). The newly thrown bpel:missingReply fault MUST encompass all orphaned IMA's, and they are no longer considered orphaned.
- If a fault handler itself throws or rethrows a fault different from bpel:missingReply to the parent scope then no check for orphaned IMA's is made, and the checking is deferred to the parent <scope>. The orphaned IMA's remain as such.
- The same behavior as in the previous bullet applies when a termination handler is executed.
- The same checking of orphaned IMA's is performed, after the activity of a compensation handler has completed without any unhandled fault. If any orphaned IMA's are detected, a bpel:missingReply fault MUST be propagated to the invoking FCT-handler and those IMA's are no longer considered orphaned.

  If an unhandled fault different from bpel:missingReply occurs during the execution of the compensation handler, that fault is propagated to the invoking FCT-handler. The checking for orphaned IMA's is deferred to the invoking FCT-handler. If any orphaned IMA's resulted from the execution of the compensation handler, they remain orphaned.

12.3. Error Handling in Business Processes

Business processes are often of long duration. They can manipulate 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 fault conditions and technical and business errors 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. The partial work done must be undone as best as possible. Error handling in WS-BPEL processes therefore leverages the 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 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 mechanism 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, in support of Long-Running Transactions (LRT's).

The notion of LRT described here is purely local and occurs within a single business process instance. There is no distributed coordination necessary regarding an agreed-upon outcome among multiple-participant services. The achievement of distributed agreement is an orthogonal problem outside the scope of this specification.

As an example, 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 business transactions, the compensation behavior is itself a part of the business logic and protocol, and must be explicitly specified. In this example, there might be penalties or fees applied for cancellation of an airline reservation depending on the class of ticket and the timing of the cancellation. 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 ACID transaction systems. Using `<scope>` activities as the definition of logical units of work, WS-BPEL addresses these requirements of LRT.

12.4. Compensation Handlers

The ability to declare compensation logic alongside forward-working logic is the underpinning of the application-controlled error-handling framework of WS-BPEL. WS-BPEL allows scopes to delineate that part of the behavior that is meant to be reversible in an application-defined way by specifying a compensation handler. Scopes with compensation and fault handlers can be nested without constraint to arbitrary depth.

12.4.1. Defining a Compensation Handler

Syntactically, a `<compensationHandler>` is simply a wrapper for an activity that performs compensation as shown below.

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

As explained in section 10.3. Invoking Web Service Operations – Invoke, there is a special shortcut for the <invoke> activity to inline a <compensationHandler> rather than explicitly using an immediately enclosing <scope>. For example:

```
<invoke partnerLink="Seller"
   portType="SP:Purchasing"
   operation="Purchase"
   inputVariable="sendPO"
   outputVariable="getResponse">
   <correlations>
      <correlation set="PurchaseOrder" initiate="yes"
         pattern="request" />
   </correlations>
   <compensationHandler>
      <invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation">
         <correlations>
            <correlation set="PurchaseOrder" pattern="request" />
         </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 on the same port of the same partner link, using the response to the purchase request as the input.

Without the <invoke> shortcut this example would be expressed as follows:

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

Note that the variable `getResponse` is not local to the `<scope>` to which the `<compensationHandler>` 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. 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 that is local to the `<scope>`, i.e., by making `getResponse` local to the `<scope>`. In this case, an explicit `<scope>` is needed for the variable declaration.

If the `<compensationHandler>` for a scope is not specified, default compensation handling for the scope is provided (see section 12.5.2. Default Compensation Order for more details).

12.4.2. Process State Usage in Compensation Handlers

A compensation handler always uses the current state of the process at the time the compensation handler is executed. This state comes from its associated scope and all enclosing scopes, and includes the state of variables, partner links and correlation sets. Compensation handlers are able to both read and write the values of all such data. Other parts of the process will see the changes made to shared data by compensation handlers, and conversely, compensation handlers will see changes made to shared data by other parts of the process. In cases where a compensation handler runs concurrently with other parts of the process, compensation handlers may need to use isolated scopes when they touch state in enclosing `<scope>`'s to avoid interference (see section 12.8. Isolated Scopes).

The process state consists of the current 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 (but uncompensated) scopes, their state is kept at the time of completion. Such scopes are not running, yet they are still reachable. This is because their compensation handlers are still available, and therefore the execution of such scopes may continue during the execution of their compensation handlers, which can be thought of as an optional continuation of the behavior of the associated scope. A scope may have been executed several times (e.g. in a `<while>` or in a `<forEach>`), so the state of the process includes the state of *all* successfully completed (and uncompensated) iteration instances of 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 use the state of the associated scope as it has been left. This includes variables, partner links, message exchanges, and correlation sets in both the associated scope and all scopes that enclose it. For the variables in the associated scope, the compensation handler starts executing with the scope snapshot. The compensation handler also has access to the current state of each enclosing scope. This state is shared with any concurrent units of logic. The compensation handler may itself have been called from the compensation handler of the parent scope. It will then share the continuation of the state of the enclosing scope that its caller is using.

The picture below shows three nested scopes P, S2 and S3, a fault handler FH(P) of the process and compensation handlers CH(S2) and CH(S3).

The picture is based on the XML below. When executing the process, the first scope P (the process itself) declares a variable V1 and initializes it to the value of 0. Scopes S2 and S3 are executed. At successful completion of S2 and S3, all variable values are set to 1 and are frozen into snapshots (in the timeline shown by dotted lines). Subsequently, a fault occurs within the process P (indicated by event "1" in the picture), which gets caught by the fault handler FH(P) of the process P. When the fault handler of the process calls the compensation handler CH(S2) of scope S2 (indicated by event "2" in the picture), the snapshot of S2's state is retrieved and used while compensating. The same applies when compensating scope S3 (indicated by event "3" in the picture).

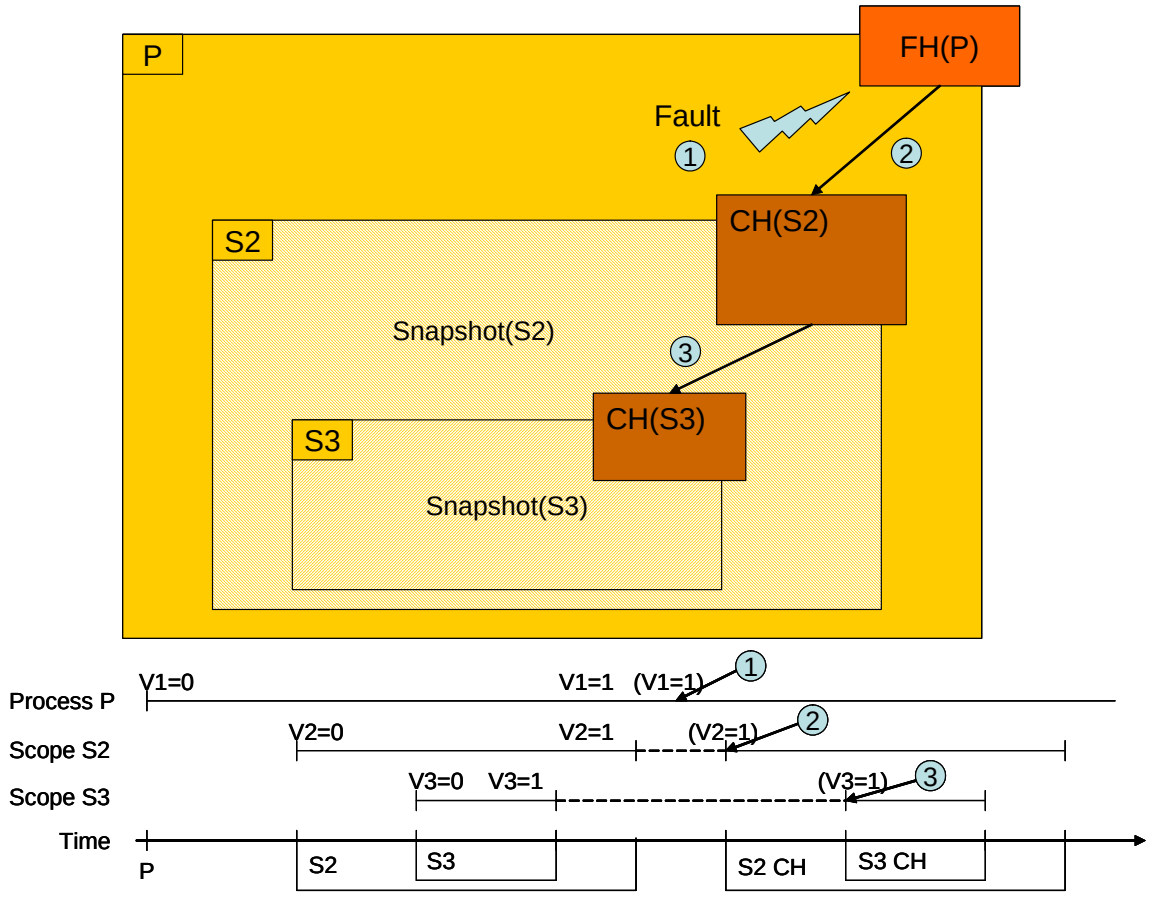


Figure 3: Variable Access in Compensation Handlers

```
<process name="P">
   <variables>
      <variable name="V1" type="xsd:int">
         <from>0</from>
      </variable>
   </variables>
   <faultHandlers>
      <catch faultName="prefix:someFault">
         <compensate />
      </catch>
   </faultHandlers>
   <scope name="S2">
```

```
      <variables>
         <variable name="V2" type="xsd:int">
            <from>0</from>
         </variable>
      </variables>
      <compensationHandler>...</compensationHandler>
      <sequence>
         <scope name="S3">
            <variables>
               <variable name="V3" type="xsd:int">
                  <from>0</from>
               </variable>
            </variables>
            <compensationHandler>
               ...
               <!-- V1, V2, and V3 ALL have the value 1
                    when this logic is reached -->
               ...
            </compensationHandler>
            <assign>
               <copy>
                  <from>1</from>
                  <to variable="V3" />
               </copy>
            </assign>
         </scope> <!-- end of scope S3 -->
         <assign>
            <copy>
               <from>1</from>
               <to variable="V1" />
            </copy>
            <copy>
               <from>1</from>
               <to variable="V2" />
            </copy>
         </assign>
         <throw faultName="prefix:someFault" />
      </sequence>
   </scope> <!-- end of scope S2 -->
</process>
```

12.4.3. Invoking a Compensation Handler

A compensation handler can be invoked by using the <compensateScope> or <compensate> (together referred to as the "compensation activities"). A compensation handler for a scope MUST be made available for invocation only when the scope completes successfully. Any attempt to compensate a scope, for which the compensation handler either has not been installed or has been installed and executed, MUST be treated as executing an <empty> activity. Therefore, handlers do not rely on state to determine which nested scopes have completed successfully.

```
<compensateScope target="NCName" standard-attributes>
   standard-elements
</compensateScope>
```

```
<compensate standard-attributes>
   standard-elements
```

```
</compensate>
```

The <compensateScope> and <compensate> activities MUST only be used within <catch>, <catchAll>, <compensationHandler>, and <terminationHandler>.

Fault handlers, compensation handlers, and termination handlers are referred to as *FCT-handlers*. For the purpose of specifying the semantics of <compensate> and <compensateScope>, a scope A is considered to *immediately enclose* another scope B, if B is enclosed in A and B is not enclosed in any other scope or FCT-handler that is itself enclosed in the outer scope A. Other structured activities (e.g. <sequence> or <forEach>) and event handlers enclosed in A do not affect the concept of immediate enclosure. This definition includes scopes that result from the <invoke> shorthand notation for fault handlers and compensation handlers.

[SA00092]Within a scope, the name of all named immediately enclosed scopes MUST be unique. This requirement MUST be statically enforced.

A <compensateScope> or <compensate> activity in an FCT-handler is used to compensate the behavior of a successfully completed scope immediately enclosed inside the scope associated with the FCT-handler. [SA00077] The value of the target attribute on a <compensateScope> activity MUST refer to the name of an immediately enclosed scope. This includes immediately enclosed scopes of an event handler (<onEvent> or <onAlarm>) associated with the same scope (see section 12.7. Event Handlers). This rule MUST be statically enforced.

FCT-handlers may themselves contain scopes. The invocation of a compensation activity is interpreted based on the immediately enclosing FCT-handler and is used to compensate the behavior of a successfully completed scope immediately enclosed inside the scope associated with that FCT-handler. There is therefore no way to use a compensation activity to compensate the scopes immediately enclosed inside an FCT-handler.

12.4.3.1. Compensation of a Specific Scope

The <compensateScope> activity causes one specified child scope to be compensated. For example:

```
<compensateScope target="RecordPayment"/>
```

The names of all named activities immediately enclosed in a scope must be unique (see section 10.1. Standard Attributes for All Activities). [SA00078] The target attribute of a <compensateScope> activity MUST refer to a scope or an invoke activity with a fault handler or compensation handler. The referenced activity MUST be immediately enclosed by the scope containing the FCT-handler with the <compensateScope> activity. If these requirements are not met then the WS-BPEL process MUST be rejected. These requirements MUST be statically enforced.

12.4.3.2. Invoking Default Compensation Behavior

The <compensate> activity causes all immediately enclosed scopes to be compensated in default order (see section 12.5.2. Default Compensation Order).

This activity is used when an FCT-handler needs to perform additional work, such as updating variables, in addition to performing default compensation for the targeted immediately enclosed scopes.

User-defined FCT-handlers may use <compensateScope> activities to compensate specific immediately enclosed scopes and/or <compensate> to compensate all immediately enclosed scopes in default order. Any repeated attempt to compensate immediately enclosed scopes is treated as executing an <empty> activity (see section 12.4.3. Invoking a Compensation Handler).

When user-defined FCT-handlers are executed, a WS-BPEL processor MUST NOT compensate immediately enclosed scopes unless the <compensate> or <compensateScope> activities are used.

12.4.4. Compensation within Repeatable Constructs or Handlers

12.4.4.1. Compensation Handler Instance Groups

Placing a scope inside a repeatable construct, such as loop or an event handler usually results in multiple instances of that scope. One scope instance is created for each repetition or event handler instantiation, respectively.

When a <compensate> or <compensateScope> activity is used to invoke the compensation handler of a scope contained in a repeatable construct, the compensation activity runs a set of installed compensation handler instances and causes the corresponding set of child scope instances to be compensated. The set of all such installed compensation handler instantiations is called a *Compensation Handler Instance Group.*

In the case of scope specific compensation (<compensateScope>), the Compensation Handler Instance Group contains the installed compensation handler instances of a particular target scope that is executed within a repeatable construct. For the case of default compensation (<compensate>), the Compensation Handler Instance Group contains the compensation handler instances of all immediately enclosed scopes that completed successfully. The compensation handler instances of immediately enclosed scopes, are treated as a single group.

If an uncaught fault occurs while executing any compensation handler instance of the group, or if compensation activities are terminated, then all running instances MUST be terminated following standard WS-BPEL activity termination semantics. All compensation handler instances of the group and compensation handler instance groups of immediately enclosed scopes are uninstalled. Completed compensation handler instances within a Compensation Handler Instance Group are not subject to further compensation.

12.4.4.2. Compensation within Repeatable Constructs

If a scope being compensated by name is nested in a <while>, <repeatUntil>, or non-parallel <forEach> loop, the invocation of the installed instances of the compensation handlers in the successive iterations MUST be in reverse order.

In the case of parallel <forEach> and event handlers, no ordering requirement is imposed on the compensation of the associated scope.

12.4.4.3. Compensation within FCT-Handlers

If a scope is enclosed inside an FCT-handler, then the enclosed scope's compensation handler is available only for the lifetime of the enclosing handler. Once the handler completes, any installed compensation handlers within it are uninstalled. [SA00079] A root scope enclosed inside a handler of the above three kinds cannot have a compensation handler associated because it is not reachable at all from anywhere within the process. Therefore, the root scope inside a handler of the above three kinds MUST NOT have a compensation handler. This rule MUST be statically enforced. Note that the root scope of an event handler (<onEvent> or <onAlarm>) can have a compensation handler.

Figure 4: Compensation within Fault Handlers shows a fault handler FH(S1) that contains a scope S2. Scope S2 cannot have a compensation handler CH(S2) as this compensation handler would be unreachable, but it may have a fault handler FH(S2) that is allowed to compensate an inner scope S3.

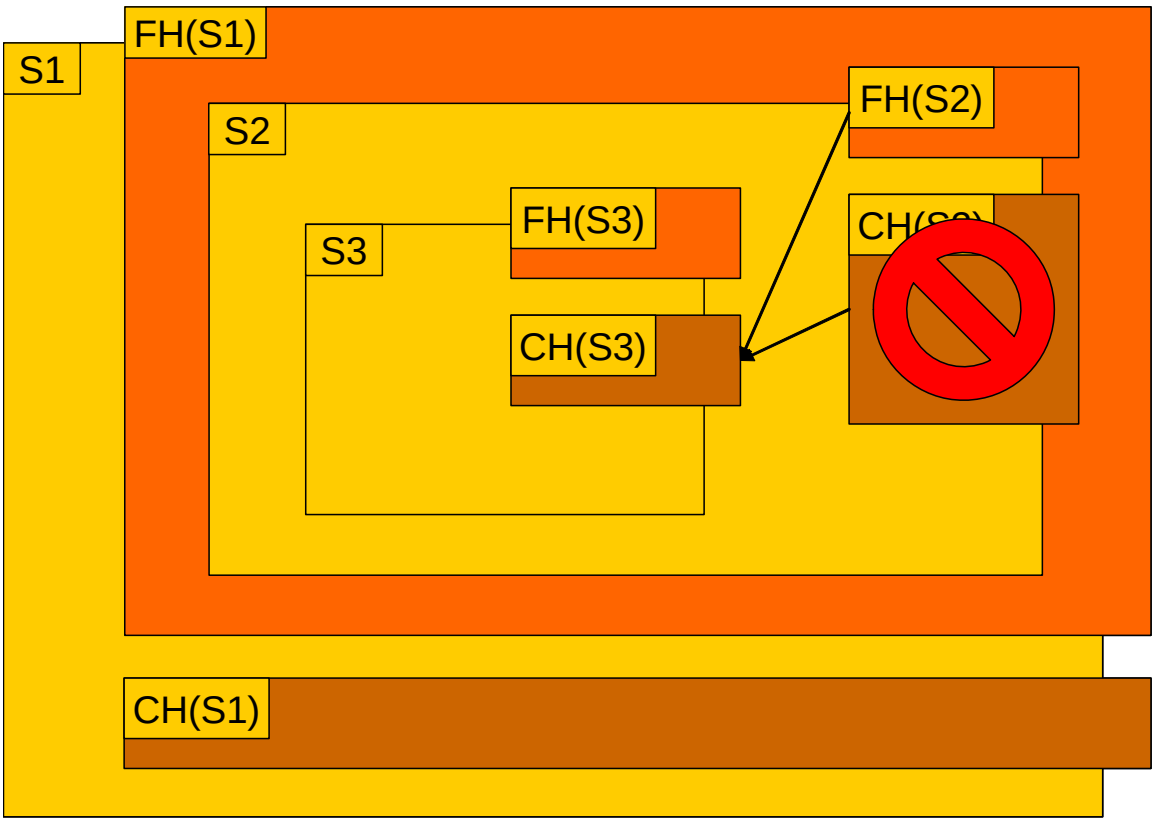


Figure 4: Compensation within Fault Handlers

Compensation within Fault Handlers

A fault in a fault handler MUST cause all running contained activities to be terminated (see section 12.6 Termination Handlers). All compensation handlers contained in the fault handler MUST be uninstalled. The fault is propagated to the enclosing scope.

Compensation within Compensation Handlers

A root scope enclosed by a compensation handler can be used to ensure “all or nothing” semantics, but not for reversing the work of a successfully completed compensation handler. If the compensation handler completes successfully then any installed compensation handlers for scopes nested within it are uninstalled. The successfully completed compensation cannot be reversed, because the root scope inside a compensation handler cannot have a its own compensation handler associated because it is not reachable at all from anywhere within the process.

A compensation handler that faults (“internal fault”) will undo its partial work by compensating all scopes immediately enclosed by the root scope according to the fault handler of the root scope. If such a fault handler is not specified explicitly, partial work will be compensated in the default order (see section 12.5.2. Default Compensation Order). This approach can be used to provide all or nothing semantics for compensation handlers. After the partial work is undone, the compensation handler MUST be uninstalled. The fault MUST be propagated to the caller of the compensation handler. This caller is a default FCT-handler of the enclosing scope or a compensation activity contained within a user-defined handler.

Figure 5: Compensation within Compensation Handlers shows a compensation handler CH(S1) that contains a scope S2. As in the preceding figure, S2 cannot have a compensation handler CH(S2) itself but may have a fault handler FH(S2) that is allowed to compensate an inner scope S3.

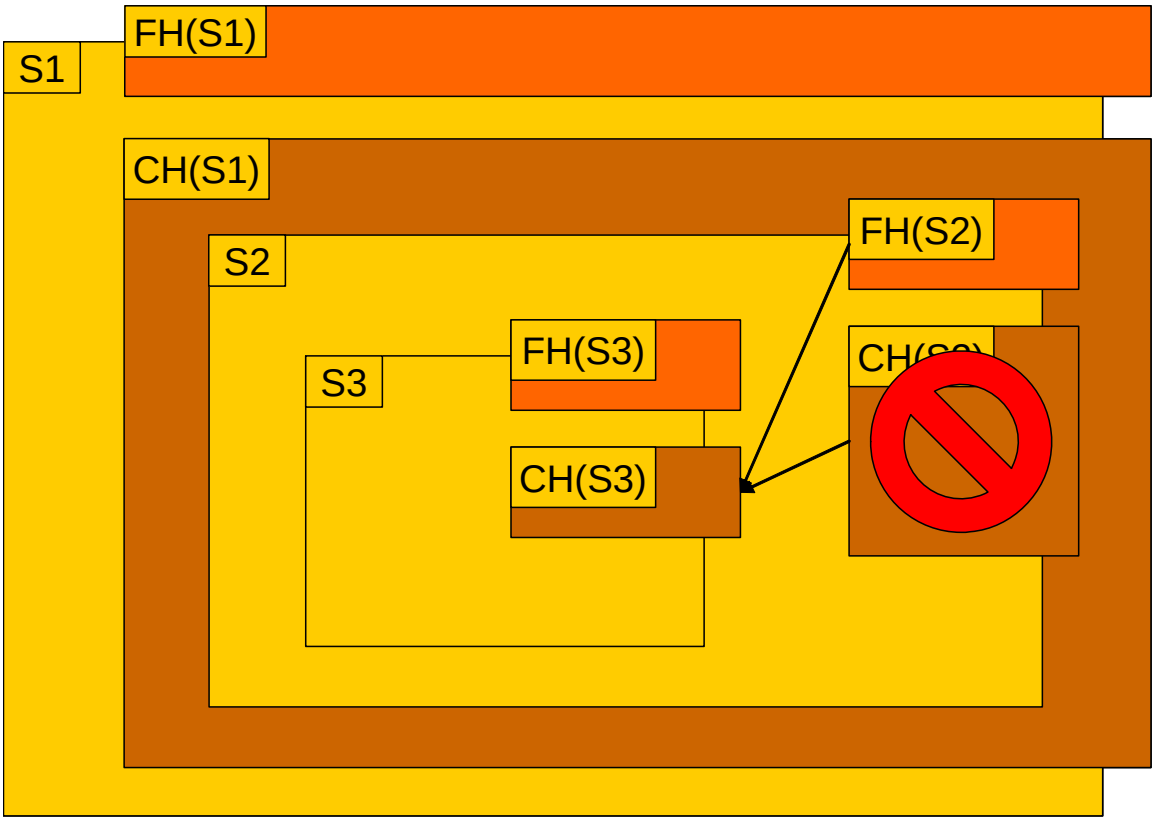


Figure 5: Compensation within Compensation Handlers

Compensation within Termination Handlers

A fault inside a termination handler MUST NOT be not propagated to the enclosing scope (see section 12.6 Termination Handlers). Other than that, all of the statements about fault handlers apply to termination handlers as well.

12.5. 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 designed to be treated as "reverse work," in that its 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 handled fault, is not considered successful completion of the attached scope. Compensation is not enabled for a scope that has had an associated fault handler invoked.

Explicit fault handlers, if used, attached to a scope provide a way to define a set of custom fault-handling activities, defined by `<catch>` and `<catchAll>` constructs. Each `<catch>` construct is defined to intercept a specific kind of fault, defined by a fault QName. An optional variable can be provided to hold the data associated with the fault. If the fault name is missing, then the catch will intercept all faults with the same type of fault data. The fault variable is specified using the `faultVariable` attribute in a `<catch>` fault handler. The variable is deemed to be implicitly 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. A <catchAll> clause can be added to catch any fault not caught by a more specific fault handler.

There are various sources of faults in WS-BPEL. A fault response to an <invoke> activity is one source of faults, where the fault name and data are based on the definition of the fault in the WSDL operation. A <throw> activity is another source, with explicitly given name and/or data. WS-BPEL defines several standard faults with their names, and there may be other platform-specific faults such as communication failures.

A fault name may be used in a WS-BPEL process without being defined elsewhere, for example in a WSDL operation; or the fault name may be missing.

```
<faultHandlers>
   <catch faultName="QName"?
      faultVariable="BPELVariableName"?
      ( faultMessageType="QName" | faultElement="QName" )? >*
      activity
   </catch>
   <catchAll>?
      activity
   </catchAll>
</faultHandlers>
```

[SA00080] There MUST be at least one <catch> or <catchAll> element within a <faultHandlers> element. This requirement MUST be statically enforced.

Data thrown with a fault can be a WSDL message type or a XML Schema element. Each <catch>, which specifies a QName as its faultName attribute value, can only catch a fault with a matching QName (see section 10.3. Invoking Web Service Operations – Invoke for the description of how to construct this QName from a fault defined in WSDL). Faults with the same name defined in multiple WSDL operations within the same WSDL namespace can be caught by the same <catch> fault handler. If the data to be caught is a WSDL message then the faultMessageType attribute is used to specify the message type's QName. If the data to be caught is a XML element then the faultElement attribute is used to specify the element definition's QName.

[SA00081] To have a defined type associated with the fault variable, the faultVariable attribute MUST only be used if either the faultMessageType or faultElement attributes, but not both, accompany it. The faultMessageType and faultElement attributes MUST NOT be used unless accompanied by faultVariable attribute.

Because of the flexibility allowed in expressing the faults that a <catch> construct can handle, it is possible for a fault to match more than one fault handler. [SA00093] While multiple fault handlers may match a fault, the <faultHandlers> element MUST NOT contain identical <catch> constructs. The <catch> constructs are considered identical in this context, when they have identical values in their faultName, faultElement and faultMessageType attributes. If an attribute is not present in a <catch>, its value is considered absent and is identical only to an absent attribute of another <catch>. A process definition that violates this condition MUST be detected by static analysis and MUST be rejected by a conformant implementation.

When faults are thrown without associated data the fault MUST be caught as follows:

1. If there is a <catch> construct 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> fault handler then the fault is passed to the <catchAll> fault handler.
3. Otherwise, the fault will be handled by the default fault handler (see section 12.5.1. Default Fault, Compensation, and Termination Handlers).

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

1. If there is a <catch> construct with a matching faultName value that has a faultVariable whose type matches the type of the runtime fault data then the fault is passed to the identified <catch> construct (see the matching criteria definition below).
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> construct with a matching faultName value that has a faultVariable whose associated faultElement's QName matches the QName of the runtime element data of the single WSDL message part, then the fault is passed to the identified <catch> construct with the faultVariable initialized to the value in the single part's element (see the matching criteria definition below).
3. Otherwise if there is a <catch> construct with a matching faultName value that does not specify a faultVariable attribute then the fault is passed to the identified <catch> construct. 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> construct without a faultName attribute that has a faultVariable whose type matches the type of the runtime fault data then the fault is passed to the identified <catch> construct (see the matching criteria definition below).
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> construct without a faultName attribute that has a faultVariable whose associated faultElement's QName matches the QName of the runtime element data of the single WSDL message part, then the fault is passed to the identified <catch> construct with the faultVariable initialized to the value in the single part's element (see the matching criteria definition below).
6. Otherwise if there is a <catchAll> fault handler then the fault is passed to the <catchAll> fault handler.
7. Otherwise, the fault will be handled by the default fault handler (see section 12.5.1. Default Fault, Compensation, and Termination Handlers).

Matching the type of a faultVariable to the runtime fault data as mentioned in points #1 and #4 above is more restrictive than in points #2 and #5. In the case of #1 and #4, a WSDL message type variable can only match a WSDL message type fault data, while an element variable can only match element-based fault data. For the case of WSDL message-based fault, they match only when their QNames are identical. For points #1 and #4, where faultElement is used, and point #2 and #5, matching is done by comparing the runtime element-based data and the faultElement's QName.

The runtime element-based data, which originates from throwing a fault with an XSD element-based variable, an XSD type-based variable or a single-part WSDL message based on an XSD element, is considered to be compatible with the globally declared element referenced by faultElement, when:

- the QName of the element-based data exactly matches the QName of the referenced element, or
- the element-based data is a member of the substitutionGroup headed by the referenced element (note: this membership relation is transitive but not symmetric).

If multiple `faultElement`-based `<catch>` constructs are compatible with element-based fault data then their matching priority is as follows:

- A `<catch>` construct with an exact QName match takes precedence.
- If no exact match exists then the matching precedence is given to a `<catch>` with a `faultElement` which has the fewest level of substitutionGroup relation in XML element declaration (see example below).

For example, `foo:Elem1`, `foo:Elem2`, `foo:Elem3`, `foo:Elem4`, `foo:Elem5` are all globally declared elements. `Elem2` is declared with its substitutionGroup attribute referring to `Elem1`. The same relationship is declared between `Elem3` and `Elem2`, and between `Elem4` and `Elem3`, and between `Elem5` and `Elem4`. Consider a scope with the following fault handlers:

```
<scope>
  <faultHandlers>
    <catch faultName="foo:BarFaultName" faultElement="foo:Elem2">
      ... catch-logic-A ...
    </catch>
    <catch faultName="foo:BarFaultName" faultElement="foo:Elem4">
      ... catch-logic-B ...
    </catch>
    <catch faultName="foo:BarFaultName">
      ... catch-logic-C ...
    </catch>
  </faultHandlers>
</scope>
```

If the fault data element is "`foo:Elem5`", the `<catch>`-logic-B based on "`foo:Elem4`" will be matched. If fault data element is "`foo:Elem3`", the `<catch>`-logic-A based on "`foo:Elem2`" will be matched. If fault data element is "`foo:Elem1`", the `<catch>`-logic-C will be matched.

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 from within the scope to which this <faultHandlers> construct is attached. 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 <catchAll> fault 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 <catchAll> fault handler.

A WS-BPEL process is allowed to rethrow the original fault caught by the nearest enclosing fault handler with a <rethrow> activity. A <rethrow> activity is allowed to be used within any fault handler and only within a fault handler. Regardless of how a fault is caught and whether a fault handler modifies the fault data, a <rethrow> activity always throws the original fault data and preserves its type.

Although the use of compensation can be a key aspect of the behavior of fault handlers, the activity within a fault handler is arbitrary, and can even be the <empty> activity. 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 process or scope in which a fault occurred is considered to have ended abnormally (i.e. completed unsuccessfully), whether or not the fault was caught and handled without rethrowing the original fault or throwing a new fault. A compensation handler is never installed for a scope which is reached by a fault.

When a fault handler for a scope completes handling a fault that occurred in that scope without throwing a fault itself, links that have that scope as the source MUST be subject to evaluation of their status.

As explained in section 10.3. Invoking Web Service Operations – Invoke, there is a special shortcut for the invoke activity to inline fault handlers rather than explicitly using an immediately enclosing scope.

The compensation handler for scope C becomes available for invocation by the FCT-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 and termination handlers are uninstalled. A WS-BPEL processor MUST NOT run more than one explicit or default FCT-handler for the same scope under any circumstances.

The behavior of fault handling for scope C MUST begin by terminating all activities that are currently active and directly enclosed within C (see section 12.6 Termination Handlers). The termination of these activities MUST occur before the specific behavior of a fault handler is started. This also applies to the default fault handlers described below. The activity of a fault handler is deemed to occur in the scope to which the fault handler is attached.

12.5.1. Default Fault, Compensation, and Termination Handlers

The visibility of scope names and therefore of compensation handlers is limited to the immediately enclosing scope. Therefore, the ability to compensate a scope would be lost if the immediately enclosing scope did not have an FCT-handler. Also many faults are not programmatic or the result of operation invocation, so it is not reasonable to expect an explicit fault handler for every fault in every scope. WS-BPEL therefore provides default fault handlers, when they are missing. Similar convenience features are applied to compensation handlers and termination handlers.

Whenever a `<catchAll>` fault handler (for any fault), `<compensationHandler>`, or `<terminationHandler>` is missing for any given `<scope>`, they MUST be implicitly created as follows.

Default fault handler:

```
<catchAll>
   <sequence>
      <compensate />
      <rethrow />
   </sequence>
</catchAll>
```

Default compensation handler:

```
<compensationHandler>
   <compensate />
</compensationHandler>
```

Default termination handler:

```
<terminationHandler>
   <compensate />
</terminationHandler>
```

12.5.2. Default Compensation Order

There are two rules for default compensation order that address different aspects of the order relation. Note that they are cumulative, 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 `<flow>`, parallel

<foreach>, or <eventHandlers>, and not otherwise constrained by links, any actual logical 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 a single default compensation behavior, the compensation handler of B MUST run to completion before the compensation handler of A is started.

In some situations, a single fault signal can trigger multiple default compensation behaviors. Rule 1 above applies to each compensation behavior individually.

Rule 1 permits scopes that executed concurrently on the forward path to also be compensated concurrently in the reverse path. The rule imposes a constraint on the order in which compensation handlers run during compensation in any default handlers of the enclosing scope, and is not meant to be fully prescriptive about the exact order and concurrency.

Of course, if one follows the strict reverse order of completion, then that necessarily respects control dependencies and is also consistent with this rule.

Informally, the second rule is needed as a result of the fact that all scopes are not isolated (see section 12.8. Isolated Scopes). 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 (see section 11.6.2. Link Semantics). This would be illegal if both such scopes were isolated. The semantics of links crossing isolated scope boundaries imply that such bidirectional links constitute a cycle. The intent of Rule 2 is to treat all scopes as if they were isolated, 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 immediately enclosed within the same 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 enclosed within S, at any depth.

Definition (Peer-Scope Dependency). If S1 and S2 are peer scopes then S2 is said to have 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: [SA00082] 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. A process definition containing a cyclic peer-scope dependency relation MUST be rejected. This MUST be enforced by static analysis.

In the following example, scope “SC1” and “SC2” are peer-scopes with respect to the process scope “P1” as their enclosing scope. Activities “InvA” and “RcvB” are within the scope-controlled set of activities of scope “SC1”, while “InvB” and “RcvA” are within the scope-controlled set of activities of scope “SC2”. Scope “SC1” is said to have a peer-scope dependency on scope “SC2” because of control link “LinkA”. Because of control link “LinkB”, there is a peer-scope dependency in the opposite direction. Hence, this process definition is not accepted by a WS-BPEL processor because of this cyclic dependency.

```
<process name="P1">
   ...
   <flow name="F1">
      ...
      <scope name="SC1">
         <flow name="F2">
            ...
            <invoke name="InvA" ...>
               <targets>
                  <target linkName="LinkA" />
               </targets>
            </invoke>
            ...
            <receive name="RcvB" ...>
               <sources>
                  <source linkName="LinkB" />
               </sources>
            </receive>
            ...
         </flow>
      </scope>
      <scope name="SC2">
         <flow name="F3">
            ...
            <invoke name="InvB" ...>
               <targets>
                  <target linkName="LinkB" />
               </targets>
            </invoke>
            ...
            <receive name="RcvA" ...>
               <sources>
                  <source linkName="LinkA" />
               </sources>
            </receive>
            ...
         </flow>
      </scope>
   </flow>
   ...
</process>
```

An effect 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. Default compensation order of a scope resulting from these rules is dependent only on the compensation of its nested scopes. The default compensation order mandated by the rules here is consistent with strict reverse order of completion. Strict reverse order of completion applied to compensation of all scopes might not be in depth-first order, and could require interleaving of nested compensations across peer scopes. Processes that require interleaving of nested compensations across peer scopes are disallowed by the rules above.

12.5.3. Relation between Compensation Handlers and Isolated Scopes

Compensation handlers may run concurrently with other activities including other compensation handlers, therefore it is necessary to allow compensation handlers to use isolation scope semantics (see section 12.8. Isolated Scopes). 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 compensation handlers MUST NOT use isolated scopes themselves because isolated scopes cannot be nested. However, their isolation environment would be 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.

12.5.4. 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 `bpel:joinFailure` reaches the scope. 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>`.

12.6 Termination Handlers

The behavior of a fault handler for a scope C begins by disabling the scope's event handlers and implicitly terminating all activities enclosed within C that are currently active (including all running event handler instances). Note that the completion condition in `<forEach>` may also trigger termination of enclosed scopes. The following paragraphs define the rules that MUST be followed for all WS-BPEL activity types.

The `<assign>` activities are sufficiently short-lived that they MAY be allowed to complete rather than being interrupted when termination is forced. The evaluation of expressions when already started is also allowed to complete. An enforced termination MAY also be allowed as WS-BPEL does not mandate a particular behavior for assignments and expression evaluations.

Each <wait>, <receive>, <reply> and <invoke> activity MUST be interrupted and terminated prematurely. When a request-response <invoke> is interrupted and terminated prematurely, the response (if received) for such a terminated activity MUST be ignored.

The <empty>, <throw> and <rethrow> activities MAY be allowed to complete. The <exit> activity, once started, MUST NOT be terminated.

All structured activity behavior is interrupted. The iteration of <while>, <repeatUntil>, and serial <forEach> MUST be interrupted and termination MUST be applied to the loop body activity. For a parallel <forEach>, termination MUST be applied to all parallel executing branches. If an <if> or <pick> activity has already selected a branch, then the termination MUST be applied to the activity of the selected branch. If either of these activities has not yet selected a branch, then the <if> or <pick> activity itself MUST be terminated immediately. The <sequence> and <flow> constructs MUST be terminated by terminating their behavior and applying termination to all nested activities currently active within them.

The <compensateScope> and <compensate> activity MUST be terminated by propagating the termination to the invoked compensation handler instances and applying termination to the activities of the compensation handlers.

Termination handlers provide the ability for scopes to control the semantics of forced termination to some degree. The syntax is as follows:

```
<terminationHandler>
   activity
</terminationHandler>
```

The forced termination of a scope begins by disabling the scope's event handlers and terminating its primary activity and all running event handler instances. Following this, the custom <terminationHandler> for the scope, if present, is run. Otherwise, the default termination handler is run.

Forced termination for a scope applies only if the scope is in normal processing mode. If the scope has already invoked fault handling behavior, then the termination handler is uninstalled, and the forced termination has no effect. The already active fault handling is allowed to complete. If the fault handler itself throws a fault, this fault is propagated to the next enclosing scope.

The termination handler for a scope is permitted to use the same range of activities as a fault handler, including the <compensateScope> or <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: (a) 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 or (b) the scope being terminated is a branch of a parallel <forEach> and the early completion mechanism has triggered the termination, as the <completionCondition> of <forEach> was fulfilled.

A fault in a termination handler MUST cause all running contained activities to be terminated (see also section 12.4.4.3. Compensation within FCT-Handlers).

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 its primary activity.

12.7. Event Handlers

Each scope, including the process scope, can have a set of event handlers. These event handlers can run concurrently and are invoked when the corresponding event occurs. The child activity within an event handler MUST be a `<scope>` activity. There are two types of events. First, events can be inbound messages that correspond to a WSDL 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>?
   <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName" )?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
      <correlations>?
         <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      <scope ...>...</scope>
   </onEvent>
   <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr</until>
      )?
      <repeatEvery expressionLanguage="anyURI"?>?
         duration-expr
      </repeatEvery>
      <scope ...>...</scope>
   </onAlarm>
</eventHandlers>
```

[SA00083] An event handler MUST contain at least one `<onEvent>` or `<onAlarm>` element. This MUST be statically enforced.

The `portType` attribute on `<onEvent>` is optional. If the `portType` attribute is included, the value of the `portType` attribute MUST match the `portType` value implied by the value of the partnerLink's myRole attribute. All instances of `<onEvent>` MUST use exactly one of `messageType`, `element`, or `<fromParts>`.

Event handlers are considered a part of the normal behavior of the scope, unlike FCT-handlers.

The activity enclosed within <onEvent> and <onAlarm> MUST be a <scope>.

When discussing event handlers, the following two terms are used to explain semantics:

- associated scope: the scope directly defined within <onEvent> or <onAlarm>
- ancestor scopes: the chain of enclosing <scope> or <process> elements of the event handler

12.7.1. Message Events

The <onEvent> element indicates that the specified event waits for a message to arrive. The interpretation of this element and its attributes is very similar to a <receive> activity. The partnerLink attribute references the partner link that contains the myRole endpoint reference on which the message is expected to arrive. [SA00084] The partnerLink reference MUST resolve to a partner link declared in the process in the following order: the associated scope first and then the ancestor scopes. This requirement MUST be enforced during static analysis. 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 port type and operation that is invoked by the partner in order to cause the event.

The variable attribute, if it exists, identifies a variable local to the event handler that will contain the message received from the partner. [SA00087] 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 element type MUST be an exact match of the element type referenced by the element attribute. The variable and messageType/element attributes constitute the implicit declaration of a variable of that name and type within the associated scope associated of the event handler. If an element attribute is used then the binding of the incoming message to the variable declared in the <onEvent> event handler occurs as specified for the receive activity in section 10.4. Providing Web Service Operations – Receive and Reply .

An alternative to the use of the variable attribute is the use of a collection of <fromPart> elements. The syntax and semantics of the <fromPart> elements as used on the <onEvent> element are the same as specified in section 10.4. Providing Web Service Operations – Receive and Reply for the receive activity. [SA00085] This includes the restriction that if <fromPart> elements are used on an <onEvent> element then the variable, element and messageType attributes MUST NOT be used on the same element, and [SA00047] the rules regarding the optional nature of the variable attribute or <fromPart> elements. When using the <fromPart> elements, each <fromPart> element constitutes an implicit declaration of a variable of that name within the associated scope of the event handler. The variable type is derived from the type of the corresponding message part. The message type of the WSDL operation can be deduced without any ambiguity, as WS-BPEL does not support WSDL with overloaded operations (see section 3. Relationship with Other Specifications).

Variables referenced by the `variable` attribute of `<fromPart>` elements or the `variable` attribute of an `<onEvent>` element are implicitly declared in the associated scope of the event handler. [SA00086] Variables of the same names MUST NOT be explicitly declared in the associated scope. This requirement MUST be enforced by static analysis.

[SA00090] If the `variable` attribute is used in the `<onEvent>` element, either the `messageType` or the `element` attribute MUST be provided in the `<onEvent>` element. This requirement MUST be enforced during static analysis.

Upon receipt of the inbound message the event handler assigns the inbound message to the variable(s) before proceeding to perform the the `<scope>` activity enclosed by the event handler. Since the variable(s) are 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(s), which is not shared with other instances. [SA00095]The variable references are resolved to the associated scope only and MUST NOT be resolved to the ancestor scopes.

The operation specified in the `<onEvent>` event handler may be either a one-way or a request-response operation. In the latter case, the event handler is expected to use a `<reply>` activity to send the response.

The usage of `<correlation>` is exactly the same as for `<receive>` activities, with the following addition: it is possible, from an event handler's inbound message operation, to use correlation sets that are declared within the associated scope. [SA00088] The resolution order of the correlation set(s) referenced by `<correlation>` MUST be first the associated scope and then the ancestor scopes.

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

```
    </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> construct is used to receive the update messages which are correlated using the travelCode correlation set, 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 not only the value in the travelCode correlation set, 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> construct comes in. When the update message is received the updateCode correlation set is initialized and its value made available only to the <onEvent> event handler instance.

Scope S2 is an immediately enclosed scope of S1. The compensation handler on scope S1 invokes the compensation handler on scope S2, which is associated with the <onEvent> event handler. If S2's compensation handler were invoked, the variable used to receive the message for the <onEvent> event handler as well as any correlation sets declared in the associated scope would be visible to the compensation handler, and as parts of the scope snapshot.

The semantics of <onEvent> are identical to those of a receive activity regarding the optional nature of the variable attribute or <fromPart> elements, the handling of race conditions, and the constraint regarding simultaneous enablement of conflicting receive actions. For the last case, see the bpel:conflictingReceive fault and its related semantics in section 10.4. Providing Web Service Operations – Receive and Reply .

When the operation specified in the <onEvent> element is a request-response operation, a message exchange is used to associate the response from a <reply> activity with the inbound message operation specified in the <onEvent> element. A message exchange is always used to pair up request and response messages. This is true even when the messageExchange attribute is not specified explicitly on the <onEvent> element, since omission of the attribute signifies use of a default message exchange (see section 10.4.1. Message Exchanges). [SA00089] When the messageExchange attribute is explicitly specified, the resolution order of the message exchange referenced by messageExchange attribute MUST be first the associated scope and then the ancestor scopes.

Event handlers do not carry the createInstance attribute, since the event handler cannot be enabled until the instance is created.

When the message constituting an event arrives, the <scope> activity specified in the corresponding event handler is executed. Business processes are enabled to receive such messages concurrently with the normal activity of the scope to which the event handler is attached, as well as concurrently with other event handler instances. This allows such events to 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 halt a process instance immediately through an external message. This event handler is attached to the <process> scope and is therefore available during the lifetime of the entire business process instance.

```
<process name="orderCar">
   ...
   <eventHandlers>
      <onEvent partnerLink="buyer"
         portType="ns:car"
         operation="haltOrder"
         messageType="ns:haltOrderMsgType"
         variable="haltDetails">
         <scope>
            <exit />
         </scope>
      </onEvent>
      ...
   </eventHandlers>
   ...
</process>
```

In this example, if the buyer invokes the haltOrder operation, the <exit> activity is executed, which results in immediate termination of the process instance without the ongoing work being compensated. Alternatively, the event handler could throw a fault to cause the ongoing work to be undone and compensated.

12.7.2. Alarm events

The <onAlarm> element marks a time-driven event. In an <onAlarm> element, the <for> and <until> expressions are mutually exclusive. There MUST be at least one <for>, <until>, or <repeatEvery> expression. 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 parent scope (the scope which directly encloses the event handler) 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. If the specified duration value in <for> is zero or negative, or a specified deadline in <until> has already been reached or passed, then the <onAlarm> event is executed immediately. The optional <repeatEvery> expression also specifies a duration. When the <repeatEvery> expression is specified, the alarm will be fired repeatedly each time the duration period expires, while the parent 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 parent scope starts. If the <repeatEvery> 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. The duration for the <repeatEvery> is calculated when the parent scope starts. If the specified duration value for <repeatEvery> is zero or negative then the standard fault bpel:invalidExpressionValue MUST be thrown.

12.7.3. Enablement of Events

The event handlers associated with a scope are enabled when the parent scope starts. If the event handler is enclosed by the <process> scope, the event handler is enabled as soon as the process instance is created. 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/example" ...>
   <wsdl:message name="orderDetails">
      <wsdl: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>
   ...
   <variables>
      <variable name="orderDetails" messageType="def:orderDetails" />
   </variables>
   ...
   <receive name="getOrder"
      partnerLink="buyer"
      operation="order"
      variable="orderDetails"
      createInstance="yes" />
   ...
</process>
```

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

12.7.4. Processing of Events

The following subsections provide rules that MUST be adhered to during processing of alarm or message events.

12.7.4.1. Alarm Events

The clock for the duration starts at the point in time when the parent scope starts. An alarm event goes off when the specified time or duration has been reached. Except for the <repeatEvery> alarm, an alarm event is executed at most once while the containing scope is active; the event is

disabled for the rest of the lifespan of the parent scope after it has occurred and the specified processing has been executed. While the parent scope is active, the `<repeatEvery>` alarm event is created repeatedly each time the duration expires. If the specified duration value for `<repeatEvery>` is zero or negative then the standard fault `bpel:invalidExpressionValue` MUST be thrown.

12.7.4.2. Message Events

A message event occurs when the appropriate message is received. When such an event occurs, the associated `<scope>` activity is executed. However, the event handler remains enabled, even for concurrent use. While the parent scope is active, a particular message event can occur multiple times (see section 12.7.7. Concurrency Considerations below for concurrency considerations).

12.7.5. Disablement of Events

When the primary activity of a scope is complete, all its contained event handlers are disabled. The already running instances of the event handlers MUST be allowed to complete, and the completion of the scope as a whole is delayed until they complete.

12.7.6. Fault Handling Considerations

When a fault occurs within the inbound message operation specified in `<onEvent>` itself (e.g. `bpel:invalidVariables` or `bpel:conflictingReceive`) or its associated scope, the fault MUST be propagated to the associated scope first. If unhandled, the fault will be propagated to the ancestor scopes chain.

12.7.7. Concurrency Considerations

Multiple `<onEvent>` and `<onAlarm>` events can occur concurrently and they are treated as concurrent activities even if they correspond to a request-response operation from the same partner link. The constraint that there can be at most one outstanding request for a request-response operation on a given partner link also applies (see `bpel:conflictingRequest` related semantics in section 10.4. Providing Web Service Operations – Receive and Reply).

When considering concurrent invocation of event handlers, including both `<onEvent>` and `<onAlarm>` with a `<repeatEvery>` expression, isolated scopes can be used to control access to shared variables (see section 12.8. Isolated Scopes).

12.8. Isolated Scopes

The isolated attribute of a scope, when set to "`yes`", provides control of concurrent access to shared resources: variables, partner links, and control dependency links. Such a scope is called an *isolated scope*. The default value of the isolated attribute is "`no`".

Suppose two concurrent isolated scopes, S1 and S2, access a common set of variables and partner links (external to them) for read or write operations. The semantics of isolated scopes

ensure that the results would be no different if all conflicting activities (read/write and write/write activities) on all shared variables and partner links were conceptually reordered so that either all such activities within S1 are completed before any in S2 or vice versa. The same isolation semantics apply to properties also. Properties are merely projections of variables and thus are always coupled with them. Access to properties is identical to access to variables, controlled by the enclosing isolated scope. It is useful to note that the semantics of isolated scopes are very similar to the standard isolation level "serializable" used in database transactions. The actual mechanisms used to ensure this are implementation dependent.

[SA00091] Isolated scopes 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.

Any message exchange referenced in a scope serves only to provide a handle to access a facet of the state of its associated partner link and is intrinsically stateless. Hence, the control exerted by the enclosing isolated scope does not apply to message exchange.

Any partner links referenced within an isolated scope have their access protected by that enclosing scope. The protection applies specifically to the `endpointReference` part, and not the message exchange parts of the partnerLink state. The same conflict isolation semantics for shared variable access are applied to the `endpointReference` part of a shared partner link state.

By definition, correlation sets are only mutable at initiation and are immutable throughout the remainder of their lifecycle. Hence any correlation sets referenced within an isolated scope do not have their access controlled by the enclosing scope. However, the initiation of a correlation set is performed in an atomic fashion – in the same sense as that of an `<assign>` operation – ensuring that the correlation set will not be partially initiated.

The used handlers in an isolated scope MUST follow these rules:

- The event handlers for an isolated scope share the isolation domain of the associated scope. The rule that isolated scopes must not be nested applies to the associated scope of an event handler as well.
- The fault handlers for an isolated scope share the isolation domain of the associated scope. In case a fault occurs in an isolated scope, the behavior of the fault handler is considered part of the isolated behavior.
- The termination handler for an isolated scope shares the isolation domain of the associated scope. When the termination handler of an isolated scope is invoked, its behavior is considered part of the isolated behavior.

- The compensation handler for an isolated scope does not share the isolation domain of the associated scope. The isolation domain ends and the scope snapshot is created when the normal processing of that isolated scope completes. Afterwards, the compensation handler is installed. If the invoker of the compensation handler (i.e. `<compensate>` or `<compensateScope>` activities or implicit invoking invoking FCT-handler of the immediately enclosing scope) is not within an isolation domain, the execution of the compensation handler associated with an isolated scope will be implicitly isolated. Such

an implicit isolation domain ends when the execution of such a compensation handler ends (see scope "FH_P" and scope "Q" in the example below). If the invoker of the compensation handler is already within an isolation domain and the invoked compensation handler is associated with an isolated scope, such a scope definition is a case of nested isolated scopes and MUST be disallowed by static analysis (if scope "FH_P" below is isolated, then such a scope definition is disallowed). (See also [SA00091]).

```
<scope name="P">
  <faultHandler>
    <catchAll>
      <scope name="FH_P">
        <sequence>
          ...
          <compensate/>
          ...
        </sequence>
      </scope>
    </catchAll>
  </faultHandler>
  <sequence>
    ...
    <scope name="Q" isolated="true">
      <compensationHandler>
        <sequence name="undoQ_Seq">...</sequence>
      </compensationHandler>
      <sequence name="doQ_Seq">...</sequence>
    </scope>
    ...
  </sequence>
</scope>
```

In the above example, the `<compensate/>` activity is NOT already within an isolation domain (assuming scope "P" is the root scope of the process). The execution of the compensation handler of scope "Q" will be isolated automatically. This isolation domain ends when the execution of the compensation handler of scope "Q" ends.

The compensation handler associated with a non-isolated scope actually shares the isolation domain of the invoker of the compensation handler, when the invoker is already within an isolation domain (see scope "FH_X" in the following example).

```
<scope name="X">
  <faultHandler>
    <catchAll>
      <scope name="FH_X" isolated="true">
        <sequence>
          ...
          <compensate />
          ...
        </sequence>
      </scope>
    </catchAll>
  </faultHandler>
  <sequence>
```

```
    ...
    <scope name="Y">
      <compensationHandler>
        <sequence name="undoY_Seq">...</sequence>
      </compensationHandler>
      <sequence name="doY_Seq"></sequence>
    </scope>
    ...
  </sequence>
</scope>
```

In the above example, the `<compensate/>` activity will invoke the compensation handler of scope "Y" (which performs sequence "undoY_seq") in the isolation domain of scope "FH_X".

The status of links leaving an isolated scope (see also section 11.6.2. Link Semantics) will not be visible at the target until the scope completes, whether successfully or unsuccessfully. If the scope completes unsuccessfully, the status of links leaving the scope is false regardless of what it was at the time the source activity completed. There are no special rules for links which enter isolated scopes.

13. WS-BPEL Abstract Processes

Abstract processes have multiple use cases. Consequently, an approach is provided for defining Abstract Processes that uses a common base, with profiles to refine it for separate use cases. The common base, defined in section 13.1. The Common Base, specifies the features that define the syntactic universe of Abstract Processes. However, the common base does not have well-defined semantics. Given this common base, a usage profile defines the necessary syntactic constraints and the semantics based on Executable WS-BPEL Processes for a particular use case for Abstract Processes. Every Abstract Process MUST identify the usage profile that defines its meaning. A profile is identified using a URI. This approach is extensible; new profiles can be defined as different areas are identified. These profiles can be defined elsewhere, outside of this specification.

Profiles are created from the common base and their characteristics are defined in section 13.2. Abstract Process Profiles and the Semantics of Abstract Processes. Two profiles are provided in this specification.

13.1. The Common Base

The common base is the “syntactic form” to which all WS-BPEL Abstract Processes MUST conform. The syntactic characteristics of the common base are:

1. The `abstractProcessProfile` attribute MUST exist. Its value refers to an existing profile definition.
2. All the constructs of Executable Processes are permitted. Thus, there is no fundamental expressive power distinction between Abstract and Executable Processes.
3. Certain syntactic constructs in WS-BPEL Executable Processes may be hidden, explicitly through the inclusion of *opaque language extensions*, and implicitly through *omission*, as detailed below in section 13.1.3. Hiding Syntactic Elements. Four types of opaque tokens are enabled: activities, expressions, attributes and from-specs.
4. An Abstract Process MUST comply with the syntactic validity constraint defined in section 13.1.4. Syntactic Validity Constraints.
5. An Abstract Process MAY omit the `createInstance` activity (`<receive>` or `<pick>`) that is mandatory for Executable WS-BPEL Processes.

13.1.1. URI

The Abstract Process syntax is denoted under the following namespace:

```
http://docs.oasis-open.org/wsbpel/2.0/process/abstract
```

13.1.2. Structure of an Abstract Process

The structure of an Abstract Process differs from that of an Executable Process only in the attributes that are permitted, as shown below:

```
<process name="NCName"
```

```
   targetNamespace="anyURI"
   abstractProcessProfile="anyURI"
   queryLanguage="anyURI"?
   expressionLanguage="anyURI"?
   suppressJoinFailure="yes|no"?
   exitOnStandardFault="yes|no"?
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract">
   ...
</process>
```

The additional top-level attribute for Abstract Processes is as follows:

- `abstractProcessProfile`. This mandatory attribute for Abstract Processes provides the URI that identifies the profile of an Abstract Process.

13.1.3. Hiding Syntactic Elements

The hiding of syntactic elements mentioned in 13.1. The Common Base, clause [3], is detailed below.

Opaque Language Extensions

Language extensions consisting of opaque tokens are used as explicit placeholders for missing details. Note that opaque tokens are not new semantically meaningful constructs but syntactic devices for indicating incompleteness. As such, opaque entities have no semantics of their own.

There are four opaque placeholders: expressions, activities, attributes and from-specs. A usage profile MAY restrict the kinds of opaque tokens allowed at its discretion. For example, a profile could specify that it allows only opaque activities, but not other kinds of opaque tokens, or a profile could specify that it allows all attributes to be opaque except the `partnerLink` attribute. However, a usage profile MUST NOT expand allowable opacity above what is allowed by the "common base". For example, a profile cannot specify that it allows a fault handler element to be opaque.

Each opaque token is a placeholder for a corresponding Executable WS-BPEL construct, as will be described below. That construct can be different in each Executable Completion (see section 13.1.4. Syntactic Validity Constraints) of an Abstract Process. For example, an opaque activity in an Abstract Process could represent an `<assign>` in one Executable Process and an `<empty>` in another Executable Process that are both valid completions of the Abstract Process.

The common base allows the following uses of opacity in Abstract Processes:

- Opaque activities are allowed.
- All WS-BPEL expressions are allowed to be opaque.
- All WS-BPEL attributes are allowed to be opaque in the common base.
- The from-spec (e.g. in `<assign>`) is allowed to be opaque.

The function of the four types of opaque tokens allowed in Abstract Processes (activities, expressions, attributes and from-specs) are described below, with examples:

Opaque activities

An opaque activity is an explicit placeholder for exactly one Executable WS-BPEL activity, and any activities that could be nested within that activity. The Executable WS-BPEL activity uses all non-opaque elements/attributes defined by the opaque activity it replaces. It also replaces any opaque attributes or expressions that are part of that opaque activity.

An opaque activity has the same standard elements and attributes common to all WS-BPEL activities (see sections 10.1. Standard Attributes for All Activities and 10.2. Standard Elements for All Activities). An opaque activity has the following form:

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

One example of using opaque activities is in the creation of a process template that marks the points of extension in a process. Another is hiding an activity that is a join point for several links when creating an Abstract Process from a known Executable Process. If that activity, on the other hand, were an activity in a `<sequence>` with no links to or from it, it could be omitted from the resulting Abstract Process. This could not be done using the `<empty>` activity, because `<empty>` explicitly means "nothing happens here". Whereas `<opaqueActivity>` means "something happens here, but it's hidden on purpose".

The reason for making an opaque activity a placeholder for one activity (and not zero or more) is that in the case of one activity there is no ambiguity regarding carrying over any attributes or elements defined on the opaque activity, or in its relation to its parent and sibling activities.

Opaque expressions

An opaque expression is a placeholder for a corresponding Executable WS-BPEL expression.

An example usage of an opaque expression is that of copying a hidden value into a known variable. Opaque expressions can be used for non-determinism: the obvious case being a process that needs to show a decision point with alternative outcomes without specifying how the decision is reached. In this case the expressions that constrain each branch may need to be left unspecified. However, it may also be convenient to make a specific value or quantity such as a price threshold unspecified, so that explicitly specified conditions relative to the threshold become non-deterministic as a result of the threshold value being unknown.

All expressions in WS-BPEL, and their corresponding opaque representations are listed below:

1. Boolean valued expressions.

- `<transitionCondition>` element of `<source>`:

```
<transitionCondition expressionLanguage="anyURI"? opaque="yes"/>
```

- `<joinCondition>` element of `<targets>`:

```
<joinCondition expressionLanguage="anyURI"? opaque="yes"/>
```

- <condition> element of <while>, <repeatUntil>, <if>, and <elseif >:

```
<condition expressionLanguage="anyURI"? opaque="yes"/>
```

2. Deadline valued expressions.

- <until> element of <onAlarm> and <wait>:

```
<until expressionLanguage="anyURI"? opaque="yes"/>
```

3. Duration valued expressions:

- <for> element of <onAlarm> and <wait>:

```
<for expressionLanguage="anyURI"? opaque="yes"/>
```

- <repeatEvery> element of <onAlarm>:

```
<repeatEvery expressionLanguage="anyURI"? opaque="yes"/>
```

4. unsignedInt valued expressions:

- <startCounterValue>, <finalCounterValue>, and <branches> elements of <forEach>:

```
<startCounterValue expressionLanguage="anyURI"? opaque="yes"/>
```

```
<finalCounterValue expressionLanguage="anyURI"? opaque="yes"/>
```

```
<branches ... expressionLanguage="anyURI"? opaque="yes"/>
```

5. Opaque expressions and queries in from-spec and to-spec, respectively:

- <from> element incorporating an expression:

```
<from expressionLanguage="anyURI"? opaque="yes"/>
```

- <from> element incorporating a query:

```
<from variable="BPELVariableName" part="NCName"?>
   <query queryLanguage="anyURI"? opaque="yes"/>?
</from>
```

- <to> element incorporating an expression:

```
<to expressionLanguage="anyURI"? opaque="yes"/>
```

- `<to>` element incorporating a query:

```
<to variable="BPELVariableName" part="NCName"?>
   <query queryLanguage="anyURI"? opaque="yes"/>?
</to>
```

Opaque from-spec

A special case for generic opaque assignment is allowed. It represents hiding any of the forms of the from-spec (see section 8.4. Assignment). The new `<opaqueFrom>` construct is used for this:

```
<opaqueFrom/>
```

Opaque attributes

An Executable WS-BPEL attribute used in an Abstract Process can have an opaque value, thereby hiding the attribute's value. We refer to these as *opaque attributes*.

For example, an opaque `variable` attribute in a `<receive>` activity hides where the data is stored once the corresponding message is received.

The value `##opaque` MUST be reserved and can be used as the value of any WS-BPEL attributes that can be opaque in an Abstract Process.

Omission

Omission may be used as a shortcut to opacity, from hereon referred to as *omission shortcut.* The omission shortcut is exactly equivalent to representing the omitted artifact with an opaque value at the omitted location. Tokens MUST only be omitted where the location can be detected deterministically. To enforce this requirement, omittable tokens are restricted to all attributes, activities, expressions and from-specs which are both (1) syntactically required by the Executable WS-BPEL XML Schema, and (2) have no default value. Note that it is allowed to omit the start activity in an Abstract Process as well (see section 13.1.3. Hiding Syntactic Elements, [5]). If the omitted token is an activity, the implied opaque activity MUST have the exact form `<opaqueActivity/>` (i.e.: no standard-elements or standard-attributes). Notice that (1) deliberately excludes any non-Schema requirements of Executable WS-BPEL.

Therefore, an Abstract Process P1 that uses the omission shortcut is always equivalent to an Abstract Process P2 that is the same as P1 but injects opaque tokens anywhere they have been omitted and does not use the omission shortcut. To illustrate, consider a process that omits the variable attribute in all invoke activities. This is equivalent to another process which is identical to P1 except that it includes the `variable` attribute on all its invokes but with the value `##opaque`, and vice versa.

13.1.4. Syntactic Validity Constraints

Definition (Executable Completion). An *Executable Completion* of an Abstract Process is defined as an Executable Process that

1. is derived only by:

    a) Changing the namespace to that of Executable WS-BPEL and removing the profile URI.
    b) Using some combination of the following syntactic transformations:

        i. Opaque Token Replacement: Replacing every opaque token (including those omitted using the omission-shortcut) with a corresponding Executable token. For example, replacing an opaque activity with an `<empty>`.
        ii. Addition of WS-BPEL constructs: Adding new WS-BPEL XML elements anywhere in the process.

2. is a valid Executable WS-BPEL process that satisfies all static validation rules mandated by this specification including both Schema-based and non-Schema-based rules.

A clarification is provided here regarding the completion rules and their application to constructs with default values in the Abstract Process, such as “createInstance” at <receive>, “validate” at <assign> and <joinCondition> within <targets>. The completion rules above do not allow changing non-opaque constructs when creating an executable completion (whether through omission-shortcut or explicitly). As stated in section "13.1.3. Hiding Syntactic Elements", a construct, which is not explicitly present in the abstract process and has a default value, is not allowed to be made opaque through omission-shortcut. Its value will be that of the default in all executable completions (e.g.: “no” for an omitted “suppressJoinFailure” attribute). Therefore, specifying its value in an executable completion is not covered by “Addition of WS-BPEL constructs.” In order to allow Execution Completion to specify the value of such constructs, explicit opaque tokens should be used in the Abstract Process. Completions can then specify the values specified using “Opaque Token Replacement”.

This is especially relevant to the addition of links. New links cannot be added as targets to existing activities with at least one link if such an addition changes an existing, non-opaque join condition (including the default join condition). The default join condition is included in this consideration because adding a new link to an activity using the default join condition effectively changes the condition to include the new link’s status. Examples where new links can be added include adding them to: 1) an activity with no existing incoming links, 2) an activity with incoming link(s) and an opaque join condition, or 3) an activity with incoming link(s) and an explicitly specified, non-opaque join condition (whose value cannot be changed in any executable completion).

Definition (Basic Executable Completion). A *Basic Executable Completion* of an Abstract Process is defined as an Executable Completion whose allowed syntactic transformations MUST be limited to:

- Opaque Token Replacement (1. (b) i. above),
- the addition of a start activity if none are present in the Abstract Process (per clause [5] of section 13.1.3. Hiding Syntactic Elements), and
- the addition of `<import>`, `<partnerLinks>`, `<partnerLink>`, `<variables>`, `<variable>` elements at the `<process>` level, as long as the declarations of any

such newly added elements are not referenced by existing constructs in the AP (before opaque token replacement).

An Abstract Process MUST be considered valid if and only if it meets the following criteria, as referred to in 13.1. The Common Base, clause [4] :

- It conforms to the WS-BPEL XML Schema for the common base, as defined in Appendix E. XML Schemas. This is purely an XML Schema check and does not enforce any non-Schema validation rules, such as requiring that every link that has a source must also have a target.
- Any extension construct, including the extension attribute, elements, extension activity and extension assign operations, is declared properly with the "namespace" and "mustUnderstand" not being opaque (including omission-shortcuts). See more details in sections 5.3. Language Extensibility and 14. Extension Declarations.
- There exists at least one Basic Executable Completion of the Abstract Process.

The purpose of the last bullet above is to improve the static validation of an Abstract Process beyond the XML Schema check. This limits the creation of ill-defined constructs in the Abstract Processes that the Schema would otherwise allow. On the other hand, the semantics of an Abstract Process comes from the range of Executable Processes that can be created from the Executable Completions (not limited to Basic Executable Completions) allowed by its profile.

There is no fundamental expressive power distinction between Abstract and Executable Processes. To accommodate the syntactic flexibility introduced by allowing opacity and omission in the syntax of Abstract Processes, the XML Schema for the Common Base of Abstract Processes does not reuse any definitions from XML Schema for the Executable Processes. The two have distinct namespaces: one for Abstract WS-BPEL Processes and one for Executable WS-BPEL Processes.

At the same time, an Abstract Process Profile may be required to extend the level of syntactic validation from that of the common base to support the inclusion of additional information necessary to it. Therefore an Abstract Profile MAY provide:

- extension constructs in its own namespace to be added to the Abstract Process,
- additional XML grammar to support its own specific syntax validation.

Abstract Processes defined using any profile MUST validate according to the grammar of the common base.

13.1.5. Interpretation of the Common Base

The common base, being extremely flexible, does not have well-defined semantics. On the other hand, Executable WS-BPEL Processes have well-defined semantics and prescribed behavior. The semantics of an Abstract Process are provided by the set of Executable WS-BPEL Processes that the particular Abstract Process represents. This set is provided in usage profiles, and varies from one profile to another. In other words, the semantics of an Abstract Process depend on its associated profile.

In addition to semantics, the consistency constraints of Executable WS-BPEL are clearly defined. The semantics of each language construct in an Abstract Process MUST be derived from that of the same construct in Executable WS-BPEL. For example, an `<invoke>` in an Abstract Process always represents invoking a Web service operation as used in Executable Processes. The difference is strictly a consequence of the opacity used in that construct (missing information) and other parts of the process affected by it (for example, opacity in a link source element may affect the link target element). Any required clarifications depending on allowed opacity will be specified in the relevant usage profile.

In the common base definition, there are no requirements how Executable realizations of a given Abstract Process should be implemented (i.e. language, platform, etc.); nor are specific relationships with such realizations implied. Again, a concrete usage profile might provide such information based on its use case.

13.2. Abstract Process Profiles and the Semantics of Abstract Processes

The common base for Abstract Processes specifies the syntactic universe within which Abstract Processes are defined. The common base does not provide any semantics for Abstract Processes since the semantics must express a specific intent for the interpretation of an Abstract Process and the common base provides no mechanism to express such intent.

It is a profile that defines a class of Abstract Processes with a shared semantic interpretation. Abstract Processes are incomplete and by definition not Executable, whether or not they contain opaque entities. The semantics of the non-opaque constructs they contain cannot be understood in isolation. Their semantics are bound by the Executable Completions that are permitted by the profile referenced by the Abstract Process. The semantics of those constructs can be realized only in the possible Executable Completions. As an edge case, a permitted completion may sometimes be virtually identical to the Abstract Process syntactically, but this is the exception rather than the rule.

A WS-BPEL Abstract Process and a WS-BPEL Executable Process are said to be *compatible* if the Executable Process is one of the Executable Completions in the set of permitted completions specified by the Abstract Process' Profile. Compatibility for Executable Processes that are not WS-BPEL processes is outside the scope of this specification. Clearly, an Executable Process can exist independently from an Abstract Process.

A profile MUST NOT violate the common base. A profile MUST define

(i) A URI that identifies the profile and is used as the value of the `abstractProcessProfile` attribute by all Abstract Processes belonging to this profile.

(ii) The set of syntactically valid Abstract Processes that belong to this profile, as a subset of the common base. Note that the subset does not have to be proper, i.e., it may include the entire common base. Example profiles include those that disallow control links or certain types of opaque tokens. Note further that the subset must be consistent with respect to the use of the omission-shortcut. Specifically, if a profile limits the use of opaque tokens in the set of Abstract Processes it covers, then it can only permit those omissions that

correspond to permitted usage of opaque tokens. For instance, if a profile does not allow attributes to be opaque, then Abstract Processes belonging to that profile cannot omit attributes using the omission-shortcut.

(iii) The set of permitted Executable Completions for Abstract Processes that belong to the set in (ii). The set of permitted Executable Completions MUST be non-empty for each Abstract Process in the set in (ii).

Any Abstract Process that belongs to a given profile MUST follow the restrictions defined in that profile.

If the allowed level of opacity in a profile leads to the inability to relate constructs in the abstract process, the profile MUST provide additional syntactic constraints to ensure that a user can match the constructs. Examples include a receive/reply pair with opaque operation attributes, or a link source/target pair with an opaque name attribute.

Another example is a profile that allows "Opaque Token Replacement" and the addition of only WS-BPEL constructs that create leaf-nodes or sub-trees. By disallowing arbitrary additions, such a profile would not allow Executable Completions to do such things as wrap an existing activity with a `<while>`, or add a `<sequence>` around activities in a `<flow>`. On the other hand, it would allow the creation of new leaf activities inside an existing `<flow>`.

13.3. Abstract Process Profile for Observable Behavior

The objective of the Abstract Process Profile for Observable Behavior is to provide precise and predictable descriptions of observable service behavior. The main application of this profile is the definition of business process contracts; that is, the behavior followed by one business partner in the context of Web services exchanges. Business process contracts are particularly relevant in automated cross enterprise interactions but have general applicability in the extension of service contracts with precise, machine processable behavioral descriptions.

There are several key differences between processes intended to represent business process contracts and Executable Processes. Foremost among them is the different way in which data are handled in each case; the rich data manipulation that occurs in Executable Processes need not be described in public process contracts. Instead, public process contracts use non-deterministic data values to hide the private aspects of executable behavior. For example, using opaque assignment supports modeling the non-deterministic effects that private computation has on external behavior.

In this profile, the use of opacity is concentrated in those features associated with data handling. Non-deterministic data values are not allowed in Executable Processes; Abstract Processes, on the other hand, use non-deterministic values to reflect the consequences of actual behavior while maintaining the details of that behavior to remain private.

13.3.1. Profile URI

The URI identifying this Abstract Process Profile is:

```
http://docs.oasis-open.org/wsbpel/2.0/process/abstract/ap11/2006/08
```

13.3.2. Subset of the Processes Allowed in the Common Base

This profile is concerned with hiding internal processing of a business partner's process while capturing all the information required to describe how the process interacts with its partners. The set of usage restrictions associated with the use of Abstract Processes in general was in fact derived from this original requirement, which was captured by the Abstract Process definition incorporated in the previous version of this specification ([BPEL4WS 1.1]).

This profile applies opacity in WS-BPEL constructs that handle data. In addition, the omission-shortcut described in 13.1.3. Hiding Syntactic Elements can be used as an alternative to explicitly specifying opaque tokens. The profile described here allows the use of opaque activities specifically for supporting the two cases where an activity is syntactically required. The first is hiding internal processing that needs to be the source or target of links in the Abstract Process, while maintaining the same flow of control in the abstract representation. The second is the use of opacity (and consequently the omission shortcut) in places where an activity is required by the WS-BPEL semantics and Schema. For example, Executable Processes are required to have an activity in a fault handler. Using an opaque activity avoids the need to use an `<empty>` activity. The use of opaque activities where an activity is not syntactically required is superfluous, because this profile's completion rules are flexible about where one can add an activity in an Executable Completion. The full completion rules are presented in the next section.

This profile restricts the use of the Abstract Process Common Base in the following manner:

- Expressions: `<joinCondition>` is not allowed to be opaque. The `<joinCondition>` has a default value, and is based only on of the status of the incoming links, and not on data handled by the process. Therefore, it is not appropriate to hide it. All other expressions may be opaque, as defined in section 13.1.3. Hiding Syntactic Elements.
- Activities: The use of `<exit>` is not allowed.
- Attributes: Only the attributes used for identification of variables and message parts of message related constructs representing partner interactions are allowed to be opaque. The full list of the attributes allowed to be opaque is shown below. The following is the complete list of attributes, belonging to the `<receive>`, `<invoke>`, `<reply>`, `<onMessage>`, or `<onEvent>` constructs, that are allowed to be opaque in this profile:
  - `variable`, `inputVariable`, `outputVariable` attributes.
  - `part` and `toVariable` attributes of the `<fromPart>` element.
  - `part` and `fromVariable` attributes of `<toPart>` element.
- From-specs: Opaque from-specs are allowed.

The level of abstraction appropriate in the description of business process contracts makes it often unnecessary to use message variables in Web service message activities, particularly when the intent is to simply constrain the sequencing of such activities and the actual message data is not relevant.

13.3.3. The Use of Opaque Variable References

Unlike Executable Processes, variables in Abstract Processes defined using this profile do not need to be initialized before being referenced since additional computation may be implicitly assumed.

Executable Processes are expected to follow constraints such as initializing variables before they are used. Clearly, Executable Completions of Abstract Processes that hide variable references and data manipulation are expected to abide by the constraints and requirements of executable processes.

13.3.4. Subset of the Executable Completions Allowed in the Base

With respect to executable BPEL completions of an abstract process that uses this profile, the intent of the profile requires a valid completion to follow the same interactions as the abstract process, with the partners that are specified by the abstract process. The executable process may, however, perform additional interaction steps relating to other partners. This section uses the term ‘existing’ to refer to constructs present in an abstract process, and the term ‘new’ to refer to those added to an abstract process while creating an executable completion.

In order to achieve the intent of the profile, a completion MUST NOT change the presence or order of interactions already in the abstract process, and it MUST NOT perform additional interactions with the partner links defined in the abstract process. The completion rules provided below aim to enforce this restriction.

Data writing may cause changes in interaction order. Changes caused by data writing are not enforced by the completion rules, but are highlighted here as an advisory note. One example is changing the value of a variable used in a condition that affects branching, in such a way that the new effective branching behavior is in direct conflict with what is specified by the abstract process. Conditions that affect the flow of control such as transition conditions, “if” or “while” expressions, among others, can have such an effect on the order of interactions. For example, adding a new `<while>` loop with a “true” condition as a child of an existing `<sequence>` would hang the process.

When creating an executable completion of an abstract process belonging to this profile, the possible locations for adding new activities are not explicitly defined: Activities may be added anywhere that the Executable Completions definition in section [see 13.1.4. Syntactic Validity Constraints] allows with the restrictions below.

In this profile, the valid executable completions of an abstract process are obtained through both 'opaque token replacement' and 'addition of BPEL constructs', with the following restrictions.

New activities (including those created to replace opaque activities) MUST NOT interact with partnerLinks already declared in the abstract process. This rule does not affect adding interactions with new partnerLinks present in the executable completion but not in the abstract process.

- The endpoint reference of any partnerLink defined in the abstract process MUST NOT be modified (whether using an `<assign>` activity or otherwise). Additionally, an endpoint

reference of any partnerLink defined in the abstract process MUST NOT be copied into the reference of a newly created partnerLink. The reason is that the former would effectively prevent subsequent interactions with that partner and the latter would add new ones. Remember that 'opaque token replacement' also replaces opaque tokens omitted through the omission-shortcut.

- The lexical parent of an existing BPEL construct (including activities in particular) present in the abstract process MUST NOT be changed in an executable completion. Hence, the nesting structure of composite activities around any activity in an abstract process remains unchanged in any legal completion. Some examples to illustrate this restriction are provided below. The word 'existing' is used in the examples to refer to constructs defined in the abstract process from which the executable completions are being created:
    - Examples of legal additions:
        - Adding a variable or a partner link to an existing scope S, even though that scope is the parent of existing activity A, except as disallowed above.
        - Adding a new link definition to an existing flow, except as disallowed above.
    - Examples of illegal additions:
        - Adding a `<while>` activity around an existing activity.
        - Adding a new scope around an existing variable definition.
- A valid executable completion MUST NOT add:
    - New branches to an existing "if-else" activity, unless it has no "else" branch, and the new branch is added after all existing branches.
    - New branches to an existing pick activity.
    - New fault, compensation, or termination handlers to an existing scope. However, they may be added at the process level.
    - `<exit>` activities.
    - New links whose targets are existing activities. The Executable Completions definition in the Base already disallows adding new links to existing activities that have existing links and use the default join condition. This profile restricts this further by disallowing the addition of new links to *any* existing activity. However, one may freely add links targeting new activities as long as those activities are not a replacement of one of the abstract process's opaque activities.
    - Declarations of variables, partner links, and correlation sets in existing scopes if they hide existing declarations that are used by existing constructs in the scope.
- Activities that throw non-standard faults (e.g. web service activities whose operations define faults, `<throw>`) MAY be added only if the exception will not be propagated to any activity existing in the Abstract Process. For example, consider adding an activity A as a child of an existing sequence S. Then, one may only add a `<throw>` within A if the fault it throws does not reach the scope of the existing sequence S. In other words, the fault must be caught and fully handled by A or its descendants, and not be re-thrown by them.

Recall from the definition of Executable Completion in the Base that if a construct is optional and has a default value, then the construct needs to be explicitly opaque, in order to allow Executable Completion to specify its value. One example that highlights that is an Abstract Process with a `<receive>` activity or other IMA that does not have the createInstance attribute.

Such an activity is always treated as a non-start activity, an Execution Completion cannot add `createInstance="yes"` to it. If one wants to make a `<receive>` activity or other IMA optionally become a start activity, the createInstance attribute has to be made explicitly opaque.

13.4. Abstract Process Profile for Templates

A high-level design-time representation may be used by a technical analyst to describe a business process in an organization. The representation may have several inputs, which may be provided in various forms including non- WS-BPEL process modeling languages as well as forms of natural languages. In support of these design-time representations, WS-BPEL defines an Abstract Process profile called the Template Profile that allows the definition of Abstract Processes which hide almost any arbitrary execution details and have explicit opaque extension points for adding behavior. These Abstract Processes allow process developers to complete execution details at a later stage – for example, adding conditions and defining endpoints for an Executable Completion.

For the remainder of section 13.4. Abstract Process Profile for Templates, the prefix associated with the Template Profile namespace URI is "`template`".

13.4.1. Profile URI

```
http://docs.oasis-open.org/wsbpel/2.0/process/abstract/simple-
template/2006/08
```

13.4.2. Opaque Start Activities

The Template Profile introduces a new `template:createInstance` extension attribute to mark an opaque activity as a start activity. This `template:createInstance` attribute carries similar semantics to the `createInstance` attribute of an IMA which are defined in both executable processes, and the common base of abstract processes. Please refer to the section below for the detailed usage of this attribute.

13.4.3. Subset of the Processes Allowed in the Common Base

All constructs allowed in the common base, such as the `<exit>` activity, are allowed in the Template Profile. All explicit opaque tokens MAY be used anywhere as allowed in the common base of Abstract Processes.

This profile restricts the common base in the following manner:

- Explicit opaque tokens – opaque activity, opaque attributes, opaque expression, and opaque from-spec – MUST be used in order to denote where WS-BPEL constructs will be added to produce an Executable Completion in all cases other than those listed under “Adding constructs without explicit opacity”.
- Omission shortcuts (see section 13.1.3. Hiding Syntactic Elements) MUST NOT be used in the Template Profile. For example, variable related attributes used in message related

constructs can not be omitted. They need to be specified with either opaque attribute values or the actual variable names.

- All start activities MUST be defined in a process of this Template Profile. That is, every IMA with a `createInstance="yes"` attribute that is added during Executable Completion MUST replace an opaque activity with `template:createInstance="yes"`. No new start activity is allowed to be added during Executable Completion.

13.4.4. Adding Constructs without explicit opacity

For the following cases, constructs MAY be added to the process definition during Execution Completion without any explicit opacity in the Abstract Process:

- Message Correlation: One or more `<correlation>` elements MAY be added to a message activity and `<onEvent>`, where no `<correlation>` or `<correlations>` is used.
- Process/Scope Declarations:
  - New data and resource declarations at a scope or process. These declarations are partner links, variables, message exchange and/or correlation sets at a scope or the process.
  - A fault handler declaration at a scope or the process. Note that compensation handlers cannot be added during Executable Completion of an Abstract Process of this profile.
  - Termination handler declaration at a scope.
  - An event handler declaration at a scope or the process.
  - Import declaration at the process
  - Extension declaration at the process

- Extensions
  - New general extension elements and attributes.

A tool, which generates Abstract Processes of Template Profile based on user inputs, is expected to use explicit opaque tokens to denote the constructs with default values (e.g. validate attribute at <assign>) in the generated WS-BPEL Abstract Processes, when users of the tool do not specify any values for those constructs. In the WS-BPEL Abstract Process itself, omitting such a construct is, as usual, equivalent to specifying it using the default value.

13.4.5. Extensions and Document Usage

This Template profile concentrates on the use of extension attributes and elements that are generally allowed in WS-BPEL. Information can be added in extensions or by natural language documentation. This information may signal the intention of the designer or provide extra semantics where needed. This is used to clarify cases where using opacity for specifying hidden syntactic links may cause ambiguity in other related parts of the process, such as those mentioned in section 13. WS-BPEL Abstract Processes.

Examples are:

- A unique identifier attribute that may be added by a designer tool to uniquely identify a WS-BPEL fragment that spans the lifetime of a business process in Abstract and Execution completion stages - as such, the activity that replaces the `<opaqueActivity>` retains that unique identifier.
- WS-BPEL template designer may add natural language as documentation or extension constructs to denote extra template information.

```
<process name="templateExample1-HomeAppraisal"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
   targetNamespace="http://example.org/template-example-1"
   xmlns:tns="http://example.org/template-example-1"
   suppressJoinFailure="yes"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:ext="http://example.com/bpel/some/extension"
   xmlns:template="http://docs.oasis-
open.org/wsbpel/2.0/process/abstract/simple-template/2006/08"
   abstractProcessProfile="http://docs.oasis-
open.org/wsbpel/2.0/process/abstract/simple-template/2006/08">

   <extensions>
      <extension
         namespace="http://example.com/bpel/some/extension"
         mustUnderstand="yes" />
   </extensions>

   <partnerLinks>
      <!-- example explanatory note: none of the 3
           referenced partnerLinks have been declared -->
      <partnerLink name="homeInfoVerifier"
         partnerLinkType="##opaque"
         myRole="##opaque"
         partnerRole="##opaque">

         <documentation>
            We have not confirmed our home information verification
            partner yet.
         </documentation>

      </partnerLink>
   </partnerLinks>

   <variables>
      <variable name="commonRequestVar" element="##opaque" />
   </variables>

   <sequence>

      <opaqueActivity template:createInstance="yes">
         <documentation>
            Pick an appraisal request from one of 3 customer referral
            channels.
         </documentation>
      </opaqueActivity>

      <assign>
         <documentation>
            Transform one of these 3 appraisal request into our own
```

```
            format.
         </documentation>
         <copy>
            <opaqueFrom/>
            <to variable="commonRequestVar" />
         </copy>
      </assign>

      <scope>
         <faultHandlers>
            <!-- example explanatory note: One can add a new <catch>
                 faultHandler for a fault from the "homeInfoVerifier"
                 partnerLink of unspecified portType yet -->
            <catchAll>
               <exit />
            </catchAll>
         </faultHandlers>
         <sequence>
            <opaqueActivity>
               <documentation>
                  Extract customer and housing info from our appraisal
                  request into a message understood by our home info
                  verification partner.
               </documentation>
            </opaqueActivity>

            <invoke partnerLink="homeInfoVerifier"
               operation="##opaque" inputVariable="##opaque"
               ext:uniqueUserFriendlyName="request verification" />

            <receive partnerLink="homeInfoVerifier"
               operation="##opaque" variable="##opaque"
               ext:uniqueUserFriendlyName="receive verification
               result" />

            <reply partnerLink="homeInfoVerifier" operation="##opaque"
               variable="##opaque"
               ext:uniqueUserFriendlyName="confirm receipt of
               verification result">
               <documentation>
                  This step confirms whether we have received the
                  verification result. It is intended to match the
                  "receive verification result" step.
               </documentation>
            </reply>

         </sequence>
      </scope>

      <opaqueActivity>
         <documentation>
            Relay the appraisal request and home info verification to
            an appraiser, who is responsible for on-site inspection.
            The appraiser may request further verification info from
            the partner through this business process. We will also
            receive the results of the appraisal from this step.
         </documentation>
         <!-- example explanatory note: An unspecified
```

```
            referral channel may trigger more than one unexpected
            fault in this process. -->
      </opaqueActivity>

      <opaqueActivity>
         <documentation>
            Send the appraisal result back to the corresponding
            referral channel.
         </documentation>
         <!-- example explanatory note: An unspecified
            referral channel may trigger more than one unexpected
            fault in this process. -->
      </opaqueActivity>

   </sequence>

</process>
```

13.4.6. Syntactic Validity

The Process Template Profile provides an XML grammar to support syntax validation beyond that provided by the common base Schema.

14. Extension Declarations

WS-BPEL is designed to be extensible. Extensions to WS-BPEL could include anything ranging from new attributes to new elements, to extended assign operations and activities, to enable restrictions or extensions of run time behavior and so on.

```
<process ...>
   ...
   <extensions>?
      <extension namespace="anyURI" mustUnderstand="yes|no" />+
   </extensions>
   ...
</process>
```

The <extension> child element under <extensions> element of the <process> element is used to declare namespaces of WS-BPEL extension attributes/elements and indicate whether they carry semantics that must be understood by a WS-BPEL processor.

If a WS-BPEL processor does not support one or more of the extensions with mustUnderstand="yes", then the process definition MUST be rejected.

Optional extensions are extensions which the WS-BPEL process MAY ignore. There is no requirement to declare any optional extensions. Optional extension can be declared using the extensions element with mustUnderstand="no". The purpose of allowing optional extensions to be declared using the extensions element is to provide a well defined location where additional information about the optional extension can be found.

The <extension> declaration element under <extensions> is itself extensible.

The same extension URI MAY be declared multiple times in the <extensions> element. If an extension URI is identified as mandatory in one <extension> element and optional in another, then the mandatory semantics have precedence and MUST be enforced. The extension declarations in an <extensions> element MUST be treated as an unordered set. That is, WS-BPEL does not provide any way to establish precedence between extension declarations based on ordering.

An extension declared through the <extension> element MUST NOT, in and of itself, cause any change to the semantics of a WS-BPEL process. Rather, the extension declaration defines whether the extensions identified by the denoted namespace must be supported or can safely be ignored.

In order to apply extension semantics to a WS-BPEL process, an extension syntax token, in the form of an element or attribute qualified by the URI value of a namespace attribute in an <extension> element that is used outside of an <extension> element, MUST appear in the WS-BPEL process definition or its directly referenced WSDL <portType> definitions, <vprop:propertyAlias> definitions or <vprop:property> definitions. It is this extension syntax token, rather than the extension declaration, that indicates the new semantics apply.

An extension syntax token can only affect WS-BPEL constructs within the syntax sub-tree of the parent element of the token. In other words, extension syntax token MUST NOT affect the semantics outside the subtree. Here are two examples to illustrate this concept further:

```
<process>
   ...
   <scope>
      <sequence>
         <invoke operation="operation1"
            foo:invokeProperty="someNature" ... />
         <invoke operation="operation2" ... />
         <invoke operation="operation3"
            foo:invokeProperty="someNature2" ... />
      </sequence>
   </scope>
   ...
</process>
```

The "`foo:invokeProperty`" extension attribute are applied to `<invoke>` activities for "`operation1`" and "`operation3`". The `<invoke>` activity for "`operation2`" must not be affected.

```
<process>
   ...
   <scope>
      <foo:invokeProperty>SomeNature</foo:invokeProperty>
      <sequence>
         <invoke operation="operation1" ... />
         <invoke operation="operation2" ... />
         <invoke operation="operation3" ... />
      </sequence>
   </scope>
   ...
</process>
```

The "`foo:invokeProperty`" extension element can be applied to all `<invoke>` activities within the `<scope>` activity where the extension element are attached to.

15. Examples

The examples in this section are not completely specified. For instance, the shipping service example imports XML Schema elements from the namespace "http://example.com/shipping/ship.xsd", which is not fully specified in this document.

15.1. Shipping Service

This example presents a WS-BPEL Abstract Process for a rudimentary shipping service. This service handles the shipment of orders, and orders are composed of a number of items. The shipping service offers two options, one for shipments where the items are shipped all together, and one for partial shipments where the items are shipped in groups until the order is fulfilled.

15.1.1. Service Description

The context for the shipping service is an interaction between a customer and the service. This is modeled with a `partnerLinkType` definition (shippingLT.wsdl):

```
<wsdl:definitions
   targetNamespace="http://example.com/shipping/partnerLinkTypes/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
   xmlns:sif="http://example.com/shipping/interfaces/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <wsdl:import location="shippingPT.wsdl"
      namespace="http://example.com/shipping/interfaces/" />

   <plnk:partnerLinkType name="shippingLT">
      <plnk:role name="shippingService"
         portType="sif:shippingServicePT" />
      <plnk:role name="shippingServiceCustomer"
         portType="sif:shippingServiceCustomerPT" />
   </plnk:partnerLinkType>

</wsdl:definitions>
```

The corresponding `message` and `portType` definitions are as follows (shippingPT.wsdl):

```
<wsdl:definitions
   targetNamespace="http://example.com/shipping/interfaces/"
   xmlns:ship="http://example.com/shipping/ship.xsd"
   xmlns:tns="http://example.com/shipping/interfaces/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:types>
      <xsd:schema>
         <!-- import ship schema -->
      </xsd:schema>
   </wsdl:types>

   <wsdl:message name="shippingRequestMsg">
```

```
      <wsdl:part name="shipOrder" type="ship:shipOrder" />
   </wsdl:message>

   <wsdl:message name="shippingNoticeMsg">
      <wsdl:part name="shipNotice" type="ship:shipNotice" />
   </wsdl:message>

   <wsdl:portType name="shippingServicePT">
      <wsdl:operation name="shippingRequest">
         <wsdl:input message="tns:shippingRequestMsg" />
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:portType name="shippingServiceCustomerPT">
      <wsdl:operation name="shippingNotice">
         <wsdl:input message="tns:shippingNoticeMsg" />
      </wsdl:operation>
   </wsdl:portType>

</wsdl:definitions>
```

15.1.2. Properties

The properties relevant to the service are:

- The shipping order ID (`shipOrderID`) is used to correlate the shipping notice(s) with the shipping order.
- Whether the order is to be shipped complete or not (`shipComplete`).
- The total number of items in the order (`itemsTotal`).
- The number of items in a ship notice (`itemsCount`). When partial shipments are acceptable, `itemsCount` and `itemsTotal` are used to track the fulfillment of the order.

The definitions for the properties and their aliases are (shippingProperties.wsdl):

```
<wsdl:definitions
   targetNamespace="http://example.com/shipping/properties/"
   xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
   xmlns:ship="http://example.com/shipping/ship.xsd"
   xmlns:sif="http://example.com/shipping/interfaces/"
   xmlns:tns="http://example.com/shipping/properties/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:import location="shippingPT.wsdl"
      namespace="http://example.com/shipping/interfaces/" />

   <!-- 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/shipping/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>

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

<vprop:propertyAlias propertyName="tns:shipOrderID"
   messageType="sif:shippingRequestMsg" part="shipOrder">
   <vprop:query>
      ship:ShipOrderRequestHeader/ship:shipOrderID
   </vprop:query>
</vprop:propertyAlias>

<vprop:propertyAlias propertyName="tns:shipOrderID"
   messageType="sif:shippingNoticeMsg" part="shipNotice">
   <vprop:query>ship:ShipNoticeHeader/ship:shipOrderID</vprop:query>
</vprop:propertyAlias>

<vprop:propertyAlias propertyName="tns:shipComplete"
   messageType="sif:shippingRequestMsg" part="shipOrder">
   <vprop:query>
      ship:ShipOrderRequestHeader/ship:shipComplete
   </vprop:query>
</vprop:propertyAlias>

<vprop:propertyAlias propertyName="tns:itemsTotal"
   messageType="sif:shippingRequestMsg" part="shipOrder">
   <vprop:query>
      ship:ShipOrderRequestHeader/ship:itemsTotal
   </vprop:query>
</vprop:propertyAlias>

<vprop:propertyAlias propertyName="tns:itemsCount"
   messageType="sif:shippingRequestMsg" part="shipOrder">
   <vprop:query>
      ship:ShipOrderRequestHeader/ship:itemsCount
   </vprop:query>
</vprop:propertyAlias>

<vprop:propertyAlias propertyName="tns:itemsCount"
   messageType="sif:shippingNoticeMsg" part="shipNotice">
   <vprop:query>ship:ShipNoticeHeader/ship:itemsCount</vprop:query>
</vprop:propertyAlias>

</wsdl:definitions>
```

15.1.3. Process

For brevity, the Abstract Process definition does not include details such as the handling of error conditions that a complete process description would likely provide. The 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
```

The WS-BPEL process is as follows:

```
<process name="shippingService"
   targetNamespace="http://example.com/shipping/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
   xmlns:plt="http://example.com/shipping/partnerLinkTypes/"
   xmlns:props="http://example.com/shipping/properties/"
   xmlns:ship="http://example.com/shipping/ship.xsd"
   xmlns:sif="http://example.com/shipping/interfaces/"
   abstractProcessProfile="http://docs.oasis-
open.org/wsbpel/2.0/process/abstract/ap11/2006/08">

   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="shippingLT.wsdl"
      namespace="http://example.com/shipping/partnerLinkTypes/" />
   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="shippingPT.wsdl"
      namespace="http://example.com/shipping/interfaces/" />
   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="shippingProperties.wsdl"
      namespace="http://example.com/shipping/properties/" />

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

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

   <correlationSets>
      <correlationSet name="shipOrder"
```

```
      properties="props:shipOrderID" />
  </correlationSets>

  <sequence>

     <receive partnerLink="customer"
        operation="shippingRequest"
        variable="shipRequest">
        <correlations>
           <correlation set="shipOrder" initiate="yes" />
        </correlations>
     </receive>

     <if>
        <condition>
           bpel:getVariableProperty('shipRequest',
              'props:shipComplete')
        </condition>
        <sequence>
           <assign>
              <copy>
                 <from variable="shipRequest"
                    property="props:shipOrderID" />
                 <to variable="shipNotice"
                    property="props:shipOrderID" />
              </copy>
              <copy>
                 <from variable="shipRequest"
                    property="props:itemsCount" />
                 <to variable="shipNotice"
                    property="props:itemsCount" />
              </copy>
           </assign>
           <invoke partnerLink="customer"
              operation="shippingNotice"
              inputVariable="shipNotice">
              <correlations>
                 <correlation set="shipOrder" pattern="request" />
              </correlations>
           </invoke>
        </sequence>
        <else>
           <sequence>
              <assign>
                 <copy>
                    <from>0</from>
                    <to>$itemsShipped</to>
                 </copy>
              </assign>
              <while>
                 <condition>
                    $itemsShipped
                    &lt;
                    bpel:getVariableProperty('shipRequest',
                       'props:itemsTotal')
                 </condition>
                 <sequence>
                    <assign>
```

```
                        <copy>
                           <opaqueFrom/>
                           <to variable="shipNotice"
                              property="props:shipOrderID" />
                        </copy>
                        <copy>
                           <opaqueFrom/>
                           <to variable="shipNotice"
                              property="props:itemsCount" />
                        </copy>
                     </assign>
                     <invoke partnerLink="customer"
                        operation="shippingNotice"
                        inputVariable="shipNotice">
                        <correlations>
                           <correlation set="shipOrder"
                              pattern="request" />
                        </correlations>
                     </invoke>
                     <assign>
                        <copy>
                           <from>
                              $itemsShipped
                              +
                              bpel:getVariableProperty('shipNotice',
                              'props:itemsCount')
                           </from>
                           <to>$itemsShipped</to>
                        </copy>
                     </assign>
                  </sequence>
               </while>
            </sequence>
         </else>
      </if>

   </sequence>

</process>
```

15.2. Ordering Service

This example expands on the shipping service to illustrate the use of an Abstract Process using the template profile. This Abstract Process describes a set of services to request, track, and confirm orders and their shipments, invoicing, and payment. The ordering service receives orders from an order processor, sends a shipping request to the shipping service, and acknowledges shipment, pickup, invoicing, and payment as each is performed.

15.2.1. Service Description

The context for the ordering service is an interaction between a consumer and the service. This is modeled in the following `partnerLinkType` definition (orderingLT.wsdl):

```
<wsdl:definitions
   targetNamespace="http://example.com/ordering/partnerLinkTypes/"
```

```
   xmlns:oif="http://example.com/ordering/interfaces/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <wsdl:import location="orderingPT.wsdl"
      namespace="http://example.com/ordering/interfaces/" />

   <plnk:partnerLinkType name="orderingServiceLT">
      <plnk:role name="orderingService"
         portType="oif:orderingPT" />
      <plnk:role name="orderingServiceResponse"
         portType="oif:orderingResponsePT" />
   </plnk:partnerLinkType>

   <plnk:partnerLinkType name="shipperLT">
      <plnk:role name="shippingService"
         portType="oif:shippingServicePT" />
      <plnk:role name="shippingServiceResponse"
         portType="oif:shippingServiceResponsePT" />
   </plnk:partnerLinkType>

   <plnk:partnerLinkType name="completionConfirmationLT">
      <plnk:role name="orderingServiceConfirmation"
         portType="oif:orderingConfirmationPT" />
   </plnk:partnerLinkType>

</wsdl:definitions>
```

The corresponding `message` and `portType` definitions are as follows (orderingPT.wsdl):

```
<wsdl:definitions
   targetNamespace="http://example.com/ordering/interfaces/"
   xmlns:order="http://example.com/ordering/order.xsd"
   xmlns:tns="http://example.com/ordering/interfaces/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:types>
      <xsd:schema>
         <!-- import ordering schema -->
      </xsd:schema>
   </wsdl:types>

   <wsdl:message name="OrderMessageType">
      <wsdl:part name="OrderMessagePart" element="order:OrderMessage"/>
   </wsdl:message>

   <wsdl:message name="OrderAckMessageType">
      <wsdl:part name="OrderAckMessagePart"
         element="order:OrderAckMessage" />
   </wsdl:message>

   <wsdl:message name="ShipRequestMessageType">
      <wsdl:part name="ShipRequestMessagePart"
         element="order:ShipRequestMessage" />
   </wsdl:message>

   <wsdl:message name="ShipNoticeMessageType">
```

```
      <wsdl:part name="ShipNoticeMessagePart"
         element="order:ShipNoticeMessage" />
   </wsdl:message>

   <wsdl:message name="ShipHistoryMessageType">
      <wsdl:part name="ShipHistoryMessagePart"
         element="order:ShipHistoryMessage" />
   </wsdl:message>

   <wsdl:message name="InvoiceAckMessageType">
      <wsdl:part name="InvoiceAckMessagePart"
         element="order: InvoiceAckMessage" />
   </wsdl:message>

   <wsdl:portType name="orderingPT">
      <wsdl:operation name="placeOrder">
         <wsdl:input message="tns:OrderMessageType" />
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:portType name="orderingResponsePT">
      <wsdl:operation name="getOrderAck">
         <wsdl:input message="tns:OrderAckMessageType" />
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:portType name="orderingConfirmationPT">
      <wsdl:operation name="getOrderConfirmation">
         <wsdl:input message="tns:OrderAckMessageType" />
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:portType name="shippingServicePT">
      <wsdl:operation name="shippingRequest">
         <wsdl:input message="tns:ShipRequestMessageType" />
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:portType name="shippingServiceCustomerPT">
      <wsdl:operation name="shippingNotice">
         <wsdl:input message="tns:ShipNoticeMessageType" />
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

Although there are more interactions between consumer and service, not all have been modeled in this example. Un-modeled interactions are opaque.

15.2.2. Properties

The properties relevant to the service are:

- The order ID (`orderID`) is used to correlate the order placement with the shipping request, shipping notice, invoice confirmation, pickup confirmation and final order confirmation. For this example, only the shipping request, shipping notice and final confirmation are defined

The order ID and aliases are defined as follows (orderingProperties.wsdl):

```
<wsdl:definitions
   targetNamespace="http://example.com/ordering/properties/"
   xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
   xmlns:oif="http://example.com/ordering/interfaces/"
   xmlns:order="http://example.com/ordering/order.xsd"
   xmlns:tns="http://example.com/ordering/properties/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:import location="orderingPT.wsdl"
      namespace="http://example.com/ordering/interfaces/" />

   <vprop:property name="orderID" type="xsd:string" />

   <vprop:propertyAlias propertyName="tns:orderID"
      messageType="oif:OrderMessageType" part="OrderMessagePart">
      <vprop:query>
         order:OrderMessageHeader/order:orderID
      </vprop:query>
   </vprop:propertyAlias>

   <vprop:propertyAlias propertyName="tns:orderID"
      messageType="oif:ShipNoticeMessageType"
      part="ShipNoticeMessagePart">
      <vprop:query>
         order:ShipNoticeMessageHeader/order:orderID
      </vprop:query>
   </vprop:propertyAlias>

</wsdl:definitions>
```

Although there are more messages between the consumer and the service, not all have been modeled. Un-modeled messages are opaque.

15.2.3. Process

This Abstract Process uses the template profile. The outline is as follows:

```
receive placeOrder
send shipOrder
if
      condition shipCompleted
            send orderNotice (indicating shipCompleted)
      else
            send orderNotice (indicating !shipCompleted)

receive pickupNotification
update shipHistory

receive invoice
send invoiceResponse

receive paymentConfirmation
```

```
send orderConfirmation
```

The WS-BPEL process is as follows:

```
<process name="OrderingServiceProcess"
   targetNamespace="http://example.com/ordering/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
   xmlns:ext="http://example.com/bpel/some/extension"
   xmlns:oif="http://example.com/ordering/interfaces/"
   xmlns:order="http://example.com/ordering/order.xsd"
   xmlns:plt="http://example.com/ordering/partnerLinkTypes/"
   xmlns:props="http://example.com/ordering/properties/"
   xmlns:tns="http://example.com/ordering/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   abstractProcessProfile="http://docs.oasis-
open.org/wsbpel/2.0/process/abstract/simple-template/2006/08"
   suppressJoinFailure="yes">

   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="orderingLT.wsdl"
      namespace="http://example.com/ordering/partnerLinkTypes/" />
   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="orderingPT.wsdl"
      namespace=" http://example.com/ordering/interfaces/" />
   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="orderingProperties.wsdl"
      namespace="http://example.com/ordering/properties/" />

   <extensions>
      <extension namespace="http://example.com/bpel/some/extension"
         mustUnderstand="no" />
   </extensions>

   <partnerLinks>
      <partnerLink name="ordering"
         partnerLinkType="plt:orderingServiceLT"
         myRole="orderingService"
         partnerRole="orderingServiceResponse" />

      <partnerLink name="shipper"
         partnerLinkType="plt:shipperLT"
         myRole="shippingServiceResponse"
         partnerRole="shippingService" />

      <partnerLink name="shippingRequester"
         partnerLinkType="##opaque"
         myRole="##opaque" />

      <partnerLink name="invoiceProcessor"
         partnerLinkType="##opaque"
         myRole="##opaque"
         partnerRole="##opaque" />

      <partnerLink name="orderingConfirmation"
         partnerLinkType="plt:completionConfirmationLT"
         partnerRole="orderingServiceConfirmation" />
   </partnerLinks>
```

```
<variables>
   <!-- Reference to the message passed as input during
        initiation -->

   <variable name="order" messageType="oif:OrderMessageType" />
   <variable name="orderAckMsg"
      messageType="oif:OrderAckMessageType" />
   <variable name="orderShippedMsg"
      element="order:OrderAckMessage" />
   <variable name="shippingRequestMsg"
      element="order:ShipRequestMessage" />
   <variable name="shippingNoticeMsg"
      element="order:ShipNoticeMessage" />
   <variable name="shipHistoryMsg"
      messageType="oif:ShippingHistoryMessageType" />
   <variable name="invoiceAckMsg"
      messageType="oif:InvoiceAckMessageType" />
</variables>

<correlationSets>
   <correlationSet name="orderCS" properties="props:orderID" />
</correlationSets>

<sequence>
   <receive partnerLink="ordering" operation="placeOrder"
      variable="order" createInstance="yes">
      <correlations>
         <correlation set="orderCS" initiate="yes" />
      </correlations>
   </receive>

   <assign>
      <copy>
         <from>
            $order.OrderMessagePart/order:OrderMessageHeader/
               order:orderID
         </from>
         <to>
            $shippingRequestMsg/order:ShipRequestMessageHeader/
               order:orderID
         </to>
      </copy>
      <copy>
         <from>$order.OrderMessagePart/order:ShippingInfo</from>
         <to>$shippingRequestMsg/order:ShippingInfo</to>
      </copy>
   </assign>

   <invoke partnerLink="shipper" operation="shippingRequest"
      inputVariable="shippingRequestMsg"
      ext:uniqueUserFriendlyName="send shipping request to
         shipper"/>

   <receive partnerLink="shipper"
      portType="oif:shippingServiceCustomerPT"
      operation="shippingNotice"
      variable="shippingNoticeMsg"
      ext:uniqueUserFriendlyName="receive response from shipper">
```

```
         <correlations>
            <correlation set="orderCS" />
         </correlations>
      </receive>

      <assign>
         <copy>
            <from>
               $order.OrderMessagePart/order:OrderMessageHeader/
                  order:orderID
            </from>
            <to>
               $orderAckMsg.OrderAckMessagePart/
                  order:OrderAckMessageHeader/order:orderID
            </to>
         </copy>
      </assign>

      <if>
         <condition opaque="yes" />
         <!--
            the first case would package the order
            acknowledgement for a completed shipment
         -->
         <assign>
            <copy>
               <opaqueFrom/>
               <to>$orderAckMsg.OrderAckMessagePart/order:Ack</to>
            </copy>
         </assign>
         <else>
            <!--
               the second case would package the order
               acknowledgement for an uncompleted shipment
            -->
            <assign>
               <copy>
                  <opaqueFrom/>
                  <to>$orderAckMsg.OrderAckMessagePart/order:Ack</to>
               </copy>
            </assign>
         </else>
      </if>

      <invoke partnerLink="ordering"
         operation="getOrderAck"
         inputVariable="orderAckMsg" />

      <receive partnerLink="shippingRequester"
         operation="##opaque"
         variable="##opaque"
         ext:uniqueUserFriendlyName="receive the pickup notification">
         <correlations>
            <correlation set="orderCS" />
         </correlations>
      </receive>

      <assign>
```

```
         <copy>
            <opaqueFrom/>
            <to>
               $shipHistoryMsg.ShipHistoryMessagePart/order:Event
            </to>
         </copy>
      </assign>

      <opaqueActivity>
         <documentation>
            If we receive notice that the ship has completed, update
            our ship history accordingly
         </documentation>
      </opaqueActivity>

      <receive partnerLink="invoiceProcessor" operation="##opaque"
         variable="##opaque"
         ext:uniqueUserFriendlyName="receive invoice for processing">
         <correlations>
            <correlation set="orderCS" />
         </correlations>
      </receive>

      <assign>
         <copy>
            <opaqueFrom/>
            <to>$invoiceAckMsg.InvoiceAckMessagePart</to>
         </copy>
      </assign>

      <invoke partnerLink="invoiceProcessor" operation="##opaque"
         inputVariable="##opaque"
         ext:uniqueUserFriendlyName="send response for the invoice" />

      <receive partnerLink="shippingRequester" operation="##opaque"
         variable="##opaque"
         ext:uniqueUserFriendlyName="receive payment confirmation">
         <correlations>
            <correlation set="orderCS" />
         </correlations>
      </receive>

      <assign>
         <copy>
            <opaqueFrom/>
            <to>$orderShippedMsg/order:Ack</to>
         </copy>
         <copy>
            <from>
               $order.OrderMessagePart/order:OrderMessageHeader/
                  order:orderID
            </from>
            <to>
               $orderShippedMsg/order:OrderAckMessageHeader/
                  order:orderID
            </to>
         </copy>
      </assign>
```

```
      <invoke partnerLink="orderingConfirmation"
         operation="getOrderConfirmation"
         inputVariable="orderShippedMsg" />

   </sequence>
</process>
```

15.3. Loan Approval Service

This example consists of a simple loan approval service. Customers of the service send loan requests, including personal information and amount being requested. Using this information, the loan service executes a simple process resulting in either a "loan approved" message or a "loan rejected" message. The decision is based on the amount requested and the risk associated with the customer. For low amounts of less than $10,000 a streamlined process is used. In the streamlined process low-risk customers are approved automatically. For higher amounts, or medium and high-risk customers, the credit request requires further processing. For each request, the loan service uses the functionality provided by two other services. In the streamlined process, used for low amount loans, a risk assessment service is used to obtain a quick evaluation of the risk associated with the customer. A full loan approval service (possibly requiring direct involvement of a loan expert) is used to obtain assessments when the streamlined approval process is not applicable.

15.3.1. Service Description

The WSDL `portType` (`loanServicePT`) used by this service is shown below. This example assumes that an independent "loan.org" consortium has provided definitions of the loan service `portType` as well as the risk assessment and full 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 `partnerLinkTypes` that relate to the use of these `portTypes`, are defined in the WSDL (loanServicePT.wsdl).

```
<wsdl:definitions
   targetNamespace="http://example.com/loan-approval/wsdl/"
   xmlns:ens="http://example.com/loan-approval/xsd/error-messages/"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
   xmlns:tns="http://example.com/loan-approval/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema/">

   <wsdl:types>
      <xsd:schema>
         <!-- import schemas -->
      </xsd:schema>
   </wsdl:types>

   <wsdl:message name="creditInformationMessage">
      <wsdl:part name="firstName" type="xsd:string" />
      <wsdl:part name="name" type="xsd:string" />
      <wsdl:part name="amount" type="xsd:integer" />
   </wsdl:message>
```

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

   <wsdl:message name="riskAssessmentMessage">
      <wsdl:part name="level" type="xsd:string" />
   </wsdl:message>

   <wsdl:message name="errorMessage">
      <wsdl:part name="errorCode" element="ens:integer" />
   </wsdl:message>

   <wsdl:portType name="loanServicePT">
      <wsdl:operation name="request">
         <wsdl:input message="tns:creditInformationMessage" />
         <wsdl:output message="tns:approvalMessage" />
         <wsdl:fault name="unableToHandleRequest"
            message="tns:errorMessage" />
      </wsdl:operation>
   </wsdl:portType>

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

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

   <plnk:partnerLinkType name="loanPartnerLT">
      <plnk:role name="loanService" portType="tns:loanServicePT" />

   </plnk:partnerLinkType>

   <plnk:partnerLinkType name="loanApprovalLT">
      <plnk:role name="approver" portType="tns:loanApprovalPT" />

   </plnk:partnerLinkType>

   <plnk:partnerLinkType name="riskAssessmentLT">
      <plnk:role name="assessor" portType="tns:riskAssessmentPT" />

   </plnk:partnerLinkType>

</wsdl:definitions>
```

15.3.2. Process

In the process, 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 executed according to the dependencies expressed by the <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 setting the suppressJoinFailure attribute to yes on the <process> element (See section 11.6.3. Dead-Path Elimination).

The operations invoked can return a fault of type loanProcessFault, therefore 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://example.com/loan-approval/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:lns="http://example.com/loan-approval/wsdl/"
   suppressJoinFailure="yes">

   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="loanServicePT.wsdl"
      namespace="http://example.com/loan-approval/wsdl/" />

   <partnerLinks>
      <partnerLink name="customer"
         partnerLinkType="lns:loanPartnerLT"
         myRole="loanService" />
      <partnerLink name="approver"
         partnerLinkType="lns:loanApprovalLT"
         partnerRole="approver" />
      <partnerLink name="assessor"
         partnerLinkType="lns:riskAssessmentLT"
         partnerRole="assessor" />
   </partnerLinks>

   <variables>
      <variable name="request"
         messageType="lns:creditInformationMessage" />
      <variable name="risk"
         messageType="lns:riskAssessmentMessage" />
      <variable name="approval"
         messageType="lns:approvalMessage" />
   </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>
               <literal>yes</literal>
            </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>
```

15.4. Auction Service

A process may have multiple activities capable of creating an instance of the process. An example can be a simplified auction house process. The process collects information from the buyer and the seller of a particular auction, report the appropriate auction results to an auction registration service, and then send the registration result back to the seller and the buyer. The process may start either by receiving the seller information, or by 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 their data. The sequence in which the seller and buyer requests arrive at the auction house is random. When a request comes in, it needs to check whether a process instance exists already or not. If no process instance already exists then one is created. When both requests have been received, the auction registration service is invoked. Because the invocation is done one-way, the auction house passes the auction ID to the auction registration service. The auction registration service returns this auction ID in its answer for the auction house to locate the proper process instance. Each buyer or seller provides an endpoint reference for the auction service to respond properly. In addition, the auction house provides its own endpoint reference to the auction registration service for the auction registration service to send the response back to the auction house.

15.4.1. Service Description

The auction service offers two `portTypes`, called `sellerPT` and `buyerPT`, with appropriate operations for accepting the data provided by the seller and the buyer. The auction service responds to the seller and buyer through appropriate `portTypes`, `sellerAnswerPT` and `buyerAnswerPT`. These `portTypes` are properly combined into two `partnerLinkTypes`, one for the seller called `sellerAuctionHouseLT` and one for the buyer called `buyerAuctionHouseLT`.

The auction service needs two `portTypes`, called `auctionRegistrationPT` and `auctionRegistrationAnswerPT`, for the invocation of the auction registration service. The `portTypes` are part of the `partnerLinkType` `auctionHouseAuctionRegistrationServiceLT` (auctionServiceInterface.wsdl).

```
<wsdl:definitions
   targetNamespace="http://example.com/auction/wsdl/auctionService/"
   xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
   xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
   xmlns:sref="http://docs.oasis-open.org/wsbpel/2.0/serviceref"
   xmlns:tns="http://example.com/auction/wsdl/auctionService/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <!-- Messages for communication with the seller -->

   <wsdl:message name="sellerData">
      <wsdl:part name="creditCardNumber" type="xsd:string" />
      <wsdl:part name="shippingCosts" type="xsd:integer" />
      <wsdl:part name="auctionId" type="xsd:integer" />
      <wsdl:part name="endpointReference" type="sref:ServiceRefType" />
   </wsdl:message>

   <wsdl:message name="sellerAnswerData">
      <wsdl:part name="thankYouText" type="xsd:string" />
   </wsdl:message>

   <!-- Messages for communication with the buyer -->

   <wsdl:message name="buyerData">
      <wsdl:part name="creditCardNumber" type="xsd:string" />
      <wsdl:part name="phoneNumber" type="xsd:string" />
      <wsdl:part name="ID" type="xsd:integer" />
      <wsdl:part name="endpointReference" type="sref:ServiceRefType" />
   </wsdl:message>

   <wsdl:message name="buyerAnswerData">
      <wsdl:part name="thankYouText" type="xsd:string" />
   </wsdl:message>

   <!-- Messages for communication with the
        auction registration service -->

   <wsdl:message name="auctionData">
      <wsdl:part name="auctionId" type="xsd:integer" />
      <wsdl:part name="amount" type="xsd:integer" />
```

```
    <wsdl:part name="auctionHouseEndpointReference"
       type="sref:ServiceRefType" />
  </wsdl:message>

  <wsdl:message name="auctionAnswerData">
    <wsdl:part name="registrationId" type="xsd:integer" />
    <wsdl:part name="auctionId" type="xsd:integer" />
  </wsdl:message>

  <!-- PortTypes for interacting with the seller -->

  <wsdl:portType name="sellerPT">
    <wsdl:operation name="submit">
      <wsdl:input message="tns:sellerData" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:portType name="sellerAnswerPT">
    <wsdl:operation name="answer">
      <wsdl:input message="tns:sellerAnswerData" />
    </wsdl:operation>
  </wsdl:portType>

  <!-- PortTypes for interacting with the buyer -->

  <wsdl:portType name="buyerPT">
    <wsdl:operation name="submit">
      <wsdl:input message="tns:buyerData" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:portType name="buyerAnswerPT">
    <wsdl:operation name="answer">
      <wsdl:input message="tns:buyerAnswerData" />
    </wsdl:operation>
  </wsdl:portType>

  <!-- PortTypes for interacting with the
       auction registration service -->

  <wsdl:portType name="auctionRegistrationPT">
    <wsdl:operation name="process">
      <wsdl:input message="tns:auctionData" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:portType name="auctionRegistrationAnswerPT">
    <wsdl:operation name="answer">
      <wsdl:input message="tns:auctionAnswerData" />
    </wsdl:operation>
  </wsdl:portType>

  <!-- Context type used for locating business process
       via auction Id -->

  <vprop:property name="auctionId" type="xsd:integer" />
  <vprop:propertyAlias propertyName="tns:auctionId"
    messageType="tns:sellerData" part="auctionId" />
```

```
   <vprop:propertyAlias propertyName="tns:auctionId"
      messageType="tns:buyerData" part="ID" />
   <vprop:propertyAlias propertyName="tns:auctionId"
      messageType="tns:auctionData" part="auctionId" />
   <vprop:propertyAlias propertyName="tns:auctionId"
      messageType="tns:auctionAnswerData" part="auctionId" />

   <!-- PartnerLinkType for seller/auctionHouse -->

   <plnk:partnerLinkType name="sellerAuctionHouseLT">
      <plnk:role name="auctionHouse" portType="tns:sellerPT" />
      <plnk:role name="seller" portType="tns:sellerAnswerPT" />
   </plnk:partnerLinkType>

   <!-- PartnerLinkType 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>

</wsdl:definitions>
```

15.4.2. Process

The WS-BPEL process for the auction house is as follows:

```
<process name="auctionService"
   targetNamespace="http://example.com/auction"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:sref=" http://docs.oasis-open.org/wsbpel/2.0/serviceref"
   xmlns:addr="http://example.com/addressing"
   xmlns:as="http://example.com/auction/wsdl/auctionService/">

   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="auctionServiceInterface.wsdl"
      namespace="http://example.com/auction/wsdl/auctionService/" />

   <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>
      <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>

   <correlationSets>
      <correlationSet name="auctionIdentification"
         properties="as:auctionId" />
   </correlationSets>

   <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="submit"
            variable="sellerData"
            createInstance="yes">
            <correlations>
               <correlation set="auctionIdentification"
                  initiate="join" />
            </correlations>
         </receive>

         <!-- Process buyer request -->

         <receive name="acceptBuyerInformation"
            partnerLink="buyer"
            portType="as:buyerPT"
            operation="submit"
            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>
                  <sref:service-ref>
                     <addr:EndpointReference>
                        <addr:Address>
                           http://example.com/auction/
                              RegistrationService/
                        </addr:Address>
                        <addr:ServiceName>
                           as:RegistrationService
                        </addr:ServiceName>
                     </addr:EndpointReference>
                  </sref:service-ref>
               </literal>
            </from>
            <to partnerLink="auctionRegistrationService" />
         </copy>
         <copy>
            <from partnerLink="auctionRegistrationService"
               endpointReference="myRole" />
            <to>$auctionData.auctionHouseEndpointReference</to>
         </copy>
         <copy>
            <from>$sellerData.auctionId</from>
            <to>$auctionData.auctionId</to>
         </copy>
         <copy>
            <from>1</from>
            <to>$auctionData.amount</to>
         </copy>
      </assign>

      <invoke name="registerAuctionResults"
         partnerLink="auctionRegistrationService"
         portType="as:auctionRegistrationPT"
         operation="process"
         inputVariable="auctionData" />

      <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>$sellerData.endpointReference</from>
                  <to partnerLink="seller" />
               </copy>
               <copy>
                  <from>
                     <literal>Thank you!</literal>
                  </from>
                  <to>$sellerAnswerData.thankYouText</to>
               </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>$buyerData.endpointReference</from>
                  <to partnerLink="buyer" />
               </copy>
               <copy>
                  <from>
                     <literal>Thank you!</literal>
                  </from>
                  <to>$buyerAnswerData.thankYouText</to>
               </copy>
            </assign>

            <invoke name="respondToBuyer"
               partnerLink="buyer"
               portType="as:buyerAnswerPT"
               operation="answer"
               inputVariable="buyerAnswerData" />

         </sequence>
```

```
      </flow>
   </sequence>
</process>
```

16. Security Considerations

Although WS-BPEL is inherently binding neutral it is strongly recommended that business process implementations use WS-Security when using a binding where messages may be modified or forged. WS-Security provides mechanisms to ensure messages have not been modified or forged while in transit or while residing at destinations. Similarly, there are mechanisms to prevent invalid or expired messages from being re-used or message headers not specifically associated with the specific message being referenced. Consequently, when using WS-Security, signatures should 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.

Appendix A. Standard Faults

The following list specifies the standard faults defined within the WS-BPEL specification. All standard fault names are qualified with the standard WS-BPEL namespace.

Table A.1. Standard Faults

Fault name	Description
`ambiguousReceive`	Thrown when a business process instance simultaneously enables two or more IMAs for the same partnerLink, portType, operation but different correlationSets, and the correlations of multiple of these activities match an incoming request message.
`completionConditionFailure`	Thrown if upon completion of a directly enclosed `<scope>` activity within `<forEach>` activity it can be determined that the completion condition can never be true.
`conflictingReceive`	Thrown when more than one inbound message activity is enabled simultaneously for the same partner link, port type, operation and correlation set(s).
`conflictingRequest`	Thrown when more than one inbound message activity is open for the same partner link, operation and message exchange.
`correlationViolation`	Thrown when the contents of the messages that are processed in an `<invoke>`, `<receive>`, `<reply>`, `<onMessage>`, or `<onEvent>` do not match specified correlation information.
`invalidBranchCondition`	Thrown if the integer value used in the `<branches>` completion condition of `<forEach>` is larger than the number of directly enclosed `<scope>` activities.
`invalidExpressionValue`	Thrown when an expression used within a WS-BPEL construct (except `<assign>`) returns an invalid value with respect to the expected XML Schema type.
`invalidVariables`	Thrown when an XML Schema validation (implicit or explicit) of a variable value fails.
`joinFailure`	Thrown when the join condition of an activity evaluates to `false` and the value of the `suppressJoinFailure` attribute is `yes`.
`mismatchedAssignmentFailure`	Thrown when incompatible types or incompatible XML infoset structure are encountered in an `<assign>` activity.
`missingReply`	Thrown when an inbound message activity has been executed, and the process instance or scope instance reaches the end of its execution without a corresponding `<reply>` activity having been executed.
`missingRequest`	Thrown when a `<reply>` activity cannot be associated with an

Fault name	Description
	open inbound message activity by matching the partner link, operation and message exchange tuple.
`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.
`selectionFailure`	Thrown when a selection operation performed either in a function such as `bpel:getVariableProperty`, or in an assignment, encounters an error.
`subLanguageExecutionFault`	Thrown when the execution of an expression results in an unhandled fault in an expression language or query language.
`uninitializedPartnerRole`	Thrown when an `<invoke>` or `<assign>` activity references a partner link whose `partnerRole` endpoint reference is not initialized.
`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.
`unsupportedReference`	Thrown when a WS-BPEL implementation fails to interpret the combination of the `reference-scheme` attribute and the content element OR just the content element alone.
`xsltInvalidSource`	Thrown when the transformation source provided in a `bpel:doXslTransform` function call was not legal (i.e., not an EII).
`xsltStylesheetNotFound`	Thrown when the named style sheet in a `bpel:doXslTransform` function call was not found.

Appendix B. Static Analysis requirement summary (Non-Normative)

The purpose of static analysis is to detect any undefined semantics or invalid semantics within a process definition that was not detected during the schema validation against the XSD found in Appendix E. XML Schemas Any process definition that fails one or more of these checks must be rejected by the WS-BPEL processor.

This appendix summarizes the requirements for static analysis specified in the main body of the specification and is provided for convenience.

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00001	A WS-BPEL processor MUST reject a WS-BPEL that refers to solicit-response or notification operations portTypes.	Section 3
SA00002	A WS-BPEL processor MUST reject any WSDL portType definition that includes overloaded operation names.	Section 3
SA00003	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.	Section 5.2
SA00004	If any referenced `queryLanguage` or `expressionLanguage` is unsupported by the WS-BPEL processor then the processor MUST reject the submitted WS-BPEL process definition.	Section 5.2
SA00005	If the `portType` attribute is included for readability, in a <receive>, <reply>, <invoke>, `<onEvent>` or `<onMessage>` element, 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.	Section 5.2
SA00006	The `<rethrow>` activity MUST only be used within a faultHandler (i.e. `<catch>` and `<catchAll>` elements).	Section 5.2
SA00007	The `<compensateScope>` activity MUST only be used from within a `faultHandler`, another `compensationHandler`, or a `terminationHandler`.	Section 5.2
SA00008	The `<compensate>` activity MUST only be used from within a `faultHandler`, another `compensationHandler`, or a `terminationHandler`.	Section 5.2
SA00009	In the case of mandatory extensions declared in the `<extensions>` element not supported by a WS-BPEL implementation, the process definition MUST be rejected.	Section 5.3

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00010	A WS-BPEL process definition MUST import all XML Schema and WSDL definitions it uses. This includes all XML Schema type and element definitions, all WSDL port types and message types as well as property and property alias definitions used by the process.	Section 5.4
SA00011	If a `namespace` attribute is specified on an `<import>` then the imported definitions MUST be in that namespace.	Section 5.4
SA00012	If no namespace is specified then the imported definitions MUST NOT contain a `targetNamespace` specification.	Section 5.4
SA00013	The value of the `importType` attribute of element `<import>` 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.	Section 5.4
SA00014	A WS-BPEL process definition MUST be rejected if the imported documents contain conflicting definitions of a component used by the importing process definition (as could be caused, for example, when the XSD redefinition mechanism is used).	Section 5.4
SA00015	To be instantiated, an executable business process MUST contain at least one `<receive>` or `<pick>` activity annotated with a `createInstance="yes"` attribute.	Section 5.5
SA00016	A `partnerLink` MUST specify the `myRole` or the `partnerRole`, or both.	Section 6.2
SA00017	The `initializePartnerRole` attribute MUST NOT be used on a `partnerLink` that does not have a partner role.	Section 6.2
SA00018	The name of a `partnerLink` MUST be unique among the names of all partnerLinks defined within the same immediately enclosing scope.	Section 6.2
SA00019	Either the `type` or `element` attributes MUST be present in a `<vprop:property>` element but not both.	Section 7.2
SA00020	A `<vprop:propertyAlias>` element MUST use one of the three following combinations of attributes: • `messageType` and `part`, • `type` or • `element`	Section 7.3

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00021	Static analysis MUST detect property usages where propertyAliases for the associated variable's type are not found in any WSDL definitions directly imported by the WS-BPEL process.	Section 7.3
SA00022	A WS-BPEL process definition MUST NOT be accepted for processing if it defines two or more propertyAliases for the same property name and WS-BPEL variable type.	Section 7.3
SA00023	The name of a variable MUST be unique among the names of all variables defined within the same immediately enclosing scope.	Section 8.1
SA00024	Variable names are `BPELVariableName`s, that is, NCNames (as defined in XML Schema specification) but in addition they MUST NOT contain the "." character.	Section 8.1
SA00025	The `messageType`, `type` or `element` attributes are used to specify the type of a variable. Exactly one of these attributes MUST be used.	Section 8.1
SA00026	Variable initialization logic contained in scopes that contain or whose children contain a start activity MUST only use idempotent functions in the from-spec.	Section 8.1
SA00027	When XPath 1.0 is used as an expression language in WS-BPEL 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 language.	Section 8.2.4
SA00028	WS-BPEL functions MUST NOT be used in joinConditions.	Section 8.2.5
SA00029	WS-BPEL variables and WS-BPEL functions MUST NOT be used in query expressions of propertyAlias definitions.	Section 8.2.6
SA00030	The arguments to `bpel:getVariableProperty` MUST be given as quoted strings. It is therefore illegal to pass into a WS-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.	Section 8.3

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00031	The second argument of the XPath 1.0 extension function `bpel:getVariableProperty(string, string)` MUST be a string literal conforming to the definition of QName in [XML Namespaces] section 3.	Section 8.3

SA00032	For <assign>, the <from> and <to> element MUST be one of the specified variants.
	
	The `<assign>` activity copies a type-compatible value from the source ("from-spec") to the destination ("to-spec"), using the `<copy>` element. Except in Abstract Processes, the from-spec MUST be one of the following variants:
	
	`<from variable="BPELVariableName" part="NCName"?>`
	`  <query queryLanguage="anyURI"?>?`
	`  queryContent`
	`  </query>`
	`</from>`
	`<from partnerLink="NCName"`
	`  endpointReference="myRole|partnerRole" />`
	`<from variable="BPELVariableName"`
	`  property="QName" />`
	`<from expressionLanguage="anyURI"?>`
	`  expression`
	`</from>`
	`<from>`
	`  <literal>literal value</literal>`
	`</from>`
	`<from/>`
	
	In Abstract Processes, the from-spec MUST be either one of the above or the opaque variant described in section 13.1.3. Hiding Syntactic Elements
	
	The to-spec MUST be one of the following variants:
	
	`<to variable="BPELVariableName" part="NCName"?>`
	`  <query queryLanguage="anyURI"?>?`
	`  queryContent`
	`  </query>`
	`</to>`
	`<to partnerLink="NCName" />`
	`<to variable="BPELVariableName"`
	`  property="QName" />`
	`<to expressionLanguage="anyURI"?>`
	`  expression`
	`</to>`
	`<to/>`
		Section 8.4

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00033	The XPath expression in <to> MUST begin with an XPath VariableReference.	Section 8.4
SA00034	When the variable used in <from> or <to> is defined using XML Schema types (simple or complex) or element, the part attribute MUST NOT be used.	Section 8.4
SA00035	In the from-spec of the partnerLink variant of <assign> the value "myRole" for attribute endpointReference is only permitted when the partnerLink specifies the attribute myRole.	Section 8.4
SA00036	In the from-spec of the partnerLink variant of <assign> the value "partnerRole" for attribute endpointReference is only permitted when the partnerLink specifies the attribute partnerRole.	Section 8.4
SA00037	In the to-spec of the partnerLink variant of assign only partnerLinks are permitted which specify the attribute partnerRole.	Section 8.4
SA00038	The literal from-spec variant returns values as if it were a from-spec that selects the children of the <literal> element in the WS-BPEL source code. The return value MUST be a single EII or Text Information Item (TII) only.	Section 8.4
SA00039	The first parameter of the XPath 1.0 extension function bpel:doXslTransform(string, node-set, (string, object)*) is an XPath string providing a URI naming the style sheet to be used by the WS-BPEL processor. This MUST take the form of a string literal.	Section 8.4
SA00040	In the XPath 1.0 extension function bpel:doXslTransform(string, node-set, (string, object)*) the optional parameters after the second parameter MUST appear in pairs. An odd number of parameters is not valid.	Section 8.4
SA00041	For the third and subsequent parameters of the XPath 1.0 extension function bpel:doXslTransform(string, node-set, (string, object)*) the global parameter names MUST be string literals conforming to the definition of QName in section 3 of [Namespaces in XML].	Section 8.4
SA00042	For <copy> the optional keepSrcElementName attribute is provided to further refine the behavior. It is only applicable when the results of both from-spec and to-spec are EIIs, and MUST NOT be explicitly set in other cases.	Section 8.4.2

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00043	For a copy operation to be valid, the data referred to by the from-spec and the to-spec MUST be of compatible types. The following situations are considered type incompatible: • the selection results of both the from-spec and the to-spec are variables of a WSDL message type, and the two variables are not of the same WSDL message type (two WSDL message types are the same if their QNames are equal). • the selection result of the from-spec is a variable of a WSDL message type and that of the to-spec is not, or vice versa (parts of variables, selections of variable parts, or endpoint references cannot be assigned to/from variables of WSDL message types directly).	Section 8.4.3
SA00044	The name of a `<correlationSet>` MUST be unique among the names of all `<correlationSet>` defined within the same immediately enclosing scope.	Section 9.1
SA00045	Properties used in a `<correlationSet>` MUST be defined using XML Schema simple types.	Section 9.2
SA00046	The `pattern` attribute used in `<correlation>` within `<invoke>` is required for request-response operations, and disallowed when a one-way operation is invoked.	Section 9.2
SA00047	One-way invocation requires only the `inputVariable` (or its equivalent `<toPart>` elements) since a response is not expected as part of the operation (see section 10.4. Providing Web Service Operations – Receive and Reply). Request-response invocation requires both an `inputVariable` (or its equivalent `<toPart>` elements) and an `outputVariable` (or its equivalent `<fromPart>` elements). If a WSDL message definition does not contain any parts, then the associated attributes `variable`, `inputVariable` or `outputVariable`, MAY be omitted,and the `<fromParts>` or `<toParts>` construct MUST be omitted.	Section 10.3 Section 10.4 Section 10.4 Section 11.5 Section 12.7

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00048	When the optional `inputVariable` and `outputVariable` attributes are being used in an `<invoke>` activity, the variables referenced by `inputVariable` and `outputVariable` MUST be messageType variables whose QName matches the QName of the input and output message type used in the operation, respectively, except as follows: if the WSDL operation used in an `<invoke>` activity uses a message containing exactly one part which itself is defined using an element, then a variable of the same element type as used to define the part MAY be referenced by the `inputVariable` and `outputVariable` attributes respectively.	Section 10.3
SA00050	When `<toParts>` is, it is required to have a `<toPart>` for every part in the WSDL message definition; the order in which parts are specified is irrelevant. Parts not explicitly represented by `<toPart>` elements would result in uninitialized parts in the target anonymous WSDL variable used by the `<invoke>` or `<reply>` activity. Such processes with missing `<toPart>` elements MUST be rejected during static analysis.	Section 10.3.1
SA00051	The `inputVariable` attribute MUST NOT be used on an `Invoke` activity that contains `<toPart>` elements.	Section 10.3.1
SA00052	The `outputVariable` attribute MUST NOT be used on an `<invoke>` activity that contains a `<fromPart>` element.	Section 10.3.1
SA00053	For all <fromPart> elements the part attribute MUST reference a valid message part in the WSDL message for the operation.	Section 5.4
SA00054	For all <toPart> elements the part attribute MUST reference a valid message part in the WSDL message for the operation.	Section 5.4
SA00055	For <receive>, if `<fromPart>` elements are used on a `<receive>` activity then the `variable` attribute MUST NOT be used on the same activity.	Section 10.4
SA00056	A "start activity" is a `<receive>` or `<pick>` activity that is annotated with a `createInstance="yes"` attribute. Activities other than the following: start activities, `<scope>`, `<flow>` and `<sequence>` MUST NOT be performed prior to or simultaneously with start activities.	Section 10.4
SA00057	If a process has multiple start activities with correlation sets then all such activities MUST share at least one common correlationSet and all common correlationSets defined on all the activities MUST have the value of the `initiate` attribute be set to "`join`".	Section 10.4

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00058	In a <receive> or <reply> activity, the variable referenced by the variable attribute MUST be a messageType variable whose QName matches the QName of the input (for <receive>) or output (for <reply>) message type used in the operation, except as follows: if the WSDL operation uses a message containing exactly one part which itself is defined using an element, then a WS-BPEL variable of the same element type as used to define the part MAY be referenced by the variable attribute of the <receive> or <reply>activity.	Section 10.4
SA00059	For <reply>, if <toPart> elements are used on a <reply> activity then the variable attribute MUST NOT be used on the same activity.	Section 10.4
SA00060	The explicit use of messageExchange is needed only where the execution can result in multiple IMA-<reply> pairs (e.g. <receive>-<reply> pair) on the same partnerLink and operation being executed simultaneously. In these cases, the process definition MUST explicitly mark the pairing-up relationship.	Section 10.4.1
SA00061	The name used in the optional messageExchange attribute MUST resolve to a messageExchange declared in a scope (where the process is considered the root scope) which encloses the <reply> activity and its corresponding IMA.	Section 10.4.1
SA00062	If <pick> has a createInstance attribute with a value of yes, the events in the <pick> MUST all be <onMessage> events.	Section 11.5
SA00063	The semantics of the <onMessage> event are identical to a <receive> activity regarding the optional nature of the variable attribute or <fromPart> elements, if <fromPart> elements on an activity then the variable attribute MUST NOT be used on the same activity (see SA00055).	Section 11.5
SA00064	For <flow>, a declared link's name MUST be unique among all <link> names defined within the same immediately enclosing <flow>.	Section 11.6
SA00065	The value of the linkName attribute of <source> or <target> MUST be the name of a <link> declared in an enclosing <flow> activity.	Section 11.6.1
SA00066	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.	Section 11.6.1

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00067	Two different links MUST NOT share the same source and target activities; that is, at most one link may be used to connect two activities.	Section 11.6.1
SA00068	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.	Section 11.6.1
SA00069	An activity MAY declare itself to be the target of one or more links by including one or more <target> elements. Each <target> element associated with a given activity MUST use a link name distinct from all other <target> elements at that activity.	Section 11.6.1
SA00070	A link MUST NOT cross the boundary of a repeatable construct or the <compensationHandler> element. This means, a link used within a repeatable construct (<while>, <repeatUntil>, <forEach>, <eventHandlers>) or a <compensationHandler> MUST be declared in a <flow> that is itself nested inside the repeatable construct or <compensationHandler>.	Section 11.6.1
SA00071	A link that crosses a <catch>, <catchAll> or <terminationHandler> element boundary MUST be outbound only, that is, it MUST have its source activity within the <faultHandlers> or <terminationHandler>, and its target activity outside of the scope associated with the handler.	Section 11.6.1
SA00072	A <link> declared in a <flow> MUST NOT create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity.	Section 11.6.1
SA00073	The expression for a join condition MUST be constructed using only Boolean operators and the activity's incoming links' status values.	Section 11.6.2
SA00074	The expressions in <startCounterValue> and <finalCounterValue> MUST return a TII (meaning they contain at least one character) that can be validated as a xsd:unsignedInt. Static analysis MAY be used to detect this erroneous situation at design time when possible (for example, when the expression is a constant).	Section 11.7
SA00075	For the <forEach> activity, <branches> is an integer value expression. Static analysis MAY be used to detect if the integer value is larger than the number of directly enclosed activities of <forEach> at design time when possible (for example, when the branches expression is a constant).	Section 11.7

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00076	For <forEach> the enclosed scope MUST NOT declare a variable with the same name as specified in the counterName attribute of <forEach>.	Section 11.7
SA00077	The value of the target attribute on a <compensateScope> activity MUST refer to the name of an immediately enclosed scope of the scope containing the FCT-handler with the <compensateScope> activity. This includes immediately enclosed scopes of an event handler (<onEvent> or <onAlarm>) associated with the same scope.	Section 12.4.3.1
SA00078	The target attribute of a <compensateScope> activity MUST refer to a scope or an invoke activity with a fault handler or compensation handler.	Section 12.4.3.1
SA00079	The root scope inside a FCT-handler MUST not have a compensation handler.	Section 12.4.4.3
SA00080	There MUST be at least one <catch> or <catchAll> element within a <faultHandlers> element.	Section 12.5
SA00081	For the <catch> construct; to have a defined type associated with the fault variable, the faultVariable attribute MUST only be used if either the faultMessageType or faultElement attributes, but not both, accompany it. The faultMessageType and faultElement attributes MUST NOT be used unless accompanied by faultVariable attribute.	Section 12.5
SA00082	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.	Section 12.5.2
SA00083	An event handler MUST contain at least one <onEvent> or <onAlarm> element.	Section 12.7
SA00084	The partnerLink reference of <onEvent> MUST resolve to a partner link declared in the process in the following order: the associated scope first and then the ancestor scopes.	Section 12.7.1
SA00085	The syntax and semantics of the <fromPart> elements as used on the <onEvent> element are the same as specified for the receive activity. This includes the restriction that if <fromPart> elements are used on an onEvent element then the variable, element and messageType attributes MUST NOT be used on the same element.	Section 12.7.1

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00086	For <onEvent>, variables referenced by the variable attribute of <fromPart> elements or the variable attribute of an <onEvent> element are implicitly declared in the associated scope of the event handler. Variables of the same names MUST NOT be explicitly declared in the associated scope..	Section 12.7.1
SA00087	For <onEvent>, 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 element type MUST be an exact match of the element type referenced by the element attribute.	Section 12.7.1
SA00088	For <onEvent>, the resolution order of the correlation set(s) referenced by <correlation> MUST be first the associated scope and then the ancestor scopes.	Section 12.7.1
SA00089	For <onEvent>, when the messageExchange attribute is explicitly specified, the resolution order of the message exchange referenced by messageExchange attribute MUST be first the associated scope and then the ancestor scopes.	Section 12.7.1
SA00090	If the variable attribute is used in the <onEvent> element, either the messageType or the element attribute MUST be provided in the <onEvent> element.	Section 12.7.1
SA00091	A scope with the isolated attribute set to "yes" is called an isolated scope. Isolated scopes MUST NOT contain other isolated scopes.	Section 12.8
SA00092	Within a scope, the name of all named immediately enclosed scopes MUST be unique.	Section 12.4.3
SA00093	Identical <catch> constructs MUST NOT exist within a <faultHandlers> element.	Section 12.5
SA00094	For <copy>, when the keepSrcElementName attribute is set to "yes" and the destination element is the Document EII of an element-based variable or an element-based part of a WSDL message-type-based variable, the name of the source element MUST belong to the substitutionGroup of the destination element. This checking MAY be enforced through static analysis of the expression/query language.	Section 8.4.2

Static Analysis Fault Code	Static analysis Description	Section Reference
SA00095	For <onEvent>, the variable references are resolved to the associated scope only and MUST NOT be resolved to the ancestor scopes.	Section 12.7.1

Appendix C. Attributes and Defaults

The following list summarizes all standard attributes for which a default value is defined.

Table C.1. Attributes and Defaults

Attribute	Default

`createInstance`
on elements
`<pick>`
`<receive>`
	`no`

`exitOnStandardFault`
on element
`<process>`
	`no`

`exitOnStandardFault`
on element
`<scope>`
	When this attribute is not specified on a `<scope>`, it inherits its value from its immediately enclosing `<scope>` (where the top-level scope is the `<process>` itself).

`expressionLanguage`
on element
`<process>`
	`urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0`

`expressionLanguage`
on elements
`<branches>`
`<condition>`
`<finalCounterValue>`
`<for>`
`<from>`
`<joinCondition>`
`<repeatEvery>`
`<startCounterValue>`
`<to>`
`<transitionCondition>`
`<until>`
	When this attribute is not specified for one of these elements, the attribute inherits its value from `<process>`.

`initializePartnerRole`
on element
`<partnerLink>`
	`no`

`initiate`
on element
`<correlation>`
	`no`

`isolated`
on element
`<scope>`
	`no`

`keepSrcElementName`
on element
`<copy>`
	`no`

`location` on element	An `<import>` element without a `location` attribute indicates that external definitions are used by the process

Attribute	Default
<import>	but makes no statement about where those definitions may be found.
messageExchange on elements <receive> <reply> <onMessage> <onEvent>	If not specified on an inbound message activity or <reply> then the activity's messageExchange is automatically associated with a default messageExchange with no name.
namespace on element <import>	An <import> element without a namespace attribute indicates that external definitions are in use which are not namespace qualified.
queryLanguage on element <process>	urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0
queryLanguage on element <query>	When this attribute is not specified for a <query> that is part of a from-spec or to-spec then the attribute inherits its value from <process>. If the <query> is part of a <vprop:propertyAlias> and the attribute is not specified its default value is: urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0
reference-scheme on element <sref:service-ref>	If not specified, the namespace URI of the content element within the wrapper MUST be used to determine the reference scheme of service endpoint.
successfulBranchesOnly on element <branches>	no
suppressJoinFailure on element <process>	no
suppressJoinFailure on each activity (standard-attribute)	When this attribute is not specified for an activity, it inherits its value from its directly enclosing activity (or from the <process> itself, if it is the primary activity of the process).
validate on element <assign>	no

Appendix D. Examples of Replacement Logic

The following provides detailed examples illustrative of copy operations as described in section 8.4.2. Replacement Logic of Copy Operations.

(a) EII-to-EII copy

XML Schema Context

```
<xsd:element name="poHeader">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:choice>
            <xsd:element name="shippingAddr" type="tns:AddressType" />
            <xsd:element name="USshippingAddr"
               type="tns:USAddressType" />
         </xsd:choice>
         <xsd:element name="billingAddr" type="tns:AddressType" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

"tns:USAddressType" is a type extended from "tns:AddressType".

- Example 1:

```
<assign>
   <copy>
      <from>$poHeaderVar1/tns:shippingAddr</from>
      <to>$poHeaderVar2/tns:billingAddr</to>
   </copy>
</assign>
```

This <copy> replaces the attributes and elements of the billing address in "poHeaderVar2" with those of shipping address in "poHeaderVar1".

Value of poHeaderVar1

```
<tns:poHeader>
   ...
   <tns:shippingAddr verified="true">
      <tns:street>123 Main Street</tns:street>
      <tns:city>SomeWhere City</tns:city>
      <tns:country>UK</tns:country>
   </tns:shippingAddr>
   ...
</tns:poHeader>
```

Value of poHeaderVar2: (prior to <copy>)

```
<tns:poHeader>
```

```
   ...
   <tns:billingAddr pobox="true" />
   ...
</tns:poHeader>
```

Value of poHeaderVar2: (subsequent to <copy>)

```
<tns:poHeader>
   ...
   <tns:billingAddr verified="true">
      <tns:street>123 Main Street</tns:street>
      <tns:city>SomeWhere City</tns:city>
      <tns:country>UK</tns:country>
   </tns:billingAddr>
   ...
</tns:poHeader>
```

- Example 2:

```
<assign>
   <copy keepSrcElementName="yes">
      <from>$poHeaderVar3/tns:USshippingAddr</from>
      <to>$poHeaderVar2/tns:shippingAddr</to>
   </copy>
</assign>
```

This <copy> replaces the attributes and elements of the shipping address in "poHeaderVar2" with those of the US shipping address in "poHeaderVar3".

Value of poHeaderVar3:

```
<tns:poHeader>
   ...
   <tns:USshippingAddr verified="true">
      <tns:street>123 Main Street</tns:street>
      <tns:city>SomeWhere City</tns:city>
      <tns:country>USA</tns:country>
      <tns:zipcode>98765</tns:zipcode>
   </tns:USshippingAddr>
   ...
</tns:poHeader>
```

Value of poHeaderVar2: (prior to <copy>)

```
<tns:poHeader>
   ...
   <tns:shippingAddr pobox="true" />
   ...
</tns:poHeader>
```

Value of poHeaderVar2: (subsequent to <copy>)

```
<tns:poHeader>
   ...
```

```
   <tns:USshippingAddr verified="true">
      <tns:street>123 Main Street</tns:street>
      <tns:city>SomeWhere City</tns:city>
      <tns:country>USA</tns:country>
      <tns:zipcode>98765</tns:zipcode>
   </tns:USshippingAddr>
   ...
</tns:poHeader>
```

(b) EII-to-AII copy

XML Data Context

Value of `creditApprovalVar`:

```
<tns:creditApplication appId="123-456">
   <tns:approvedLimit code="AXR">4500</tns:approvedLimit>
</tns:creditApplication>
```

- Example 1:

```
<assign>
   <copy>
      <from>$creditApprovalVar/tns:approvedLimit</from>
      <to>$approvalNotice2Var/@amt</to>
   </copy>
</assign>
```

This <copy> replaces the content of the amount attribute in "`approvalNotice2Var`" with that of the approved limit in "`creditApprovalVar`".

Value of `approvalNotice2Var`: (prior to <copy>)

```
<tns2:approvalNotice amt="" />
```

Value of `approvalNotice2Var`: (subsequent to <copy>)

```
<tns2:approvalNotice amt="4500" />
```

(c) EII-to-TII copy

XML Data Context

Value of `creditApprovalVar`:

```
<tns:creditApplication appId="123-456">
   <tns:approvedLimit code="AXR">4500</tns:approvedLimit>
</tns:creditApplication>
```

- Example 1:

```
<assign>
   <copy>
      <from>$creditApprovalVar/tns:approvedLimit</from>
      <to>$approvalNotice3Var/text()</to>
   </copy>
</assign>
```

This <copy> replaces the content of "approvalNotice3Var" with that of the approved limit in "creditApprovalVar".

Value of approvalNotice3Var: (prior to <copy>)

```
<tns3:approvalNotice>0</tns3:approvalNotice>
```

Value of approvalNotice3Var: (subsequent to <copy>)

```
<tns3:approvalNotice>4500</tns3:approvalNotice>
```

- Example 2:

```
<assign>
   <copy>
      <from>$creditApprovalVar/tns:approvedLimit</from>
      <to>$approvalNotice4Var/text()</to>
   </copy>
</assign>
```

Value of approvalNotice4Var: (prior to <copy>)

```
<tns4:approvalNotice></tns4:approvalNotice>
```

As no text node exists under"tns4:approvalNotice", a selectionFailure fault will be thrown, and no replacement logic executed.

- Example 3: EII-to-EII (for comparison to EII-to-TII)

```
<assign>
   <copy>
      <from>$creditApprovalVar/tns:approvedLimit</from>
      <to>$approvalNotice4Var</to>
   </copy>
</assign>
```

This <copy> replaces the attributes and elements of "approvalNotice4Var" with those of the approved limit in "creditApprovalVar".

Value of approvalNotice4Var: (prior to EII-to-EII <copy>)

```
<tns4:approvalNotice></tns4:approvalNotice>
```

Value of approvalNotice4Var: (subsequent to EII-to-EII <copy>)

```
<tns4:approvalNotice code="AXR">4500</tns4:approvalNotice>
```

(d) AII-to-AII copy

XML Data Context

Value of `orderDetailVar`:

```
<tns:orderDetail amt="2299" />
```

- Example 1:

```
<assign>
   <copy>
      <from>$orderDetailVar/@amt</from>
      <to>$billingDetailVar/@amt</to>
   </copy>
</assign>
```

This <copy> replaces the content of the amount attribute in "`billingDetailVar`" with that of the amount if "`orderDetailVar`".

Value of `billingDetailVar`: (prior to <copy>)

```
<tns:billingDetail amt="" />
```

Value of `billingDetailVar`: (subsequent to <copy>)

```
<tns:billingDetail amt="2299" />
```

(e) AII-to-EII copy

XML Data Context

Value of `orderDetailVar`:

```
<tns:orderDetail amt="3399" />
```

- Example 1:

```
<assign>
   <copy>
      <from>$orderDetailVar/@amt</from>
      <to>$billingDetailVar/tns1:billingAmount</to>
   </copy>
</assign>
```

This <copy> replaces the content of the billing amount in "`billingDetailVar`" with that of the amount attribute in "`orderDetailVar`".

Value of `billingDetailVar`: (prior to <copy>)

```
<tns1:billingDetail id="8675309">
   <tns1:billingAmount code="F00B2R"></tns1:billingAmount>
</tns1:billingDetail>
```

Value of `billingDetailVar`: (subsequent to <copy>)

```
<tns1:billingDetail id="8675309">
   <tns1:billingAmount code="F00B2R">3399</tns1:billingAmount>
</tns1:billingDetail>
```

(f) AII-to-TII copy

XML Data context.

Value of `orderDetailVar`:

```
<tns:orderDetail amt="4499" />
```

- Example 1:

```
<assign>
   <copy>
      <from>$orderDetailVar/@amt</from>
      <to>$billingAmount2Var/text()</to>
   </copy>
</assign>
```

This <copy> replaces the content of "`billingAmount2Var`" with that of the amount attribute in "`orderDetailVar`".

Value of `billingAmount2Var`: (prior to <copy>)

```
<tns2:billingAmount>0</tns2:billingAmount>
```

Value of `billingAmount2Var`: (subsequent to <copy>)

```
<tns2:billingAmount>4499</tns2:billingAmount>
```

(g) TII-to-TII copy

XML Data context

Value of `postalCodeVar`:

```
<tns:postalCode>95110</tns:postalCode>
```

- Example 1:

```
<assign>
   <copy>
      <from>$postalCodeVar/text()</from>
      <to>$shippingPostalCodeVar/text()</to>
   </copy>
</assign>
```

This <copy> replaces the content of "shippingPostalCodeVar" with that of "postalCodeVar".

Value of shippingPostalCodeVar: (prior to <copy>)

```
<tns:shippingPostalCode>0</tns:shippingPostalCode>
```

Value of shippingPostalCodeVar: (subsequent to <copy>)

```
<tns:shippingPostalCode>95110</tns:shippingPostalCode>
```

(h) TII-to-AII copy

XML Data Context

Value of postalCodeVar:

```
<tns:postalCode>94304</tns:postalCode>
```

- Example 1:

```
<assign>
   <copy>
      <from>$postalCodeVar/text()</from>
      <to>$shippingAddress1Var/@postCode</to>
   </copy>
</assign>
```

This <copy> replaces the content of the post code attribute of "shippingAddress1Var" with the content of "postalCodeVar".

Value of shippingAddress1Var: (prior to <copy>)

```
<tns1:shippingAddress postCode="" />
```

Value of approvalNotice1Var: (subsequent to <copy>)

```
<tns1:shippingAddress postCode="94304" />
```

(i) TII-to-EII copy

XML Data Context

Value of `postalCodeVar`:

```
<tns:postalCode>94107</tns:postalCode>
```

- Example 1:

```
<assign>
   <copy>
      <from>$postalCodeVar/text()</from>
      <to>$shippingAddress2Var/tns2:postalCode</to>
   </copy>
</assign>
```

This <copy> replaces the content of the postal code element in "`shippingAddress2Var`" with that of "`postalCodeVar`".

Value of `shippingAddress2Var`: (prior to <copy>)

```
<tns2:shippingAddress id="9035768">
   <tns2:postalCode></tns2:postalCode>
</tns2:shippingAddress>
```

Value of `shippingAddress2Var`: (subsequent to <copy>)

```
<tns2:shippingAddress id="9035768">
   <tns2:postalCode>94107</tns2:postalCode>
</tns2:shippingAddress>
```

Appendix E. XML Schemas

Schema for Executable Process for WS-BPEL 2.0

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Copyright (c) OASIS Open 2003-2006. All Rights Reserved.
-->
<xsd:schema
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  elementFormDefault="qualified" blockDefault="#all">
  <xsd:annotation>
    <xsd:documentation>
      Schema for Executable Process for WS-BPEL 2.0
      Last modified date: 18th October, 2006
    </xsd:documentation>
  </xsd:annotation>
  <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="http://www.w3.org/2001/xml.xsd" />
  <xsd:element name="process" type="tProcess">
    <xsd:annotation>
      <xsd:documentation>
        This is the root element for a WS-BPEL 2.0 process.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="tProcess">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="extensions" minOccurs="0" />
          <xsd:element ref="import" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="partnerLinks" minOccurs="0" />
          <xsd:element ref="messageExchanges" minOccurs="0" />
          <xsd:element ref="variables" minOccurs="0" />
          <xsd:element ref="correlationSets" minOccurs="0" />
          <xsd:element ref="faultHandlers" minOccurs="0" />
          <xsd:element ref="eventHandlers" minOccurs="0" />
          <xsd:group ref="activity" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:NCName" use="required" />
        <xsd:attribute name="targetNamespace" type="xsd:anyURI"
          use="required" />
        <xsd:attribute name="queryLanguage" type="xsd:anyURI"
          default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" />
        <xsd:attribute name="expressionLanguage" type="xsd:anyURI"
          default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" />
        <xsd:attribute name="suppressJoinFailure" type="tBoolean"
          default="no" />
        <xsd:attribute name="exitOnStandardFault" type="tBoolean"
          default="no" />
      </xsd:extension>
```

```
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tExtensibleElements">
    <xsd:annotation>
      <xsd:documentation>
        This type is extended by other component types to allow
        elements and attributes from other namespaces to be added at
        the modeled places.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element ref="documentation" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:element name="documentation" type="tDocumentation" />
  <xsd:complexType name="tDocumentation" mixed="true">
    <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="source" type="xsd:anyURI" />
    <xsd:attribute ref="xml:lang" />
  </xsd:complexType>
  <xsd:group name="activity">
    <xsd:annotation>
      <xsd:documentation>
        All standard WS-BPEL 2.0 activities in alphabetical order.
        Basic activities and structured activities. Addtional
        constraints: - rethrow activity can be used ONLY within a
        fault handler (i.e. "catch" and "catchAll" element) -
        compensate or compensateScope activity can be used ONLY within
        a fault handler, a compensation handler or a termination
        handler
      </xsd:documentation>
    </xsd:annotation>
    <xsd:choice>
      <xsd:element ref="assign" />
      <xsd:element ref="compensate" />
      <xsd:element ref="compensateScope" />
      <xsd:element ref="empty" />
      <xsd:element ref="exit" />
      <xsd:element ref="extensionActivity" />
      <xsd:element ref="flow" />
      <xsd:element ref="forEach" />
      <xsd:element ref="if" />
      <xsd:element ref="invoke" />
      <xsd:element ref="pick" />
      <xsd:element ref="receive" />
      <xsd:element ref="repeatUntil" />
      <xsd:element ref="reply" />
      <xsd:element ref="rethrow" />
      <xsd:element ref="scope" />
      <xsd:element ref="sequence" />
      <xsd:element ref="throw" />
```

```
      <xsd:element ref="validate" />
      <xsd:element ref="wait" />
      <xsd:element ref="while" />
    </xsd:choice>
  </xsd:group>
  <xsd:element name="extensions" type="tExtensions" />
  <xsd:complexType name="tExtensions">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="extension" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="extension" type="tExtension" />
  <xsd:complexType name="tExtension">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="namespace" type="xsd:anyURI"
          use="required" />
        <xsd:attribute name="mustUnderstand" type="tBoolean"
          use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="import" type="tImport" />
  <xsd:complexType name="tImport">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="namespace" type="xsd:anyURI"
          use="optional" />
        <xsd:attribute name="location" type="xsd:anyURI"
          use="optional" />
        <xsd:attribute name="importType" type="xsd:anyURI"
          use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="partnerLinks" type="tPartnerLinks" />
  <xsd:complexType name="tPartnerLinks">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="partnerLink" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="partnerLink" type="tPartnerLink" />
  <xsd:complexType name="tPartnerLink">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="name" type="xsd:NCName" use="required" />
        <xsd:attribute name="partnerLinkType" type="xsd:QName"
          use="required" />
        <xsd:attribute name="myRole" type="xsd:NCName" />
        <xsd:attribute name="partnerRole" type="xsd:NCName" />
```

```
      <xsd:attribute name="initializePartnerRole" type="tBoolean" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="messageExchanges" type="tMessageExchanges" />
<xsd:complexType name="tMessageExchanges">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="messageExchange" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="messageExchange" type="tMessageExchange" />
<xsd:complexType name="tMessageExchange">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="name" type="xsd:NCName" use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="variables" type="tVariables" />
<xsd:complexType name="tVariables">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="variable" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="variable" type="tVariable" />
<xsd:complexType name="tVariable">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="from" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="name" type="BPELVariableName"
        use="required" />
      <xsd:attribute name="messageType" type="xsd:QName"
        use="optional" />
      <xsd:attribute name="type" type="xsd:QName" use="optional" />
      <xsd:attribute name="element" type="xsd:QName" use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="BPELVariableName">
  <xsd:restriction base="xsd:NCName">
    <xsd:pattern value="[^\.]+" />
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="correlationSets" type="tCorrelationSets" />
<xsd:complexType name="tCorrelationSets">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
```

```
          <xsd:element ref="correlationSet" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="correlationSet" type="tCorrelationSet" />
  <xsd:complexType name="tCorrelationSet">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="properties" type="QNames" use="required" />
        <xsd:attribute name="name" type="xsd:NCName" use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="QNames">
    <xsd:restriction>
      <xsd:simpleType>
        <xsd:list itemType="xsd:QName" />
      </xsd:simpleType>
      <xsd:minLength value="1" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="faultHandlers" type="tFaultHandlers" />
  <xsd:complexType name="tFaultHandlers">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="catch" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="catchAll" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="catch" type="tCatch">
    <xsd:annotation>
      <xsd:documentation>
        This element can contain all activities including the
        activities compensate, compensateScope and rethrow.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="tCatch">
    <xsd:complexContent>
      <xsd:extension base="tActivityContainer">
        <xsd:attribute name="faultName" type="xsd:QName" />
        <xsd:attribute name="faultVariable" type="BPELVariableName" />
        <xsd:attribute name="faultMessageType" type="xsd:QName" />
        <xsd:attribute name="faultElement" type="xsd:QName" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="catchAll" type="tActivityContainer">
    <xsd:annotation>
      <xsd:documentation>
        This element can contain all activities including the
        activities compensate, compensateScope and rethrow.
      </xsd:documentation>
```

```
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="tActivityContainer">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:group ref="activity" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="eventHandlers" type="tEventHandlers" />
  <xsd:complexType name="tEventHandlers">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element onAlarm needs to be a Local
        Element Declaration, because there is another onAlarm element
        defined for the pick activity.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="onEvent" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element name="onAlarm" type="tOnAlarmEvent"
            minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="onEvent" type="tOnEvent" />
  <xsd:complexType name="tOnEvent">
    <xsd:complexContent>
      <xsd:extension base="tOnMsgCommon">
        <xsd:sequence>
          <xsd:element ref="scope" />
        </xsd:sequence>
        <xsd:attribute name="messageType" type="xsd:QName"
          use="optional" />
        <xsd:attribute name="element" type="xsd:QName" use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tOnMsgCommon">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlations needs to be a
        Local Element Declaration, because there is another
        correlations element defined for the invoke activity.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element name="correlations" type="tCorrelations"
            minOccurs="0" />
          <xsd:element ref="fromParts" minOccurs="0" />
```

```
      </xsd:sequence>
      <xsd:attribute name="partnerLink" type="xsd:NCName"
        use="required" />
      <xsd:attribute name="portType" type="xsd:QName"
        use="optional" />
      <xsd:attribute name="operation" type="xsd:NCName"
        use="required" />
      <xsd:attribute name="messageExchange" type="xsd:NCName"
        use="optional" />
      <xsd:attribute name="variable" type="BPELVariableName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tCorrelations">
  <xsd:annotation>
    <xsd:documentation>
      XSD Authors: The child element correlation needs to be a Local
      Element Declaration, because there is another correlation
      element defined for the invoke activity.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element name="correlation" type="tCorrelation"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tCorrelation">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="set" type="xsd:NCName" use="required" />
      <xsd:attribute name="initiate" type="tInitiate" default="no" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="tInitiate">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="yes" />
    <xsd:enumeration value="join" />
    <xsd:enumeration value="no" />
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="tOnAlarmEvent">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:choice>
          <xsd:sequence>
            <xsd:group ref="forOrUntilGroup" />
            <xsd:element ref="repeatEvery" minOccurs="0" />
          </xsd:sequence>
          <xsd:element ref="repeatEvery" />
        </xsd:choice>
        <xsd:element ref="scope" />
```

```
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:group name="forOrUntilGroup">
  <xsd:choice>
    <xsd:element ref="for" />
    <xsd:element ref="until" />
  </xsd:choice>
</xsd:group>
<xsd:element name="for" type="tDuration-expr" />
<xsd:element name="until" type="tDeadline-expr" />
<xsd:element name="repeatEvery" type="tDuration-expr" />
<xsd:complexType name="tActivity">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="targets" minOccurs="0" />
        <xsd:element ref="sources" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName" />
      <xsd:attribute name="suppressJoinFailure" type="tBoolean"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="targets" type="tTargets" />
<xsd:complexType name="tTargets">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="joinCondition" minOccurs="0" />
        <xsd:element ref="target" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="joinCondition" type="tCondition" />
<xsd:element name="target" type="tTarget" />
<xsd:complexType name="tTarget">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="linkName" type="xsd:NCName"
        use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="sources" type="tSources" />
<xsd:complexType name="tSources">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="source" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="source" type="tSource" />
```

```
<xsd:complexType name="tSource">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="transitionCondition" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="linkName" type="xsd:NCName"
        use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="transitionCondition" type="tCondition" />
<xsd:element name="assign" type="tAssign" />
<xsd:complexType name="tAssign">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element ref="copy" />
          <xsd:element ref="extensionAssignOperation" />
        </xsd:choice>
      </xsd:sequence>
      <xsd:attribute name="validate" type="tBoolean" use="optional"
        default="no" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="copy" type="tCopy" />
<xsd:complexType name="tCopy">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="from" />
        <xsd:element ref="to" />
      </xsd:sequence>
      <xsd:attribute name="keepSrcElementName" type="tBoolean"
        use="optional" default="no" />
      <xsd:attribute name="ignoreMissingFromData" type="tBoolean"
        use="optional" default="no" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="from" type="tFrom" />
<xsd:complexType name="tFrom" mixed="true">
  <xsd:sequence>
    <xsd:element ref="documentation" minOccurs="0"
      maxOccurs="unbounded" />
    <xsd:any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded" />
    <xsd:choice minOccurs="0">
      <xsd:element ref="literal" />
      <xsd:element ref="query" />
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="expressionLanguage" type="xsd:anyURI" />
  <xsd:attribute name="variable" type="BPELVariableName" />
  <xsd:attribute name="part" type="xsd:NCName" />
  <xsd:attribute name="property" type="xsd:QName" />
```

```
    <xsd:attribute name="partnerLink" type="xsd:NCName" />
    <xsd:attribute name="endpointReference" type="tRoles" />
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:element name="literal" type="tLiteral" />
  <xsd:complexType name="tLiteral" mixed="true">
    <xsd:sequence>
      <xsd:any namespace="##any" processContents="lax" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="query" type="tQuery" />
  <xsd:complexType name="tQuery" mixed="true">
    <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="queryLanguage" type="xsd:anyURI" />
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:simpleType name="tRoles">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="myRole" />
      <xsd:enumeration value="partnerRole" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="to" type="tTo" />
  <xsd:complexType name="tTo" mixed="true">
    <xsd:sequence>
      <xsd:element ref="documentation" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element ref="query" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI" />
    <xsd:attribute name="variable" type="BPELVariableName" />
    <xsd:attribute name="part" type="xsd:NCName" />
    <xsd:attribute name="property" type="xsd:QName" />
    <xsd:attribute name="partnerLink" type="xsd:NCName" />
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:element name="extensionAssignOperation"
    type="tExtensionAssignOperation" />
  <xsd:complexType name="tExtensionAssignOperation">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="compensate" type="tCompensate" />
  <xsd:complexType name="tCompensate">
    <xsd:complexContent>
      <xsd:extension base="tActivity" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="compensateScope" type="tCompensateScope" />
  <xsd:complexType name="tCompensateScope">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
```

```
      <xsd:attribute name="target" type="xsd:NCName" use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="empty" type="tEmpty" />
<xsd:complexType name="tEmpty">
  <xsd:complexContent>
    <xsd:extension base="tActivity" />
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="exit" type="tExit" />
<xsd:complexType name="tExit">
  <xsd:complexContent>
    <xsd:extension base="tActivity" />
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="extensionActivity" type="tExtensionActivity" />
<xsd:complexType name="tExtensionActivity">
  <xsd:sequence>
    <xsd:any namespace="##other" processContents="lax" />
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="flow" type="tFlow" />
<xsd:complexType name="tFlow">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element ref="links" minOccurs="0" />
        <xsd:group ref="activity" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="links" type="tLinks" />
<xsd:complexType name="tLinks">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="link" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="link" type="tLink" />
<xsd:complexType name="tLink">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="name" type="xsd:NCName" use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="forEach" type="tForEach" />
<xsd:complexType name="tForEach">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element ref="startCounterValue" />
        <xsd:element ref="finalCounterValue" />
```

```
        <xsd:element ref="completionCondition" minOccurs="0" />
        <xsd:element ref="scope" />
      </xsd:sequence>
      <xsd:attribute name="counterName" type="BPELVariableName"
        use="required" />
      <xsd:attribute name="parallel" type="tBoolean" use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="startCounterValue" type="tExpression" />
<xsd:element name="finalCounterValue" type="tExpression" />
<xsd:element name="completionCondition" type="tCompletionCondition" />
<xsd:complexType name="tCompletionCondition">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="branches" minOccurs="0" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="branches" type="tBranches" />
<xsd:complexType name="tBranches">
  <xsd:complexContent>
    <xsd:extension base="tExpression">
      <xsd:attribute name="successfulBranchesOnly" type="tBoolean"
        default="no" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="if" type="tIf" />
<xsd:complexType name="tIf">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element ref="condition" />
        <xsd:group ref="activity" />
        <xsd:element ref="elseif" minOccurs="0"
          maxOccurs="unbounded" />
        <xsd:element ref="else" minOccurs="0" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="elseif" type="tElseif" />
<xsd:complexType name="tElseif">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="condition" />
        <xsd:group ref="activity" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="else" type="tActivityContainer" />
<xsd:element name="invoke" type="tInvoke" />
<xsd:complexType name="tInvoke">
```

```
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlations needs to be a
        Local Element Declaration, because there is another
        correlations element defined for the non-invoke activities.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element name="correlations"
            type="tCorrelationsWithPattern" minOccurs="0" />
          <xsd:element ref="catch" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="catchAll" minOccurs="0" />
          <xsd:element ref="compensationHandler" minOccurs="0" />
          <xsd:element ref="toParts" minOccurs="0" />
          <xsd:element ref="fromParts" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="partnerLink" type="xsd:NCName"
          use="required" />
        <xsd:attribute name="portType" type="xsd:QName"
          use="optional" />
        <xsd:attribute name="operation" type="xsd:NCName"
          use="required" />
        <xsd:attribute name="inputVariable" type="BPELVariableName"
          use="optional" />
        <xsd:attribute name="outputVariable" type="BPELVariableName"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tCorrelationsWithPattern">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlation needs to be a Local
        Element Declaration, because there is another correlation
        element defined for the non-invoke activities.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element name="correlation"
            type="tCorrelationWithPattern" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tCorrelationWithPattern">
    <xsd:complexContent>
      <xsd:extension base="tCorrelation">
        <xsd:attribute name="pattern" type="tPattern" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="tPattern">
    <xsd:restriction base="xsd:string">
```

```
      <xsd:enumeration value="request" />
      <xsd:enumeration value="response" />
      <xsd:enumeration value="request-response" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="fromParts" type="tFromParts" />
  <xsd:complexType name="tFromParts">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="fromPart" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="fromPart" type="tFromPart" />
  <xsd:complexType name="tFromPart">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="part" type="xsd:NCName" use="required" />
        <xsd:attribute name="toVariable" type="BPELVariableName"
          use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="toParts" type="tToParts" />
  <xsd:complexType name="tToParts">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="toPart" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="toPart" type="tToPart" />
  <xsd:complexType name="tToPart">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="part" type="xsd:NCName" use="required" />
        <xsd:attribute name="fromVariable" type="BPELVariableName"
          use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="pick" type="tPick" />
  <xsd:complexType name="tPick">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element onAlarm needs to be a Local
        Element Declaration, because there is another onAlarm element
        defined for event handlers.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element ref="onMessage" maxOccurs="unbounded" />
```

```
        <xsd:element name="onAlarm" type="tOnAlarmPick"
          minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="createInstance" type="tBoolean"
        default="no" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="onMessage" type="tOnMessage" />
<xsd:complexType name="tOnMessage">
  <xsd:complexContent>
    <xsd:extension base="tOnMsgCommon">
      <xsd:sequence>
        <xsd:group ref="activity" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tOnAlarmPick">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:group ref="forOrUntilGroup" />
        <xsd:group ref="activity" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="receive" type="tReceive" />
<xsd:complexType name="tReceive">
  <xsd:annotation>
    <xsd:documentation>
      XSD Authors: The child element correlations needs to be a
      Local Element Declaration, because there is another
      correlations element defined for the invoke activity.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element name="correlations" type="tCorrelations"
          minOccurs="0" />
        <xsd:element ref="fromParts" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="partnerLink" type="xsd:NCName"
        use="required" />
      <xsd:attribute name="portType" type="xsd:QName"
        use="optional" />
      <xsd:attribute name="operation" type="xsd:NCName"
        use="required" />
      <xsd:attribute name="variable" type="BPELVariableName"
        use="optional" />
      <xsd:attribute name="createInstance" type="tBoolean"
        default="no" />
      <xsd:attribute name="messageExchange" type="xsd:NCName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
```

```
  </xsd:complexType>
  <xsd:element name="repeatUntil" type="tRepeatUntil" />
  <xsd:complexType name="tRepeatUntil">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:group ref="activity" />
          <xsd:element ref="condition" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="reply" type="tReply" />
  <xsd:complexType name="tReply">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlations needs to be a
        Local Element Declaration, because there is another
        correlations element defined for the invoke activity.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element name="correlations" type="tCorrelations"
            minOccurs="0" />
          <xsd:element ref="toParts" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="partnerLink" type="xsd:NCName"
          use="required" />
        <xsd:attribute name="portType" type="xsd:QName"
          use="optional" />
        <xsd:attribute name="operation" type="xsd:NCName"
          use="required" />
        <xsd:attribute name="variable" type="BPELVariableName"
          use="optional" />
        <xsd:attribute name="faultName" type="xsd:QName" />
        <xsd:attribute name="messageExchange" type="xsd:NCName"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="rethrow" type="tRethrow" />
  <xsd:complexType name="tRethrow">
    <xsd:complexContent>
      <xsd:extension base="tActivity" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="scope" type="tScope" />
  <xsd:complexType name="tScope">
    <xsd:annotation>
      <xsd:documentation>
        There is no schema-level default for "exitOnStandardFault" at
        "scope". Because, it will inherit default from enclosing scope
        or process.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
```

```
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element ref="partnerLinks" minOccurs="0" />
          <xsd:element ref="messageExchanges" minOccurs="0" />
          <xsd:element ref="variables" minOccurs="0" />
          <xsd:element ref="correlationSets" minOccurs="0" />
          <xsd:element ref="faultHandlers" minOccurs="0" />
          <xsd:element ref="compensationHandler" minOccurs="0" />
          <xsd:element ref="terminationHandler" minOccurs="0" />
          <xsd:element ref="eventHandlers" minOccurs="0" />
          <xsd:group ref="activity" />
        </xsd:sequence>
        <xsd:attribute name="isolated" type="tBoolean" default="no" />
        <xsd:attribute name="exitOnStandardFault" type="tBoolean" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="compensationHandler" type="tActivityContainer">
    <xsd:annotation>
      <xsd:documentation>
        This element can contain all activities including the
        activities compensate and compensateScope.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="terminationHandler" type="tActivityContainer">
    <xsd:annotation>
      <xsd:documentation>
        This element can contain all activities including the
        activities compensate and compensateScope.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="sequence" type="tSequence" />
  <xsd:complexType name="tSequence">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:group ref="activity" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="throw" type="tThrow" />
  <xsd:complexType name="tThrow">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:attribute name="faultName" type="xsd:QName"
          use="required" />
        <xsd:attribute name="faultVariable" type="BPELVariableName" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="validate" type="tValidate" />
  <xsd:complexType name="tValidate">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:attribute name="variables" type="BPELVariableNames"
```

```
          use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="BPELVariableNames">
    <xsd:restriction>
      <xsd:simpleType>
        <xsd:list itemType="BPELVariableName" />
      </xsd:simpleType>
      <xsd:minLength value="1" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="wait" type="tWait" />
  <xsd:complexType name="tWait">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:choice>
          <xsd:element ref="for" />
          <xsd:element ref="until" />
        </xsd:choice>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="while" type="tWhile" />
  <xsd:complexType name="tWhile">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element ref="condition" />
          <xsd:group ref="activity" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tExpression" mixed="true">
    <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI" />
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:complexType name="tCondition" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="condition" type="tBoolean-expr" />
  <xsd:complexType name="tBoolean-expr" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tDuration-expr" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
    </xsd:complexContent>
  </xsd:complexType>
```

```
  <xsd:complexType name="tDeadline-expr" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="tBoolean">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="yes" />
      <xsd:enumeration value="no" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Schema for Abstract Process Common Base for WS-BPEL 2.0

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Copyright (c) OASIS Open 2006. All Rights Reserved.
-->
<xsd:schema
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd-derived="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
  targetNamespace="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
  elementFormDefault="qualified" blockDefault="#all">
  <xsd:annotation>
    <xsd:documentation>
      Schema for Abstract Process Common Base for WS-BPEL 2.0 Last
      modified date: 18th October, 2006
    </xsd:documentation>
  </xsd:annotation>
  <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="http://www.w3.org/2001/xml.xsd" />
  <xsd:element name="process" type="tProcess">
    <xsd:annotation>
      <xsd:documentation>
        This is the root element for a WS-BPEL 2.0 process.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="tProcess">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="extensions" minOccurs="0" />
          <xsd:element ref="import" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="partnerLinks" minOccurs="0" />
          <xsd:element ref="messageExchanges" minOccurs="0" />
          <xsd:element ref="variables" minOccurs="0" />
          <xsd:element ref="correlationSets" minOccurs="0" />
          <xsd:element ref="faultHandlers" minOccurs="0" />
          <xsd:element ref="eventHandlers" minOccurs="0" />
          <xsd:group ref="activity" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="targetNamespace"
```

```
          type="xsd-derived:anyURI" use="optional" />
        <xsd:attribute name="queryLanguage" type="xsd-derived:anyURI"
          default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" />
        <xsd:attribute name="expressionLanguage"
          type="xsd-derived:anyURI"
          default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0" />
        <xsd:attribute name="suppressJoinFailure" type="tBoolean"
          default="no" />
        <xsd:attribute name="exitOnStandardFault" type="tBoolean"
          default="no" />
        <xsd:attribute name="abstractProcessProfile" type="xsd:anyURI"
          use="required" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tExtensibleElements">
    <xsd:annotation>
      <xsd:documentation>
        This type is extended by other component types to allow
        elements and attributes from other namespaces to be added at
        the modeled places.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element ref="documentation" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:element name="documentation" type="tDocumentation" />
  <xsd:complexType name="tDocumentation" mixed="true">
    <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="source" type="xsd-derived:anyURI" />
    <xsd:attribute ref="xml:lang" />
  </xsd:complexType>
  <xsd:group name="activity">
    <xsd:annotation>
      <xsd:documentation>
        All standard WS-BPEL 2.0 activities in alphabetical order.
        Basic activities and structured activities. Addtional
        constraints: - rethrow activity can be used ONLY within a
        fault handler (i.e. "catch" and "catchAll" element) -
        compensate or compensateScope activity can be used ONLY within
        a fault handler, a compensation handler or a termination
        handler
      </xsd:documentation>
    </xsd:annotation>
    <xsd:choice>
      <xsd:element ref="assign" />
      <xsd:element ref="compensate" />
      <xsd:element ref="compensateScope" />
      <xsd:element ref="empty" />
      <xsd:element ref="exit" />
```

```
      <xsd:element ref="extensionActivity" />
      <xsd:element ref="flow" />
      <xsd:element ref="forEach" />
      <xsd:element ref="if" />
      <xsd:element ref="invoke" />
      <xsd:element ref="pick" />
      <xsd:element ref="receive" />
      <xsd:element ref="repeatUntil" />
      <xsd:element ref="reply" />
      <xsd:element ref="rethrow" />
      <xsd:element ref="scope" />
      <xsd:element ref="sequence" />
      <xsd:element ref="throw" />
      <xsd:element ref="validate" />
      <xsd:element ref="wait" />
      <xsd:element ref="while" />
      <xsd:element ref="opaqueActivity" />
    </xsd:choice>
  </xsd:group>
  <xsd:element name="extensions" type="tExtensions" />
  <xsd:complexType name="tExtensions">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="extension" minOccurs="0"
            maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="extension" type="tExtension" />
  <xsd:complexType name="tExtension">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="namespace" type="xsd-derived:anyURI"
          use="optional" />
        <xsd:attribute name="mustUnderstand" type="tBoolean"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="import" type="tImport" />
  <xsd:complexType name="tImport">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="namespace" type="xsd-derived:anyURI"
          use="optional" />
        <xsd:attribute name="location" type="xsd-derived:anyURI"
          use="optional" />
        <xsd:attribute name="importType" type="xsd-derived:anyURI"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="partnerLinks" type="tPartnerLinks" />
  <xsd:complexType name="tPartnerLinks">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
```

```
      <xsd:sequence>
        <xsd:element ref="partnerLink" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="partnerLink" type="tPartnerLink" />
<xsd:complexType name="tPartnerLink">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="name" type="xsd-derived:NCName"
        use="optional" />
      <xsd:attribute name="partnerLinkType" type="xsd-derived:QName"
        use="optional" />
      <xsd:attribute name="myRole" type="xsd-derived:NCName" />
      <xsd:attribute name="partnerRole" type="xsd-derived:NCName" />
      <xsd:attribute name="initializePartnerRole" type="tBoolean" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="messageExchanges" type="tMessageExchanges" />
<xsd:complexType name="tMessageExchanges">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="messageExchange" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="messageExchange" type="tMessageExchange" />
<xsd:complexType name="tMessageExchange">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="name" type="xsd-derived:NCName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="variables" type="tVariables" />
<xsd:complexType name="tVariables">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="variable" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="variable" type="tVariable" />
<xsd:complexType name="tVariable">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:group ref="fromGroup" minOccurs="0" />
```

```
      </xsd:sequence>
      <xsd:attribute name="name" type="BPELVariableName"
        use="optional" />
      <xsd:attribute name="messageType" type="xsd-derived:QName"
        use="optional" />
      <xsd:attribute name="type" type="xsd-derived:QName"
        use="optional" />
      <xsd:attribute name="element" type="xsd-derived:QName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="BPELVariableName">
  <xsd:union>
    <xsd:simpleType>
      <xsd:restriction base="xsd:NCName">
        <xsd:pattern value="[^\.]+" />
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="##opaque" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
<xsd:element name="correlationSets" type="tCorrelationSets" />
<xsd:complexType name="tCorrelationSets">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="correlationSet" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="correlationSet" type="tCorrelationSet" />
<xsd:complexType name="tCorrelationSet">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="properties" type="QNames" use="optional" />
      <xsd:attribute name="name" type="xsd-derived:NCName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="QNames">
  <xsd:restriction>
    <xsd:simpleType>
      <xsd:list itemType="xsd-derived:QName" />
    </xsd:simpleType>
    <xsd:minLength value="1" />
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="faultHandlers" type="tFaultHandlers" />
<xsd:complexType name="tFaultHandlers">
  <xsd:complexContent>
```

```
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="catch" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="catchAll" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="catch" type="tCatch">
    <xsd:annotation>
      <xsd:documentation>
        This element can contain all activities including the
        activities compensate, compensateScope and rethrow.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="tCatch">
    <xsd:complexContent>
      <xsd:extension base="tActivityContainer">
        <xsd:attribute name="faultName" type="xsd-derived:QName" />
        <xsd:attribute name="faultVariable" type="BPELVariableName" />
        <xsd:attribute name="faultMessageType"
          type="xsd-derived:QName" />
        <xsd:attribute name="faultElement" type="xsd-derived:QName" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="catchAll" type="tActivityContainer">
    <xsd:annotation>
      <xsd:documentation>
        This element can contain all activities including the
        activities compensate, compensateScope and rethrow.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="tActivityContainer">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:group ref="activity" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="eventHandlers" type="tEventHandlers" />
  <xsd:complexType name="tEventHandlers">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element onAlarm needs to be a Local
        Element Declaration, because there is another onAlarm element
        defined for the pick activity.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="onEvent" minOccurs="0"
```

```
            maxOccurs="unbounded" />
          <xsd:element name="onAlarm" type="tOnAlarmEvent"
            minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="onEvent" type="tOnEvent" />
  <xsd:complexType name="tOnEvent">
    <xsd:complexContent>
      <xsd:extension base="tOnMsgCommon">
        <xsd:sequence>
          <xsd:element ref="scope" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="messageType" type="xsd-derived:QName"
          use="optional" />
        <xsd:attribute name="element" type="xsd-derived:QName"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tOnMsgCommon">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlations needs to be a
        Local Element Declaration, because there is another
        correlations element defined for the invoke activity.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element name="correlations" type="tCorrelations"
            minOccurs="0" />
          <xsd:element ref="fromParts" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="partnerLink" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="portType" type="xsd-derived:QName"
          use="optional" />
        <xsd:attribute name="operation" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="messageExchange"
          type="xsd-derived:NCName" use="optional" />
        <xsd:attribute name="variable" type="BPELVariableName"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tCorrelations">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlation needs to be a Local
        Element Declaration, because there is another correlation
        element defined for the invoke activity.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
```

```
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element name="correlation" type="tCorrelation"
            minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tCorrelation">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:attribute name="set" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="initiate" type="tInitiate" default="no" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="tInitiate">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="yes" />
      <xsd:enumeration value="join" />
      <xsd:enumeration value="no" />
      <xsd:enumeration value="##opaque" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="tOnAlarmEvent">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:group ref="forOrUntilGroup" minOccurs="0" />
          <xsd:element ref="repeatEvery" minOccurs="0" />
          <xsd:element ref="scope" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:group name="forOrUntilGroup">
    <xsd:choice>
      <xsd:element ref="for" minOccurs="0" />
      <xsd:element ref="until" minOccurs="0" />
    </xsd:choice>
  </xsd:group>
  <xsd:element name="for" type="tDuration-expr" />
  <xsd:element name="until" type="tDeadline-expr" />
  <xsd:element name="repeatEvery" type="tDuration-expr" />
  <xsd:complexType name="tActivity">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="targets" minOccurs="0" />
          <xsd:element ref="sources" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd-derived:NCName" />
        <xsd:attribute name="suppressJoinFailure" type="tBoolean"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

```
<xsd:element name="targets" type="tTargets" />
<xsd:complexType name="tTargets">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="joinCondition" minOccurs="0" />
        <xsd:element ref="target" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="joinCondition" type="tCondition" />
<xsd:element name="target" type="tTarget" />
<xsd:complexType name="tTarget">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="linkName" type="xsd-derived:NCName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="sources" type="tSources" />
<xsd:complexType name="tSources">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="source" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="source" type="tSource" />
<xsd:complexType name="tSource">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="transitionCondition" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="linkName" type="xsd-derived:NCName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="transitionCondition" type="tCondition" />
<xsd:element name="assign" type="tAssign" />
<xsd:complexType name="tAssign">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element ref="copy" minOccurs="0" />
          <xsd:element ref="extensionAssignOperation" minOccurs="0" />
        </xsd:choice>
      </xsd:sequence>
      <xsd:attribute name="validate" type="tBoolean" use="optional"
        default="no" />
```

```
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="copy" type="tCopy" />
<xsd:complexType name="tCopy">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:group ref="fromGroup" minOccurs="0" />
        <xsd:element ref="to" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="keepSrcElementName" type="tBoolean"
        use="optional" default="no" />
      <xsd:attribute name="ignoreMissingFromData" type="tBoolean"
        use="optional" default="no" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:group name="fromGroup">
  <xsd:choice>
    <xsd:element ref="opaqueFrom" />
    <xsd:element ref="from" />
  </xsd:choice>
</xsd:group>
<xsd:element name="opaqueFrom" type="tExtensibleElements" />
<xsd:element name="from" type="tFrom" />
<xsd:complexType name="tFrom" mixed="true">
  <xsd:sequence>
    <xsd:element ref="documentation" minOccurs="0"
      maxOccurs="unbounded" />
    <xsd:any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded" />
    <xsd:choice minOccurs="0">
      <xsd:element ref="literal" minOccurs="0" />
      <xsd:element ref="query" minOccurs="0" />
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="expressionLanguage"
    type="xsd-derived:anyURI" />
  <xsd:attribute name="variable" type="BPELVariableName" />
  <xsd:attribute name="part" type="xsd-derived:NCName" />
  <xsd:attribute name="property" type="xsd-derived:QName" />
  <xsd:attribute name="partnerLink" type="xsd-derived:NCName" />
  <xsd:attribute name="endpointReference" type="tRoles" />
  <xsd:attribute name="opaque" type="xsd-derived:tOpaqueBoolean" />
  <xsd:anyAttribute namespace="##other" processContents="lax" />
</xsd:complexType>
<xsd:element name="literal" type="tLiteral" />
<xsd:complexType name="tLiteral" mixed="true">
  <xsd:sequence>
    <xsd:any namespace="##any" processContents="lax" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="query" type="tQuery" />
<xsd:complexType name="tQuery" mixed="true">
  <xsd:sequence>
    <xsd:any processContents="lax" minOccurs="0"
      maxOccurs="unbounded" />
```

```
    </xsd:sequence>
    <xsd:attribute name="queryLanguage" type="xsd-derived:anyURI" />
    <xsd:attribute name="opaque" type="xsd-derived:tOpaqueBoolean" />
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:simpleType name="tRoles">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="myRole" />
      <xsd:enumeration value="partnerRole" />
      <xsd:enumeration value="##opaque" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="to" type="tTo" />
  <xsd:complexType name="tTo" mixed="true">
    <xsd:sequence>
      <xsd:element ref="documentation" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element ref="query" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="opaque" type="xsd-derived:tOpaqueBoolean" />
    <xsd:attribute name="expressionLanguage"
      type="xsd-derived:anyURI" />
    <xsd:attribute name="variable" type="BPELVariableName" />
    <xsd:attribute name="part" type="xsd-derived:NCName" />
    <xsd:attribute name="property" type="xsd-derived:QName" />
    <xsd:attribute name="partnerLink" type="xsd-derived:NCName" />
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:element name="extensionAssignOperation"
    type="tExtensionAssignOperation" />
  <xsd:complexType name="tExtensionAssignOperation">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="compensate" type="tCompensate" />
  <xsd:complexType name="tCompensate">
    <xsd:complexContent>
      <xsd:extension base="tActivity" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="compensateScope" type="tCompensateScope" />
  <xsd:complexType name="tCompensateScope">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:attribute name="target" type="xsd-derived:NCName"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="empty" type="tEmpty" />
  <xsd:complexType name="tEmpty">
    <xsd:complexContent>
      <xsd:extension base="tActivity" />
    </xsd:complexContent>
  </xsd:complexType>
```

```
<xsd:element name="exit" type="tExit" />
<xsd:complexType name="tExit">
  <xsd:complexContent>
    <xsd:extension base="tActivity" />
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="extensionActivity" type="tExtensionActivity" />
<xsd:complexType name="tExtensionActivity">
  <xsd:sequence>
    <xsd:any namespace="##other" processContents="lax" />
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="flow" type="tFlow" />
<xsd:complexType name="tFlow">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element ref="links" minOccurs="0" />
        <xsd:group ref="activity" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="links" type="tLinks" />
<xsd:complexType name="tLinks">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="link" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="link" type="tLink" />
<xsd:complexType name="tLink">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="name" type="xsd-derived:NCName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="forEach" type="tForEach" />
<xsd:complexType name="tForEach">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element ref="startCounterValue" minOccurs="0" />
        <xsd:element ref="finalCounterValue" minOccurs="0" />
        <xsd:element ref="completionCondition" minOccurs="0" />
        <xsd:element ref="scope" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="counterName" type="BPELVariableName"
        use="optional" />
      <xsd:attribute name="parallel" type="tBoolean" use="optional" />
    </xsd:extension>
  </xsd:complexContent>
```

```
  </xsd:complexType>
  <xsd:element name="startCounterValue" type="tExpression" />
  <xsd:element name="finalCounterValue" type="tExpression" />
  <xsd:element name="completionCondition" type="tCompletionCondition" />
  <xsd:complexType name="tCompletionCondition">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="branches" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="branches" type="tBranches" />
  <xsd:complexType name="tBranches">
    <xsd:complexContent>
      <xsd:extension base="tExpression">
        <xsd:attribute name="successfulBranchesOnly" type="tBoolean"
          default="no" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="if" type="tIf" />
  <xsd:complexType name="tIf">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element ref="condition" minOccurs="0" />
          <xsd:group ref="activity" minOccurs="0" />
          <xsd:element ref="elseif" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="else" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="elseif" type="tElseif" />
  <xsd:complexType name="tElseif">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="condition" minOccurs="0" />
          <xsd:group ref="activity" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="else" type="tActivityContainer" />
  <xsd:element name="invoke" type="tInvoke" />
  <xsd:complexType name="tInvoke">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlations needs to be a
        Local Element Declaration, because there is another
        correlations element defined for the non-invoke activities.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
```

```
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element name="correlations"
            type="tCorrelationsWithPattern" minOccurs="0" />
          <xsd:element ref="catch" minOccurs="0"
            maxOccurs="unbounded" />
          <xsd:element ref="catchAll" minOccurs="0" />
          <xsd:element ref="compensationHandler" minOccurs="0" />
          <xsd:element ref="toParts" minOccurs="0" />
          <xsd:element ref="fromParts" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="partnerLink" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="portType" type="xsd-derived:QName"
          use="optional" />
        <xsd:attribute name="operation" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="inputVariable" type="BPELVariableName"
          use="optional" />
        <xsd:attribute name="outputVariable" type="BPELVariableName"
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tCorrelationsWithPattern">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlation needs to be a Local
        Element Declaration, because there is another correlation
        element defined for the non-invoke activities.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:element name="correlation"
            type="tCorrelationWithPattern" minOccurs="0"
            maxOccurs="unbounded" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tCorrelationWithPattern">
    <xsd:complexContent>
      <xsd:extension base="tCorrelation">
        <xsd:attribute name="pattern" type="tPattern" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="tPattern">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="request" />
      <xsd:enumeration value="response" />
      <xsd:enumeration value="request-response" />
      <xsd:enumeration value="##opaque" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="fromParts" type="tFromParts" />
```

```
<xsd:complexType name="tFromParts">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="fromPart" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="fromPart" type="tFromPart" />
<xsd:complexType name="tFromPart">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="part" type="xsd-derived:NCName"
        use="optional" />
      <xsd:attribute name="toVariable" type="BPELVariableName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="toParts" type="tToParts" />
<xsd:complexType name="tToParts">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:sequence>
        <xsd:element ref="toPart" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="toPart" type="tToPart" />
<xsd:complexType name="tToPart">
  <xsd:complexContent>
    <xsd:extension base="tExtensibleElements">
      <xsd:attribute name="part" type="xsd-derived:NCName"
        use="optional" />
      <xsd:attribute name="fromVariable" type="BPELVariableName"
        use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="pick" type="tPick" />
<xsd:complexType name="tPick">
  <xsd:annotation>
    <xsd:documentation>
      XSD Authors: The child element onAlarm needs to be a Local
      Element Declaration, because there is another onAlarm element
      defined for event handlers.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element ref="onMessage" minOccurs="0"
          maxOccurs="unbounded" />
        <xsd:element name="onAlarm" type="tOnAlarmPick"
```

```
            minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="createInstance" type="tBoolean"
          default="no" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="onMessage" type="tOnMessage" />
  <xsd:complexType name="tOnMessage">
    <xsd:complexContent>
      <xsd:extension base="tOnMsgCommon">
        <xsd:sequence>
          <xsd:group ref="activity" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tOnAlarmPick">
    <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
        <xsd:sequence>
          <xsd:group ref="forOrUntilGroup" minOccurs="0" />
          <xsd:group ref="activity" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="receive" type="tReceive" />
  <xsd:complexType name="tReceive">
    <xsd:annotation>
      <xsd:documentation>
        XSD Authors: The child element correlations needs to be a
        Local Element Declaration, because there is another
        correlations element defined for the invoke activity.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element name="correlations" type="tCorrelations"
            minOccurs="0" />
          <xsd:element ref="fromParts" minOccurs="0" />
        </xsd:sequence>
        <xsd:attribute name="partnerLink" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="portType" type="xsd-derived:QName"
          use="optional" />
        <xsd:attribute name="operation" type="xsd-derived:NCName"
          use="optional" />
        <xsd:attribute name="variable" type="BPELVariableName"
          use="optional" />
        <xsd:attribute name="createInstance" type="tBoolean"
          default="no" />
        <xsd:attribute name="messageExchange"
          type="xsd-derived:NCName" use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

```
<xsd:element name="repeatUntil" type="tRepeatUntil" />
<xsd:complexType name="tRepeatUntil">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:group ref="activity" minOccurs="0" />
        <xsd:element ref="condition" minOccurs="0" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="reply" type="tReply" />
<xsd:complexType name="tReply">
  <xsd:annotation>
    <xsd:documentation>
      XSD Authors: The child element correlations needs to be a
      Local Element Declaration, because there is another
      correlations element defined for the invoke activity.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:element name="correlations" type="tCorrelations"
          minOccurs="0" />
        <xsd:element ref="toParts" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="partnerLink" type="xsd-derived:NCName"
        use="optional" />
      <xsd:attribute name="portType" type="xsd-derived:QName"
        use="optional" />
      <xsd:attribute name="operation" type="xsd-derived:NCName"
        use="optional" />
      <xsd:attribute name="variable" type="BPELVariableName"
        use="optional" />
      <xsd:attribute name="faultName" type="xsd-derived:QName" />
      <xsd:attribute name="messageExchange"
        type="xsd-derived:NCName" use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="rethrow" type="tRethrow" />
<xsd:complexType name="tRethrow">
  <xsd:complexContent>
    <xsd:extension base="tActivity" />
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="scope" type="tScope" />
<xsd:complexType name="tScope">
  <xsd:annotation>
    <xsd:documentation>
      There is no schema-level default for "exitOnStandardFault" at
      "scope". Because, it will inherit default from enclosing scope
      or process.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="tActivity">
```

```
      <xsd:sequence>
        <xsd:element ref="partnerLinks" minOccurs="0" />
        <xsd:element ref="messageExchanges" minOccurs="0" />
        <xsd:element ref="variables" minOccurs="0" />
        <xsd:element ref="correlationSets" minOccurs="0" />
        <xsd:element ref="faultHandlers" minOccurs="0" />
        <xsd:element ref="compensationHandler" minOccurs="0" />
        <xsd:element ref="terminationHandler" minOccurs="0" />
        <xsd:element ref="eventHandlers" minOccurs="0" />
        <xsd:group ref="activity" minOccurs="0" />
      </xsd:sequence>
      <xsd:attribute name="isolated" type="tBoolean" default="no" />
      <xsd:attribute name="exitOnStandardFault" type="tBoolean" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="compensationHandler" type="tActivityContainer">
  <xsd:annotation>
    <xsd:documentation>
      This element can contain all activities including the
      activities compensate and compensateScope.
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="terminationHandler" type="tActivityContainer">
  <xsd:annotation>
    <xsd:documentation>
      This element can contain all activities including the
      activities compensate and compensateScope.
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:element name="sequence" type="tSequence" />
<xsd:complexType name="tSequence">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:sequence>
        <xsd:group ref="activity" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="throw" type="tThrow" />
<xsd:complexType name="tThrow">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:attribute name="faultName" type="xsd-derived:QName"
        use="optional" />
      <xsd:attribute name="faultVariable" type="BPELVariableName" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="validate" type="tValidate" />
<xsd:complexType name="tValidate">
  <xsd:complexContent>
    <xsd:extension base="tActivity">
      <xsd:attribute name="variables" type="BPELVariableNames"
```

```
          use="optional" />
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="BPELVariableNames">
    <xsd:restriction>
      <xsd:simpleType>
        <xsd:list itemType="BPELVariableName" />
      </xsd:simpleType>
      <xsd:minLength value="1" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="wait" type="tWait" />
  <xsd:complexType name="tWait">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:choice>
          <xsd:element ref="for" minOccurs="0" />
          <xsd:element ref="until" minOccurs="0" />
        </xsd:choice>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="while" type="tWhile" />
  <xsd:complexType name="tWhile">
    <xsd:complexContent>
      <xsd:extension base="tActivity">
        <xsd:sequence>
          <xsd:element ref="condition" minOccurs="0" />
          <xsd:group ref="activity" minOccurs="0" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tExpression" mixed="true">
    <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage"
      type="xsd-derived:anyURI" />
    <xsd:attribute name="opaque" type="xsd-derived:tOpaqueBoolean" />
    <xsd:anyAttribute namespace="##other" processContents="lax" />
  </xsd:complexType>
  <xsd:complexType name="tCondition" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="condition" type="tBoolean-expr" />
  <xsd:complexType name="tBoolean-expr" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tDuration-expr" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
```

```
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="tDeadline-expr" mixed="true">
    <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression" />
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="tBoolean">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="yes" />
      <xsd:enumeration value="no" />
      <xsd:enumeration value="##opaque" />
    </xsd:restriction>
  </xsd:simpleType>
  <!-- SCHEMA NOTE: new types and element introduced for Abstract WS-BPEL -->
  <xsd:simpleType name="tOpaqueStr">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="##opaque" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="QName">
    <xsd:union memberTypes="xsd:QName tOpaqueStr" />
  </xsd:simpleType>
  <xsd:simpleType name="NCName">
    <xsd:union memberTypes="xsd:NCName tOpaqueStr" />
  </xsd:simpleType>
  <xsd:simpleType name="anyURI">
    <xsd:union memberTypes="xsd:anyURI tOpaqueStr" />
  </xsd:simpleType>
  <xsd:simpleType name="tOpaqueBoolean">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="yes" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:element name="opaqueActivity" type="tOpaqueActivity" />
  <xsd:complexType name="tOpaqueActivity">
    <xsd:complexContent>
      <xsd:extension base="tActivity" />
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Partner Link Type Schema for WS-BPEL 2.0

```
<?xml version='1.0' encoding="UTF-8"?>
<!--
  Copyright (c) OASIS Open 2003-2006. All Rights Reserved.
-->
<xsd:schema targetNamespace="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
      xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
      blockDefault="#all">

  <xsd:annotation>
    <xsd:documentation>
      Partner Link Type Schema for WS-BPEL 2.0
      Last modified date: 17th August, 2006
```

```
    </xsd:documentation>
  </xsd:annotation>

  <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
              schemaLocation="http://www.w3.org/2001/xml.xsd"/>

  <xsd:element name="partnerLinkType" type="plnk:tPartnerLinkType"/>
  <xsd:complexType name="tPartnerLinkType">
    <xsd:complexContent>
      <xsd:extension base="plnk:tExtensibleElements">
        <xsd:sequence>
          <xsd:element ref="plnk:role" minOccurs="1" maxOccurs="2"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:NCName" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="tExtensibleElements">
    <xsd:annotation>
      <xsd:documentation>
        This type is extended by other component types to allow elements and
        attributes from other namespaces to be added at the modeled places.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element ref="plnk:documentation" minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
               maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:complexType>

  <xsd:element name="documentation">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="source" type="xsd:anyURI"/>
      <xsd:attribute ref="xml:lang"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="role" type="plnk:tRole"/>
  <xsd:complexType name="tRole">
    <xsd:complexContent>
      <xsd:extension base="plnk:tExtensibleElements">
        <xsd:attribute name="name" type="xsd:NCName" use="required"/>
        <xsd:attribute name="portType" type="xsd:QName" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

</xsd:schema>
```

Variable Properties Schema for WS-BPEL 2.0

```xml
<?xml version='1.0' encoding="UTF-8"?>
<!--
  Copyright (c) OASIS Open 2003-2006. All Rights Reserved.
-->
<xsd:schema targetNamespace="http://docs.oasis-open.org/wsbpel/2.0/varprop"
      xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
      blockDefault="#all">

  <xsd:annotation>
    <xsd:documentation>
      Variable Properties Schema for WS-BPEL 2.0
      Last modified date: 22th August, 2006
    </xsd:documentation>
  </xsd:annotation>

  <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
              schemaLocation="http://www.w3.org/2001/xml.xsd"/>

  <xsd:element name="property">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="vprop:tExtensibleElements">
          <xsd:attribute name="name" type="xsd:NCName" use="required"/>
          <xsd:attribute name="type" type="xsd:QName"/>
          <xsd:attribute name="element" type="xsd:QName"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="tExtensibleElements">
    <xsd:annotation>
      <xsd:documentation>
        This type is extended by other component types to allow elements and
        attributes from other namespaces to be added at the modeled places.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element ref="vprop:documentation" minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
               maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:complexType>

  <xsd:element name="documentation">
    <xsd:complexType mixed="true">
      <xsd:sequence>
        <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="source" type="xsd:anyURI"/>
      <xsd:attribute ref="xml:lang"/>
    </xsd:complexType>
  </xsd:element>
```

```
  <xsd:element name="propertyAlias">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="vprop:tExtensibleElements">
          <xsd:sequence>
            <xsd:element ref="vprop:query" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="propertyName" type="xsd:QName"
                         use="required"/>
          <xsd:attribute name="messageType" type="xsd:QName"/>
          <xsd:attribute name="part" type="xsd:NCName"/>
          <xsd:attribute name="type" type="xsd:QName"/>
          <xsd:attribute name="element" type="xsd:QName"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="query" type="vprop:tQuery"/>
  <xsd:complexType name="tQuery" mixed="true">
    <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="queryLanguage" type="xsd:anyURI"
                   default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
  </xsd:complexType>

</xsd:schema>
```

Service Reference Schema for WS-BPEL 2.0

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Copyright (c) OASIS Open 2006. All Rights Reserved.
-->
<xsd:schema targetNamespace="http://docs.oasis-
open.org/wsbpel/2.0/serviceref"
      xmlns:sref="http://docs.oasis-open.org/wsbpel/2.0/serviceref"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
      blockDefault="#all">

  <xsd:annotation>
    <xsd:documentation>
      Service Reference Schema for WS-BPEL 2.0
      Last modified date: 17th August, 2006
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="service-ref" type="sref:ServiceRefType">
    <xsd:annotation>
      <xsd:documentation>
        This element can be used within a from-spec.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="ServiceRefType">
```

```
    <xsd:annotation>
      <xsd:documentation>
        This type definition is for service reference container.
        This container is used as envelope to wrap around the actual endpoint
        reference value, when a BPEL process interacts the endpoint reference
        of a partnerLink. It provides pluggability of different versions of
        service referencing schemes being used within a BPEL program. The
        design pattern here is similar to those of expression language.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:any namespace="##other" processContents="lax"/>
    </xsd:sequence>
    <xsd:attribute name="reference-scheme" type="xsd:anyURI"/>
  </xsd:complexType>

</xsd:schema>
```

Appendix F. References

1. Normative References

[BPEL4WS 1.1] BEA, IBM, Microsoft, SAP and Siebel, “Business Process Execution Language for Web Services Version 1.1”, S. Thatte, et al., May 2003. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

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

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

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

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

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

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

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

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

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

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

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

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

2. Non-Normative References

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

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

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

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

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

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

Appendix G. Committee Members (Non-Normative)

The following individuals are current members of the committee:

- Alastair Green, Choreology Ltd
- Alejandro Guizar, Redhat, formerly JBoss Inc
- Alex Yiu, Oracle
- Alexandre Alves, BEA Systems, Inc.
- Allen Brookes, Rogue Wave Software
- Ashish Agrawal, Adobe Systems
- Assaf Arkin, Intalio, Inc.
- Axel Martens, IBM
- Balinder Malhi, Microsoft Corporation
- Bernd Eckenfels, Seeburger, AG
- Canyang Liu, SAP AG
- Charles Fenton, Sterling Commerce
- Charlton Barreto, Adobe Systems
- Christopher Keller, Active Endpoints, Inc.
- Danny van der Rijn, TIBCO Software Inc.
- Dennis Curry, EDS
- Diane Jordan, IBM
- Dieter Koenig, IBM
- Dulipala Jagannadham, Hewlett-Packard
- Fabienne Marquardt, IBM
- Fang Gu, Changfeng Open Standards Platform Software Alliance
- Francisco Curbera, IBM
- Frank Leymann, IBM
- Frank Ryan, Active Endpoints, Inc.
- Greg Carter, Metastorm, Inc.
- Hadrian Zbarcea, IONA Technologies
- Harvey Reed, Mitre Corporation
- Hyun Jung, Korean National Computerization Agency
- Innamuri venubabu, CrimsonLogic Pte Ltd
- Ivana Trickovic, SAP AG
- J. Darrel Thomas, EDS
- James Pasley, Cape Clear Software
- Jianguang Geng, Changfeng Open Standards Platform Software Alliance
- Jim Alateras, IPsphere Forum
- John Evdemon, Microsoft Corporation
- Kent Horng, webMethods, Inc.
- Kristofer Agren, Individual
- Layna Fischer, Workflow Management Coalition (WfMC)
- Mark David, Gensym Corporation
- Mark Ford, Active Endpoints, Inc.
- Mark Little, Redhat, formerly JBoss Inc.

- Martin Chapman, Oracle
- Michael Kleinhenz, The OpenDocument Foundation, Inc.
- Mike Marin, IBM, formerly FileNet Corporation
- Monica J. Martin, Sun Microsystems
- Muruga Chinnananchi, Oracle
- Nickolaos Kavantzas, Oracle
- Nitin Raut, IBM
- Nobuyuki Sambuichi, Hitachi Systems & Services, Ltd.
- Peter Furniss, Choreology Ltd
- Prasad Yendluri, webMethods, Inc.
- Rakesh Saha, Oracle
- Ralph Stout, iWay Software
- Rania Khalaf, IBM
- Ricardo Jimenez-Peris, Individual
- Rob Bartel, Corel Corporation
- Rob Williams, Concurrence, Inc.
- Ron Ten-Hove, Sun Microsystems
- Sally St. Amand, Individual
- Satish Thatte, Microsoft Corporation
- Simon Moser, IBM
- Subramanian Hariharan, Oracle
- Sumeet Malhotra, Unisys Corporation
- Takatoshi Kitano, NEC Corporation
- Thomas Erl, SOA Systems Inc.
- Thomas Schulze, IBM
- Venkatraman Balasubramanian, Individual
- Vinkesh Mehta, Deloitte Consulting LLP
- Willemde Pater, Oracle
- William Barnhill, Booz Allen Hamilton
- Wolfgang Dostal, IBM
- Wu Chou, Avaya, Inc.
- Yin-Leng Husband, Hewlett-Packard

The following individuals were previously members of the committee:

- Ajay Gummadi, Individual
- Alex Chan, Cisco Systems, Inc.
- Andrew Francis, Individual
- Andrew Pugsley, Hewlett-Packard
- Aniruddha Thakur, Oracle
- Anthony Roby, Accenture
- Art Machado, PeopleSoft
- Arun Candadai, Individual
- Ashok Anand, BAHWAN CYBERTEK INC
- B.J Fesq, Individual
- Bala Kamallakharan, Cap Gemini Ernst & Young
- Ben Bloch, Systinet
- Bill Flood, Sybase

- Bill Pope, Individual
- Bimal Mehta, Microsoft Corporation
- Bob Schmidt, Microsoft Corporation
- Brian Carroll, Serena
- Chad Kulesa, SPS Commerce
- Christopher Kurt, Microsoft Corporation
- Chunbo Huang, BEA Systems, Inc.
- Claus von Riegen, SAP
- Daniel Dominguez, Parasoft
- Darran Rolls, Sun Microsystems
- Dave Bettin, Attachmate
- Dave Chappell, Sonic Software
- David Bolene, Individual
- David Burdett, CommerceOne
- David Hayes, OpenStorm Software, Inc.
- David Ingham, Arjuna Technologies Limited
- David Webber, Individual
- Debra Kellington, Convergys
- Derick Townsend, OpenStorm Software, Inc.
- Dieter Roller, IBM
- Donald Steiner, WebV2, Inc.
- Doug Knowles, Novell
- Edwin Khodabakchian, Oracle
- Eunju Kim, Korean National Computerization Agency
- Fred Carter, AmberPoint
- Fred Cummins, EDS
- Ganesh Vednere, Cap Gemini Ernst & Young
- Genadi Genov, Seeburger, AG
- George Brown, Intel
- Glenn Mi, Oracle
- Gloria Vargas, Reuters
- Goran Olsson, Oracle
- Goutham Sukumar, Microsoft Corporation
- Greg Ritzinger, Novell
- Hedy Alban, Individual
- Heidi Buelow, Rogue Wave Software
- Howard Smith, Business Process Management Initiative
- James Rust, CTO and VP Strategy
- Jean-Luc Giraud, Axway software
- Jeff Mischkinsky, Oracle
- Jim Clune, Parasoft
- Jog Raj, Popkin Software & Systems, Inc.
- John Parkinson, Cap Gemini Ernst & Young
- John Wunder, Lockheed Martin
- John Yunker, Individual
- Jon Pyke, Workflow Management Coalition (WfMC)
- Justin Brunt, Staffware plc
- Keith Swenson, Fujitsu

- Kelvin Lawrence, IBM
- Ken Pugsley, PeopleSoft
- Kenji Nagahashi, Fujitsu
- Kent Below, IBM
- Kenwood Tsai, Documentum
- Kevin Hein, Teamplate
- Kireet Reddy, Oracle
- Lalitha Prakash, BAHWAN CYBERTEK INC
- Linda DeMichiel, Sun Microsystems
- Maciej Szefler, FiveSight Technologies
- Manoj Das, Oracle
- Marc-Thomas Schmidt, IBM
- Martin Smith, US Department of Homeland Security
- Martin Owen, Popkin Software & Systems, Inc.
- Matthew Pryor, Business Process Management Initiative (BP...
- Melanie Kudela, Uniform Code Council, Inc.
- Michael DeBellis, Fujitsu
- Michael Rowley, BEA Systems, Inc.
- Michael Winters, IBM
- Mike Blevins, BEA Systems, Inc.
- Mike Gilger, Workflow Management Coalition (WfMC)
- Muthu Ramadoss, BAHWAN CYBERTEK INC
- Neelakantan Kartha, Sterling Commerce
- Parijat Sinha, Convergys
- Patrick Hogan, Individual
- Paul Lipton, Computer Associates
- Pete Wenzel, SeeBeyond Technology Corporation
- Phil Rossomando, Unisys Corporation
- Philip Lee, Business Process Management Initiative (BP...
- Pinaki Shah, E2Open
- Rajaraman Sowmya, BAHWAN CYBERTEK INC
- Rajesh Manglani, Uniform Code Council, Inc.
- Rajesh Pradhan, Iopsis Software
- Ram Jeyaraman, Sun Microsystems
- Ran Tamir, BMC Software
- Randall Anderson, Macgregor
- Ravi Akireddy, Individual
- Richard Katz, Individual
- Robert Haugen, Choreology Ltd
- Robert Carpenter, Intel
- Roshan Punnoose, Booz Allen Hamilton
- Ryan Cairns, OpenStorm Software, Inc.
- Samih Fadli, Momentum SI
- Sanjeev Kumar, Individual
- Sazi Temel, BEA Systems, Inc.
- Scott Hinkelman, IBM
- Scott Tattrie, Teamplate
- Scott Woodgate, Microsoft Corporation

- Sid Askary, Individual
- Srinivas Padmanabhuni, Infosys Technologies
- Stephen White, IBM
- Steve Brown, Metastorm
- Steve Ross-Talbot, Enigmatec Corporation Ltd
- Stuart Wheater, Arjuna Technologies Limited
- Subhra Bose, Reuters
- Sundari Revanur, Oracle
- Sun-Ho Kim, Individual
- Terry Bjornsen, Booz Allen Hamilton
- Tim Moses, Entrust
- Tony Andrews, Microsoft Corporation
- Tony Fletcher, Choreology Ltd
- Ugo Corda, SeeBeyond Technology Corporation
- Van Wiles, BMC Software
- Vaughn Bullard, AmberPoint
- Vishwanath Shenoy, Infosys Technologies
- William Vambenepe, Hewlett-Packard
- Yanming Li, France Telecom
- Yaron Goland, BEA Systems, Inc.
- Yoko Seki, Hitachi
- Yuji Sakata, Individual
- Yuzo Fujishima, NEC Corporation
- Ziyang Duan, Reuters