



Business Process Execution Language for Web Services, Version 1.0

31 July 2002

This version:

<http://www.ibm.com/developerworks/library/ws-bpel/> (available in [PDF](#))

Authors:

Francisco Curbera, IBM [<curbera@us.ibm.com>](mailto:curbera@us.ibm.com)
Yaron Goland, BEA Systems [<ygoland@bea.com>](mailto:ygoland@bea.com)
Johannes Klein, Microsoft [<joklein@microsoft.com>](mailto:joklein@microsoft.com)
Frank Leymann, IBM [<LEY1@de.ibm.com>](mailto:LEY1@de.ibm.com)
Dieter Roller, IBM [<ROL@de.ibm.com>](mailto:ROL@de.ibm.com)
Satish Thatte, Microsoft [<satisht@microsoft.com>](mailto:satisht@microsoft.com)
Sanjiva Weerawarana, IBM [<sanjiva@us.ibm.com>](mailto:sanjiva@us.ibm.com)

Editor:

Satish Thatte, Microsoft [<satisht@microsoft.com>](mailto:satisht@microsoft.com)

Copyright© Copyright© 2002 [BEA Systems](#), [International Business Machines Corporation](#), [Microsoft Corporation, Inc.](#) All rights reserved. The presentation, distribution, or other dissemination of the information contained in this specification is not a license, either expressly or impliedly, to any intellectual property owned or controlled by BEA or IBM or Microsoft and/or any other third party. BEA, IBM, Microsoft, and/or any other third party may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you any license to BEA's or IBM's or Microsoft's or any other third party's patents, trademarks, copyrights, or other intellectual property. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. This specification and the information contained herein is provided on an "AS IS" basis and to the maximum extent permitted by applicable law, BEA, IBM and Microsoft provides the document AS IS AND WITH ALL FAULTS, and hereby disclaim all other warranties and conditions, either express, implied, or statutory, including, but not limited to, any (if any) implied warranties, duties, or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, all with regard to the document. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO THE DOCUMENT. IN NO EVENT WILL BEA OR IBM OR MICROSOFT BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THIS DOCUMENT, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

Abstract

This document defines a notation for specifying business process behavior based on Web Services. This notation is called *Business Process Execution Language for Web Services* (abbreviated to BPEL4WS in the rest of this document). Processes in BPEL4WS export and import functionality by using Web Service interfaces exclusively.

Business processes can be described in two ways. *Executable* business processes model actual behavior of a participant in a business interaction. *Business protocols*, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The process descriptions for business protocols are called *abstract processes*. BPEL4WS is meant to be used to model the behavior of both executable and abstract processes.

BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intracorporate and the business-to-business spaces.

Status of this Document

This is an initial public draft release of the BPEL4WS specification. We anticipate a number of extensions to the feature set of BPEL4WS that are discussed briefly at the end of the document. BPEL4WS represents a convergence of the ideas in the [XLANG](#) and [WSFL](#) specifications. Both XLANG and WSFL are superseded by the BPEL4WS specification.

BPEL4WS and related specifications are provided as-is and for review and evaluation only. BEA, IBM and Microsoft hope to solicit your contributions and suggestions in the near future. BEA, IBM and Microsoft make no warranties or representations regarding the specifications in any manner whatsoever.

Table of Contents

1. [Introduction](#)
2. [Notational Conventions](#)
3. [Relationship with WSDL](#)
4. [Defining a Business Process](#)
 - 4.1. [Initial Example](#)
 - 4.2. [The structure of a business process](#)
 - 4.3. [Language Extensibility](#)
 - 4.4. [The Lifecycle of a Business Process](#)
5. [Service Linking, Partners, and Service References](#)
 - 5.1. [Service Linking](#)
 - 5.2. [Partners](#)
 - 5.3. [Service References](#)
6. [Message Properties](#)
 - 6.1. [Motivation](#)
 - 6.2. [Defining Properties](#)
7. [Correlation](#)
 - 7.1. [Message Correlation](#)
 - 7.2. [Defining and Using Correlation Sets](#)
8. [Data Handling](#)
 - 8.1. [Expressions](#)
 - 8.1.1. [Boolean Expressions](#)
 - 8.1.2. [Deadline-Valued Expressions](#)
 - 8.1.3. [Duration-Valued Expressions](#)
 - 8.1.4. [General Expressions](#)
 - 8.2. [Containers](#)
 - 8.3. [Assignment](#)
 - 8.3.1. [Atomicity of Assignment](#)
 - 8.3.2. [Assignment Example](#)
 - 8.4. [Summary of Differences Between Abstract and Executable Processes](#)
9. [Basic Activities](#)
 - 9.1. [Standard Attributes for Each Activity](#)
 - 9.2. [Standard Elements for Each Activity](#)
 - 9.3. [Invoking Web Service Operations](#)
 - 9.4. [Providing Web Service Operations](#)
 - 9.5. [Updating Container Contents](#)
 - 9.6. [Signaling Faults](#)
 - 9.7. [Terminating the Service Instance](#)
 - 9.8. [Waiting](#)
 - 9.9. [Doing Nothing](#)

- 10. [Structured Activities](#)
 - 10.1. [Sequence](#)
 - 10.2. [Switch](#)
 - 10.3. [While](#)
 - 10.4. [Pick](#)
 - 10.5. [Flow](#)
 - 10.5.1. [Link Semantics](#)
 - 10.5.2. [Dead-Path-Elimination \(DPE\)](#)
 - 10.5.3. [Flow Graph Example](#)
 - 10.5.4. [Links and Structured Activities](#)
- 11. [Scopes](#)
 - 11.1. [Error Handling in Business Processes](#)
 - 11.2. [Compensation Handlers](#)
 - 11.2.1. [Defining a Compensation Handler](#)
 - 11.2.2. [Invoking a Compensation Handler](#)
 - 11.3. [Fault Handlers](#)
 - 11.3.1. [Implicit Fault and Compensation Handlers](#)
 - 11.3.2. [Semantics of Activity Termination](#)
 - 11.3.3. [Handling Faults That Occur Inside Fault and Compensation Handlers](#)
 - 11.3.4. [Serializable Scopes](#)
- 12. [Examples](#)
 - 12.1. [Shipping Service](#)
 - 12.1.1. [Service Description](#)
 - 12.1.2. [Message Properties](#)
 - 12.1.3. [Process](#)
 - 12.2. [Loan Approval](#)
 - 12.2.1. [Service Description](#)
 - 12.2.2. [Process](#)
 - 12.3. [Multiple Start Activities](#)
 - 12.3.1. [Service Description](#)
 - 12.3.2. [Process](#)
- 13. [Future Directions](#)
 - 13.1. [Scopes](#)
 - 13.1.1. [Containers](#)
 - 13.1.2. [Event Handlers](#)
 - 13.1.3. [Overlapping Scopes](#)
 - 13.1.4. [Atomic Scopes](#)
 - 13.1.5. [Compensation](#)
 - 13.2. [Lifecycle and Query](#)
 - 13.2.1. [Suspend/Resume](#)
 - 13.2.2. [Query](#)
 - 13.3. [Service Composition](#)
 - 13.4. [Relationship to WS-Transaction Specification](#)
- 14. [Security Considerations](#)
- 15. [Acknowledgments](#)
- 16. [References](#)
- 17. [Appendices](#)
 - 17.1. [Appendix A – Standard Faults](#)
 - 17.2. [Appendix B. - Attributes and Defaults](#)
 - 17.3. [Appendix C – Coordination Protocol](#)
 - 17.3.1. [Coordination Protocol for BPEL4WS Scopes](#)
 - 17.4. [Appendix D - XSD Schemas](#)
 - 17.4.1. [BPEL4WS Schema](#)
 - 17.4.2. [Service Link Type Schema](#)
 - 17.4.3. [Service References Schema](#)
 - 17.4.4. [Message Properties Schema](#)

1. Introduction

The goal of the Web Services effort is to achieve universal interoperability between applications by using Web standards. Web Services use a loosely coupled integration model to allow flexible integration of heterogeneous systems in a variety of domains including business-to-consumer, business-to-business and enterprise application integration. The following basic specifications originally defined the Web Services space: SOAP, Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). SOAP defines an XML messaging protocol for basic service interoperability. WSDL introduces a common grammar for describing services. UDDI provides the infrastructure required to publish and discover services in a systematic way. Together, these specifications allow applications to find each other and interact following a loosely coupled, 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 synchronous or uncorrelated asynchronous interactions. Models for business interactions typically assume sequences of peer-to-peer message exchanges, both synchronous and asynchronous, within stateful, long-running interactions involving two or more parties. To define such business interactions, a formal description of the message exchange protocols used by business processes in their interactions is needed. The definition of such *business protocols* involves precisely specifying the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal implementation. There are two good reasons to separate the public aspects of business process behavior from internal or private aspects. One is that businesses obviously do not want to reveal all their internal decision making and data management to their business partners. The other is that, even where this is not the case, separating public from private process provides the freedom to change private aspects of the process implementation without affecting the public business protocol.

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

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

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

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

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

BPEL4WS uses a notion of message properties to identify protocol-relevant data embedded in messages. Properties can be viewed as "transparent" data relevant to public aspects as opposed to the "opaque" data that internal/private functions use. Transparent data affects the public business protocol in a direct way, whereas opaque data is significant primarily to back-end systems and affects the business protocol only by creating nondeterminism because the way it

affects decisions is opaque. We take it as a principle that any data that is used to affect the behavior of a business protocol must be transparent and hence viewed as a property.

The implicit effect of opaque data manifests itself through nondeterminism in the behavior of services involved in business protocols. Consider the example of a purchasing protocol. The seller has a service that receives a purchase order and responds with either acceptance or rejection based on a number of criteria, including availability of the goods and the credit of the buyer. Obviously, the decision processes are opaque, but the fact of the decision must be reflected as behavior alternatives in the external business protocol. In other words, the protocol requires something like a switch activity in the behavior of the seller's service but the selection of the branch taken is nondeterministic. Such nondeterminism can be modeled by allowing the assignment of a nondeterministic or opaque value to a message property, typically from an enumerated set of possibilities. The property can then be used in defining conditional behavior that captures behavioral alternatives without revealing actual decision processes. BPEL4WS explicitly allows the use of nondeterministic data values to make it possible to capture the essence of public behavior while hiding private aspects.

Defining business protocols and defining executable business processes require very similar concepts. The concepts required for defining business protocols and those required for defining executable business processes form a continuum, and BPEL4WS is designed to cover this continuum. BPEL4WS defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. The interaction with each partner occurs through Web Service interfaces, and the structure of the relationship at the interface level is encapsulated in what we call a service link. The BPEL4WS 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. BPEL4WS also introduces systematic mechanisms for dealing with business exceptions and processing faults. Finally, BPEL4WS introduces a mechanism to define how individual or composite activities within a process are to be compensated in cases where exceptions occur or a partner requests reversal.

The basic concepts of BPEL4WS can be applied in one of two ways. A BPEL4WS process can define a business protocol role, using the notion of *abstract process*. For example, in a supply-chain protocol, the buyer and the seller are two distinct roles, each with its own abstract process. Their relationship is typically modeled as a service link. Abstract processes use all the concepts of BPEL4WS but approach data handling in a way that reflects the level of abstraction required to describe public aspects of the business protocol. Specifically, abstract processes handle only protocol-relevant data. BPEL4WS provides a way to identify protocol-relevant data as *message properties*. In addition, abstract processes use nondeterministic data values to hide private aspects of behavior.

It is also possible to use BPEL4WS to define an executable business process. The logic and state of the process determine the nature and sequence of the Web Service interactions conducted at each business partner, and thus the interaction protocols. While a BPEL4WS process definition is not required to be complete from a private implementation point of view, the language effectively defines a portable execution format for business processes that rely exclusively on Web Service resources and XML data. Moreover, such processes execute and interact with their partners in a consistent way regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Even where private implementation aspects use platform-dependent functionality, which is likely in many if not most realistic cases, the continuity of the basic conceptual model between abstract and executable processes in BPEL4WS makes it possible to export and import the public aspects embodied in business protocols as process or role templates while maintaining the intent and structure of the protocols. This is arguably the most attractive prospect for the use of BPEL4WS 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.

BPEL4WS is layered on top of several XML specifications: WSDL 1.1, XML Schema 1.0, and XPath1.0. WSDL messages and XML Schema type definitions provide the data model used by BPEL4WS processes. XPath provides support for data manipulation. All external resources and partners are represented as WSDL services. BPEL4WS provides extensibility to accommodate future versions of these standards, specifically the XPath and related standards used in XML computation.

2. Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 [\[13\]](#)

Namespace URIs of the general form "some-URI" represent some application-dependent or context-dependent URI as defined in RFC 2396 [\[14\]](#)

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 `<!-- descriptions -->` as follows: ? (0 or 1), * (0 or more), + (1 or more). The characters [and] are used to indicate that contained items are to be treated as a group with respect to the ?, *, or + characters.
- Elements and attributes separated by | and grouped by (and) are meant to be syntactic alternatives.
- The XML namespace prefixes (defined below) are used to indicate the namespace of the element being defined.
- Examples starting with `<?xml` contain enough information to conform to this specification; other examples are fragments and require additional information to be specified in order to conform.

XSD schemas and WSDL definitions are provided as a formal definition of grammars [\[7\]\[3\]](#).

3. Relationship with WSDL

BPEL4WS depends on the following XML-based specifications: WSDL 1.1, XML Schema 1.0, and XPath 1.0. Among these, WSDL has the most influence on the BPEL4WS language. The BPEL4WS process model is layered on top of the service model defined by WSDL 1.1. At the core of the BPEL4WS process model is the notion of peer-to-peer interaction between services described in WSDL; both the process and its partners are modeled as WSDL services. A business process defines how to coordinate the interactions between a process instance and its partners. In this sense, a BPEL4WS 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, BPEL4WS defines the message exchange protocols followed by the business process of a specific role in the interaction.

The definition of a BPEL4WS business process also follows the WSDL model of separation between the abstract message contents used by the business process and deployment information (messages and portType versus binding and address information). In particular, a BPEL4WS process represents all partners and interactions with these partners in terms of abstract WSDL interfaces (portTypes and operations); no references are made to the actual services used by a process instance. A BPEL4WS process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them. Note that the description of the deployment of a BPEL4WS process is out of scope for this specification.

4. Defining a Business Process

Business processes can be described in two ways. **Executable** business processes model actual behavior of a participant in a business interaction. In executable processes, no attempt is made to separate externally visible or "public" aspects of a business process from its internal workings. **Business protocols**, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The processes involved in a business protocol are called **abstract processes**. Abstract processes are not typically executable. They are meant to couple Web Service interface definitions with behavioral specifications that can be used to both constrain the implementation of business roles and define in precise terms the behavior that each party in a business protocol can expect from the others. BPEL4WS is meant to be used to define both kinds of processes. The difference between the two lies *exclusively* in the different feature sets for data handling that are available in the two kinds of processes. These differences are defined precisely in the section on [Data Handling](#).

4.1. Initial Example

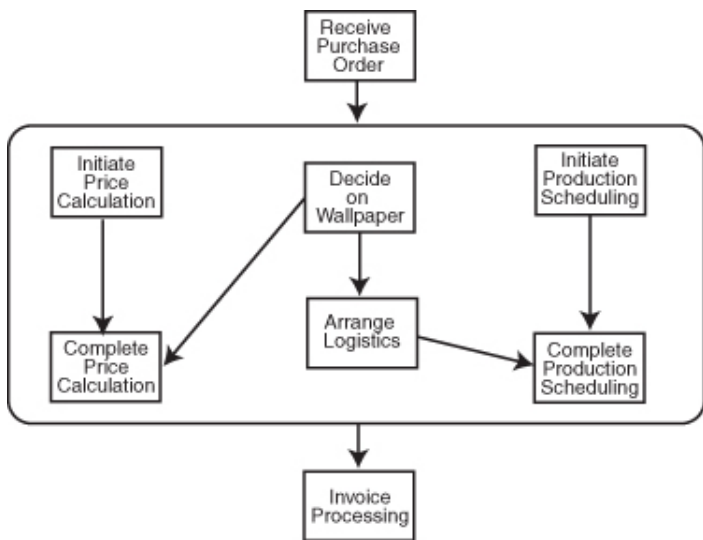
Before describing the structure of business processes in detail, this section presents a simple example of a BPEL4WS process for handling a purchase order. The aim is to introduce the most basic structures and some of the fundamental concepts of the language.

The operation of the process is very simple, and is represented in [Figure 1](#). 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 BPEL4WS processes. It is used here informally as an aid to understanding.

On receiving the purchase order from a customer, the process initiates three tasks in parallel: 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 in parallel, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer.

Figure 1. Initial example



The WSDL portType offered by the service to its customers (purchaseOrderPT) is shown in the following WSDL document in [Listing 1](#). Other WSDL definitions required by the business process are included in the same WSDL document for simplicity; in particular, the portTypes for the Web Services providing price calculation, shipping selection and scheduling, and production scheduling functions are also defined there. Observe that there are no bindings or service elements in the WSDL document. A BPEL4WS process is defined "in the abstract" by referencing only the portTypes of the services involved in the process, and not their possible deployments. Defining business processes in this way allows the reuse of business process definitions over multiple deployments of compatible services.

The service link types included at the bottom of [Listing 1](#) represent the interaction between the purchase order service and each of the parties with which it interacts (see [Service Linking, Partners, and Service References](#)). Service link types can be used to represent dependencies between services, regardless of whether a BPEL4WS business process is defined for one or more of those services. Each service link type defines up to two "role" names, and lists the portTypes that each role must support for the interaction to be carried out successfully. In this example, two link types, "purchaseLT" and "schedulingLT", list a single role because, in the corresponding service interactions, one of the parties provides all the invoked operations: The "purchaseLT" service link represents the connection between the purchase order service and the requesting customer, where only the purchase order service needs to offer a service operation ("sendPurchaseOrder"); the "schedulingLT" service link represents the interaction between the purchase order service and the scheduling service, in which only operations of the latter are invoked. The two other service link types, "invoiceLT" and "shippingLT", define two roles because both the user of the invoice calculation and the user of the shipping service (the invoice or the shipping schedule) must provide callback operations to enable asynchronous notifications to be asynchronously sent ("invoiceCallbackPT" and "shippingCallbackPT" portTypes).

Listing 1. Sample WSDL definitions

```

<definitions targetNamespace="http://manufacturing.org/wsdl/purchase"
  xmlns:sns="http://manufacturing.org/xsd/purchase"
  xmlns:pos="http://manufacturing.org/wsdl/purchase"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/07/service-link/">

```

```

<import namespace="http://manufacturing.org/xsd/purchase"
        location="http://manufacturing.org/xsd/purchase.xsd"/>

<message name="POMessage">
    <part name="customerInfo" type="sns:customerInfo"/>
    <part name="purchaseOrder" type="sns:purchaseOrder"/>
</message>
<message name="InvMessage">
    <part name="IVC" type="sns:Invoice"/>
</message>
<message name="orderFaultType">
    <part name="problemInfo" type="xsd:string"/>
</message>
<message name="shippingRequestMessage">
    <part name="customerInfo" type="sns:customerInfo"/>
</message>
<message name="shippingInfoMessage">
    <part name="shippingInfo" type="sns:shippingInfo"/>
</message>
<message name="scheduleMessage">
    <part name="schedule" type="sns:scheduleInfo"/>
</message>

<!-- portTypes supported by the purchase order process -->

<portType name="purchaseOrderPT">
    <operation name="sendPurchaseOrder">
        <input message="pos:POMessage"/>
        <output message="pos:InvMessage"/>
        <fault name="cannotCompleteOrder"
                message="pos:orderFaultType"/>
    </operation>
</portType>
<portType name="invoiceCallbackPT">
    <operation name="sendInvoice">
        <input message="pos:InvMessage"/>
    </operation>
</portType>
<portType name="shippingCallbackPT">
    <operation name="sendSchedule">
        <input message="pos:scheduleMessage"/>
    </operation>
</portType>

<!-- portType supported by the invoice services -->

<portType name="computePricePT">
    <operation name="initiatePriceCalculation">
        <input message="pos:POMessage"/>
    </operation>
    <operation name="sendShippingPrice">
        <input message="pos:shippingInfoMessage"/>
    </operation>
</portType>

<!-- portType supported by the shipping service -->

<portType name="shippingPT">
    <operation name="requestShipping">
        <input message="pos:shippingRequestMessage"/>
        <output message="pos:shippingInfoMessage"/>
        <fault name="cannotCompleteOrder"
                message="pos:orderFaultType"/>
    </operation>

```



```

</portType>

<!-- portType supported by the production scheduling process -->

<portType name="schedulingPT">
  <operation name="requestProductionScheduling">
    <input message="pos:POMessage" />
  </operation>
  <operation name="sendShippingSchedule">
    <input message="pos:scheduleMessage" />
  </operation>
</portType>

<slnk:serviceLinkType name="purchaseLT">
  <slnk:role name="purchaseService">
    <slnk:portType name="pos:purchaseOrderPT" />
  </slnk:role>
</slnk:serviceLinkType>

<slnk:serviceLinkType name="invoiceLT">
  <slnk:role name="invoiceService">
    <slnk:portType name="pos:computePricePT" />
  </slnk:role>
  <slnk:role name="invoiceRequester">
    <portType name="pos:invoiceCallbackPT" />
  </slnk:role>
</slnk:serviceLinkType>

<slnk:serviceLinkType name="shippingLT">
  <slnk:role name="shippingService">
    <slnk:portType name="pos:shippingPT" />
  </slnk:role>
  <slnk:role name="shippingRequester">
    <portType name="pos:shippingCallbackPT" />
  </slnk:role>
</slnk:serviceLinkType>

<slnk:serviceLinkType name="schedulingLT">
  <slnk:role name="schedulingService">
    <slnk:portType name="pos:schedulingPT" />
  </slnk:role>
</slnk:serviceLinkType>

</definitions>

```

The business process for the order service is defined next in [Listing 2](#). There are four major sections in this process definition:

- The `<containers>` section defines the data containers used by the process, providing their definitions in terms of WSDL message types. Containers allow processes to maintain state data and process history based on messages exchanged.
- The `<partners>` section defines the different parties that interact with the business process in the course of processing the order. The four partners shown here correspond to the sender of the order (customer), as well as the providers of price (invoiceProvider), shipment (shippingProvider), and manufacturing scheduling services (schedulingProvider). Each partner is characterized by a service link type and a role name. This information identifies the functionality that must be provided by the business process and by the partner for the relationship to succeed, that is, the portTypes that the purchase order process and the partner need to implement.
- The `<faultHandlers>` section contains fault handlers defining the activities that must be executed in response to faults resulting from the invocation of the assessment and approval services. In BPEL4WS, all faults, whether internal or resulting from a service invocation, are identified by a qualified name. In particular, each WSDL fault is identified in BPEL4WS by a qualified name formed by the target namespace of the WSDL document in which the relevant portType and fault are defined, and the nname of the fault. It is important to note, however, that because WSDL 1.1 does not require that fault names be unique within the namespace where the operation is

defined, all faults sharing a common name and defined in the same namespace are indistinguishable. In spite of this serious WSDL limitation, BPEL4WS provides a uniform naming model for faults, in the expectation that future versions of WSDL will provide a better fault-naming model.

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

Listing 2. Business process for the order services

```
<process name="purchaseOrderProcess"
  targetNamespace="http://acme.com/ws-bp/purchase"
  xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  xmlns:lns="http://manufacturing.org/wsdl/purchase">

  <partners>
    <partner name="customer"
      serviceLinkType="lns:purchaseLT"
      myRole="purchaseService"/>
    <partner name="invoiceProvider"
      serviceLinkType="lns:invoiceLT"
      myRole="invoiceRequester"
      partnerRole="invoiceService"/>
    <partner name="shippingProvider"
      serviceLinkType="lns:shippingLT"
      myRole="shippingRequester"
      partnerRole="shippingService"/>
    <partner name="schedulingProvider"
      serviceLinkType="lns:schedulingLT"
      partnerRole="schedulingService"/>
  </partners>

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

  <faultHandlers>
    <catch faultName="lns:cannotCompleteOrder"
      faultContainer="POFault">
      <reply partner="customer"
        portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder"
        container="POFault"
        faultName="cannotCompleteOrder"/>
    </catch>
  </faultHandlers>

  <sequence>
    <receive partner="customer"
      portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder"
      container="PO">
    </receive>
```

```

<flow>

  <links>
    <link name="ship-to-invoice" />
    <link name="ship-to-scheduling" />
  </links>

  <sequence>
    <assign>
      <copy>
        <from container="PO" part="customerInfo" />
        <to container="shippingRequest"
            part="customerInfo" />
      </copy>
    </assign>

    <invoke partner="shippingProvider"
            portType="lns:shippingPT"
            operation="requestShipping"
            inputContainer="shippingRequest"
            outputContainer="shippingInfo">
      <source linkName="ship-to-invoice" />
    </invoke>

    <receive partner="shippingProvider"
            portType="lns:shippingCallbackPT"
            operation="sendSchedule"
            container="shippingSchedule">
      <source linkName="ship-to-scheduling" />
    </receive>

  </sequence>

  <sequence>

    <invoke partner="invoiceProvider"
            portType="lns:computePricePT"
            operation="initiatePriceCalculation"
            inputContainer="PO">
    </invoke>
    <invoke partner="invoiceProvider"
            portType="lns:computePricePT"
            operation="sendShippingPrice"
            inputContainer="shippingInfo">
      <target linkName="ship-to-invoice" />
    </invoke>

    <receive partner="invoiceProvider"
            portType="lns:invoiceCallbackPT"
            operation="sendInvoice"
            container="Invoice" />

  </sequence>

  <sequence>
    <invoke partner="schedulingProvider"
            portType="lns:schedulingPT"
            operation="requestProductionScheduling"
            inputContainer="PO">
    </invoke>
    <invoke partner="schedulingProvider"
            portType="lns:schedulingPT"
            operation="sendShippingSchedule"
            inputContainer="shippingSchedule">
  </sequence>

```

```

        <target linkName="ship-to-scheduling" />
    </invoke>
</sequence>
</flow>

    <reply partner="customer"
        portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder"
        container="Invoice" />
</sequence>

</process>

```

The structure of the main processing section is defined by the outer `<sequence>` element, which states that the three elements contained inside are executed in order. The customer request is received (`<receive>` element), then processed (inside a `<flow>` section that enables concurrent execution), 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 executed 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 example makes the implicit assumption that the customer request can be processed in a reasonable amount of time, justifying the requirement that the invoker wait for a synchronous response (because this service is offered as a request-response operation). When that assumption does not hold, the interaction with the customer is better modeled as a pair of asynchronous message exchanges. In that case, the "sendPurchaseOrder" operation is a one-way operation and the asynchronous response is sent by invoking a second one-way operation on a customer "callback" interface. In addition to changing the signature of "sendPurchaseOrder" and defining a new portType to represent the customer callback interface, two modifications need to be made in the preceding example to support an asynchronous response to the customer. First, the service link type "purchaseLT" that represents the process-customer connection needs to include a second role ("customer") listing the customer callback portType. Second, the `<reply>` activity in the process needs to be replaced by an `<invoke>` on the customer callback operation.

The processing taking place inside the `<flow>` element consists of three `<sequence>` blocks running concurrently. The synchronization dependencies between activities in the three parallel 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 execution of the activities nested directly inside a flow proceeds in parallel. In the example, however, the presence of two links introduces control dependencies between the activities executed inside each sequence. For example, while the price calculation can be started immediately after the request is received, shipping price can only be added to the invoice after the shipper information has been obtained; this dependency is represented by the link (named "ship-to-invoice") that connects the first call on the shipping provider ("requestShipping") with sending shipping information to the price calculation service ("sendShippingPrice"). Likewise, shipping scheduling information can only be sent to the manufacturing scheduling service after it has been received from the shipper service; thus the need for the second link ("ship-to-scheduling").

Observe that information is passed between the different activities in an implicit way through the sharing of globally visible data containers. In this example, the control dependencies represented by links are related to corresponding data dependencies, in one case on the availability of the shipper rates and in another on the availability of a shipping schedule. The information is passed from the activity that generates it to the activity that uses it by means of two global data containers ("shippingInfo" and "shippingSchedule"). The current version of BPEL4WS supports only global data containers, but future versions will include locally scoped data as well. Moreover, in the future, data flow will be allowed through links in addition to using links to express synchronization dependencies.

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

Finally, it is important to observe how an assignment activity is used to transfer information between data containers. The simple assignments shown in this example transfer a message part from a source container to a message part in a

target container, but more complex forms of assignments are also possible.

4.2. The structure of a business process

This section provides a quick summary of the BPEL4WS 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 shown in Listing 3.

Listing 3. Basic BPEL4WS structure

```
<process name="ncname" targetNamespace="uri"
  queryLanguage="anyURI"?
  expressionLanguage="anyURI"?
  suppressJoinFailure="yes|no"?
  enableInstanceCompensation="yes|no"?
  abstractProcess="yes|no"?
  xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/">

  <partners>?
    <!-- Note: At least one role must be specified. -->
    <partner name="ncname" serviceLinkType="qname"
      myRole="ncname"? partnerRole="ncname"?>+
    </partner>
  </partners>

  <containers>?
    <!-- Note: The message type may be indicated with the messageType
      attribute or with an inlined <wsdl:message> element within. -->
    <container name="ncname" messageType="qname"?>
      <wsdl:message name="ncname">?
      ...
    </wsdl:message>
    </container>
  </containers>

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

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

  <compensationHandler>?
    activity
  </compensationHandler>

  activity
</process>
```

The top-level attributes are as follows:

- **queryLanguage.** This attribute specifies the XML query language used for selection of nodes in assignment, property definition, and other uses. The default for this attribute is XPath 1.0, represented by the URI of the XPath

1.0 specification: <http://www.w3.org/TR/1999/REC-xpath-19991116>.

- **expressionLanguage**. This attribute specifies the expression language used in the process. The default for this attribute is XPath 1.0, represented by the URI of the XPath 1.0 specification: <http://www.w3.org/TR/1999/REC-xpath-19991116>.
- **suppressJoinFailure**. This attribute determines whether the `joinFailure` fault will be suppressed for all activities in the process. The default for this attribute is "no".
- **enableInstanceCompensation**. This attribute determines whether the process instance as a whole can be compensated by platform-specific means. The default for this attribute is "no".
- `abstractProcess`. This attribute specifies whether the process being defined is abstract (rather than executable). The default for this attribute is "no".

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

- `<receive>`
- `<reply>`
- `<invoke>`
- `<assign>`
- `<throw>`
- `<terminate>`
- `<wait>`
- `<empty>`
- `<sequence>`
- `<switch>`
- `<while>`
- `<pick>`
- `<flow>`
- `<scope>`
- `<compensate>`

The syntax of each of these elements is considered in the following sections. The `<receive>` construct allows the business process to do a blocking wait for a matching message to arrive.

Listing 4. The `<receive>` construct

```
<receive partner="ncname" portType="qname" operation="ncname"
         container="ncname" createInstance="yes|no"?
         standard-attributes>
  standard-elements
  <correlations?>
    <correlation set="ncname" initiation="yes|no">+
  </correlations>
</receive>
```

The `<reply>` construct allows the business process to send a message in reply to a message that was received through a `<receive>`. The combination of a `<receive>` and a `<reply>` forms a request-response operation on the WSDL portType for the process.

```
<reply partner="ncname" portType="qname" operation="ncname"
        container="ncname" faultName="qname"?
        standard-attributes>
  standard-elements
  <correlations?>
    <correlation set="ncname" initiation="yes|no"?>+
  </correlations>
</reply>
```

The `<invoke>` construct allows the business process to invoke a one-way or request-response operation on a portType offered by a partner.

```

<invoke partner="ncname" portType="qname" operation="ncname"
    inputContainer="ncname" outputContainer="ncname"?
    standard-attributes>
    standard-elements
    <correlations>?
        <correlation set="ncname" initiation="yes|no"?
            pattern="in|out|out-in"/>+
    </correlations>
    <catch faultName="qname" faultContainer="ncname"?>*
        activity
    </catch>
    <catchAll>?
        activity
    </catchAll>
    <compensationHandler>?
        activity
    </compensationHandler>
</invoke>

```

The `<assign>` construct can be used to update the values of containers with new data. An `<assign>` construct can contain any number of elementary assignments assignments. The syntax of the assignment activity is:

```

<assign standard-attributes>
    standard-elements
    <copy>+
        from-spec
        to-spec
    </copy>
</assign>

```

where the various options for ***from-spec*** and ***to-spec*** tokens are:

```

<from container="ncname" part="ncname"? query="queryString"?/>
<from partner="ncname" serviceReference="myRole|partnerRole"/>
<from container="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>
<from opaque="yes"/>

<to container="ncname" part="ncname"? query="queryString"?/>
<to partner="ncname"/>
<to container="ncname" property="qname"/>

```

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

```

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

```

The `<terminate>` construct allows you to immediately terminate a business process.

```

<terminate standard-attributes>
    standard-elements

```

```
</terminate>
```

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

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

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

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

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

```
<sequence standard-attributes>
  standard-elementsactivity+
</sequence>
```

The `<switch>` construct allows you to select exactly one branch of execution from a set of choices.

```
<switch standard-attributes>
  standard-elements
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise?>
    activity
  </otherwise>
</switch>
```

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

```
<while condition="bool-expr" standard-attributes>
  standard-elementsactivity
</while>
```

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

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partner="ncname" portType="qname"
    operation="ncname" container="ncname">+
  <correlations?>
    <correlation set="ncname" initiation="yes|no"?>+
```



```

    </correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>*
    activity
  </onAlarm>
</pick>

```

The `<flow>` construct allows you to specify one or more activities to be executed in parallel. Links can be used within parallel activities to define arbitrary control structures.

```

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

  activity+
</flow>

```

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

```

<scope containerAccessSerializable="yes|no" standard-attributes>
  standard-elements
  <faultHandlers>?
    ... see above under <process> for syntax ...
  </faultHandlers>
  <compensationHandler>?
    ... see above under <process> for syntax ...
  </compensationHandler>
  activity
</scope>

```

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

```

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

```

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

```

name="ncname"?
joinCondition="bool-expr"?
suppressJoinFailure="yes|no"?

```

where the default values are as follows:

- name. No default value (that is, unnamed)
- joinCondition. The logical OR of the liveness status of all links that are targeted at this activity
- suppressJoinFailure. No

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

```
<target linkName="ncname" />*
<source linkName="ncname" transitionCondition="bool-expr"? />*
```

where the default value of the "transitionCondition" attribute is "true()", the truth-value function from the default expression language XPath 1.0.

4.3. Language Extensibility

BPEL4WS contains constructs that are generally sufficient for expressing abstract and executable business processes. In some cases, however, it might be necessary to "extend" the BPEL4WS language with additional constructs from other XML namespaces.

BPEL4WS supports extensibility by allowing namespace-qualified attributes to appear on any BPEL4WS element and by allowing elements from other namespaces to appear within BPEL4WS defined elements. This is allowed in the XML Schema specifications for BPEL4WS.

All extension namespaces used in a BPEL4WS document MUST be declared. An extension namespace is declared by using the following syntax:

```
<extension namespace="anyURI" />*
```

Extensions MUST NOT change the semantics of any element or attribute from the BPEL4WS namespace.

4.4. The Lifecycle of a Business Process

As noted in the introduction, the interaction model that is directly supported by WSDL is essentially a stateless client-server model of synchronous or uncorrelated asynchronous interactions. BPEL4WS, builds on WSDL by assuming that all external interactions of the business process occur through Web Service operations. However, BPEL4WS 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 BPEL4WS business process definition can be thought of as a template for creating business process instances.

The creation of a process instance in BPEL4WS is always implicit; activities that receive messages (that is, receive activities and pick activities) can be annotated to indicate that the occurrence of that activity causes a new instance of the business process to be created. This is done by setting the createInstance attribute of such an activity to "yes". When a message is received by such an activity, an instance of the business process is created if it does not already exist (see [Providing Web Service Operations](#) and [Pick](#)).

To be instantiated, each business process must contain at least one such "start activity." This must be an initial activity in the sense that there is no basic activity that logically precedes it in execution flow.

If more than one start activity is enabled for concurrent execution, then all such activities must use at least one correlation set and must use the same correlation sets (see [Correlation](#) and the [Multiple Start Activities](#) example).

If exactly one start activity is expected to be executed, the use of correlation sets is unconstrained. This includes a pick with multiple onMessage branches; each such branch can use different correlation sets or no correlation sets.

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

- When the activity that defines the behavior of the process as a whole completes. In this case the termination is

normal.

- When a fault reaches the process scope, and is either handled or not handled. In this case the termination is considered abnormal even if the fault is handled and the fault handler does not rethrow any fault. A compensation handler is never installed for a scope that terminates abnormally.
- When a process instance is explicitly terminated by a terminate activity (see [Terminating the Service Instance](#)). In this case the termination is abnormal.
- If a compensation handler is specified for the business process as a whole (see [Compensation Handlers](#)), a business process instance can be compensated after normal completion by platform-specific means. This functionality is enabled by setting the enableInstanceCompensation attribute of the process to "yes".

5. Service Linking, Partners, and Service References

A very important, if not the most important, use case for BPEL4WS will be in describing cross-enterprise business interactions in which the business processes of each enterprise interact through Web Service interfaces with the processes of other enterprises. An important requirement for realistic modeling of business processing in this environment is the ability to model a partner process. WSDL already describes the functionality of a service provided by a partner, at both the abstract and concrete levels. The relationship of a business process to a partner is typically peer-to-peer, requiring a two-way dependency at the service level. In other words, a partner represents both a consumer of a service provided by the business process and a provider of a service to the business process. This is especially the case when the interactions are based on asynchronous messaging rather than on remote procedure calls. The notion of service links is used to directly model peer-to-peer partner relationships. Service links define the shape of a relationship with a partner by defining the message and port types used in the interactions in both directions. However, the actual partner service may be dynamically determined within the process. BPEL4WS defines a notion of service reference to represent the dynamic data required to describe a partner service.

It is important to emphasize that the notions of service link and service reference used here are preliminary. There is as yet no generally agreed specification for these concepts as they relate to Web Services, and we expect standard definitions for them to emerge in future. The BPEL4WS specification will be updated to conform to the expected future standard.

5.1. Service Linking

A service link type characterizes the relationship between two services by defining the "roles" played by each of the services in the relationship and specifying the portTypes provided by each role. The following example illustrates the basic syntax of a service link type declaration:

```
<serviceLinkType name="BuyerSellerLink"
  xmlns="http://schemas.xmlsoap.org/ws/2002/07/service-link/">
  <role name="Buyer">
    <portType name="buy:BuyerPortType"/>
  </role>
  <role name="Seller">
    <portType name="sell:SellerPortType"/>
  </role>
</serviceLinkType>
```

Each role can include any number of WSDL portTypes.

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

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

The extensibility mechanism of WSDL 1.1 is used to define serviceLinkType as a new definition type to be placed as an immediate child element of a <wSDL:definitions> element in all cases. This allows reuse of the WSDL target namespace specification and, more importantly, its import mechanism to import portTypes. For cases where a service

link type declaration is linking the portTypes of two different services, the service link type declaration can be placed in a separate WSDL document (with its own targetNamespace).

The syntax for defining a service link type is:

```
<definitions name="ncname" targetNamespace="uri"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/07/service-link/">
  ...
  <slnk:serviceLinkType name="ncname">
    <slnk:role name="ncname">
      <slnk:portType name="qname"/>+
    </slnk:role>
    <slnk:role name="ncname">?
      <slnk:portType name="qname"/>+
    </slnk:role>
  </slnk:serviceLinkType>
  ...
</definitions>
```

This defines a service link type in the namespace indicated by the value of the "targetNamespace" attribute of the WSDL document element. The portTypes identified within `<slnk:role>` are referenced by using QNames as for all top-level WSDL definitions.

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

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

5.2. Partners

The services with which a business process interacts are modeled as partners in BPEL4WS. Each partner is characterized by a serviceLinkType. More than one partner can be characterized by the same serviceLinkType. For example, a certain procurement process might use more than one vendor for its transactions, but might use the same serviceLinkType for all vendors.

```
<partners>
  <partner name="ncname" serviceLinkType="qname"
    myRole="ncname"? partnerRole="ncname"?>+
  </partner>
</partners>
```

Each partner is named, and this name is used for all service interactions with that partner. This is critical, for example, in correlating responses to different partners for simultaneous requests of the same kind (see [Invoking Web Service Operations](#) and [Providing Web Service Operations](#)).

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

Note that the partner declarations specify the static shape of the relationships that the BPEL4WS process will employ in its behavior. When such a process is deployed and executed, all the partners for whom the partnerRole attribute is present must be resolved to actual services before operations on the partner's service can be invoked. The relevant information about a partner service can be set as part of business process deployment. This is outside the scope of BPEL4WS. However, it is also possible to select and assign actual partner services dynamically, and BPEL4WS provides the mechanisms to do so. The dynamic content of the information regarding a partner's service is conceptually

encapsulated in a service reference. In fact, because the partners are likely to be stateful, the service endpoint information needs to be extended with instance-specific information. BPEL4WS allows the service references implicitly present in partner definitions to be both extracted and assigned dynamically, and also to be set more than once. The form of a service reference is described in the following paragraphs. See [Assignment](#) for the mechanisms used for dynamic assignment of service references to partners.

5.3. Service References

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

The fundamental use of service references is to serve as the mechanism for dynamic communication of port-specific data for services. A service reference makes it possible in BPEL4WS to dynamically select a provider for a particular type of service and to invoke their operations. BPEL4WS provides a general mechanism for correlating messages to stateful instances of a service, and therefore service references that carry instance-neutral port information are often sufficient. However, in general it is necessary to carry additional instance-identification tokens in the service reference itself.

A service reference is defined as a typed reference that includes port-specific data for a service, and optionally additional data regarding instance-identification tokens and other relevant properties. Relevant WSDL schemas are used wherever possible to avoid redundancy.

The syntactic structure of a service reference is:

```
<sref:serviceReference
  xmlns:sref="http://schemas.xmlsoap.org/ws/2002/07/service-reference/"
  <wsdl:definitions> ... </wsdl:definitions?
  <sref:service name="qname" />
  <sref:referenceProperties?
    <sref:property name="qname">+
      <!-- any element content -->
    </sref:property>
  </sref:referenceProperties>
</sref:serviceReference>
```

At a minimum, a service reference is the qualified name of a `<wsdl:service>` element where that element is either inlined within the service reference or assumed to be already known by the recipient of the service reference. The following is a minimal example of a service reference:

```
<sref:serviceReference
  xmlns:sref="http://schemas.xmlsoap.org/ws/2002/07/service-reference/"
  xmlns:ns="http://example.com/services/"
  <sref:service name="ns:myService" />
</sref:serviceReference>
```

If the service is not assumed to be already known by reference, its definition can be inlined in the service reference as a way of dynamically communicating the service definition part of a WSDL document.

```
<sref:serviceReference
  xmlns:sref="http://schemas.xmlsoap.org/ws/2002/07/service-reference/"
  xmlns:ns="http://example.com/services/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  <wsdl:definitions
    targetNamespace="http://example.com/services/" ...>
```

```

    <wsdl:service name="myService">
        ...
    </wsdl:service>
</wsdl:definitions>
<sref:service name="ns:myService"/>
</sref:serviceReference>

```

The inlined `<wsdl:definitions>` element **MUST NOT** contain more than one inline service definition. The standard use of `<wsdl:definitions>` includes only a single service definition. The presence of any other definitions can affect the portability of service references.

Finally, the service reference might require additional tokens for purposes such as identifying one or more service instances of interest. The service reference schema does not assign any particular significance to these tokens. They are permitted as a convenient way to carry metadata for a variety of purposes. The key point is that this data is always associated with globally named properties (see [Message Properties](#)). As such, they have semantics that are assumed to be known by the receiver of the service reference.

```

<sref:serviceReference
  xmlns:sref="http://schemas.xmlsoap.org/ws/2002/07/service-reference/"
  xmlns:ns="http://example.com/services/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<sref:service name="ns:myService"/>
<sref:referenceProperties>
  <sref:property name="ns:instanceID">
    74b9f5d0-33fb-4a81-b02b-5b760641c1d6
  </sref:property>
</sref:referenceProperties>
</sref:serviceReference>

```

Every partner in a BPEL4WS process instance is assigned a unique service reference in the course of the deployment or during the execution of the process.

6. Message Properties

6.1. Motivation

The data in a message consists conceptually of two parts: application data and protocol-relevant data, where the protocols can be business protocols or infrastructure protocols providing higher quality of service. An example of business protocol data is the correlation tokens that are used in correlation sets (see [Correlation](#)). Examples of infrastructure protocols are security, transaction, and reliable messaging protocols. The business protocol data is usually found embedded in the application-visible message parts, whereas the infrastructure protocols almost always add implicit extra parts to the message types to represent protocol headers that are separate from application data. Such implicit parts are often called message context because they relate to security context, transaction context, and other similar middleware context of the interaction. Business processes might need to gain access to and manipulate both kinds of protocol-relevant data. The notion of message properties is defined as a general way of naming and representing distinguished data elements within a message, whether in application-visible data or in message context. For a full accounting of the service description aspects of infrastructure protocols, it is necessary to define notions of service policies, endpoint properties, and message context. This work is outside the scope of BPEL4WS. Message properties are defined here in a sufficiently general way to cover message context consisting of implicit parts, but the use in this specification focuses on properties embedded in application-visible data that is used in the definition of business protocols and abstract business processes.

6.2. Defining Properties

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

sequence-number property does. Properties can occur anywhere in a message, including in the message context.

A typical use for a property in BPEL4WS 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 global name is given to this use of the type by defining a property, as in the following example:

```
<definitions name="properties"
  targetNamespace="http://example.com/properties.wsdl"
  xmlns:tns="http://example.com/properties.wsdl"
  xmlns:txtyp="http://example.com/taxTypes.xsd"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

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

  ...
</wsdl:definitions>
```

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

The WSDL extensibility mechanism is used to define properties so that the target namespace and other useful aspects of WSDL are available. The BPEL4WS standard namespace, "http://schemas.xmlsoap.org/ws/2002/07/business-process/", is used for property definitions. The syntax for a property definition is a new kind of WSDL definition as follows:

```
<wsdl:definitions name="ncname"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/">
  <bpws:property name="ncname" type="qname"/>
  ...
</wsdl:definitions>
```

Properties used in business protocols are typically embedded in application-visible message data. The notion of aliasing is introduced to map a global property to a field in a specific message part. The property name becomes an alias for the message part and location, and can be used as such in [Expressions](#) and [Assignment](#) in abstract business processes.

```
<definitions name="properties"
  targetNamespace="http://example.com/properties.wsdl"
  xmlns:tns="http://example.com/properties.wsdl"
  xmlns:txtyp="http://example.com/taxTypes.xsd"
  xmlns:txmsg="http://example.com/taxMessages.wsdl"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- define a correlation property -->
  <bpws:property name="taxpayerNumber" type="txtype:SSN"/>
  ...
  <bpws:propertyAlias propertyName="tns:taxpayerNumber"
    messageType="txmsg:taxpayerInfo" part="identification"
    query="/socialsecnumber"/>
  </bpws:propertyAlias>
</definitions>
```

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

The syntax for a `propertyAlias` definition is:

```
<definitions name="ncname"
  ...
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/">
  <bpws:propertyAlias propertyName="qname"
    messageType="qname" part="ncname" query="queryString"/>
  ...
</wsdl:definitions>
```

The interpretation of the message, part, and query attributes is the same as in the corresponding *from-spec* in copy assignments (see [Assignment](#)).

7. Correlation

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

In the object-oriented world, such stateful interactions are mediated by *object references*, which intrinsically provide the ability to reach a specific object (instance) with the right state and history for the interaction. This works reasonably well in tightly coupled implementations where a dependency on the structure of the implementation is normal. In the loosely coupled world of Web Services, the use of such references would create a fragile web of implementation dependencies that would not survive the independent evolution of business process implementation details at each business partner. In this world, the answer is to rely on the business data and communication protocol headers that define the wire-level contract between partners and to avoid the use of implementation-specific tokens for instance routing whenever possible.

Consider the usual supply-chain situation where a buyer sends a purchase order to a seller. Suppose that the buyer and seller have a stable business relationship and are statically configured to send documents related to the purchasing interaction to the URLs associated with the relevant WSDL service ports. The seller needs to asynchronously return an acknowledgement for the order, and the acknowledgement must be routed to the correct business process instance at the buyer. The obvious and standard mechanism to do this is to carry a business token in the order message (such as a purchase order number) that is copied into the acknowledgement for correlation. The token can be in the message envelope in a header or in the business document (purchase order) itself. In either case, the exact location and type of the token in the relevant messages is fixed and instance independent. Only the value of the token is instance dependent. Therefore, the structure and position of the correlation tokens in each message can be expressed declaratively in the business process description. The BPEL4WS notion of correlation set, described in the following section, provides this feature. The declarative information allows a BPEL4WS-compliant infrastructure to use correlation tokens to provide instance routing automatically.

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

7.1. Message Correlation

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

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

BPEL4WS addresses correlation scenarios by providing a declarative mechanism to specify correlated groups of operations within a service instance. A set of correlation tokens can be defined as a set of properties shared by all messages in the correlated group. Such a set of properties is called a *correlation set*.

Correlation sets are instantiated within the scope of an instance of the business process. Like the instantiation of a business process, the instantiation of a correlation set is triggered by a specially marked operation. In the current version of BPEL4WS, a correlation set can be instantiated only once during the lifetime of a business process instance. Its value, once initialized, can be thought of as an alias for the identity of the business process instance.

In multiparty business protocols, each participant process in a correlated message exchange acts either as the initiator or as a follower of the exchange. The initiator process sends the first message (as part of an operation invocation) that starts the conversation, and therefore defines the values of the properties in the correlation set that tag the conversation. All other participants are followers that instantiate their correlation sets in the conversation by receiving an incoming message that provides the values of the properties in the correlation set. Both initiator and followers must mark the first activity in their respective groups as the activity that initializes the correlation set.

7.2. Defining and Using Correlation Sets

The examples in this section show correlation being used on almost every messaging activity (receive, reply, and invoke). This is because BPEL4WS 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 BPEL4WS can be reduced to those activities that establish the conversational connections.

Each correlation set in BPEL4WS is a named group of properties that, taken together, serve to define a way of identifying an application-level conversation within a business protocol instance. A given message can carry multiple correlation sets. After initialization, the values of the properties for a correlation set in a business process instance must be identical for all the messages in all the operations that carry the correlation set. If at execution time this constraint is violated, the standard fault `bpws:correlationViolation` MUST be thrown by a compliant implementation. The same fault MUST be thrown if an activity attempts to use a correlation set that has not been instantiated. As the following examples illustrate, a correlation set is instantiated when the activity within which it is used applies the attribute `initiation="yes"` to the set.

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

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

```

<definitions name="properties"
  targetNamespace="http://example.com/supplyCorrelation.wsdl"
  xmlns:tns="http://example.com/supplyCorrelation.wsdl"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

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

```

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

```

<definitions name="correlatedMessages"
  targetNamespace="http://example.com/supplyMessages.wsdl"
  xmlns:tns="http://example.com/supplyMessages.wsdl"
  xmlns:cor="http://example.com/supplyCorrelation.wsdl"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!--define schema types for PO and invoice information -->
  <types>
    <xsd:complexType name="PurchaseOrder">
      <xsd:element name="CID" type="xsd:string"/>
      <xsd:element name="order" type="xsd:int"/>
      ...
    </xsd:complexType>
  </types>
  <types>
    <xsd:complexType name="PurchaseOrderResponse">
      <xsd:element name="CID" type="xsd:string"/>
      <xsd:element name="order" type="xsd:int"/>
      ...
    </xsd:complexType>
  </types>
  <types>
    <xsd:complexType name="PurchaseOrderReject">
      <xsd:element name="CID" type="xsd:string"/>
      <xsd:element name="order" type="xsd:int"/>
      <xsd:element name="reason" type="xsd:string"/>
      ...
    </xsd:complexType>
  </types>
  <types>
    <xsd:complexType name="Invoice">
      <xsd:element name="VID" type="xsd:string"/>
      <xsd:element name="invNum" type="xsd:int"/>
    </xsd:complexType>
  </types>
  <message name="POMessage">
    <part name="PO" type="tns:PurchaseOrder"/>
  </message>
  <message name="POResponse">
    <part name="RSP" type="tns:PurchaseOrderResponse"/>
  </message>
  <message name="POReject">
    <part name="RJCT" type="tns:PurchaseOrderReject"/>

```

```

</message>
<message name="InvMessage">
  <part name="IVC" type="tns:Invoice"/>
</message>
<bpws:propertyAlias propertyName="cor:customerID"
  messageType="tns:POMessage" part="PO"
  query="/CID"/>
<bpws:propertyAlias propertyName="cor:orderNumber"
  messageType="tns:POMessage" part="PO"
  query="/Order"/>
<bpws:propertyAlias propertyName="cor:vendorID"
  messageType="tns:InvMessage" part="IVC"
  query="/VID"/>
<bpws:propertyAlias propertyName="cor:invoiceNumber"
  messageType="tns:InvMessage" part="IVC"
  query="/InvNum"/>
...
</definitions>

```

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

```

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

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

```

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

```

<correlationSets
  xmlns:cor="http://example.com/supplyCorrelation.wsdl">
  <!-- Order numbers are particular to a customer,
    this set is carried in application data -->
  <correlationSet name="PurchaseOrder"
    properties="cor:customerID cor:orderNumber"/>

  <!-- Invoice numbers are particular to a vendor,
    this set is carried in application data -->

```

```
<correlationSet name="Invoice"
  properties="cor:vendorID cor:invoiceNumber"/>
</correlationSets>
```

Correlation sets are used in invoke, receive, and reply activities (see [Invoking Web Service Operations](#) and [Providing Web Service Operations](#)) to indicate which correlation sets occur in the messages being sent and received. The `initiation` attribute is used to indicate whether the set is being initialized. When the attribute is set to "yes" the set is initialized with the values of the properties occurring in the message being sent or received. Finally, in the case of invoke, when the operation invoked is synchronous request/response, a `pattern` attribute is used to indicate whether the correlation applies to the outbound (request) message, the inbound (response) message, or both. These ideas are explained in more detail in the context of the use of correlation in the rest of this example.

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

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

```
<receive partner="Buyer" portType="SP:PurchasingPT"
  operation="AsyncPurchase"
  container="PO">
  <correlations>
    <correlation set="PurchaseOrder" initiation="yes">
  </correlations>
</receive>

<invoke partner="Buyer" portType="SP:BuyerPT"
  operation="AsyncPurchaseResponse" inputContainer="POResponse">
  <correlations>
    <correlation set="PurchaseOrder" initiation="no" pattern="out">
    <correlation set="Invoice" initiation="yes" pattern="out">
  </correlations>
</invoke>
```

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

```
<invoke partner="Buyer" portType="SP:BuyerPT"
  operation="AsyncPurchaseReject" inputContainer="POReject">
  <correlations>
    <correlation set="PurchaseOrder" pattern="out">
  </correlations>
</invoke>
```

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

```
<invoke partner="Seller" portType="SP:PurchasingPT" operation="SyncPurchase"
  inputContainer="sendPO"
  outputContainer="getResponse">
```

```

<correlations>
  <correlation set="PurchaseOrder" initiation="yes"
    pattern="out">
  <correlation set="Invoice" initiation="yes"
    pattern="in">
</correlations>
<catch faultName="SP:RejectPO" faultContainer="POReject">
  <!-- handle the fault -->
</catch>
</invoke>

```

Note that an `invoke` consists of two messages: an outgoing request message and an incoming reply message. The correlation sets applicable to each message must be separately considered because they can be different. In this case the `PurchaseOrder` correlation applies to the outgoing request that initializes it, while the `Invoice` correlation applies to the incoming reply and is initialized by the reply. Because the `PurchaseOrder` correlation is initialized by an outgoing message, the buyer is the initiator of that correlation but a follower of the `Invoice` correlation because the values of the correlation properties for `Invoice` are set by the seller in the reply received by the buyer.

8. Data Handling

Business processes model stateful interactions. The state involved consists of messages received and sent as well as other relevant data such as time-out values. The maintenance of the state of a business process requires the use of state variables, which are called containers. 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. BPEL4WS provides these features for XML data types and WSDL message types. The XML family of standards in these areas is still evolving, and using the process-level attributes for query and expression languages provides for the incorporation of future standards.

The differences between abstract and executable processes are exclusively in the data-handling feature set available to each kind of process. Executable processes are permitted to use the full range of data manipulation and assignment features but are not permitted to use nondeterministic values. Abstract processes are restricted to limited manipulation of values contained in message properties but are permitted to use nondeterministic values to reflect the consequences of hidden private behavior. Detailed differences are specified in the following sections, and summarized at the end of the data-handling section.

8.1. Expressions

BPEL4WS uses several types of expressions. The kinds of expressions used are as follows (relevant usage contexts are listed in parentheses):

- Boolean-valued expressions (transition conditions, join conditions, while condition, and switch cases)
- Deadline-valued expressions ("until" attribute of `onAlarm` and `wait`)
- Duration-valued expressions ("for" attribute of `onAlarm` and `wait`)
- General expressions (assignment)

BPEL4WS provides an extensible mechanism for the language used in these expressions. The language is specified by the `expressionLanguage` attribute of the process element. Compliant implementations of the current version of BPEL4WS MUST support the use of XPath 1.0 as the expression language. XPath 1.0 is indicated by the default value of the `expressionLanguage` attribute, which is:

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

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

BPEL4WS introduces several extension functions to XPath's built-in functions to enable XPath 1.0 expressions to access information from the process. The extensions are defined in the standard BPEL4WS namespace

"http://schemas.xmlsoap.org/ws/2002/07/business-process/". The prefix "bpws:" is associated with this namespace.

Any qualified names used within XPath expressions are resolved by using namespace declarations currently in scope in the BPEL4WS document at the location of the expression.

The following functions are defined by this specification:

```
bpws:getContainerData ('containerName', 'partName', 'locationPath?')
```

This function extracts values from containers. The first argument names the source container for the data, the second names the part to select from that container, and the third optional argument, when present, provides an absolute location path (with '/' meaning the root of the document fragment representing the entire part) to identify the root of a subtree within the document fragment representing the part. The return value of this function is a node set containing the single node representing either an entire part (if the third argument is absent) or the result of the selection based on the locationPath. If the given locationPath selects a node set of a size other than one, then the standard fault `bpws:selectionFailure` MUST be thrown by a compliant implementation.

The use of `bpws:getContainerData` is *prohibited in expressions used within abstract processes*.

```
bpws:getContainerProperty ('containerName', 'propertyName')
```

This function extracts global property values from containers. The first argument names the source container for the data and the second is the qualified name (QName) of the global property to select from that container (see [Message Properties](#)). If the given property does not appear in any of the parts of the container's message type, then the standard fault `bpws:selectionFailure` MUST be thrown by a compliant implementation. The return value of this function is a node set containing the single node representing the property. If the given property definition selects a node set of a size other than one, then the standard fault `bpws:selectionFailure` MUST be thrown by a compliant implementation.

```
bpws:getLinkStatus ('linkName')
```

This function returns a Boolean indicating the status of the link (see [Link Semantics](#)). If the status of the link is positive the value is true, and if the status is negative the value is false. This function MUST NOT be used anywhere except in a join condition. The linkName argument MUST refer to the name of an incoming link for the activity associated with the join condition. These restrictions MUST be statically enforced.

These BPEL4WS-defined extension functions are available for use within all XPath 1.0 expressions.

The syntax of XPath 1.0 expressions for BPEL4WS is considered in the following paragraphs.

8.1.1. Boolean Expressions

These are expressions that conform to the XPath 1.0 [Expr](#) production where the evaluation results in Boolean values.

8.1.2. Deadline-Valued Expressions

These are expressions that conform to the XPath 1.0 [Expr](#) production where the evaluation results in values that are of the XML Schema types [dateTime](#) or [date](#). Note that XPath 1.0 is not XML Schema aware. As such, none of the built-in functions of XPath 1.0 are capable of producing or manipulating [dateTime](#) or [date](#) values. However, it is possible to write a constant (literal) that conforms to XML Schema definitions and use that as a deadline value or to extract a field from a container (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.

8.1.3. Duration-Valued Expressions

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

8.1.4. General Expressions

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

Expressions with operators are restricted as follows:

- All numeric values including arbitrary constants are permitted with the equality or relational operators (<, <=, =, !=, >=, >).
- Values of integral (short, int, long, unsignedShort, and so on) type including constants are permitted in numeric expressions, provided that only integer arithmetic is performed. In practice, this means that division is disallowed. It is difficult to enforce this restriction in XPath 1.0 because XPath 1.0 lacks integral support for types. The restriction should be taken as a statement of intent that will be enforced in the future when expression languages with more refined type systems become available.
- Only equality operators (=, !=) are permitted when used with values of string type including constants.

These restrictions reflect XPath 1.0 syntax and semantics. Future alternative standards in this space are expected to provide stronger type systems and therefore support more nuanced constraints. The restrictions are motivated by the fact that XPath general expressions are meant to be used to perform business protocol-related computation such as retry loops, line-item counts, and so on, that must be transparent in the process definition. They are not meant to provide arbitrary computation. This is the motivation for the constraint that numerical expressions deal only with integer computation, and for disallowing arbitrary string manipulation through expressions.

8.2. Containers

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. BPEL4WS treats all state uniformly as a collection of messages.

Containers provide the means for holding messages that constitute the state of a business process. The messages held are often those that have been received from partners or are to be sent to partners. But BPEL4WS does not require this. Containers can hold messages that act as "temporary variables" for computation and are never exchanged with partners.

The type of each container MUST be a WSDL message type. However, BPEL4WS allows the message type to be inlined in the container declaration rather than being defined in a WSDL document. This alternative is provided to avoid cluttering WSDL documents with types for the "temporary variable" case. The syntax of the containers declaration is:

```
<containers>
  <container name="ncname" messageType="qname"?>+
    <wsdl:message name="ncname" >?
      ...
    </wsdl:message>
  </container>
</containers>
```

An example of a container declaration with an inlined message type is:

```
<container name="orderDetails">
  <wsdl:message name="orderDetails">
    <part name="processDuration" type="xsd:duration"/>
  </wsdl:message>
</container>
```

If the same message type is declared in a WSDL document with the targetNamespace "http://example.com/orders", the declaration could be:

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

Containers can be specified as input or output containers for invoke, receive, and reply activities (see [Invoking Web Service Operations](#) and [Providing Web Service Operations](#)). When an invoke operation returns a fault message, this causes a fault in the current scope. The fault container in the corresponding fault handler is initialized with the fault message received (see [Scopes](#) and [Fault Handlers](#)).

At the beginning of a process all containers are uninitialized. Containers can be initialized by a variety of means including assignment and receiving a message. Containers can be partially initialized with property assignment or when some but not all parts in the message type of the container are assigned values. An attempt to use any part of a container before it is initialized MUST result in the standard `bpws:uninitializedContainer` fault. A compliant implementation MAY choose to perform static (pre-execution) analysis to detect and prevent the possibility of this fault. Such analysis is necessarily pessimistic and therefore might in some cases prevent the use of processes that would not fault, either in specific uses or in any use.

Although abstract processes are not executable, BPEL4WS requires that all message properties in a message must be initialized before the message can be used, for example before the container of the message is used as the inputContainer in a Web Service operation invocation.

8.3. Assignment

Copying data from one container to another is a common task within a business process. The *assign* activity can be used to copy data from one container to another, as well as to construct and insert new data using expressions. The use of expressions is primarily motivated by the need to perform simple computation (such as incrementing sequence numbers) that is required for describing business protocol behavior. Expressions operate on message selections, properties, and literal constants to produce a new value for a container property or selection. Finally, this activity can also be used to copy service references to and from partner links.

The *assign* activity contains one or more elementary assignments.

```
<assign standard-attributes>
  standard-elements
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>
```

The assign activity copies a type-compatible value from the source ("from-spec") to the destination ("to-spec"). The *from-spec* MUST be one of the following forms except for the opaque form available in abstract processes:

```
<from container="ncname" part="ncname"? query="queryString"?/>
<from partner="ncname" serviceReference="myRole|partnerRole"/>
<from container="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>
<from opaque="yes">
```

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


```
<to container="ncname" part="ncname"? query="queryString"?/>
<to partner="ncname" />
<to container="ncname" property="qname" />
```

In all variants the container attribute provides the name of a container, and the part attribute provides the name of a part within that container.

In the first *from-spec* and *to-spec* variants, the value of the query attribute is a query string to identify a single value within a source or target container part. BPEL4WS provides an extensible mechanism for the language used in these queries. The language is specified by the attribute "queryLanguage" of the <process> element. Compliant implementations of the current version of BPEL4WS MUST support the use of XPath 1.0 as the query language. XPath 1.0 is indicated by the default value of the queryLanguage attribute, which is:

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

For XPath 1.0, the value of the query attribute MUST be an absolute [locationPath](#) (with '/' meaning the root of the document fragment representing the entire part). It is used to identify the root of a subtree within the document fragment representing the part. The location path MUST select exactly one node. If the location path selects zero nodes or more than one node, then the standard fault `bpws:selectionFailure` MUST be thrown by a compliant implementation. Note that the part and query attributes are optional. Furthermore, the use of the query attribute is *prohibited in abstract processes*.

The second *from-spec* and *to-spec* variants allow dynamic manipulation of the service references associated with partners. The value of the partner attribute is the name of a partner declared in the process. In the case of from-specs, the role must also be specified because a process might need to communicate a service reference corresponding to either its own role or the partner's role. The value "myRole" means that the service reference of the process with respect to that partner is the source, while the value "partnerRole" means that the partner's service reference is the source. For the to-spec, the assignment is only possible to the partnerRole, hence there is no need to specify the role. The type of the value used in partner-style from/to-specs is always a service reference (see [Service Linking, Partners, and Service References](#)).

The third *from-spec* and *to-spec* variants allow explicit manipulation of message properties (see [Message Properties](#)). The property forms are especially useful for abstract processes, because they provide a way to clearly define how distinguished data elements in messages are being used.

The fourth ("expression") *from-spec* variant allows processes to perform simple computations on properties and containers (for example, increment a sequence number).

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

The sixth *from-spec* variant (<from opaque="yes">) allows an opaque value to be assigned based on non-deterministic choice. *This variant can only appear within abstract, i.e., non-executable processes.* The value of this form in the interpretation of assignment is chosen nondeterministically from the XSD value space of the target. It can only be used in assignments where the "to-spec" refers to a container property. The XSD type of the target property must be one of the following:

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

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

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

- The from-spec is a container and the to-spec is a container. In this case both containers MUST be of the same

message type, where two message types are said to be equal if their qualified names are the same.

- One of the from or to specifications is a container and the other is not. This is not legal because parts of containers, selections of container parts, or service references cannot be assigned to/from containers directly.
- In all other cases, the types of the source and destination are XML Schema types, and the constraint is that the source value MUST possess the type associated with the destination. Note that this does not require the types associated with the source and destination to be the same. In particular, the source type MAY be a subtype of the destination type.

If any of the matching constraints above is violated, the standard fault `bpws:mismatchedAssignmentFailure` MUST be thrown by a compliant implementation.

8.3.1. Atomicity of Assignment

An important characteristic of assignment in BPEL4WS is that assignment activities are *atomic*. If there is any fault during the execution of an assignment activity, the destination containers are left unchanged as they were at the start of the activity. This applies regardless of the number of assignment elements within the overall assignment activity.

8.3.2. Assignment Example

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

```
<complexType name="address">
  <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>
```

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

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

Also assume the following BPEL4WS container declarations:

```
<container name="c1" messageType="x:person"/>
<container name="c2" messageType="x:person"/>
```

The first example illustrates copying one container to another:

```
<assign>
  <copy>
    <from container="c1"/>
```

```
<to container="c2"/>
</copy>
</assign>
```

8.4. Summary of Differences Between Abstract and Executable Processes

The use of `bpws:getContainerData` is prohibited in XPath expressions used in abstract processes.

In assignments used in abstract processes, the *from-spec* and *to-spec* may not use queries within a container part, that is, the use of the `query` attribute is prohibited.

The special nondeterministic *from-spec* form for copy assignment based on the attribute `opaque` is permitted within abstract processes and prohibited in executable processes.

9. Basic Activities

9.1. Standard Attributes for Each Activity

Each activity has optional standard attributes: a name, a join condition, and an indicator whether a join fault should be suppressed if it occurs. A join condition is used to specify requirements about parallel paths reaching at an activity. See [Flow](#) for a full discussion of the last two attributes. The default value of `suppressJoinFailure` is `no`.

```
name="ncname" ?
joinCondition="bool-expr" ?
suppressJoinFailure="yes/no" ?>
```

The value of the `joinCondition` attribute is a Boolean-valued expression in the expression language indicated for this document (see [Expressions](#)). The default value of the join condition for the default expression language XPath is the logical OR of the link status of all incoming links of this activity.

9.2. Standard Elements for Each Activity

Each BPEL4WS activity has optional nested standard elements `<source>` and `<target>`. The use of these elements is required for establishing synchronization relationships through links (see [Flow](#)). Each link is defined independently and given a name. The link name is used as value of the `linkName` attribute of the `<source>` element. An activity MAY declare itself to be the source of one or more links by including one or more `<source>` elements. Each `<source>` element MUST use a distinct link name. Similarly, an activity MAY declare itself to be the target of one or more links by including one or more `<target>` elements. Each `<source>` element associated with a given activity MUST use a link name distinct from all other `<source>` elements at that activity. Each `<target>` element associated with a given activity MUST use a link name distinct from all other `<target>` elements at that activity. Each `<source>` element MAY optionally specify a transition condition that functions as a guard for following this specified link (see [Flow](#)). If the transition condition is omitted, it is deemed to be present with the constant value `true`.

```
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<target linkName="ncname"/>*
```

9.3. Invoking Web Service Operations

Web Services provided by partners (see [Service Linking, Partners, and Service References](#)) can be used to perform work in a BPEL4WS business process. Invoking an operation on such a service is a basic activity. Recall that such an operation can be a synchronous request/response or an asynchronous one-way operation. BPEL4WS uses the same basic syntax for both with some additional options for the synchronous case.

An asynchronous invocation specifies only the input container of the operation because it does not expect a response

as part of the operation (see [Providing Web Service Operations](#)). A synchronous invocation specifies both an input container and an output container. One or more correlation sets can be specified to correlate the business process instance with a stateful service at the partner's side (see [Correlation](#)).

In the case of a synchronous invocation, the operation might return a WSDL fault message. This results in a BPEL4WS fault. Such a fault can be caught locally by the activity, and in this case the specified activity will be performed. If a fault is not caught locally by the activity it is thrown to the scope that encloses the activity (see [Scopes](#) and [Fault Handlers](#)).

Note that a WSDL fault is identified in BPEL4WS by a qualified name formed by the target namespace of the corresponding portType and the fault name. This uniform naming mechanism must be followed even though it does not accurately match WSDL's fault-naming model. Because WSDL does not require that fault names be unique within the namespace where the service operation is defined, all faults sharing a common name and defined in the same namespace are indistinguishable in BPEL4WS. In WSDL 1.1 it is necessary to specify a portType name, an operation name, and the fault name to uniquely identify a fault. This limits the ability to use fault-handling mechanisms to deal with invocation faults. This is an important shortcoming of the WSDL fault model that will be removed in future versions of WSDL.

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

Semantically, the specification of local fault and/or compensation handlers is equivalent to the presence of an implicit scope immediately enclosing the activity and providing those handlers. The name of such an implicit scope is always the same as the name of the activity it encloses.

```
<invoke partner="ncname" portType="qname" operation="ncname"
  inputContainer="ncname" outputContainer="ncname"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiation="yes|no"?
      pattern="in|out|out-in"/>+
  </correlations>
  <catch faultName="qname" faultContainer="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
</invoke>
```

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

```
<invoke partner="Seller" portType="SP:Purchasing"
  operation="SyncPurchase"
  inputContainer="sendPO"
  outputContainer="getResponse">
  <compensationHandler>
    <invoke partner="Seller" portType="SP:Purchasing"
      operation="CancelPurchase"
      inputContainer="getResponse"
      outputContainer="getConfirmation">
    </compensationHandler>
</invoke>
```

9.4. Providing Web Service Operations

A business process provides services to its partners through receive activities and corresponding reply activities. A receive activity specifies the partner it expects to receive from, and the port type and operation that it expects the partner to invoke.

In addition, receive activities play a role in the lifecycle of a business process. The only way to instantiate a business process in BPEL4WS is to annotate a receive activity with the `createInstance` attribute set to "yes" (see [Pick](#) for a variant). The default value of this attribute is "no". A receive activity annotated in this way MUST be an initial activity in the process, that is, the only other basic activities potentially enabled to execute prior to or simultaneously with such a receive activity MUST be similarly annotated receive activities.

It is permissible to have the `createInstance` attribute set to "yes" for a set of concurrent initial activities. In this case the intent is to express the possibility that any one of a set of *required* inbound messages can create the process instance because the order in which these messages arrive cannot be predicted. All such receive activities MUST use the same correlation sets (see [Correlation](#)). Compliant implementations MUST ensure that only one of the inbound messages carrying the same correlation set tokens actually instantiates the business process (usually the first one to arrive, but this is implementation dependent). The other incoming messages in the concurrent initial set MUST be delivered to the corresponding receive activities in the already created instance.

A business process instance MUST NOT simultaneously enable two receive actions for the same partner, portType, and operation. Note that `receive` is a blocking operation in the sense that its execution will not complete until a matching message is received by the process instance. The constraint expressed here therefore forbids a logical fault, but not a race condition. However, this specification does not provide a static analysis algorithm to detect such a fault, but stipulates that if two `receive` actions for the same partner, portType, and operation are in fact simultaneously enabled during execution of a business process instance, then the standard fault `bpws:conflictingReceive` MUST be thrown by a compliant implementation. For the purposes of this constraint, an `onMessage` clause in a `pick`, is equivalent to a `receive` (see [Pick](#)).

```
<receive partner="ncname" portType="qname" operation="ncname"
  container="ncname" createInstance="yes|no"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiation="yes|no"?>+
  </correlations>
</receive>
```

A `reply` activity is used to send a response to a request previously accepted through a `receive` activity. Such responses are only meaningful for synchronous interactions. An asynchronous response is always sent by invoking the corresponding one-way operation on the partner's service. The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner for a particular portType and operation MUST NOT be outstanding at any given time during execution. If this constraint is violated in a business process instance, then the standard fault `bpws:conflictingRequest` MUST be thrown by a compliant implementation. Note that this is semantically different from the `bpws:conflictingReceive`, because it is possible to create the `conflictingRequest` by consecutively receiving the same request from a specific partner for a particular portType and operation. For the purposes of this constraint, an `onMessage` clause in a `pick` is equivalent to a `receive` (see [Pick](#)).

```
<reply partner="ncname" portType="qname" operation="ncname"
  container="ncname" faultName="qname"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiation="yes|no"?>+
  </correlations>
</reply>
```

9.5. Updating Container Contents

Container update occurs through the assignment activity, which is described in [Assignment](#).

9.6. Signaling Faults

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

BPEL4WS does not require fault names to be defined prior to their use in a `throw` element. An application or process-specific fault name can be directly used by using an appropriate QName as the value of the `faultName` attribute and providing a container with the fault data if required. This provides a very lightweight mechanism to introduce application-specific faults.

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

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

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

9.7. Terminating the Service Instance

The `terminate` activity can be used to immediately abandon all execution within the business process instance that executes the `terminate` activity. All currently running activities MUST be terminated as soon as possible without any fault handling or compensation behavior.

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

9.8. Waiting

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

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

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

```
<sequence>
  <wait until="2002-12-24T18:00+01:00"/>
  <invoke partner="CallServer" portType="AutomaticPhoneCall">
```

```
        operation="TextToSpeech"
        inputContainer="seasonalGreeting">
</invoke>
</sequence>
```

9.9. Doing Nothing

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

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

10. Structured Activities

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

The structured activities of BPEL4WS include:

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

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

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

10.1. Sequence

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

```
<sequence standard-attributes>
  standard-elementsactivity+
</sequence>
```

Example:

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

```
<pick>
  ...
</pick>
</sequence>
```

10.2. Switch

The `switch` structured activity supports conditional behavior in a pattern that occurs quite often. The activity consists of an ordered list of one or more conditional branches defined by case elements, followed optionally by an `otherwise` branch. The case branches of the `switch` are considered in the order in which they appear. The first branch whose condition holds true is taken and provides the activity performed for the switch. If no branch with a condition is taken, then the `otherwise` branch is taken. If the `otherwise` branch is not explicitly specified, then an `otherwise` branch with an empty activity is deemed to be present. The `switch` activity is complete when the activity of the selected branch completes.

```
<switch standard-attributes>
  standard-elements
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise?>
    activity
  </otherwise>
</switch>
```

Example:

```
<switch xmlns:inventory="http://supply-chain.org/inventory"
  xmlns:FLT="http://example.com/faults">
  <case condition="bpws:getContainerProperty(stockResult,level) > 100">
    <flow>
      <!-- perform fulfillment work -->
    </flow>
  </case>
  <case condition="bpws:getContainerProperty(stockResult,level) >= 0">
    <throw faultName="FLT:OutOfStock"
  container="RestockEstimate"/>
  </case>
  <otherwise>
    <throw faultName="FLT:ItemDiscontinued"/>
  </otherwise>
</switch>
```

10.3. While

The `while` activity supports repeated execution of a specified iterative activity. The iterative activity is executed until the given Boolean while condition no longer holds true.

```
<while condition="bool-expr" standard-attributes>
  standard-elementsactivity
</while>
```

Example:


```

    ...
    <container name="orderDetails">
      <wsdl:message name="orderDetails">
        <part name="numberOfItems" type="xsd:integer"/>
      </wsdl:message>
    </container>
    ...
    <while condition=
      "bpws:getContainerProperty(orderDetails,numberOfItems) > 100">
      <scope>
        ...
      </scope>
    </while>

```

10.4. Pick

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

The form of `pick` is a set of branches of the form event/activity, and exactly one of the branches will be selected based on the occurrence of the event associated with it before any others. Note that after the `pick` activity has accepted an event for handling, the other events are no longer accepted by that `pick`. The possible events are the arrival of some message in the form of the invocation of an inbound one-way or request/response operation, or an "alarm" based on a timer (in the sense of an *alarm clock*).

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

Each `pick` activity MUST include at least one `onMessage` event. Enablement of each `onMessage` handler is equivalent to enablement of the corresponding `receive` activity for the semantics of the occurrence of the `bpws:conflictingReceive` and `bpws:conflictingRequest` faults (see [Providing Web Service Operations](#)).

```

<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partner="ncname" portType="qname"
    operation="ncname" container="ncname">+
    <correlations?
      <correlation set="ncname" initiation="yes|no"?>+
    </correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>+
    activity
  </onAlarm>
</pick>

```

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

```
<pick>
```

```

<onMessage partner="buyer"
    portType="orderEntry"
    operation="inputLineItem"
    container="lineItem">
    <!-- activity to add line item to order -->
</onMessage>
<onMessage partner="buyer"
    portType="orderEntry"
    operation="orderComplete"
    container="completionDetail">
    <!-- activity to perform order completion -->
</onMessage>

<!-- set an alarm to go after 3 days and 10 hours -->
<onAlarm for="P3DT10H">
    <!-- handle timeout for order completion -->
</onAlarm>
</pick>

```

10.5. Flow

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

```

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

```

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

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

```

<sequence>
  <flow>
    <invoke partner="Seller" .../>
    <invoke partner="Shipper" .../>
  </flow>
  <invoke partner="Bank" .../>
</sequence>

```

More generally, a flow activity creates a set of concurrent activities directly nested within it. It further enables expression of synchronization dependencies between activities that are nested directly or indirectly within it. The `link` construct is used to express these synchronization dependencies. A link has a name and all the links of a flow activity **MUST** be defined separately within the flow activity (see [Future Directions](#) for a discussion of the reasons why links are defined separately). The standard `source` and `target` elements of an activity are used to link two activities. The source of the link **MUST** specify a `source` element specifying the link's name and the target of the link **MUST** specify a `target`

element specifying the link's name. The source activity MAY also specify a transition condition through the `transitionCondition` attribute of the source element. If the `transitionCondition` attribute is omitted, it is deemed to be present with a value of "true". 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 MAY be nested arbitrarily deeply within the (structured) activities that are directly nested within the flow, except for the boundary-crossing restrictions.

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

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

In general, a link is said to *cross the boundary* of a syntactic construct if the source activity for the link is nested within the construct but the target activity is not, or *vice versa*, if the target activity for the link is nested within the construct but the source activity is not.

A link MUST NOT cross the boundary of a while activity, a serializable scope, or a compensation handler (see [Scopes](#)). In addition, a link that crosses a fault-handler boundary MUST be outbound, that is, it MUST have its source activity within the fault handler and its target activity within a scope that encloses the scope associated with the fault handler. Finally, a link MUST NOT create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity, where an activity A logically precedes an activity B if the initiation of B semantically requires the completion of A. Therefore, directed graphs created by links are always acyclic.

10.5.1. Link Semantics

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

Every activity that is the target of a link has an implicit or explicit `joinCondition` attribute associated with it. This applies even when an activity has exactly one incoming link. If the explicit `joinCondition` is missing, the implicit condition requires the status of *at least one* incoming link to be positive (see below for an explanation of link status). A join condition is a Boolean expression (see [Expressions](#)). The expression for a join condition for an activity MUST be constructed using only Boolean operators and the `bpws:getLinkStatus` function (see [Expressions](#)) applied to incoming links at the activity.

Without considering links, the semantics of business processes, scopes, and structured activities determine when a

given activity is ready to start execution. For example, the second activity in a sequence is ready to start execution as soon as the first activity completes. An activity that defines the behavior of a branch in a switch is ready to execute if and when that branch is chosen for execution. Similarly, an activity nested directly within a flow is ready to start execution as soon as the flow itself starts execution, because flow is fundamentally a concurrency construct.

If an activity that is ready for execution in this sense has incoming links, then it does not start execution until the status of all its incoming links has been determined and the (implicit or explicit) join condition associated with the activity has been evaluated. The precise semantics of link status evaluation are as follows:

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

- Determine the *status* of all outgoing links for A. The status will be either *positive* or *negative*. To determine the status for each link its `transitionCondition` is evaluated. If the value is `true` the status is positive, otherwise it is negative.
- For each activity B that has a synchronization dependency on A, check whether:
 - B is ready for execution (except for its dependency on incoming links) in the sense described above.
 - The status of all incoming links for B has been determined.
- If both these conditions are true, then evaluate the join condition for B. If the join condition evaluates to false, a standard `bpws:joinFailure` fault is thrown, otherwise the execution of activity B is started.

If, during the execution of structured activity S, the semantics of S dictate that activity X nested within S will not be executed as part of the execution of S, then the status of all outgoing links from X is set to negative. An example is an activity within a branch that is not taken in a switch activity, or activities that were not completed in a scope in which processing was halted due to a fault, including a `bpws:joinFailure` (see [Scopes](#) and [Compensation Handlers](#)).

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

10.5.2. Dead-Path-Elimination (DPE)

In cases where the control flow is largely defined by networks of links, the normal interpretation of a false join condition for activity A is that A should not be performed, rather than that a fault has occurred. Moreover, there is a need to propagate the consequences of this decision by assigning a negative status to the outgoing links for A. BPEL4WS makes it easy to express these semantics by using an attribute `suppressJoinFailure` on an activity. A value of "yes" for this attribute has the effect of suppressing the `bpws:joinFailure` fault for the activity and all nested activities, except where the effect is overridden by using the `suppressJoinFailure` attribute with a value of "no" in a nested activity. Suppressing the `bpws:joinFailure` is equivalent to the fault being logically caught by a special default handler attached to an implicit scope that immediately encloses just the activity with the join condition. The default handler behavior is an `empty` activity, that is, the handler suppresses the fault and does nothing about it. However, because the activity with the join condition was not executed, its outgoing links are automatically assigned a negative status according to the rules of [Link Semantics](#). Thus within an activity with the value of the `suppressJoinFailure` attribute set to "yes", the semantics of a join condition that evaluates to false are to skip execution of the associated activity and to set the status of all outgoing links from that activity to negative. This is called dead-path-elimination because in a graph-like interpretation of networks of links with transition conditions, these semantics have the effect of propagating negative link status transitively along entire paths formed by consecutive links until a join condition is reached that evaluates to true.

Note that the *name* of the implicit scope (created to suppress the `bpws:joinFailure`) that immediately encloses an activity with a join condition is exactly the same as the name of the activity itself. In case this is an invoke activity (see [Invoking Web Service Operations](#)) with an inlined fault or compensation handler, the implicit scope for the fault and compensation handlers is *merged* with the implicit scope described here, which adds an additional fault handler for the `bpws:joinFailure`.

The default value of the `suppressJoinFailure` attribute is "no". This is to avoid unexpected behavior in simple use cases where complex graphs are not involved and links without transition conditions are used for synchronization. The designers of such use cases are likely to be naive about link semantics and are likely to be surprised by the consequences of a default interpretation that suppresses a well-defined fault. For example, consider the interpretation of the [Initial Example](#) with the `suppressJoinFailure` attribute set to "yes". Suppose further that the invocations of the `shippingProvider` are enclosed in a scope that provides a fault handler (see [Scopes](#) and [Fault Handlers](#)). If one of

these invocations were to fault, the status of the outgoing link from the invocation would be negative, and the (implicit) join condition at the target of the link would be false, but the resulting `bpws:joinFailure` would be implicitly suppressed and the target activity would be `silently` skipped within the sequence instead of causing the expected fault.

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

10.5.3. Flow Graph Example

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

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

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

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

10.5.4. Links and Structured Activities

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

The following flow is intended to perform the sequence of activities A, B, and C. Activity B has a synchronization dependency on the two activities X and Y outside of the sequence, that is, B is a target of links from X and Y. The join condition at B is missing, and therefore implicitly assumed to be the default, which is the disjunction of the status of the links targeted to B. The condition is therefore true if at least one of the incoming links has a positive status. In this case that condition reduces to the Boolean condition $P(X,B) \text{ OR } P(Y,B)$ based on the transition conditions on the links.

In the flow, the sequence S and the two receive activities X and Y are all concurrently started when the flow starts execution. Within S, after activity A is completed, B cannot start execution 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 their execution, the join condition for B is evaluated.

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

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

```
<flow suppressJoinFailure="no">
  <links>
    <link name="XtoB"/>
    <link name="YtoB"/>
  </links>

  <sequence name="S">
    <receive name="A" ...>
      ...
    </receive>
    <receive name="B" ...>
      <target linkName="XtoB"/>
      <target linkName="YtoB"/>
      ...
    </receive>
    <receive name="C" ...>
      ...
    </receive>
  </sequence>

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

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

```
<flow suppressJoinFailure="no">
  <links>
    <link name="AtoB"/>
    <link name="BtoC"/>
    <link name="XtoB"/>
```

```

    <link name="YtoB" />
</links>
<receive name="A">
  <source linkName="AtoB" />
</receive>
<receive name="B">
  <target linkName="AtoB" />
  <target linkName="XtoB" />
  <target linkName="YtoB" />
  <source linkName="BtoC" />
</receive>
<receive name="C">
  <target linkName="BtoC" />
</receive>
<receive name="X">
  <source linkName="XtoB" transitionCondition="P(X,B)" />
</receive>
<receive name="Y">
  <source linkName="YtoB" transitionCondition="P(Y,B)" />
</receive>
</flow>

```

11. Scopes

The execution context for each activity is provided by a *scope*. A *scope* can provide fault handlers, a compensation handler, data containers, and correlation sets.

In the current version of BPEL4WS, containers and correlation sets are only permitted at the global scope defined by the entire *process*. See [Future Directions](#) for a discussion of expected changes in future versions.

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

```

<scope containerAccessSerializable="yes|no" standard-attributes>
  standard-elements
  <faultHandlers?>
    ...
  </faultHandlers>
  <compensationHandler?>
    ...
  </compensationHandler>
  activity
</scope>

```

Each scope has a primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities within it to arbitrary depth. The scope is shared by all the nested activities. In the following example, the scope has a primary `flow` activity, which contains two concurrent `invoke` activities. Either of the `invoke` activities can receive one or more types of fault responses. The fault handlers for the scope are shared by both `invoke` activities and can be used to catch the faults caused by the possible fault responses.

```

<scope>
  <faultHandlers?>
    ...
  </faultHandlers>
  <flow>
    <invoke partner="Seller" portType="Sell:Purchasing"
            operation="SyncPurchase"

```

```
        inputContainer="sendPO"
        outputContainer="getResponse" />
    <invoke partner="Shipper"
        portType="Ship:TransportOrders"
        operation="OrderShipment"
        inputContainer="sendShipOrder"
        outputContainer="shipAck" />

</flow>
</scope>
```

11.1. Error Handling in Business Processes

Business processes are often of long duration and use asynchronous messages for communication. They also manipulate sensitive business data in back-end databases and line-of-business applications. Error handling in this environment is both difficult and business critical. The use of ACID transactions is usually limited to local updates because of trust issues and because locks and isolation cannot be maintained for the long periods during which technical and business errors and fault conditions can occur in a business process instance. As a result, the overall business transaction can fail or be cancelled after many ACID transactions have been committed during its progress, and the partial work done must be undone as best as possible. Error handling in business processes therefore relies heavily on the well-known concept of *compensation*, that is, application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned. There is a long history of work in this area regarding the use of Sagas [10] and open nested transactions [11]. BPEL4WS provides a variant of such a compensation protocol by providing the ability for flexible control of the reversal. BPEL4WS achieves this by providing the ability to define fault handling and compensation in an application-specific manner, resulting in a feature called Long-Running (Business) Transactions (LRTs).

It is important to understand that the notion of LRT described here is meant to be used purely within a platform-specific implementation. There is no prescribed requirement that the business process be distributed or span multiple vendors and platforms. For such environments, it is expected that the WS-Transaction specification [12] would be utilized to register participants interested in the reversal notifications provided by the LRT implementation. See [Appendix C](#) for a detailed model of BPEL4WS LRTs based on WS-Transaction concepts.

Additionally, it is important to understand that the notion of LRT described here is purely local and occurs within a single business process instance. There is no distributed coordination regarding an agreed-upon outcome among multiple-participant services. The achievement of distributed agreement is an orthogonal problem outside the scope of BPEL4WS, to be solved by using the protocols described in the WS-Transaction specification. The need to compose WS-transaction with BPEL4WS is recognized. This is addressed further in the section on [Future Directions](#).

As an example of an LRT, consider the planning and fulfillment of a travel itinerary. This can be viewed as an LRT in which individual service reservations can use nested transactions within the scope of the overall LRT. If the itinerary is cancelled, the reservation transactions must be compensated for by cancellation transactions, and the corresponding payment transactions must be compensated accordingly. For ACID transactions in databases the transaction coordinator(s) and the resources that they control know all of the uncommitted updates and the order in which they must be reversed, and they are in full control of such reversal. In the case of business transactions, the compensation behavior is itself a part of the business logic and protocol, and must be explicitly specified. For example, there might be penalties or fees applied for cancellation of an airline reservation depending on the class of ticket and the timing. If a payroll advance has been given to pay for the travel, the reservation must be successfully cancelled before the payroll advance for it can be reversed in the form of a payroll deduction. This means the compensation actions might need to run in the same order as the original transactions, which is not the standard or default in most transaction systems. Using activity scopes as the definition of logical units of work, the LRT feature of BPEL4WS addresses these requirements.

11.2. Compensation Handlers

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

11.2.1. Defining a Compensation Handler

A compensation handler in the current version of BPEL4WS is simply a wrapper for a compensation activity as shown below. In many scenarios the compensation handler needs to receive data about the current state of the world and return data regarding the results of the compensation. This is discussed further below and in [Future Directions](#).

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

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

```
<invoke partner="Seller" portType="SP:Purchasing"  
  operation="SyncPurchase"  
  inputContainer="sendPO"  
  outputContainer="getResponse">  
  <correlations>  
    <correlation set="PurchaseOrder" initiation="yes"  
      pattern="out" />  
  </correlations>  
  
  <compensationHandler>  
    <invoke partner="Seller" portType="SP:Purchasing"  
      operation="CancelPurchase"  
      inputContainer="getResponse"  
      outputContainer="getConfirmation">  
      <correlations>  
        <correlation set="PurchaseOrder" pattern="out" />  
      </correlations>  
    </invoke>  
  </compensationHandler>  
</invoke>
```

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

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

```
<scope>  
  <compensationHandler>  
    <invoke partner="Seller" portType="SP:Purchasing"  
      operation="CancelPurchase"  
      inputContainer="getResponse"  
      outputContainer="getConfirmation">  
      <correlations>  
        <correlation set="PurchaseOrder" pattern="out" />  
      </correlations>  
    </invoke>  
  </compensationHandler>  
  <invoke partner="Seller" portType="SP:Purchasing"  
    operation="SyncPurchase"  
    inputContainer="sendPO"  
    outputContainer="getResponse">  
    <correlations>  
      <correlation set="PurchaseOrder" initiation="yes"  
        pattern="out" />  
    </correlations>
```

```
</invoke>
</scope>
```

Note that the container `getResponse` can be reused later for other purposes before compensation is invoked. But the compensation handler needs the specific response to the `invoke` operation that is being reversed. BPEL4WS semantics state that the compensation handler, if invoked, will see a frozen snapshot of all containers, as they were when the execution of the scope being compensated was completed. In other words, if the compensation handler shown here is used, the contents of `getResponse` that it will see and use are exactly the contents at the time of the completion of the `invoke` activity it compensates. This also means that compensation handlers cannot update live data in the containers that the business process is using. They live entirely in a snapshot world. A compensation handler, once installed, can be thought of as a completely self-contained action that is not affected by, and does not affect, the global state of the business process instance. It can only affect external entities.

It is not realistic to expect compensation activities to always be oblivious to the current state of the world. In fact, compensation both affects and is affected by the current state. However, the shape of the world within which compensation is run is difficult to anticipate. It is therefore necessary to allow the two-way interaction between compensation activities and the live world to take place in a tightly controlled manner. In the future, BPEL4WS will add input and output parameters to compensation handlers for this purpose (see [Future Directions](#)).

As stated in [The Lifecycle of a Process](#), if a compensation handler is specified for the business process as a whole, a business process instance can be compensated *after normal completion* by platform-specific means. This functionality is enabled by setting the `enableInstanceCompensation` attribute of the process to "yes".

11.2.2. Invoking a Compensation Handler

The compensation handler can be invoked by using the `compensate` activity, which names the scope for which the compensation is to be performed, that is, the scope whose compensation handler is to be executed. A compensation handler for a scope is available for invocation only when the scope completes its execution normally. Invoking a compensation handler that has not been installed is equivalent to the `empty` activity (it is a no-op)—this ensures that fault handlers do not have to rely on state to determine which nested scopes have completed successfully. If an installed compensation handler is invoked more than once, a compliant implementation **MUST** throw the standard `bpws:repeatedCompensation` fault.

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

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

The ability to explicitly execute the `compensate` activity is the underpinning of the application-controlled error-handling framework of BPEL4WS. This activity can be used only in the following parts of a business process:

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

Example:

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

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

If the compensation handler for a scope is absent, the default compensation handler invokes the compensation handlers for the immediately enclosed scopes in the reverse order of the completion of those scopes.

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

11.3. Fault Handlers

Fault handling in a business process can be thought of as a mode switch from the normal processing in a scope. The optional fault handlers attached to a scope provide a way to define a set of custom fault-handling activities, syntactically defined as `catch` activities. Each `catch` activity is defined to intercept a specific kind of fault, defined by a globally unique fault QName and a container for the data associated with the fault. If the fault name is missing, then the catch will intercept all faults with the right type of fault data. The fault container is optional because a fault might not have additional data associated with it. A fault response to an `invoke` activity is one source of faults, with obvious name and data aspects based on the definition of the fault in the WSDL operation. A programmatic `throw` activity is another source, again with explicitly given name and data. BPEL4WS defines several standard faults with their names and data, and there might be other platform-specific faults such as communication failures that can occur during the execution of a business process. A `catchAll` clause can be added to catch any fault not caught by a more specific catch handler.

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

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

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

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

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

```
<invoke partner="Seller"
  portType="SP:Purchasing"
  operation="SyncPurchase"
  inputContainer="sendPO"
  outputContainer="getResponse">
  <catch faultName="SP:POFault" faultContainer="POFault">
    <!-- handle the fault -->
  </catch>
</invoke>
```

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

```
<scope>
  <faultHandlers>
    <catch faultName="SP:POFault" faultContainer="POFault">
      <!-- handle the fault -->
    </catch>
  </faultHandlers>
  <invoke partner="Seller"
    portType="SP:Purchasing"
    operation="SyncPurchase"
    inputContainer="sendPO"
    outputContainer="getResponse">
  </invoke>
</scope>
```

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

Note that availability also applies to [Implicit Fault and Compensation Handlers](#).

The execution of a fault handler for scope C begins by implicitly terminating execution of all currently executing activities directly enclosed within C (see [Semantics of Activity Termination](#)). The termination of currently executing activities occurs before the specific behavior of a fault handler is started. This also applies to the implicit fault handlers described below.

11.3.1. Implicit Fault and Compensation Handlers

Because the visibility of scope names and therefore of compensation handlers is limited to the next enclosing scope, the ability to compensate a scope would be lost if the enclosing scope did not have a compensation handler or was missing a fault handler for some fault. Because many faults are not programmatic or the result of operation invocation, it is not reasonable to expect an explicit handler for every fault in every scope. BPEL4WS therefore provides default compensation and fault handlers when these are missing. The behavior of these implicit handlers is to run available compensation handlers in the reverse order of completion of the corresponding scopes. This is defined in more precise terms below.

Whenever a fault handler (for any fault) or the compensation handler is missing for any given scope, they are implicitly created with the following behavior:

Fault handler:

- Run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes.
- Rethrow the fault to the next enclosing scope.

Compensation handler:

- Run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes.

11.3.2. Semantics of Activity Termination

As stated above, the execution of a fault handler for scope C begins by implicitly terminating execution of all currently executing activities directly enclosed within C. The following paragraphs define what this means for all BPEL4WS activity types.

The `invoke`, `reply`, and `assign` activities are sufficiently short-lived that they are allowed to complete rather than being interrupted when termination is forced. The evaluation of expressions when already started is also allowed to complete. The `receive` activity is interrupted and terminated prematurely as is `wait`. The notion of termination does not apply to `empty`, `terminate`, and `throw`.

All structured activity behavior is interrupted. The iteration of `while` is interrupted and termination is applied to the loop body activity. If `switch` has selected a branch, then the termination is applied to the activity of the selected branch. The same applies to `pick`. If either of these activities has not yet selected a branch, then the `switch` and the `pick` are terminated without further execution. The `sequence` and `flow` constructs are terminated by terminating their execution and applying termination to all nested activities currently active within them.

Scopes provide the ability to control the semantics of forced termination to some degree. When the activity being terminated is in fact a scope, the execution of the scope is interrupted and the fault handler for the standard `bpws:forcedTermination` fault is run. Note that this applies only if the scope is in normal processing mode. If the scope has already experienced an internal fault and is executing a fault handler, then as stated above, all other fault handlers including the handler for `bpws:forcedTermination` are uninstalled, and the forced termination has no effect. The already executing fault handler is allowed to complete its execution.

The fault handler for the `bpws:forcedTermination` fault is programmed like other fault handlers, but this fault handler cannot rethrow any fault. Even if an uncaught fault occurs during its execution, it is not rethrown to the next enclosing scope. This is because the enclosing scope has already faulted, which is what is causing the forced termination of the nested scope.

In other respects this is a normal fault handler. Its execution begins by implicitly (recursively) terminating execution of all currently executing activities directly enclosed within its associated scope. It can execute compensate activities. And when it is missing, it is provided by using the same implicit behavior that is used for all other implicit fault handlers.

Note that forced termination of nested scopes occurs in innermost-first order as a result of the rule (quoted above) that execution of any fault handler begins by implicitly (recursively) terminating execution of all currently executing activities directly enclosed within its associated scope.

11.3.3. Handling Faults That Occur Inside Fault and Compensation Handlers

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

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

11.3.4. Serializable Scopes

When the `containerAccessSerializable` attribute is set to "yes", the scope provides concurrency control in governing access to shared containers. Such a scope is called a *serializable scope*. Serializable scopes must not be nested. A scope marked with `containerAccessSerializable="yes"` must be a leaf scope.

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

The use of error handling features in a serializable scope is governed by the following rules:

- The fault handlers for a serializable scope share the serializability domain of the associated scope, that is, in case

a fault occurs in a serializable scope, the execution of the fault handler is considered part of the serializable behavior (in commonly used implementation terms, locks are not released when making the transition to the fault handler). This is because the repair of the fault needs a shared isolation environment to provide predictable behavior.

- The compensation handler for a serializable scope does not share the serializability domain of the associated scope.
- For a serializable scope with a compensation handler, the creation of the state snapshot for compensation is part of the serializable behavior. In other words, it is always possible to reorder execution steps as if the scope had sufficiently exclusive access to the shared containers all the way to completion, including the creation of the snapshot.

It is useful to note that the semantics of serializable scopes are very similar to the standard isolation level "serializable" used in database transactions.

12. Examples

12.1. Shipping Service

This example presents the use of a BPEL4WS abstract process to describe a rudimentary shipping service. This service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The shipping service offers two types of shipment: shipments where the items are held and shipped together and shipment where the items are shipped piecemeal until all of the order is accounted for.

12.1.1. Service Description

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

```
<slnk:serviceLinkType name="shippingLT"
  xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/07/service-link/">
  <slnk:role name="shippingService">
    <slnk:portType name="shippingServicePT" />
  </slnk:role>
  <slnk:role name="shippingServiceCustomer">
    <slnk:portType name="shippingServiceCustomerPT" />
  </slnk:role>
</slnk:serviceLinkType>
```

The corresponding message and portType definitions are as follows:

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

<message name="shippingRequestMsg">
  <part name="shipOrder" type="ship:shipOrder" />
</message>

<message name="shippingNoticeMsg">
  <part name="shipNotice" type="ship:shipNotice" />
</message>

<portType name="shippingServicePT">
  <operation name="shippingRequest">
    <input message="shippingRequestMsg" />
  </operation>
</portType>

<portType name="shippingServiceCustomerPT">
```

```

        <operation name="shippingNotice">
            <input message="shippingNoticeMsg" />
        </operation>
    </portType>

</wsdl:definitions>

```

12.1.2. Message Properties

The properties relevant to the service behavior are:

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

Here are the definitions for the properties and their aliases:

```

<wsdl:definitions
    targetNamespace="http://example.com/shipProps/"
    xmlns:sns="http://ship.org/wsdl/shipping"
    xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/">

    <!-- types used in abstract processes are required to be finite domains.
         The itemCountType is restricted by range -->

    <wsdl:types>
        <xsd:schema>
            <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>

    <!-- a message type is defined here for internal use of the
         abstract process; it is not inlined because we need to
         define a property relative to it -->

    <wsdl:message name="itemsShipped">
        <part name="value" type="ship:itemCountType" />
    </wsdl:message>

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

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

    <bpws:propertyAlias propertyName="tns:shipOrderID"
        messageType="sns:shippingNoticeMsg"
        part="shipNotice"
        query="/ShipNoticeHeader/shipOrderID" />

```

```

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

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

<bpws:propertyAlias propertyName="tns:itemsCount"
    messageType="sns:shippingNoticeMsg"
    part="shipNotice"
    query="/ShipNoticeHeader/itemsCount"/>

<bpws:propertyAlias propertyName="tns:numItemsShipped"
    messageType="tns:itemsShipped"
    part="value"
    query="/" />

</wsdl:definitions>

```

12.1.3. Process

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

```

receive shipOrder
switch
  case shipComplete
    send shipNotice
  otherwise
    itemsShipped := 0
    while itemsShipped < itemsTotal
      itemCount := opaque // non-deterministic assignment
                          // corresponding e.g. to
                          // internal interaction with
                          // back-end system

      send shipNotice
      itemsShipped = itemsShipped + itemCount

```

And here is the more complete version:

```

<process name="shippingService"
    targetNameSpace="http://acme.com/shipping"
    xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
    xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
    xmlns:sns="http://ship.org/wSDL/shipping"
    xmlns:props="http://example.com/shipProps/"
    abstractProcess="yes">

  <partners>
    <partner name="customer"
        serviceLinkType="sns:shippingLT"
        partnerRole="shippingServiceCustomer"
        myRole="shippingService"/>
  </partners>

```



```

<containers>
  <container name="shipRequest"
    messageType="sns:shippingRequestMsg" />
  <container name="shipNotice"
    messageType="sns:shippingNoticeMsg" />
  <container name="itemsShipped"
    messageType="props:itemsShipped" />
</containers>

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

<sequence>

  <receive partner="customer"
    portType="sns:shippingServicePT"
    operation="shippingRequest"
    container="shipRequest">
    <correlations>
      <correlation set="shipOrder" initiation="yes" />
    </correlations>
  </receive>

  <switch>
    <case condition=
      "bpws:getContainerProperty('shipRequest','props:shipComplete') " >
      <sequence>
        <assign>
          <copy>
            <from container="shipRequest" property="props:itemsCount" />
            <to container="shipNotice" property="props:itemsCount" />
          </copy>
        </assign>
        <invoke partner="customer"
          portType="sns:shippingServiceCustomerPT"
          operation="shippingNotice"
          inputContainer="shipNotice">
          <correlations>
            <correlation set="shipOrder" pattern="out" />
          </correlations>
        </invoke>
      </sequence>
    </case>
    <otherwise>
      <sequence>
        <assign>
          <copy>
            <from expression="0" />
            <to container="itemsShipped" part="value" />
          </copy>
        </assign>
        <while condition=
          "bpws:getContainerProperty('itemsShipped','props:numItemsShipped') <
          bpws:getContainerProperty('shipRequest','props:itemsTotal') ">
          <sequence>
            <assign>
              <copy>
                <from opaque="yes" />
                <to container="shipNotice" property="props:itemsCount" />
              </copy>
            </assign>
          </sequence>
        </while>
      </sequence>
    </otherwise>
  </switch>

```

```

    <invoke partner="customer"
           portType="sns:shippingServiceCustomerPT"
           operation="shippingNotice"
           inputContainer="shipNotice">
      <correlations>
        <correlation set="shipOrder" pattern="out" />
      </correlations>
    </invoke>
    <assign>
      <copy>
        <from expression=
          "bpws:getContainerProperty('itemsShipped',
            'props:numItemsShipped')+
            bpws:getContainerProperty('shipNotice',
            'props:itemsCount')"/>
        <to container="itemsShipped" part="value"/>
      </copy>
    </assign>
  </sequence>
</while>
</sequence>
</otherwise>
</switch>
</sequence>

</process>

```

12.2. Loan Approval

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

12.2.1. Service Description

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

```

<definitions
  targetNamespace="http://loans.org/wsd/loan-approval"
  xmlns="http://schemas.xmlsoap.org/wsd/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/07/service-link/"
  xmlns:lms="http://loans.org/wsd/loan-approval">

  <message name="creditInformationMessage">
    <part name="firstName" type="xsd:string"/>
    <part name="name" type="xsd:string"/>
  </message>

```

```

    <part name="amount" type="xsd:integer" />
</message>

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

<message name="riskAssessmentMessage">
    <part name="level" type="xsd:string" />
</message>

<message name="errorMessage">
    <part name="errorCode" type="xsd:integer" />
</message>

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

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

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

<slnk:serviceLinkType name="loanServiceLinkType">
    <slnk:role name="customer">
        <slnk:portType name="lns:loanServicePT" />
    </slnk:role>
</slnk:serviceLinkType>

<slnk:serviceLinkType name="loanApprovalLinkType">
    <slnk:role name="approver">
        <slnk:portType name="lns:loanApprovalPT" />
    </slnk:role>
</slnk:serviceLinkType>

<slnk:serviceLinkType name="riskAssessmentLinkType">
    <slnk:role name="assessor">
        <slnk:portType name="lns:riskAssessmentPT" />
    </slnk:role>
</slnk:serviceLinkType>

</definitions>

```

12.2.2. Process

In the business process defined below, the interaction with the customer is represented by the initial <receive> and the matching <reply> activities. The use of risk assessment and loan approval services is represented by <invoke> elements. All these activities are contained within a <flow>, and their (potentially parallel) execution is staged according to the dependencies expressed by corresponding <link> elements. Note that the transition conditions attached to the <source> elements of the links determine which links get activated. Dead path elimination is enabled by the value "yes" taken by the "suppressJoinFailure" attribute on the <process> element. This implies that as certain links are set false the consequences of this decision can be propagated and the execution of certain activities can be skipped.

Because the operations invoked can return a fault of type "loanProcessFault", a fault handler is provided. When a fault occurs, control is transferred to the fault handler, where a <reply> element is used to return a fault response of type "unableToHandleRequest" to the loan requester.

```
<process name="loanApprovalProcess"
  targetNamespace="http://acme.com/loanprocessing"
  xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  xmlns:lns="http://loans.org/wsd1/loan-approval"
  suppressJoinFailure="yes">

  <partners>
    <partner name="customer"
      serviceLinkType="lns:loanServiceLinkType"
      partnerRole="customer"/>
    <partner name="approver"
      serviceLinkType="lns:loanApprovalLinkType"
      partnerRole="approver"/>
    <partner name="assessor"
      serviceLinkType="lns:riskAssessmentLinkType"
      partnerRole="assessor"/>
  </partners>

  <containers>
    <container name="request"
      messageType="lns:creditInformationMessage"/>
    <container name="risk"
      messageType="lns:riskAssessmentMessage"/>
    <container name="approval"
      messageType="lns:approvalMessage"/>
    <container name="error"
      messageType="lns:errorMessage"/>
  </containers>

  <faultHandlers>
    <catch faultName="lns:loanProcessFault"
      faultContainer="error">
      <reply partner="customer"
        portType="lns:loanServicePT"
        operation="request"
        container="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 partner="customer"
  portType="lns:loanServicePT"
  operation="request"
  container="request" createInstance="yes">
  <source linkName="receive-to-assess"
    transitionCondition=
      "bpws:getContainerData('request','amount')< 10000"/>
  <source linkName="receive-to-approval"
    transitionCondition=
      "bpws:getContainerData('request','amount')>=10000"/>
</receive>

<invoke partner="assessor"
  portType="lns:riskAssessmentPT"
  operation="check"
  inputContainer="request"
  outputContainer="risk">
  <target linkName="receive-to-assess"/>
  <source linkName="assess-to-setMessage"
    transitionCondition=
      "bpws:getContainerData('risk','level')='low'"/>
  <source linkName="assess-to-approval"
    transitionCondition=
      "bpws:getContainerData('risk','level')!='low'"/>
</invoke>

<assign>
  <target linkName="assess-to-setMessage"/>
  <source linkName="setMessage-to-reply"/>
  <copy>
    <from expression="'yes'"/>
    <to container="approval" part="accept"/>
  </copy>
</assign>

<invoke partner="approver"
  portType="lns:loanApprovalPT"
  operation="approve"
  inputContainer="request"
  outputContainer="approval">
  <target linkName="receive-to-approval"/>
  <target linkName="assess-to-approval"/>
  <source linkName="approval-to-reply" />
</invoke>

<reply partner="customer"
  portType="lns:loanServicePT"
  operation="request"
  container="approval">
  <target linkName="setMessage-to-reply"/>
  <target linkName="approval-to-reply"/>
</reply>
</flow>

</process>

```

12.3. Multiple Start Activities

A process can have multiple activities that create a process instance. An example of this situation is a (simplified) business process run by an auction house. The purpose of the business process is to collect information from the buyer

and the seller of a particular auction, report the appropriate auction results to some auction registration service, and then send the registration result back to the seller and the buyer. Thus the business process starts with two activities, one for receiving the seller information and one for receiving the buyer information. Because a particular auction is uniquely identified by an auction ID, the seller and the buyer need to provide this information when sending in their data. The sequence in which the seller and buyer requests arrive at the auction house is random. Thus, when such a request comes in, it needs to be checked whether a business process instance exists already or not. If not, a business process instance is created. After both requests have been received, the auction registration service is invoked. Because the invocation is done asynchronously, the auction house passes the auction ID to the auction registration service. The auction registration service returns this auction ID in its answer so that the auction house can locate the proper business process instance. Because there are many buyers and sellers, each of them needs to provide their service references, so that the auction service can respond properly. In addition, the auction house needs to provide its own service reference to the auction registration service so that the auction registration service can send the response back to the auction house.

12.3.1. Service Description

The auction service offers two port types, called sellerPT and buyerPT, with appropriate operations for accepting the data provided by the seller and the buyer. Because the processing time of the business process is lengthy, the auction service responds to the seller and buyer through appropriate port types, sellerAnswerPT and buyerAnswerPT. These portTypes are properly combined into two service link types, one for the seller called sellerAuctionHouseLT and one for the buyer called buyerAuctionHouseLT.

The auction service needs two port types, called auctionRegistrationPT and auctionRegistrationAnswerPT, that provide for the invocation of the auction registration service. The port types are part of the appropriate service link type auctionHouseAuctionRegistrationServiceLT.

```
<definitions
  targetNamespace="http://www.auction.com/wsdl/auctionService"
  xmlns:tns="http://www.auction.com/wsdl/auctionService"
  xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/07/service-link/"
  xmlns:sref="http://schemas.xmlsoap.org/ws/2002/07/service-reference/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- Messages for communication with the seller -->

  <message name="sellerData">
    <part name="creditCardNumber" type="xsd:string"/>
    <part name="shippingCosts" type="xsd:integer"/>
    <part name="auctionId" type="xsd:integer"/>
    <part name="serviceReference" type="sref:serviceReferenceType"/>
  </message>
  <message name="sellerAnswerData">
    <part name="thankYouText" type="xsd:string"/>
  </message>

  <!-- Messages for communication with the buyer -->

  <message name="buyerData">
    <part name="creditCardNumber" type="xsd:string"/>
    <part name="phoneNumber" type="xsd:string"/>
    <part name="ID" type="xsd:integer"/>
    <part name="serviceReference" type="sref:serviceReferenceType"/>
  </message>
  <message name="buyerAnswerData">
    <part name="thankYouText" type="xsd:string"/>
  </message>

  <!-- Messages for communication with the auction registration service -->

  <message name="auctionDetails">
    <part name="auctionId" type="xsd:integer"/>
```

```

    <part name="amount" type="xsd:integer" />
</message>
<message name="auctionDetailsAnswer">
    <part name="registrationId" type="xsd:integer" />
    <part name="auctionId" type="xsd:integer" />
    <part name="auctionHouseServiceReference"
        type="sref:serviceReferenceType" />
</message>
<!-- Port types for interacting with the seller -->
<portType name="sellerPT">
    <operation name="submit">
        <input message="tns:sellerData" />
    </operation>
</portType>
<portType name="sellerAnswerPT">
    <operation name="answer">
        <input message="tns:sellerAnswerData" />
    </operation>
</portType>
<!-- Port types for interacting with the buyer -->
<portType name="buyerPT">
    <operation name="submit">
        <input message="tns:buyerData" />
    </operation>
</portType>
<portType name="buyerAnswerPT">
    <operation name="answer">
        <input message="tns:buyerAnswerData" />
    </operation>
</portType>
<!-- Port types for interacting with the auction registration service -->
<portType name="auctionRegistrationPT">
    <operation name="process">
        <input message="tns:auctionDetails" />
    </operation>
</portType>
<portType name="auctionRegistrationAnswerPT">
    <operation name="answer">
        <input message="tns:auctionRegAnswer" />
    </operation>
</portType>
<!-- Context type used for locating business process via auction Id -->
<bpws:property name="auctionId"
    type="xsd:string" />
<bpws:propertyAlias propertyName="tns:auctionId"
    messageType="tns:sellerData"
    part="auctionId" />
<bpws:propertyAlias propertyName="tns:auctionId"
    messageType="tns:buyerData"
    part="ID" />
<bpws:propertyAlias propertyName="tns:auctionId"
    messageType="tns:auctionDetailsAnswer"
    part="auctionId" />
<bpws:propertyAlias propertyName="tns:auctionId"

```

```

        messageType="tns:auctionDetails"
        part="auctionId"/>

<!-- Service link type for seller/auctionHouse -->

<slnk:serviceLinkType name="tns:sellerAuctionHouseLT">
  <slnk:role name="auctionHouse">
    <slnk:portType name="tns:sellerPT"/>
  </slnk:role>
  <slnk:role name="seller">
    <slnk:portType name="tns:sellerAnswerPT"/>
  </slnk:role>
</slnk:serviceLinkType>

<!-- Service link type for buyer/auctionHouse -->

<slnk:serviceLinkType name="buyerAuctionHouseLT">
  <slnk:role name="auctionHouse">
    <slnk:portType name="tns:buyerPT"/>
  </slnk:role>
  <slnk:role name="buyer">
    <slnk:portType name="tns:buyerAnswerPT"/>
  </slnk:role>
</slnk:serviceLinkType>

<!-- Service link type for auction house/auction
registration service -->

<slnk:serviceLinkType name="auctionHouseAuctionRegistrationServiceLT">
  <slnk:role name="auctionRegistrationService">
    <slnk:portType name="tns:auctionRegistrationPT"/>
  </slnk:role>
  <slnk:role name="auctionHouse">
    <slnk:portType name="tns:auctionRegistrationAnswerPT"/>
  </slnk:role>
</slnk:serviceLinkType>
</definitions>

```

12.3.2. Process

The BPEL4WS definition for the business process offered by the auction house follows:

```

<process name="auctionService"
  targetNamespace="http://www.auction.com"
  containerAccessSerializable="no"
  xmlns:as="http://www.auction.com/wsdl/auctionService"
  xmlns:sref="http://schemas.xmlsoap.org/ws/2002/07/service-reference/"
  xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/">

  <!-- Partners -->

  <partners>
    <partner name="seller"
      serviceLinkType="as:sellerAuctionHouseLT"
      myRole="auctionHouse" partnerRole="seller"/>
    <partner name="buyer"
      serviceLinkType="as:buyerAuctionHouseLT"
      myRole="auctionHouse" partnerRole="buyer"/>
    <partner name="auctionRegistrationService"
      serviceLinkType=

```



```

        "as:auctionHouseAuctionRegistrationServiceLT"
        myRole="auctionHouse"
        partnerRole="auctionRegistrationService"/>
</partners>

<!-- Containers -->

<containers>
  <container name="sellerData" messageType="as:sellerData"/>
  <container name="sellerAnswerData" messageType="as:sellerAnswerData"/>
  <container name="buyerData" messageType="as:buyerData"/>
  <container name="buyerAnswerData" messageType="as:buyerAnswerData"/>
  <container name="auctionRegistrationData"
    messageType="as:auctionDetails"/>
  <container name="auctionRegistrationResponse"
    messageType="as:auctionRegAnswer"/>
</containers>

<!-- Correlation set for correlating buyer and seller request
as well as auction house and auction registration service
exchange -->

<correlationSets>
  <correlationSet name="auctionIdentification"
    properties="as:auctionId"/>
</correlationSets>

<!-- Structure of the business process -->

<sequence>

<!-- Process buyer and seller request in parallel
Either one can create a process instance -->

  <flow>

<!-- Process seller request -->

    <receive name="acceptSellerInformation"
      partner="seller"
      portType="as:sellerPT"
      operation="provide"
      container="sellerData"
      createInstance="yes">
      <correlations>
        <correlation set="auctionIdentification"
          initiation="yes"/>
      </correlations>

    </receive>

<!-- Process buyer request -->

    <receive name="acceptBuyerInformation"
      partner="buyer"
      portType="as:buyerPT"
      operation="provide"
      container="buyerData"
      createInstance="yes">
      <correlations>
        <correlation set="auctionIdentification"
          initiation="yes"/>
      </correlations>
    </receive>

```

```

    </flow>

<!-- Invoke auction registration service
by setting the target service reference
and setting my own service reference for call back
and receiving the answer
Correlation of request and answer is via auction Id -->

    <assign>
        <copy>
            <from>
                <sref:serviceReference>
                    <sref:service name="ars:RegistrationService"
                        xmlns:ars="http://auctionRegistration.com"/>
                </sref:serviceReference>
            </from>
            <to partner="auctionRegistrationService"/>
        </copy>
    </assign>

    <assign>
        <copy>
            <from partner="auctionRegistrationService"
                serviceReference="myRole"/>
            <to container="auctionRegistrationData"
                part="auctionHouseServiceRef"/>
        </copy>
    </assign>

    <invoke name="registerAuctionResults"
        partner="auctionRegistrationService"
        portType="as:auctionRegistrationPT"
        operation="process"
        inputContainer="auctionRegistrationData">
        <correlations>
            <correlation set="auctionIdentification"/>
        </correlations>
    </invoke>

    <receive name="receiveAuctionRegistrationInformation"
        partner="auctionRegistrationService"
        portType="as:auctionRegistrationAnswerPT"
        operation="answer"
        container="auctionRegistrationAnswerData">

        <correlations>
            <correlation set="auctionIdentification"/>
        </correlations>
    </receive>

<!-- Send responses back to seller and buyer -->

    <flow>

<!-- Process seller response by
setting the seller to the service reference provided by the seller
and invoking the response -->

    <sequence>

        <assign>
            <copy>

```

```

        <from container="sellerData"
            part="serviceReference" />
        <to partner="seller" />
    </copy>
</assign>

    <invoke name="respondToSeller"
        partner="seller"
        portType="as:sellerAnswerPT"
        operation="answer"
        inputContainer="sellerAnswerData" />

</sequence>

<!-- Process buyer response by
    setting the buyer to the service reference provided by the buyer
    and invoking the response -->

    <sequence>

        <assign>
            <copy>
                <from container="buyerData"
                    part="serviceReference" />
                <to partner="buyer" />
            </copy>
        </assign>

        <invoke name="respondToBuyer"
            partner="buyer"
            portType="as:buyerAnswerPT"
            operation="answer"
            inputContainer="buyerAnswerData" />

    </sequence>

</flow>

</sequence>

</process>

```

13. Future Directions

BPEL4WS is a work in progress. A number of required features are absent from the specification. These features will be included in future versions. The following sections discuss a few of the salient extensions that will be added.

13.1. Scopes

Scopes provide the capability of grouping activities together and assigning properties to the scope, such as fault handlers and compensation handlers. The following characteristics need to be added to scopes.

13.1.1. Containers

Containers can currently be declared only at the process level, making the entire process the scope in which a container is accessible. In other words, all containers are global. In the future, the declaration of containers within all scopes will be permitted, providing the capability to define containers that are visible and accessible only within local scopes. This simplifies, among other things, the writing of compensation handlers.

Local containers also make it attractive to transmit data through links in addition to synchronization signals. The combination of local containers and link-based data transmission is expected to greatly reduce the need for explicit

concurrency control using serializable scopes.

Further analysis of the issues raised by concurrent access to shared containers will reveal opportunities for feature simplification and improved avoidance of deadlock and interference. The language will be improved in these respects in future versions.

13.1.2. Event Handlers

A business process often needs to be prepared for unscheduled business events that can occur at any time during a specified section of the process execution. Scopes are used to delineate process execution sections, and a notion of event handlers will be added to scopes to model processing of asynchronous events. The whole process might have a set of event handlers that are available throughout process execution. Two types of events will be supported. First, events will be supported that correspond to an inbound request/response or one-way operation in WSDL. For example, a status query is likely to be a request/response operation, whereas a cancellation might be a one-way operation. Second, events can be timer alarms that go off after user-set times. Event handlers are expected to be considered a part of the normal behavior of the scope, unlike fault and compensation handlers.

13.1.3. Overlapping Scopes

Scopes are currently nested within each other. Because scopes are associated with many different properties, it is often the case that scopes with different properties have some activities in common. This might require that scopes are allowed to overlap. This is an area for further investigation.

13.1.4. Atomic Scopes

Business processes are often "transactional" in the sense that they perform activities that require predictable and consistent outcomes. Although durability is in some sense a private implementation matter, there is often a requirement to perform a set of activities in such a way that either all complete successfully or they are not performed (or their effects erased). For example, receiving a request, performing some data manipulation, and then answering the request can often be coupled together in this way. This is usually referred to as atomicity, and the ability to define scopes with the atomicity property will be added in the future.

13.1.5. Compensation

Compensation is the means by which the effects of parts of as well as the whole business process can be undone. The current specification provides for basic support. Further refinements for compensation behavior need to be added. For example, compensations are currently entirely self-contained. They are not influenced by the current live state of the process instance in which they run, nor are they able to influence that state. This is clearly unrealistic in general. Input and output parameters will be added to compensation handlers to allow the live state to be influenced in both directions in a controlled way. Other potential enhancements include compensation retry in case of failure, and new default compensation modes in addition to the "reverse order" mode currently supported.

13.2. Lifecycle and Query

13.2.1. Suspend/Resume

Sometimes it is necessary to suspend (halt) the execution of a business process for some time or until explicitly resumed by an appropriate action. The suspend/resume type of activities are mainly intended to be used in event handlers to suspend and resume the processing of either the complete business process or of a particular scope only. Also, the terminate activity currently terminates the whole process. There are situations where it is sufficient to only terminate the processing within a particular scope.

13.2.2. Query

The current draft does not support the capability of querying the state of a business process, something that is needed when business processes are carried out. The specification of options, for example the details of the information provided, must be supported.

13.3. Service Composition

Because a BPEL4WS process in general provides and consumes multiple Web Service interfaces, such a process can be thought of as composing a set of Web Services from other Web Services. The relationship concepts enabled by service links are also important in establishing patterns of composition. This is clearly an area that needs further development, and future versions of BPEL4WS will play a part in these developments.

13.4. Relationship to WS-Transaction Specification

BPEL4WS defines a notion of long-running business transactions (LRTs) based on the syntactic construct of *scopes*. For environments where a business process is distributed or spans vendor implementations, it is expected that the WS-Transaction specification [12] will provide the mechanism to:

- Define a global agreement context for a distributed scope that can be understood across the business transaction.
- Allow participants across the distributed process to register for fault handling and compensation notifications (according to the behavior outlined for an LRT).

See [Appendix C](#) for a detailed model of BPEL4WS LRTs based on WS-Transaction concepts.

Additionally, it is often the case that multiple business process instances need to work together to achieve consistent completion of a shared unit of work. Coordination of individual LRTs in these processes can also require the use of coordination protocols listed in the WS-Transaction specification (according to the agreement behaviors outlined in WS-Transaction).

14. Security Considerations

Because messages can be modified or forged, it is strongly RECOMMENDED that business process implementations use WS-Security to ensure messages have not been modified or forged while in transit or while residing at destinations. Similarly, invalid or expired messages could be re-used or message headers not specifically associated with the specific message could be referenced. Consequently, when using WS-Security, signatures MUST include the semantically significant headers and the message body (as well as any other relevant data) so that they cannot be independently separated and re-used.

Messaging protocols used to communicate among business processes are subject to various forms of replay attacks. In addition to the mechanisms listed above, messages SHOULD include a message timestamp (as described in WS-Security) within the signature. Recipients can use the timestamp information to cache the most recent messages for a business process and detect duplicate transmissions and prevent potential replay attacks.

It should also be noted that business process implementations are subject to various forms of denial-of-service attacks. Implementers of business process execution systems compliant with this specification should take this into account.

15. Acknowledgments

Achille Fokoue, Ashok Malhotra, and Bob Schloss for their help with developing and verifying the XML Schemas.

Tony Andrews and Marc Levy for their help in defining abstract processes.

Tony Hoare and Marc Shapiro for thoughtful comments on the language concepts.

Jonathan Marsh for suggesting the generalization of the dependency on external (query and expression) languages.

Tom Freund and Tony Storey for inducing us to precisely define the relationship with the coordination framework in WS-Transaction.

Martin Nally for his help on improving the usability of the language.

16. References

- 2 W3C Note "[Simple Object Access Protocol \(SOAP\) 1.1](#)"
- 3 W3C Note "[Web Services Definition Language \(WSDL\) 1.1](#)"
- 4 Industry Initiative "[Universal Description, Discovery and Integration](#)"
- 5 XLANG: [Web Services for Business Process Design](#)
- 6 WSFL: [Web Service Flow Language 1.0](#)
- 7 W3C Proposed Recommendation "[XML Schema Part 1: Structures](#)"
- 8 W3C Proposed Recommendation "[XML Schema Part 2: Datatypes](#)"
- 9 W3C Recommendation "[XML Path Language \(XPath\) Version 1.0](#)"
- 10 "Sagas," H. Garcia-Molina and K. Salem, Proc. ACM SIGMOD (1987).
- 11 "Trends in systems aspects of database management," I.L. Traiger, Proc. 2nd Intl. Conf. on Databases (ICOD-2), Wiley & Sons 1983.
- 12 "Web Services Transaction", IBM & Microsoft, 2002. [Web Services Transaction \(WS-Transaction\)](#)
- 13 "Key words for use in RFCs to Indicate Requirement Levels," [RFC 2119](#), S. Bradner, Harvard University, March 1997.
- 14 "Uniform Resource Identifiers (URI): Generic Syntax," [RFC 2396](#), T. Berners-Lee, R. Fielding, L. Masinter, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

17. Appendices

17.1. Appendix A – Standard Faults

The following list specifies the standard faults defined within the BPEL4WS specification. All these faults are named within the BPEL4WS namespace standard prefix bpws: corresponding to URI "http://schemas.xmlsoap.org/ws/2002/07/business-process/".

Fault name	Reason
selectionFailure	Thrown when a selection operation performed either in a function such as bpws:getContainerData, or in an assignment, encounters an error.
conflictingReceive	Thrown when more than one receive activity or equivalent (currently, onMessage branch in a pick activity) are enabled simultaneously for the same partner, port type, and operation.
conflictingRequest	Thrown when more than one synchronous inbound request from the same partner for a particular port type and operation are active.
mismatchedAssignmentFailure	Thrown when incompatible types are encountered in an assign activity.
joinFailure	Thrown when the join condition of an activity evaluates to false.
forcedTermination	Thrown as a result of a fault in an enclosing scope.

correlationViolation	Thrown when the contents of the messages that are processed in an <code>invoke</code> , <code>receive</code> , or <code>reply</code> activity do not match specified correlation information.
uninitializedContainer	Thrown when there is an attempt to access the value of an uninitialized part in a message container.
repeatedCompensation	Thrown when an installed compensation handler is invoked more than once.

17.2. Appendix B. - Attributes and Defaults

The following list specifies the defaults for all standard attributes at the process and activity level. The table does not include activity-specific attributes (such as `partner` in an `invoke` activity).

Parameter	Default
<code>queryLanguage</code>	http://www.w3.org/TR/1999/REC-xpath-19991116
<code>expressionLanguage</code>	http://www.w3.org/TR/1999/REC-xpath-19991116
<code>suppressJoinFailure</code>	No
<code>containerAccessSerializable</code>	No
<code>abstractProcess</code>	No
<code>initiation</code>	No
<code>pattern</code>	No default
<code>createInstance</code>	No
<code>enableInstanceCompensation</code>	No
<code>joinCondition</code>	Disjunction of the status of the incoming links

17.3. Appendix C – Coordination Protocol

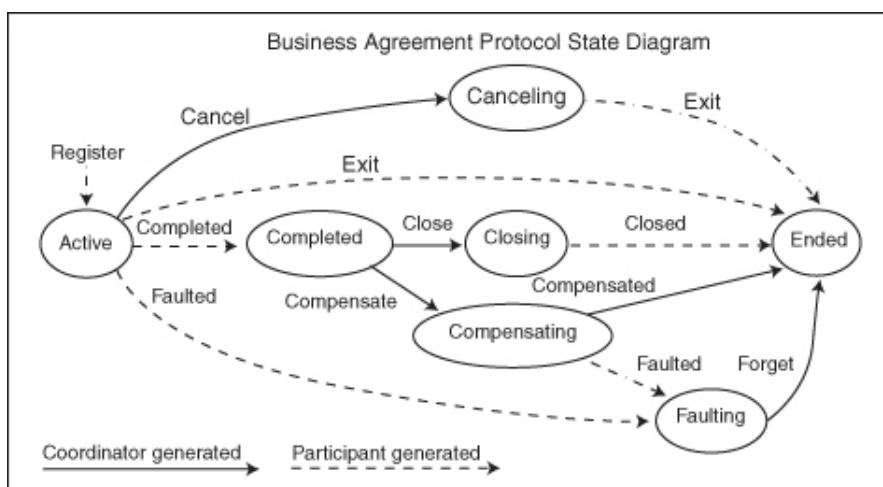
It is valuable to express the fault and compensation handling relationship between scopes by using the protocol framework of WS-Transaction [12]. Specifically, this section shows how the relationship between an enclosing scope and each of its nested scopes can be modeled using the BusinessAgreement protocol defined in the WS-Transaction specification. The BusinessAgreement protocol is designed to enable distributed coordination of business activities. BPEL4WS usage of the protocol makes the assumption of localized behavior in a single service, and as a result several of the features of the protocol, including the acknowledgement signal *Forget*, and the *Error* and *Replay* messages, are not actually needed in BPEL4WS.

17.3.1. Coordination Protocol for BPEL4WS Scopes

1. The execution of a nested scope may complete successfully. In this case a compensation handler is installed for the nested scope. This is modeled with a *completed* signal from the nested scope to its parent scope.
2. The execution of a nested scope may encounter a fault internally. In this case the scope always terminates unsuccessfully.
 - a. If the fault handler rethrows a fault to its enclosing scope, this is modeled as a *faulted* signal from the nested scope to its parent scope.
 - b. If the fault is handled and not rethrown, the scope exits gracefully from the work of its parent scope. This is modeled as an *exit* signal from the nested scope to its parent scope.
3. After a nested scope has completed, (a fault or compensation handler for) the parent scope may ask it to compensate itself by invoking its compensation handler. The compensate action is modeled with a *compensate* signal from the parent scope to the nested scope.
4. Upon successful completion of the compensation, the nested scope sends the *compensated* signal to its parent scope.
5. The compensation handler may itself fault internally. In this case:

- a. If the fault is not handled by a scope within the compensation handler, it is rethrown to the parent scope. This is modeled as a *faulted* signal from the nested scope to its parent scope.
 - b. If the fault is handled and not rethrown, we assume that the compensation was able to complete successfully. In this case the nested scope sends the *compensated* signal to its parent scope.
6. If there is a fault in the parent scope independent of the work of the nested scope, the parent scope will ask the nested scope to prematurely abandon its work by sending a *cancel* signal.
 7. The nested scope, upon receiving the cancel signal, will interrupt and terminate its execution (exactly as though there were an internal fault), and return an *exit* signal to the parent.
 8. Finally, when a parent scope decides that the compensation for a completed nested scope is not needed anymore it sends a *close* signal to the nested scope. After discarding the compensation handler the nested scope responds with a *closed* signal.
 9. In case there is a race between the *completed* signal from the nested scope and the *cancel* signal from the parent scope, the *completed* signal wins, i.e., the nested scope is deemed to have completed and the *cancel* signal is ignored.
 10. In case a *cancel* signal is sent to a nested scope that has already faulted internally, the *cancel* signal is ignored and the scope will eventually send either a *faulted* or an *exit* signal to the parent.

Figure 2. BusinessAgreement Protocol State Diagram



The BusinessAgreement protocol state diagram in Figure 2 above summarizes the preceding discussion. In the diagram, the parent (enclosing) scope generates *cancel*, *compensate*, and *close* signals and the nested scope generates *completed*, *faulted*, *exit*, and *closed* signals. It is important to emphasize that the states represent the state of the relationship between the parent scope and one specific nested scope. However, it is very nearly the case that the states represent the state of the nested scope itself, except in case of signal races. Note that the signal races discussed in points 9 and 10 above are not reflected in the diagram since the diagram only reflects real protocol states.

17.4. Appendix D - XSD Schemas

17.4.1. BPEL4WS Schema

```
<?xml version='1.0' encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  targetNamespace="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
  elementFormDefault="qualified">

  <import namespace="http://schemas.xmlsoap.org/wSDL/" />

  <complexType name="tExtensibleElements">
    <annotation>
      <documentation>This type is extended by other component types
        to allow elements and attributes
        from other namespaces to be added. .</documentation>
    </annotation>
  </complexType>
</schema>
```



```

    <sequence>
      <any namespace="##other" minOccurs="0"
        maxOccurs="unbounded" processContents="lax" />
    </sequence>
    <anyAttribute namespace="##other" processContents="lax" />
  </complexType>

  <element name="process" type="bpws:tProcess" />
  <complexType name="tProcess">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <sequence>
          <element name="partners" type="bpws:tPartners" minOccurs="0"
            />
          <element name="containers" type="bpws:tContainers" minOccurs="0"
            />
          <element name="correlationSets" type="bpws:tCorrelationSets"
            minOccurs="0" />
          <element name="faultHandlers" type="bpws:tFaultHandlers"
            minOccurs="0" />
          <element name="compensationHandler"
            type="bpws:tCompensationHandler" minOccurs="0" />
          <group ref="bpws:activity" />
        </sequence>
        <attribute name="name" type="NCName" />
        <attribute name="targetNamespace" type="anyURI" />
        <attribute name="suppressJoinFailure" type="bpws:tBoolean"
          default="no" />
        <attribute name="containerAccessSerializable" type="bpws:tBoolean"
          default="no" />
        <attribute name="enableInstanceCompensation" type="bpws:tBoolean"
          default="no" />
        <attribute name="abstractProcess" type="bpws:tBoolean"
          default="no" />
      </extension>
    </complexContent>
  </complexType>

  <group name="activity">
    <choice>
      <element name="empty" type="bpws:tEmpty" />
      <element name="invoke" type="bpws:tInvoke" />
      <element name="receive" type="bpws:tReceive" />
      <element name="reply" type="bpws:tReply" />
      <element name="assign" type="bpws:tAssign" />
      <element name="wait" type="bpws:tWait" />
      <element name="throw" type="bpws:tThrow" />
      <element name="terminate" type="bpws:tTerminate" />
      <element name="flow" type="bpws:tFlow" />
      <element name="switch" type="bpws:tSwitch" />
      <element name="while" type="bpws:tWhile" />
      <element name="sequence" type="bpws:tSequence" />
      <element name="pick" type="bpws:tPick" />
      <element name="scope" type="bpws:tScope" />
    </choice>
  </group>

  <complexType name="tPartners">
    <complexContent>
      <extension base="bpws:tExtensibleElements">
        <sequence>

```

```

        <element name="partner" type="bpws:tPartner"
            minOccurs="1" maxOccurs="unbounded" />
    </sequence>
</extension>
</complexContent>
</complexType>

<complexType name="tPartner">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <attribute name="name" type="NCName" use="required" />
            <attribute name="serviceLinkType" type="QName"
                use="required" />
            <attribute name="myRole" type="NCName" />
            <attribute name="partnerRole" type="NCName" />
        </extension>
    </complexContent>
</complexType>

<complexType name="tFaultHandlers">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <sequence>
                <element name="catch" type="bpws:tCatch"
                    minOccurs="0" maxOccurs="unbounded" />
                <element name="catchAll"
                    type="bpws:tActivityOrCompensateContainer"
                    minOccurs="0" />
            </sequence>
        </extension>
    </complexContent>
</complexType>

<complexType name="tCatch">
    <complexContent>
        <extension base="bpws:tActivityOrCompensateContainer">
            <attribute name="faultName" type="QName"
                use="optional" />
            <attribute name="faultContainer" type="NCName"
                use="optional" />
        </extension>
    </complexContent>
</complexType>

<complexType name="tActivityContainer">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <sequence>
                <group ref="bpws:activity" />
            </sequence>
        </extension>
    </complexContent>
</complexType>

<complexType name="tActivityOrCompensateContainer">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <choice>
                <group ref="bpws:activity" />
                <element name="compensate" type="bpws:tCompensate" />
            </choice>
        </extension>
    </complexContent>
</complexType>

```

```

    </complexContent>
</complexType>

<complexType name="tOnMessage">
  <complexContent>
    <extension base="bpws:tExtensibleElements">

      <sequence>
        <element name="correlations" type="bpws:tCorrelations"
minOccurs="0"/>
        <group ref="bpws:activity"/>
      </sequence>
      <attribute name="partner" type="NCName" use="required"/>
      <attribute name="portType" type="QName" use="required"/>
      <attribute name="operation" type="NCName" use="required"/>
      <attribute name="container" type="NCName" use="required"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tOnAlarm">
  <complexContent>
    <extension base="bpws:tActivityContainer">
      <attribute name="for" type="bpws:tDuration-expr"
use="optional"/>
      <attribute name="until" type="bpws:tDeadline-expr"
use="optional"/>
    </extension>
  </complexContent>
</complexType>

<complexType name="tCompensationHandler">
  <complexContent>
    <extension base="bpws:tActivityOrCompensateContainer"/>
  </complexContent>
</complexType>

<complexType name="tContainers">
  <complexContent>
    <extension base="bpws:tExtensibleElements">

      <sequence>
        <element name="container" type="bpws:tContainer"
maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tContainer">

  <!-- container does not allow extensibility elements
because otherwise its content model would be non-
deterministic -->
  <sequence>
    <element name="message" type="wsdl:tMessage"
minOccurs="0">
      <unique name="part">
        <selector xpath="wsdl:part"/>
        <field xpath="@name"/>
      </unique>

```

```

        </element>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="messageType" type="QName"
use="optional"/>
    <anyAttribute namespace="##other" processContents="lax"/>

</complexType>

<complexType name="tCorrelationSets">
    <complexContent>
        <extension base="bpws:tExtensibleElements">

            <sequence>
                <element name="correlationSet" type="bpws:tCorrelationSet"
maxOccurs="unbounded"/>
            </sequence>
        </extension>
    </complexContent>

</complexType>

<complexType name="tCorrelationSet">
    <complexContent>
        <extension base="bpws:tExtensibleElements">

            <attribute name="properties" use="required">
                <simpleType>
                    <list itemType="QName"/>
                </simpleType>
            </attribute>
            <attribute name="name" type="NCName" use="required"/>
        </extension>
    </complexContent>

</complexType>

<complexType name="tActivity">
    <complexContent>
        <extension base="bpws:tExtensibleElements">

            <sequence>
                <element name="target" type="bpws:tTarget" minOccurs="0"
maxOccurs="unbounded"/>
                <element name="source" type="bpws:tSource" minOccurs="0"
maxOccurs="unbounded"/>
            </sequence>
            <attribute name="name" type="NCName" use="optional"/>
            <attribute name="joinCondition" type="bpws:tBoolean-expr"
use="optional"/>
            <attribute name="suppressJoinFailure" type="bpws:tBoolean"
use="optional"/>
        </extension>
    </complexContent>

</complexType>

<complexType name="tSource">
    <complexContent>
        <extension base="bpws:tExtensibleElements">

```

```

        <attribute name="linkName" type="NCName" use="required"/>
        <attribute name="transitionCondition" type="bpws:tBoolean-expr"
            use="optional"/>
    </extension>
</complexContent>

</complexType>

<complexType name="tTarget">
    <complexContent>
        <extension base="bpws:tExtensibleElements">

            <attribute name="linkName" type="NCName" use="required"/>
        </extension>
    </complexContent>

</complexType>

<complexType name="tEmpty">
    <complexContent>
        <extension base="bpws:tActivity"/>
    </complexContent>

</complexType>

<complexType name="tCorrelations">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <sequence>
                <element name="correlation" type="bpws:tCorrelation"
minOccurs="1
                " maxOccurs="unbounded" />
            </sequence>
        </extension>
    </complexContent>

</complexType>

<complexType name="tCorrelation">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <attribute name="set" type="NCName" use="required"/>
            <attribute name="initiation" type="bpws:tBoolean"
                use="optional" default="no"/>
            <attribute name="pattern" use="optional">
                <simpleType>
                    <restriction base="string">
                        <enumeration value="in" />
                        <enumeration value="out" />
                        <enumeration value="out-in" />
                    </restriction>
                </simpleType>
            </attribute>
        </extension>
    </complexContent>

</complexType>

<complexType name="tInvoke">
    <complexContent>
        <extension base="bpws:tActivity">
            <sequence>
                <element name="correlations" type="bpws:tCorrelations"
                    minOccurs="0" maxOccurs="1"/>
                <element name="catch" type="bpws:tCatch" minOccurs="0"
                    maxOccurs="unbounded"/>
            </sequence>
        </extension>
    </complexContent>

```

```

        <element name="catchAll"
type="bpws:tActivityOrCompensateContainer"
        minOccurs="0"/>
        <element name="compensationHandler"
type="bpws:tCompensationHandler" minOccurs="0"/>
    </sequence>
    <attribute name="partner" type="NCName" use="required"/>
    <attribute name="portType" type="QName" use="required"/>
    <attribute name="operation" type="NCName" use="required"/>
    <attribute name="inputContainer" type="NCName" use="required"/>
    <attribute name="outputContainer" type="NCName"
use="optional"/>
    </extension>
</complexContent>
</complexType>

<complexType name="tReceive">
    <complexContent>
        <extension base="bpws:tActivity">
            <sequence>
                <element name="correlations" type="bpws:tCorrelations"
minOccurs="0"/>
            </sequence>
            <attribute name="partner" type="NCName" use="required"/>
            <attribute name="portType" type="QName" use="required"/>
            <attribute name="operation" type="NCName" use="required"/>
            <attribute name="container" type="NCName" use="required"/>
            <attribute name="createInstance" type="bpws:tBoolean"
use="optional"/>
        </extension>
    </complexContent>
</complexType>

<complexType name="tReply">
    <complexContent>
        <extension base="bpws:tActivity">
            <sequence>
                <element name="correlations" type="bpws:tCorrelations"
minOccurs="0"/>
            </sequence>
            <attribute name="partner" type="NCName" use="required"/>
            <attribute name="portType" type="QName" use="required"/>
            <attribute name="operation" type="NCName" use="required"/>
            <attribute name="container" type="NCName" use="required"/>
            <attribute name="faultName" type="QName" use="optional"/>
        </extension>
    </complexContent>
</complexType>

<complexType name="tAssign">
    <complexContent>
        <extension base="bpws:tActivity">
            <sequence>
                <element name="copy" type="bpws:tCopy" minOccurs="1"
maxOccurs="unbounded"/>
            </sequence>
        </extension>
    </complexContent>
</complexType>

<complexType name="tCopy">
    <complexContent>
        <extension base="bpws:tExtensibleElements">

```

```

        <sequence>
            <element ref="bpws:from" />
            <element ref="bpws:to" />
        </sequence>
    </extension>
</complexContent>
</complexType>

<element name="from" type="bpws:tFrom" />
<complexType name="tFrom">
    <complexContent>
        <extension base="bpws:tExtensibleElements">

            <attribute name="container" type="NCName" />
            <attribute name="part" type="NCName" />
            <attribute name="select" type="string" />
            <attribute name="property" type="QName" />
            <attribute name="partner" type="NCName" />
            <attribute name="expression" type="string" />
            <attribute name="opaque" type="bpws:tBoolean" />
        </extension>
    </complexContent>
</complexType>
<element name="to">
    <complexType>
        <complexContent>
            <restriction base="bpws:tFrom">
                <attribute name="expression" type="string"
                    use="prohibited" />
                <attribute name="opaque" type="bpws:tBoolean"
                    use="prohibited" />
            </restriction>
        </complexContent>
    </complexType>
</element>

<complexType name="tWait">
    <complexContent>
        <extension base="bpws:tActivity">
            <attribute name="for" type="bpws:tDuration-expr" use="optional" />
            <attribute name="until" type="bpws:tDeadline-expr" use="optional" />
        </extension>
    </complexContent>
</complexType>

<complexType name="tThrow">
    <complexContent>
        <extension base="bpws:tActivity">
            <attribute name="faultName" type="QName" use="required" />
            <attribute name="faultContainer" type="NCName" />
        </extension>
    </complexContent>
</complexType>

<complexType name="tCompensate">
    <complexContent>
        <extension base="bpws:tActivity">
            <attribute name="scope" type="NCName" use="optional" />
        </extension>
    </complexContent>
</complexType>

```

```

</complexType>

<complexType name="tTerminate">
  <complexContent>
    <extension base="bpws:tActivity" />
  </complexContent>
</complexType>

<complexType name="tFlow">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="links" type="bpws:tLinks" minOccurs="0" />
        <group ref="bpws:activity" maxOccurs="unbounded" />
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tLinks">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <element name="link" type="bpws:tLink"
          maxOccurs="unbounded" />
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tLink">
  <complexContent>
    <extension base="bpws:tExtensibleElements">
      <attribute name="name" type="NCName" use="required" />
    </extension>
  </complexContent>
</complexType>

<complexType name="tSwitch">
  <complexContent>
    <extension base="bpws:tActivity">
      <sequence>
        <element name="case" maxOccurs="unbounded">
          <complexType>
            <complexContent>
              <extension base="bpws:tActivityContainer">
                <attribute name="condition" type="bpws:tBoolean-
expr"
                    use="required" />
              </extension>
            </complexContent>
          </complexType>
        </element>
        <element name="otherwise" type="bpws:tActivityContainer"
minOccurs="0" />
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="tWhile">
  <complexContent>

```



```

    <extension base="bpws:tActivity">
        <sequence>
            <group ref="bpws:activity" />
        </sequence>
        <attribute name="condition" type="bpws:tBoolean-expr"
            use="required" />
    </extension>
</complexContent>
</complexType>

<complexType name="tSequence">
    <complexContent>
        <extension base="bpws:tActivity">
            <sequence>
                <group ref="bpws:activity" maxOccurs="unbounded" />
            </sequence>
        </extension>
    </complexContent>
</complexType>

<complexType name="tPick">
    <complexContent>
        <extension base="bpws:tActivity">
            <sequence>
                <element name="onMessage" type="bpws:tOnMessage"
                    maxOccurs="unbounded" />
                <element name="onAlarm" type="bpws:tOnAlarm"
                    minOccurs="0" maxOccurs="unbounded" />
            </sequence>
            <attribute name="createInstance" type="bpws:tBoolean"
                use="optional" />
        </extension>
    </complexContent>
</complexType>

<complexType name="tScope">
    <complexContent>
        <extension base="bpws:tActivity">
            <sequence>
                <element name="faultHandlers" type="bpws:tFaultHandlers"
minOccurs="0" />
                <element name="compensationHandler"
                    type="bpws:tCompensationHandler" minOccurs="0" />
                <group ref="bpws:activity" />
            </sequence>
            <attribute name="containerAccessSerializable" type="bpws:tBoolean"
                use="required" />
        </extension>
    </complexContent>
</complexType>

<simpleType name="tListOfNCNames">
    <list itemType="NCName" />
</simpleType>

<simpleType name="tBoolean-expr">
    <restriction base="string" />
</simpleType>

<simpleType name="tDuration-expr">
    <restriction base="string" />
</simpleType>

<simpleType name="tDeadline-expr">

```

```

    <restriction base="string"/>
</simpleType>

<simpleType name="tBoolean">
    <restriction base="string">
        <enumeration value="yes"/>
        <enumeration value="no"/>
    </restriction>
</simpleType>
</schema>

```

17.4.2. Service Link Type Schema

```

<?xml version='1.0' encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/07/service-link/"
    targetNamespace="http://schemas.xmlsoap.org/ws/2002/07/service-link/"
    elementFormDefault="qualified">

    <element name="serviceLinkType" type="slnk:tServiceLinkType"/>

    <complexType name="tServiceLinkType">
        <sequence>
            <element name="role" type="slnk:tRole" minOccurs="1" maxOccurs="2"/>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
    </complexType>

    <complexType name="tRole">
        <sequence>
            <element name="portType" minOccurs="1"
                maxOccurs="unbounded">
                <complexType>
                    <attribute name="name" type="QName" use="required"/>
                </complexType>
            </element>
        </sequence>
        <attribute name="name" type="NCName" use="required"/>
    </complexType>
</schema>

```

17.4.3. Service References Schema

```

< ?xml version='1.0' encoding= " UTF-8 " ? >
< schema xmlns= " http://www.w3.org/2001/XMLSchema "
    xmlns:sref= " http://schemas.xmlsoap.org/ws/2002/07/service-reference/ "
    xmlns:wSDL= " http://schemas.xmlsoap.org/wSDL/ "
    targetNamespace= " http://schemas.xmlsoap.org/ws/2002/07/service-reference/ "
    elementFormDefault= " qualified " >

    < import namespace= " http://schemas.xmlsoap.org/wSDL/ " / >

    < element name= " serviceReference " type= " sref:serviceReferenceType " / >

    < complexType name= " serviceReferenceType " >
        < sequence >
            < element ref= " wSDL:definitions " minOccurs= " 0 " / >
            < element name= " service " type= " QName " / >
            < element name= " referenceProperties " minOccurs= " 0 " >
                < complexType >

```

```

        < sequence >
            < element name= " property " type= " sref:referencePropertyType "
                maxOccurs= " unbounded " / >
        < /sequence >
    < /complexType >
< /element >
< /sequence >
< /complexType >

< complexType name= " referencePropertyType " >
    < sequence >
        < any namespace= " ##other " minOccurs= " 0 " / >
    < /sequence >
    < attribute name= " name " type= " QName " / >
< /complexType >
< /schema >

```

17.4.4. Message Properties Schema

```

<?xml version='1.0' encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
    xmlns:wsbp="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
    elementFormDefault="qualified">

    <element name="property">
        <complexType>
            <attribute name="name" type="NCName"/>
            <attribute name="type" type="QName"/>
        </complexType>
    </element>

    <element name="propertyAlias">
        <complexType>
            <attribute name="propertyName" type="QName" />
            <attribute name="messageType" type="QName"/>
            <attribute name="part" type="NCName"/>
            <attribute name="select" type="string"/>
        </complexType>
    </element>
</schema>

```



[IBM home](#) | [Products & services](#) | [Support & downloads](#) | [My account](#)

[Select a country](#)



[Go to developerWorks](#)

Our apologies...

The document you requested does not exist at the following location:

Please correct the URL above and click "Go" or select one of the links shown below. You can also use the search function in the masthead to find the information you're looking for.

If you believe you have received this message in error, please use the [developerWorks feedback form](#) to tell us about this error.

Note: Some web browser versions have problems with URLs that do not end in a slash "/" . You may wish to try the URL again and include a slash "/" at the end.

404 multifail

[About IBM](#) | [Privacy](#) | [Legal](#) | [Contact](#)