# BPMI.org

# Business Process Modeling Language (BPML)

Working Draft 0.4

3/8/2001

Author

Assaf ARKIN (Intalio, Inc.) arkin@intalio.com

Editor

Ashish AGRAWAL (Intalio, Inc.) ashish@intalio.com

Please send comments on this Draft to: bpml-spec-comments@bpmi.org

Business Process Modeling Language (BPML) Specification ("Specification")
Draft: 0.4
Status: Working Draft
Release: 8 March 2001

# Table of Contents

# Section 1 Introduction

A process is a specific ordering of work activities across time and place, with a beginning, an end, and clearly defined inputs and outputs: a structure for action[1]. Business processes are both internal and external to autonomous business entities, and drive their collaboration to achieve shared business goals by enabling highly fluid process networks. Such business goals include end-to-end efficiency, transformation empowerment, and value management.

Business processes are adaptive structures for action through which many participants—IT systems, applications, users, partners, and other processes—play a variety of roles. Business Process Management enables the collaboration of such participants in a reliable, scalable, and secure manner, by supporting dynamic process topologies that allow the boundary between processes and participants to be determined on-the-fly by long-term and real-time business goals, while retaining synchronized public interfaces associated with trading partner agreements.

The Business Process Management Initiative (BPMI.org) has been created to develop, publish, maintain, and promote a common meta-language (BPML) enabling all participants involved in the process design, deployment, execution, maintenance, and optimization to manage business activities in a process-oriented fashion, while preserving the integrity of end-to-end business processes over their entire lifecycle.

[1] Thomas H. Davenport, Mission Critical: Realizing the Promise of Enterprise Systems, Harvard Business School Press.

## 1.1 Status of this Document

This document is the fourth version of the BPML working draft to be submitted for comments by members of the BPMI initiative on March 8, 2001.

The working draft is expected to keep pace with the working drafts of the XML Schema specification until it evolves into a W3C recommendation. The October 24th 2000 XML Schema candidate release is the basis for the schemas presented in this working draft.

## 1.2 Organization

- Section 2, Concepts, introduces the reader to the key concepts underlying BPML.
- Section 3, BPML by Example, introduces BPML through an example that models trouble ticket management processes between a customer and a service provider.
- Section 4, BPML Conventions, covers the terminology and schema conventions used in BPML.
- Section 5, Process Definition, specifies how processes are defined in BPML, including message, participant, and activity definitions.
- Section 6, Process Data, specifies the manner in which instance data is managed within the context of a process.
- Section 7, Transaction And Exceptions, specifies the transaction models and fault handling behaviors in BPML processes.
- Section 8, BPML Elements, provides a reference of the BPML elements as defined in the BPML XML schema.

## 1.3 Acknowledgments

The editor would like to acknowledge contributions from Alex BOISVERT (Intalio, Inc.), Jean-Jacques DUBRAY (eXcelon Corporation), Antoine LONJON (MEGA International), Riad MOHAMMED (Intalio, Inc.), and Howard N. SMITH (CSC), who helped shape this version of the specification.

Section 2

# Concepts

BPML defines a business process as an interaction between participants and the execution of activities according to a defined set of rules in order to achieve a common goal.

This section lists the requirements that were the basis for BPML and introduces the reader to the key concepts that make BPML a distinguished business process modeling language.

## 2.1 Requirements

While enterprise business process semantics are partially represented in other models, such as the WfMC Workflow model, RosettaNet PIPs, and UML diagrams, we have defined BPML in order to cover a wider set of requirements. In particular:

- BPML must enable the coordination of collaborative business processes among trading partners, and abstract standard business-to-business protocols such as RosettaNet, ebXML, and BizTalk.
- BPML must integrate existing applications as process components, and knit process components together to address new business value propositions.
- BPML must enable the interleaving of processes and transactions that execute along independent lifelines.
- BPML must enable the modeling of business processes that are totally independent of the back-office systems or business-to-business protocol.
- BPML must address business process reliability in mission-critical deployments by explicit exception handling, accommodating for time-out and human intervention, and guaranteeing that processes maintain a consistent state.
- BPML must enable process implementations to change across systems, over time, and dynamically in response to changing conditions.
- BPML must enable different Business Process Management Systems to exchange process models and share a common process repository.

- BPML must express data in a rich format that can declare the structure, relations and types of its data members.
- BPML must enable the different types of activities that can occur in enterprise business processes, including:
  - Information retrieval
  - Information update and dissemination
  - Synchronous and asynchronous communication
  - Short and long-lived transactions
  - Decision-making
  - Interaction with users
  - Access to local and remote resources

## 2.2 Messages

Message exchange is fundamental to collaborative e-Business protocols such as RosettaNet, ebXML, and BizTalk. BPML employs a message-based model in which all participants in the process interact through the exchange of messages, and the process defines the manner in which messages flow between participants, as well as the information conveyed in each message.

Each process includes a definition of all messages communicated between the process and its participants. Messages that are specific to a process are included in the process definition, while messages that are specific to a participant are imported from the process definition of that participant.

XML Schema is used to define the structure and type of the message content. The use of XML Schema as a type definition language does not restrict messages to XML content. Message definitions are meta data and can be used to represent information, tasks, and even material goods, such as products shipped from a supplier to the buyer.

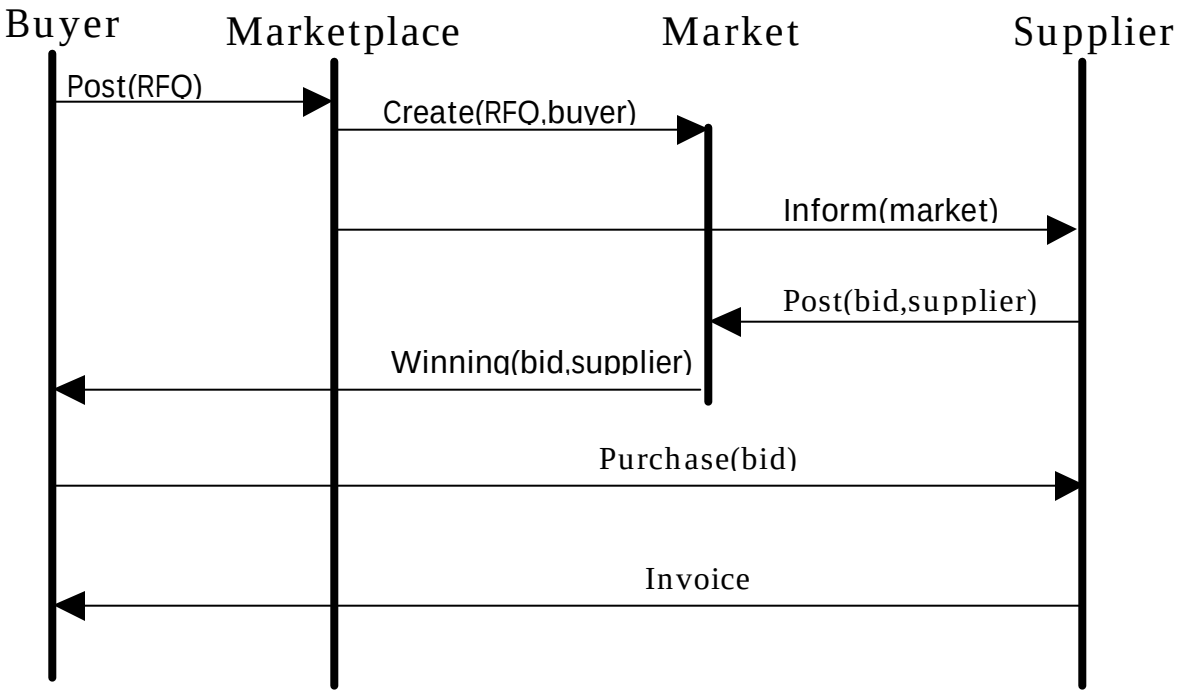
## 2.3 Participants

The participants of a process are entities with which the process interacts. Participants in a process are varied and include IT systems, applications, users, partners, and other processes. The process definition reflects two types of participants: static and dynamic. Static participants are determined when the process is modeled. They are known in

advance and the process will always communicate with the same set of static participants. Abstraction is achieved through the use of organizational roles, business channels, and generic application interfaces. Examples of such participants are "sales manager", "marketplace", and "billing service".

Dynamic participants are unknown at the time the process is modeled. They are communicated from a participant to the process (or vice versa) before being enlisted in the process. Dynamic participants allow the process to take advantage of an ever-changing business environment and emerging collaboration models, such as marketplaces and business advertising services. Support for dynamic participants sets BPML apart from traditional business process modeling techniques, which are based on a static process topology.

The following scenario illustrates the manner in which processes interact with each other dynamically, by leveraging the ability to communicate processes as participants.




Buyer, marketplace and supplier processes interacting by Communicating processes as participants.

When the buyer posts an RFQ (Request For Quote) to a marketplace, the marketplace instantiates a new market process to find the best matching offer. Different market processes can be used to locate the best match (e.g. standard or reverse auction, best price, or best terms).

The marketplace identifies a suitable list of suppliers that can provide the goods and inform each of them of the existence of a new market. Communicating the market process to the supplier is important, as no two markets are alike.

When the market closes, the winning bid is sent back to the buyer, informing the buyer which supplier has placed the best bid. The buyer can then interact directly with the supplier to complete the purchase.

## 2.4 Activities

Processes are based on the execution of activities and the flow of information across activities and between activities and participants.

Simple activities are used to model the consumption and production of messages, tasks, data, or goods. They also model operations or actions, and communication of failure.

The consume activity represents the acceptance and consumption of a message, but can also be used to model how the process accepts a task, queries data, or receives goods.

The produce activity represents the production and delivery of a message, but can also be used to model how the process delegates a task, stores data, or creates goods.

The operation activity is used to model atomic actions, such as invoking a service, opening a new account, or changing the terms of an order.

Complex activities are used to model a flow of control, whether that flow is serial, parallel or conditional. They are also used to model compound states, consisting of multiple sub-activities representing sub-states.

The sequence activity models the serial execution of activities.

The all activity models the parallel execution of activities.

The choice activity is used to model a branch between mutually exclusive flows.

Process activities are used to manage the process data, to spawn and join nested processes, to suspend and complete the process, and to repeat activities or states. The latter allows the modeling of conditional and non-conditional loops.

BPML supports the join pattern, allowing a process to engage in synchronized consumption or production of messages for coordinating activities across multiple participants.

The following diagram illustrates such an example. The buyer asks the supplier whether a product is available and asks the carrier whether the product can be delivered on time. The buyer requires a positive reply from both participants in order to proceed, and will regard a negative reply from either party as failure. We have used dotted rectangles to represent

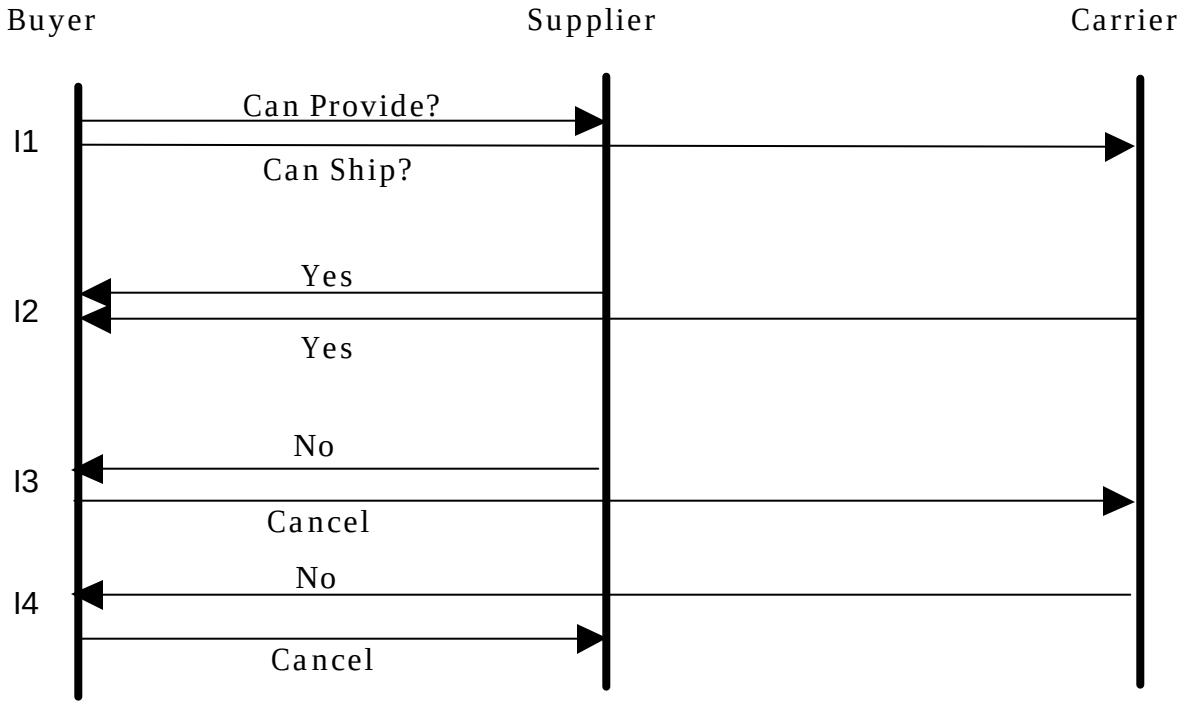
joined messages that must be sent/received together. The interactions I2, I3, and I4 are mutually exclusive.

## 2.5 Rules

Complex business logic demands that a process select one of several alternative activities, or discriminate the information upon which it acts. This is expressed in the form of rules that affect activity selection (branching and repeating) and govern message consumption.

Process branching occurs as the result of a decision made by the process, and models the manner in which process execution will be affected by information collected and created during the life of the process.

Participant branching occurs as the result of a decision made by a participant and communicated to the process in the form of a message. Participant branching is used to model collaborative processes in which the process reacts to requests or reports from its participants.




Buyer process uses join pattern to synchronize with supplier and carrier

BPML allows rules to express conditions that are based on information known to the process, or information communicated to the process. The latter can be used to provide business level validation for messages, to link processes to events occurring in the environment, or to express data extraction.

BPML uses XPath as the basis for its expression language. XPath provides a declarative expression language with rich semantics for expressing condition logic, calculations, and selection predicates.

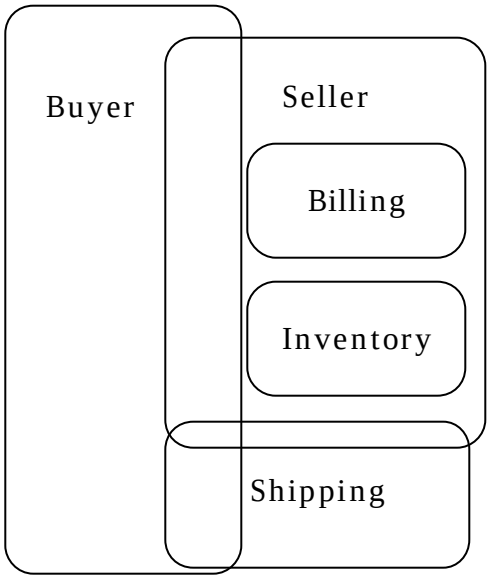
## 2.6 Transactions

BPML supports two transaction models: coordinated and extended.

The coordinated transaction model provides an all-or-nothing guarantee by assuring that all participants of the transaction agree to either complete the transaction or abort it. This model relies on the two-phase-commit protocol in order to enable distributed transaction coordination, and is also known as closed, flat transactions, due to its ability to support transaction isolation.

The extended transaction model relaxes the isolation requirement while preserving the all-or-nothing nature of transactions, aided by process-level forward and backward recovery. It allows for long living transactions to acquire resources for a short period of time, enables arbitrary levels of nesting, and allows for transaction interleaving that occurs in complex, multi-party collaborations. Extended transactions are used for the modeling of long-lived business transactions. They are also known as open nested transactions or Sagas.

The following diagram illustrates interleaving transactions and nested transactions merging together. The buyer transaction spans the order and receipt of goods. The seller transaction interacts with it and involves two nested transactions for billing and inventory management. The seller transaction concludes by initiating a carrier transaction to ship the goods, but the buyer transaction does not conclude until the goods have been received.




Interleaving of buyer, seller and shipping transactions

In order to guarantee full recovery, BPML supports the notion of compensating activities that revert the result of previously completed activities. Compensating activities are used to recover from activities that cannot be rolled back, e.g. sending an e-mail or shipping a package.

## 2.7 Processes

Process abstracts define the interaction between the process and its participants. The abstract omits information not pertinent to that particular set of participants, and as such, cannot be executed.

Abstracts are used to describe a system, business partner, or user for the purpose of enlisting them as participants in a process. Collaborative processes such as those described by ebXML and RosettaNet, Web services such as those described by WSDL, and software APIs can all be mapped to process abstracts.

Process executions define a complete and valid process such that it can be executed to completion. An execution can lead to an implementation of the process that may be executed by a software system.

BPML uses processes to manage the lifecycle and availability of resources and to constrain the manner in which they are used. Such processes are used to model data and document

management, work items, services, software components, and even equipment and perishable goods.

Nested processes are used to define processes that have a lifetime that is independent of their parent processes, unlike sub-activities, which are bound to the lifetime of their parent activity. Nested processes are made available within the states of the parent process and are initiated in the context of their parent process.

Nested processes are used to model services and resources that are available to participants with some restriction. For example, a service agreement process can expose a nested process for each service available under the agreement, but can exclude services when the account is overdue or the agreement has expired.

Section 3

# BPML by Example

To illustrate the power of BPML, we will model a business process involving trouble ticket management between a customer and a service provider. Such processes are common in the telecommunication industry as part of the quality-of-service obligation between service providers and their customers.

The customer of the service is affected by a problem and interested in a timely resolution. The service provider is responsible for providing a resolution and does so in an orderly fashion through the management of a trouble ticket.

There are several processes involved in the management of a trouble ticket:

- A process on the customer side is responsible for communicating problems (or troubles) to the service provider, as well as receiving notification of problems detected by the service provider and the resolution of a problem.
- A process on the service provider side is responsible for opening new trouble tickets, whether issued by the customer or the service provider, and for communicating with the customer to provide progress reports and billing.
- The trouble ticket itself is also modeled as a process, by virtue of having a lifecycle independent of either the customer or supplier processes, as is the service process.

We use the namespace prefix `cust` to denote the business domain of the customer, the namespace prefix `sp` for the service provider, `tt` to denote trouble ticket, and `srv` to describe a service.

## 3.1 Abstract Process

We start by defining a process involving the management of the ticket. This is an abstract process that defines the manner in which a ticket can be queried and closed. It is not an implementation process: the service provider may implement the process in various ways, e.g. using a software application component or a database table.

We start by defining the possible status code for a trouble ticket. At this point, we consider only two possible statuses: open and closed.

We use XML Schema to define the status codes. The use of XML Schema does not restrict us to XML; it merely provides a rich syntax for describing typed and structured data. The status may be maintained as a numerical value in a relational database, an enumerated field in an object, or any other means.

Next, the process defines a number of messages. They are used to obtain the status of a ticket and dose the ticket. We elected to provide two synchronous operations, requiring that we define four messages: two for input and two for output.

Even though most of the messages are empty and are merely used to communicate a request or acknowledgment of completion, we define them in order to declare unique names that can be used to distinguish requests from responses.

Last, we define the two operations that can be performed. Each operation consists of two messages, one input and one output. For clarity we chose to name each operation, although this is not mandatory. The distinction between the messages is sufficient to determine which operation will be performed.

Operations are equivalent to methods in a programming language, stored procedures in a database, and request/response messages in a communication protocol. We can easily see how this process can be implemented as a software application, database application, or Web service.

The following document illustrates how to expose the process using BPML.

Abstract process for a trouble ticket

```
<package name="TroubleTicket" namespace="http://bpmi.org/TroubleTicket">

<abstract name="Ticket">

  <xsd:schema>
    <xsd:simpleType name="statusCode" base="xsd:Name">
      <xsd:enumeration value="open"/>
      <xsd:enumeration value="closed"/>
    </xsd:simpleType>
  </xsd:schema>

  <message name="getStatusInput" type="request"/>
  <message name="getStatusOutput" type="response">
    <xsd:element name="status" type="statusCode"/>
  </message>

  <message name="closeInput" type="request"/>
  <message name="closeOutput" type="response"/>
```

```
  <choice>
    <operation name="getStatus">
      <input message="getStatusInput"/>
      <output message="getStatusOutput"/>
    </operation>

    <operation name="closeTicket">
      <input message="closeInput"/>
      <output message="closeOutput"/>
    </operation>
  </choice>

</abstract>
```

## 3.2 Nested Processes

We now define the manner in which the customer can open a ticket. This is an operation offered by the service provider, and therefore is listed under the service provider's process. Again, we define an abstract process.

Abstract process for a service provider

```
<abstract name="Provider">

  <message name="openTicketInput" type="request">
    <xsd:element name="trouble" type="xsd:string"/>
  </message>

  <message name="openTicketOutput" type="response">
    <xsd:element name="ticket" type="Ticket"/>
  </message>

  <operation name="openTicket">
    <input message="openTicketInput"/>
    <output message="openTicketOutput"/>
  </operation>

</abstract>
```

The operation `openTicket` receives a trouble ticket description and returns an open ticket, an instance of the `Ticket` process defined previously.

Each time we perform the `openTicket` operation, a new ticket process is created and the new process instance is returned. A way to declare that in BPML is by expanding the service provider process to encapsulate the ticket process.

For brevity, this example omits the message definitions in the `Provider` process and the body of the `Ticket` process.

Expanded abstract process for a service provider including a trouble ticket abstract process

```
<abstract name="Provider">

  ...

  <sequence>

    <operation name="openTicket">
      <input message="openTicketInput"/>
      <output message="openTicketOutput"/>
    </operation>

    <abstract name="Ticket">
      ...
    </abstract>

  </sequence>

</abstract>
```

The result of the `openTicket` operation is a reference to the newly created ticket process. From this example, it is not obvious how the customer can get the ticket status given the reference to the ticket. For that, we introduce the concept of participants.

## 3.3 Participants

A participant is any entity with which the process can interact. The process interacts with participants by exchanging messages or performing operations.

In its most simple form, a participant is represented by a URI. If the participant is a Web service or a user's e-mail, it will typically be a URL. If the participant is an application, it may be the name of the application or a component in that application. It may also be a virtual name, such as an organizational role, the DUNS identifier of a business partner, etc.

When two tickets are created, each one will constitute a different participant. If we decide to implement the ticket as a database table, this might be the primary key in the table. Using an application server, it might be a reference to a component instance.

A participant's name can be passed between processes so they can direct their communication. For example, if the provider realizes that the trouble is the fault of a different provider, it can communicate that to the customer, and the customer can then open a new trouble ticket directly with the appropriate provider.

We now illustrate what the customer process will look like, taking advantage of the participant element. We illustrate only a portion of a larger customer process for tracking a trouble.

The first operation involves the creation of a trouble ticket using the service provider participant. From that operation we learn which ticket process to refer to and use it as the participant for the second operation, in which we retrieve the current status of the trouble ticket.

Portion of the customer process for tracking a trouble ticket

```
<process name="TrackTrouble">

  ...

  <operation name="createTicket">
    <participant select="Provider"/>
    <output message="openTicketInput"/>
    <input message="openTicketOutput">
      <assign target="ticketParticipant" select="ticket/text()"/>
    </input>
  </operation>

  ...

  <operation name="getTicketStatus">
    <participant select="ticketParticipant"/>
    <output message="getStatustInput"/>
    <input message="getStatusOutput">
      <assign target="ticketStatus" select="status/text()"/>
    </input>
  </operation>

  ...

</process>
```

## 3.4 Assignment

In addition to participants, the previous example introduced the notion of assignment. As the process executes, it obtains information from participants and communicates it to other participants.

The process needs to be selective about what information is communicated to which participant. We can expect the provider process to hold substantial data, only a portion of which is directly relevant to the customer.

The process decides what information is relevant and communicates that subset using the `assign` element. This element takes a value known to the process and includes it in an output message, and similarly, takes a value from an input message and makes it available to the process.

The direction in which an assignment is done—i.e. from process to message, or from message to process—depends on the context in which the `assign` element appears. Under the `input` element, the assignment is from the message to the process; under the `output` element, the assignment is from the process to the message.

While assignment between a process and a message is a powerful concept, often a process merely requires the routing of information from one participant to another. For such cases we introduce a more declarative syntax that performs assignment directly between messages, allowing us to explicitly model information routing.

Enhancing the previous example to take advantage of this form of assignment allows us to remove two instances of the `assign` element. The first becomes redundant by referencing the `openTicketOutput` message directly for the purpose of determining the trouble ticket participant.

To determine the status further down in the process (not shown here) we can always reference the value of `getStatusOutput/ticketStatus`.

Portion of the customer process for tracking a trouble ticket with assignment

```
<process name="TrackTrouble">

  ...

  <operation name="createTicket">
    <participant select="Provider"/>
    <output message="openTicketInput"/>
    <input message="openTicketOutput"/>
  </operation>

  ...
```

```
  <operation name="getTicketStatus">
    <participant from="openTicketOutput/ticketParticipant"/>
    <output message="getStatustInput"/>
    <input message="getStatusOutput"/>
  </operation>

  ...

</process>
```

## 3.5 Consume and Produce

In order to provide quality-of-service, the provider takes a proactive approach and notifies the customer when the trouble ticket is closed and the problem has been brought to a resolution.

For this, the provider must be able to send a message to the customer. In this example, we expand the provider process and, to be brief, include only those elements that changed from the previous example.

Expanded service provider process for customer notification when closing a trouble ticket

```
<abstract name="Provider">

  <message name="openTicketInput" type="request">
    <xsd:element name="trouble" type="xsd:string"/>
    <xsd:element name="customer" type="Customer"/>
  </message>

  ...

  <sequence>

    <operation name="openTicket">
      <input message="openTicketInput"/>
      <output message="openTicketOutput">
        <assign target="ticket" select="Ticket"/>
      </output>
    </operation>

    <abstract name="Ticket">
      ...
```

```
    </abstract>

    ...

    <produce name="notifyCustomer">
      <participant select="openTicketInput/customer"/>
      <output message="ticketClosed"/>
    </produce>

  </sequence>

</abstract>
```

The produce activity is used to send a message from the provider to the customer. It can be used to establish asynchronous communication between processes, as in this case, but also serves other purposes.

For example, if we were interested in recording the closing of a trouble ticket, we could produce a second message directed at the database. By doing so, we cause a new row to be added to a table in the database, thereby producing a permanent record.

On the customer side, the message will be handled by the consume activity. This activity is the counterpart of produce, and again is not limited to asynchronous messaging. For example, an end-of-month process will consume all accounts payable and execute a process for paying each bill, including those due to the service provider.

The operator activity is a synchronous coupling of a consume activity and a produce activity. We will see how these three activities play along later on.

## 3.6 Executable Process

Now it is time to look at some aspects of the customer process. We plan to implement an executable process on the customer side to open and track a trouble ticket initiated by the customer.

First, we need to establish how the customer knows which provider to contact regarding a problem. For that, we define a service process that binds customer and provider together.

Abstract process for a service

```
<abstract name="Service">

  <message name="getCustomerInput" type="request"/>
  <message name="getCustomerOutput" type="response">
    <xsd:element name="customer" type="Customer"/>
```

```
  </message>

  <message name="getProviderInput" type="request"/>
  <message name="getProviderOutput" type="response">
    <xsd:element name="provider" type="Provider"/>
  </message>

  <operation name="getCustomer">
    <input message="getCustomerInput"/>
    <output message="getCustomerOutput"/>
  </operation>

  <operation name="getProvider">
    <input message="getProviderInput"/>
    <output message="getProviderOutput"/>
  </operation>

</abstract>
```

We also need to define a customer abstract process that allows the provider to notify the customer when the trouble ticket has been closed.

Abstract process for customer notification by the service provider

```
<abstract name="Customer">

  <message name="ticketClosed" type="data"/>

  <consume name="notifyCustomer">
    <input message="ticketClosed"/>
  </consume>

</abstract>
```

On the customer side, we define a process used for reporting and tracking the status of the trouble ticket. This time, we define a process that will execute to completion, introducing the process element.

Unlike and abstract process, an executable process defines the mechanism by which the process will execute to completion. As such, it takes implementation details into account.

We can imagine a user interface, whether graphical or Web based, that is layered on top of the customer process. In this simple scenario, the user interface consists of just two forms, one for submitting a trouble description, and one for viewing the status of the trouble ticket.

The first form accepts the trouble description and the service, and communicates with the process as participant `reportTroubleForm`.

Once the trouble has been reported, a new process instance is created to follow the status of this specific trouble, and a reference to that process is returned. The second user form will use that reference in order to query the status of the trouble ticket.

Next, the customer process communicates with the relevant service process in order to find which provider is responsible for the service. The process then communicates with that participant in order to create a new trouble ticket.

When the problem is brought to a resolution, the provider will automatically notify the customer that the ticket has been closed. To receive this notification, the customer process communicates itself as a participant to the provider's process.

The first part of the customer process is defined below.

First part of the customer process for reporting and tracking a trouble ticket

```
<process name="TrackTrouble">
  <supports abstract="Customer"/>

  <message name="troubleReportInput" type="request">
    <xsd:element name="service" type="Service"/>
    <xsd:element name="trouble" type="xsd:string"/>
  </message>
  <message name="troubleReportOutput" type="response">
    <xsd:element name="cookie" type="TrackTrouble"/>
  </message>

  <message name="getStatusInput" type="request"/>
  <message name="getStatusOutput" type="response">
    <xsd:element name="status" type="statusCode"/>
  </message>

  <sequence name="reportAndTrack">

    <operation name="reportTrouble">
      <participant name="reportTroubleForm"/>
      <input name="troubleReportInput"/>
      <output name="troubleReportOutput">
        <assign target="cookie" select="TrackTrouble/text()"/>
      </output>
    </operation>

    <operation name="findProvider">
```

```
      <participant select="troubleReportInput/service"/>
      <output message="getProviderInput"/>
      <input message="getProviderOutput"/>
    </operation>

    <operation name="createTicket">
      <participant select="getProviderOutput/provider"/>
      <output message="openTicketInput">
        <assign select="troubleReportInput/trouble"/>
        <assign select="TrackTrouble/text()" target="customer"/>
      </output>
      <input message="openTicketOutput"/>
    </operation>

    ...

    <consume name="notifyCustomer">
      <input message="ticketClosed"/>
    </consume>

    ...

  </sequence>

</process>
```

## 3.7 Sequence and Choice

Obtaining the status of a trouble ticket from the provider is free of charge, but since the trouble ticket process runs on the provider's system, there might be a noticeable latency involved.

The customer process has been designed with a user interface in mind, and as such aims to provide immediate feedback. The solution is to always report the ticket status currently known to the customer process, and only change that status in response to a provider initiated notification.

For this purpose, we expand the customer process with two additional states:

Immediately after opening the trouble ticket, the customer process enters the open state. It remains in this state until the provider communicates closure of the ticket. Any request for status at this state will automatically indicate that the ticket is open.

The second state is close, entered into following the provider notification. The process will remain in this state perpetually, and any request for the status in this state will automatically indicate that the ticket is closed.

The reminder of the customer process is defined below.

Second part of the customer process for reporting and tracking a trouble ticket

```
<process name="TrackTrouble">
  <supports abstract="Customer"/>

  ...

  <choice name="open">

    <sequence>

      <operation name="getStatus">
        <input message="getStatusInput"/>
        <output message="getStatusOutput">
          <assign target="status" select="open"/>
        </output>
      </operation>

      <repeat ref="open"/>

    </sequence>

    <consume name="notifyCustomer">
      <input message="ticketClosed"/>
    </consume>

  </choice>

  <sequence name="close">

    <operation name="getStatus">
      <input message="getStatusInput"/>
      <output message="getStatusOutput">
        <assign target="status" select="close"/>
      </output>
    </operation>

    <repeat ref="close"/>
```

```
  </sequence>

</process>
```

As the process executes, it always resides in one known state or the other. The duration this process spends in any given state depends on the rule that governs its completion and fires a transition to the next state.

The sequence activity lists a number of activities (at least one), which the process will follow sequentially in order to transition from this state to the next state.

The choice element lists a number of activities (at least two) that the process is given the choice of taking in order to transition from its current state to the next one. A choice models participant branching.

The repeat activity is used to trace the path of the process back to a previous state. In this case, each time the status is requested, the process iterates back to the same step. The process transitions to a different state when a provider notification arrives.

## 3.8 Switch & Rules

Two types of branching can occur in the process. Process branching is based on a decision made by the process, while participant branching is based on a decision made by a participant and communicated to the process.

Process branching depends on the application of a rule. To illustrate this, we allow the provider to reject any trouble ticket for a service not offered by that provider.

Example illustrating the use of rules in a process branch

```
<sequence>

  <consume name="createTicket">
    <input message="openTicketInput"/>
  </consume>

  <operation name="getProvider">
    <participant select="openTicketInput/service"/>
    <output message="getProviderInput"/>
    <input message="getProviderOutput"/>
  </operation>

  <switch>

    <case condition="setProviderOutput/provider = Provider”>
```

```
      <sequence>
        <produce name="acceptTicket">
          <output message="openTicketOutput">
            <assign select="Ticket/uri" target="ticket"/>
          </output>
        </produce>

        <abstract name="Ticket">
          ...
        </abstract>

        ...

      <sequence>
    </case>

    <otherwise>
      <exception name="rejectTicket" code="noSuchService"/>
    <otherwise>

  </switch>

</sequence>
```

After obtaining the trouble report, the process attempts to determine which provider is associated with this service. The case condition is met if this process and the provider process associated with the service are one and the same.

A branch is taken based on the outcome of the rule. If the condition is met, then a new ticket process is created and communicated back to the customer. If the condition is not met, then an error (`exception`) is communicated back to the customer.

A rule can also be defined where it is evaluated using information conveyed in the message, in which case it can be used for consuming messages.

To understand why the later case is important, consider another part of the provider process in which all scheduled maintenance work for the next week are processed in order to create trouble tickets. The process will consume a list of messages from a table of scheduled work and act upon it, as illustrated below.

Example illustrating the use of rules in consuming a message

```
<message name="scheduledWork" type="data">
  <xsd:element name="plannedStart" type="xsd:timeInstance"/>
  <xsd:element name="ticket" type="Ticket"/>
  <xsd:element name="service" type="Service"/>
```

```
</message>

...

<consume name="getScheduledWork">
  <participant select="scheduleList"/>
  <input message="scheduledWork">
    <rule condition=”thisWeek(scheduledWork/plannedStart) and
                not(scheduledWork/ticket)”/>
    </rule>
  </input>
</consume>
```

## 3.9 Parallel Activities

While some processes execute sequentially, others benefit from the ability to utilize resources concurrently and, as a result, complete faster.

For example, consider a situation where the closing of a ticket also entails billing of the customer. We would like to assure that the ticket is closed, the customer is notified, and the work is billed, all in one unit of work. We can accomplish that by encapsulating all three activities into a transactional sequence (we will review transactions in detail in a later section).

Since we are communicating with two different participants (customer and financial services), nothing prevents the communication to occur in parallel. We can choose to use the `all` construct to aggregate activities that will execute in parallel.

Assuming that the ticket process has been expanded to record all work performed to resolve the problem and indicate whether each work item is billable or not, we would simply pass the ticket process along to the billing service.

Example illustrating parallel execution for closing, notification, and billing

```
<all>

  <operation name="closeTicket">
    <participant select="ticket"/>
    <output message="closeInput"/>
    <input message="closeOutput"/>
  </operation>

  <produce name="notifyCustomer">
    <participant select="openTicketInput/customer"/>
```

```
    <output message="ticketClosed"/>
  </produce>

  <produce name="billCustomer">
    <participant select=" BillingService"/>
    <output message="billTicket">
      <assign select="ticket" target="ticket"/>
      <assign select="openTicketInput/customer"/>
    </output>
  </produce>

</all>
```

The `all` activity is synchronous in nature, and only completes once all the activities specified underneath it have completed. There are other contexts in which parallel processing can occur.

For example, consider the customer, provider, and ticket processes. These are three parallel processes that are all ongoing at the same time. They interact with each other at various points, but they do not depend on each other in order to complete.

For example, when a trouble is reported, the service provider spawns a trouble ticket process that executes in parallel to the service provider process. Multiple ticket processes can execute concurrently, interacting with multiple provider and customer processes, in a topology that changes dynamically.

## 3.10 Nested Processes (revisited)

The ability to engage in multiple concurrent processes in a dynamic topology is based on a model that supports nested process declarations and takes advantage of the ability to communicate processes as participants.

The definition of a process does not cause a process to execute, it merely implies that a process is defined and might execute if asked to. This applies to both top-level processes and nested processes. Hence, the inclusion of the `process` element within a complex activity does not automatically lead to the execution of the nested process.

During its lifetime, a process might wish to execute some of its nested processes, communicate them as participants to other processes, or communicate itself as a participant to other processes.

The `def` and `uri` sub-elements are part of the process data that is automatically updated to reflect process instances and available process definitions. These sub-elements hold the participant `uri` of the process instance (`uri`) and process definition(`def`).

In order to communicate a process as a participant, the name of that process must be assigned into a message using the `def` sub-element. In order to interact with a process, the name of that process must be used as a participant, using the `uri` sub-element.

In order to communicate itself or one of its parent processes as a participant, a process instance assigns its name into a message using the `uri` sub-element.

In order to execute a nested process, the parent process may either communicate with that process as a participant, or use the `spawn` activity. The `spawn` activity can only be used with a nested process defined in the same context, and is introduced to enable declarative nested process invocation.

Inform the participant that a `Ticket` process is defined

```
<produce>
  <output message="sp:exposeTicketProcess">
    <assign select="Ticket/def" target="ticket"/>
  </output>
</produce>

<process name="Ticket">
  ...
</process>
```

```
<spawn name="Ticket"/>
```

Instantiate a new `Ticket` process and communicate it to a participant

```
<produce>
  <output message="newTicketProcess">
    <assign select="Ticket/uri" target="ticket"/>
  </output>
</produce>

<process name="Ticket">
  ...
</process>
```

The `Ticket` process communicates itself to a participant

```
<process name="Ticket"/>

  <produce>
    <output message="thisTicketProcess">
      <assign select="Ticket/uri" target="ticket"/>
    </output>
  </produce>
```

```
</process>
```

Whenever a nested process is instantiated, it behaves as a sub-process of the parent process in which it was declared, having access to a copy of the process data available to the parent process at the time of instantiation. Its lifetime is independent of that of the parent process.

A nested process is available only in the context in which it is defined. If this context is bound to a state of the process, the nested process is available only while the process resides in this state. It is an error to communicate a nested process outside of a state in which it is available.

A process may also use the `join` activity to wait for the completion of all its nested processes before proceeding. This form of synchronization is used with parallel process execution, where the `all` activity guarantees synchronization with sub-activities.

## 3.11 Transactions

In order to guarantee consistency and reliability, processes make use of transactions. Transactions enable the process to guarantee that a set of activities will complete as a single unit of work, or if an error occurs, recover in a consistent manner.

Transactional behavior can only be affixed to a complex activity, which can be either a `sequence`, `choice`, `foreach`, or `all`, by specifying the desired transactional behavior using the `transaction` element.

If the complex activity requires transactional behavior, it will use the `required` type. A new transaction will then be created, unless one already exists (e.g. from a parent activity or a participant).

To declare that the complex activity must have its own transaction, it will use the `new` or `nested` types. To declare that it does not wish to be part of a transaction, it will use the `none` type. If the activity supports a transaction, but does not require one, it will use the `supported` type (default type).

BPML supports two types of transactional models: coordinated and extended. The coordinated model is also known as closed flat or ACID transactions, while the extended model is also known as open nested transactions, or Sagas. They are used to support short-lived transactions and long-lived transactions, respectively.

A coordinated transaction specifies that all its participants must agree to successfully complete the transaction, or must all uniformly abort it. This happens through the coordination of the transaction completion. The most common approach for implementing this coordination is the use of a two-phase commit protocol such as the one defined by

CORBA OTS (and its Java counterpart JTS) and X/Open X/A (supported by most database management systems).

Coordinated transactions are not applicable in all circumstances. In particular, most implementations of coordinated transactions involve a form of resource locking that can be disruptive if locks are maintained for a long period of time.

In addition, coordinated transactions do not allow for transaction interleaving that occurs in complex processes. Transaction interleaving demands that the isolation requirements of ACID transactions be relaxed in order to allow multiple transactions to interact around the same data.

Long-lived business transactions are supported through the extended transaction model. The extended model assures that, as the process progresses from state to state, it maintains a permanent record that can be reconstructed in the event of a failure, allowing the process to complete.

If the transaction is coordinated and all participants can recover from it, no specific handling is required. However, if the transaction is not coordinated, or recovery is not automated, the process must model a form of recovery known as compensating transactions.

The compensate element can be associated with any activity to describe the manner in which the process will revert changes done by this activity and return to a previous state, if the transaction aborts. In the event of the transaction aborting, compensating activities will be executed for full recovery.

Compensating activities can be used in both coordinated and extended transactions, allowing the two transaction models to be nested within each other.

In a previous example we showed how the provider process bills the customer for any work done. We assumed that billing would happen in a larger transaction. An obvious question arises: What will happen if after being billed the customer decides that the trouble has not been resolved? In this case the transaction will abort, and any billing will be refunded by a compensating activity.

Bill the customer within the transaction, refund the customer if the transaction aborts

```
<produce name="billCustomer">

  <participant name="BillingService"/>

  <output message="billTicket">
    <assign select="ticket"/>
    <assign select="openTicketInput/customer"/>
  </output>
```

```
  <compensate>
    <participant name="BillingService"/>
    <output message="refundBillTicket">
      <assign select="ticket"/>
      <assign select="openTicketInput/customer"/>
    </output>
  </compensate>

</produce>
```

## 3.12 Exceptions

We use the term exception to refer to any error that occurs while executing the process, whether the result of a local error or an error communicated to the process by one of its participants.

The onException element can be associated with a complex activity to define the behavior expected in the event of an exception. When an exception occurs in a complex activity for which an onException element is specified, the process will continue by performing the activities listed within the onException element.

The process may engage in recovery, select a different path of execution, or communicate failure to its participants using the exception activity.

In a previous example, we showed how the provider process might close a ticket, notify the customer, and bill the customer at the same time. We now recast that example using the transaction element.

We are not interested in preserving state over a long duration, but simply to ensure that all three participants engage in coordinated interaction. Hence, we use the coordinated transactional model. If an error occurs, we would like to track it and inform an operator immediately.

Since the billing service does not support coordinated transactions, we also need to communicate the transaction failure to it.

Example illustrating parallel execution with guaranteed transactional behavior

```
<all>
  <transaction type="nested" model="coordinated"/>

  <operation name="closeTicket">
    ...
  </operation>
```

```
  <produce name="notifyCustomer">
    ...
  </produce>

  <produce name="billCustomer">
    ...
  </produce>

  <onException>

    <sequence>
      <produce name="notifyOperator">
        <participant name="operator"/>
        <output message="transactionFailed"/>\
      </produce>

      <exception code="transactionFault">
        <participant name="BillingService"/>
      </exception>
    </sequence>

  </onException>

</all>
```

## 3.13 Time Constraints

Some processes are said to execute forever. Yet, some processes, or segments of processes, must be finite in time.

While there is no arbitrary limit on the amount of time a trouble ticket remains open, quality of service determines that the trouble be resolved in the quickest possible manner. The trouble ticket process can express that by adding another state called escalated. If the ticket has remained open for more than 24 hours, it will immediately change to escalated and will engage in some interaction with the provider process to assure that more attention is placed on solving the problem.

Example of time constraint on a state, leading to a different state

```
<abstract name="Ticket">

  <xsd:schema>
    <xsd:simpleType name="statusCode" base="xsd:Name">
```

```
        <xsd:enumeration value="open"/>
        <xsd:enumeration value="escalated"/>
        <xsd:enumeration value="closed"/>
      </xsd:simpleType>
    </xsd:schema>

    ...

</abstract>

<process name="SelfEscalatingTicket">
  <supports abstract="Ticket"/>

  ...

  <sequence>
    <completeBy duration="'PT24H'"/>

    ...

    <operation name="closeTicket">
      <input message="closeInput"/>
      <output message="closeOutput"/>
    </operation>

    <onException code="bpml:timeout">

      <produce name="escalate">
        <participant select="provider"/>
        <output message="escalateTicket">
          <assign select="Ticket/uri" target="ticket"/>
        </output>
      </produce>

      ...

    </onException>

  </sequence>

</process>
```

Time constraints are introduced using the element `completeBy`, which indicates time duration. It defines a boundary for the completion of the activity in which it is included

relative to the point in time at which the process entered the state represented by this activity.

Section 4

# BPML Conventions

## 4.1 Terminology

The terminology used to describe BPML is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of a BPML processor:

For Compatibility: A feature of this specification included solely to ensure that schemas that use this feature remain compatible.

May: Conforming documents and processors are permitted to but need not behave as described.

Must: Conforming documents and processors are required to behave as described; otherwise they are in error.

Error: A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it. All error recovery must be conforming.

The term BPML document refers to document instances that define a process or an abstract process.

The term BPML XML Schema refers to the schema that a document subscribes to, as defined in this specification.

The term BPML element refers to an element defined in the BPML XML Schema and used to construct a BPML document.

## 4.2 Use of Namespaces

BPML relies heavily on the use of namespaces to distinguish between the various domains in which processes are defined.

All process, abstract, message, participant and activity definitions within a BPML document are associated with the namespace URI declared in the package root element.

## 4.3 Use of Extension Elements

BPML allows extensions to be included in a BPML document that are specific to a given implementation. Such extensions can be used to provide implementation-specific details, markings for analysis, and cues for optimization.

Each extension element must be prefixed with a namespace to enable validation and indicate which implementation is required to understand and support it.

Implementations that are not aware of a particular namespace shall not attempt to process extension elements defined in this namespace. Implementations that are aware of a particular namespace may reject a BPML document if an error is encountered when validating or understanding such elements.

The any element will appear in the BPML XML Schema as follows:

```
<xsd:group ref="extension"/>
```

## 4.4 Annotations

BPML elements can be annotated with textual information. Annotations are included using the annotation element. We have elected not to use the XML Schema annotation element, since it introduces complexity by enabling both human readable and software readable annotations. The latter is achieved with the use of extension elements.

The BPML XML Schema allows annotations for multiple languages using the xml:lang attribute. Multiple occurrences are allowed as long as each annotation element specifies a different language.

We recommend the use of XHTML 1.0 for the content of annotations.

## 4.5 Meta Data

Meta data can be used to provide additional information about various BPML elements for the purpose of advertising, searching, and categorizing.

The `meta` element may contain a list of elements in various namespaces. Each element constitutes a name-value pair, with the fully qualified element name serving as the name and its content as the value. Such elements need not validate against any known schema.

We recommend the use of the Dublin Core Metadata Element Set. We use the namespace prefix `dublin` and namespace URI http://purl.org/DC/documents/rec-dces-19990702.htm.

## 4.6 Namespace Handling

In order to assure proper namespace handling of textual content, the BPML XML Schema makes a distinction between mixed content, whitespace preserving content, and names.

Mixed content is supported using the type `xsd:anyType`, allowing any type of content to be included without requiring validation. We recommend the use of XHTML 1.0, which is both XML valid and can be rendered by many visual tools.

Whitespace preserving content is supported using the type `xsd:string`. The default for any document is whitespace preserving, and therefore there is no need to explicitly use the `xml:space` attribute.

When whitespaces are allowed for the purpose of readability but ignored for the purpose of processing we use one of the following types: `xsd:token`, `xsd:Name`, `xsd:Qname`, or `xsd:uriReference`.

## 4.7 Use of Keys

We plan to leverage the ability of XML Schema to describe keys and name uniqueness within a context. The use of keys allows us to define a context in which a name is unique, e.g. the name of an activity must be unique within a given process. As such, the BPML XML Schema does not use the `ID` and `IDREF` attribute types.

Keys will be incorporated in a future version of BPML.

## 4.8 Use of name, ref, and from Attributes

The `name` attribute is used to name a BPML element. The name must be unique in the context in which it is defined. The name must be unique in a larger context in which it may be referenced.

The name attribute must be used whenever a BPML element will be referenced elsewhere in the process or from a different process. This attribute is always of type xsd:NCName, the namespace is inferred from the package root element.

While the name attribute may be used for labeling, typed labels and human readable labels can also be provided using the meta and annotation elements respectively.

The ref attribute is used to refer to a BPML element. This attribute is always of type xsd:NCName and must match an element of the same type.

The select attribute is used to refer to process data by expression. The manner in which process data is named depends on the context in which this data is assigned to the process. This attribute is always of type xsd:string and expressed in the form of an XPath expression.

## 4.9 Exception Codes

The following exception codes are pre-defined in all BPML processes.

bpml:timeout

An activity has failed to complete in the time constraint specified by the completeBy element.

Section 5

# Process Definition

The process definition entails the definition of messages used for interaction between the process and its participants, static participants with which a process will always interact, activities that will occur in order to execute the process to completion, and any abstracts supported by the process.

## 5.1 Messages

Messages can be produced by a process and delivered to a participant, delivered by a participant and consumed by the process, or used to perform operations through a synchronous request/response exchange.

The use of messages does not restrict interactions to a form of message passing. Storing and retrieving data, method invocation, and work item management, are all modeled in the form of message-based interactions.

A process must define all messages that are unique to that process. If the process interacts with other processes, it can import their message definitions.

Messages are given unique names so they can be identified within a process. Identification is required in order to direct a message to the proper activity, whether multiple activities are waiting to consume messages, or a message arrives when no activity is able to accept it.

Messages are defined as a sequence of elements or complex types. The XML Schema `xsd:element` and `xsd:complexType` are used in the definition.

Top-level elements are accessed within the message content by their fully qualified names. Top-level elements are akin to message parts.

Element recurrence is supported using the `minOccurs` and `maxOccurs` attributes. Complex types allow `sequence`, `choice`, and `all` constructs to be used.

Contents of arbitrary structure can be specified using `xsd:anyType`. Contents of arbitrary type can be specified using `xsd:string` or `xsd:binary`.

A message can be defined as a request, implying that a response is expected, as a response to a previous request, or independently as a data message. There is no need to define fault messages.

## 5.2 Participants

A participant is statically defined, or dynamically acquired during the execution of a process. Each process is considered a candidate participant to other processes.

Static participants are declared and named (using the `name` attribute). The process will always interact with the same set of static participants.

Static participants can be referenced (using the `select` attribute), allowing the same participant to be defined once and used in multiple points. Participant referencing is also used to refer to a process as a participant.

Participants can be communicated by the process or to the process in the contents of a message. A process may reference a participant previously communicated in a message or otherwise available from the process data (see later) using the `select` attribute.

A process may communicate itself or a nested process as a participant. It uses the `def` sub-element to communicate a definition, and the `uri` sub-element attribute to communicate an instance.

If a participant refers to a process definition, the first interaction with that participant will instantiate a new process instance. If the participant refers to a process instance, all interactions will target that instance.

A message can only be produced to a known participant (whether static or dynamic). It is possible to deliver a message to multiple participants, if multiple participants have been acquired under the same name.

A message can be consumed from any participant, or restricted to a known participant. It is possible to restrict a message to multiple participants, if multiple participants have been acquired under the same name. Only one message will be consumed at once.

## 5.3 Activities

BPML separates between the definition of simple activities, complex activities, and process activities. Activities play a dual role: They are used to represent flow(s) of control, as well as manage the state(s) of the process.

## Type of Activities

A process involves three types of activities: simple, complex, and process.

Simple activities involve interaction with the environment, typically in the form of a message exchange or an atomic operation against a participant. Simple activities are prone to communication and operational faults, and may require time constraints and compensating activities to be defined.

Complex activities are a composition of sub-activities that organize the serial, parallel, or conditional execution of their sub-activities. Complex activities are prone to all faults in sub-activities, and may require exception handling and transaction control to assure the proper execution of their sub-activities.

Process activities only affect the process and do not involve interaction with the environment; hence no exception handling or time constraints are required. Process activities may cause the process to complete, a complex activity to be repeated, a transaction to be aborted, or a nested process to be spawned.

## Flow of Control

The execution of a process entails one or more flows of control, modeled in the form of complex activities.

The `sequence` activity models a flow of control in which all activities execute serially. The flow concludes once the last activity in the sequence has completed.

The `choice` activity models a flow of control in which one activity will execute. The flow concludes after one activity has completed.

The `switch` activity models a flow of control in which zero or more activities will execute, based on the outcome of rules. The flow concludes after all activities have completed or immediatelty if no activity should be executed.

The `foreach` activity models a flow of control in which a sequence of activities will execute over a set of values. The flow concludes after the last sub-activities have completed over the last value in the set.

The `all` activity models a flow of control that comprises multiple flows of control. The flow concludes once all sub-flows have concluded.

A complex activity (sequence or choice) can be repeated indefinitely without recursion.

## States

Each activity represents a state. Complex activities model compound states that include a sub-state for each sub-activity.

The state is entered when it has been determined that the activity will execute. This occurs for the first activity in a sequence, when the previous activity in a sequence has completed, for all activities in an `all` construct, when an activity has been selected in a `switch`, or when a message is available for consumption.

The state lasts until the activity has completed, whether directly or by dependency on sub-activities.

The state may be subject to a time constraint, in which case failure to complete the activity in the specified time duration will cause an exception to occur. Sub-activities are subject to the time constraint of their parent activity.

If an exception occurs and an exception handling activity has been specified, the exception handling activity will execute within the same state. However, the exception handling activity is not subject to any time constraint.

A process can remain within a state indefinitely by repeating a sequence or choice.

### Transactions and Faults

A flow of control may suspend and resume from a transaction context, and may create new transactions. Each complex activity defines its participation in a transaction inherited from a parent activity or communicated from a participant.

Compensating activities are used to recover from completed activities in the event that a transaction aborts and must engage in backward-recovery.

An exception might occur while initiating or executing an activity, prior to its completion. Exception handling activities are used to recover from errors occurring while executing an activity, in order to proceed with the remaining flow of control.

An exception is propagated from a sub-activity to a parent activity, if the sub-activity or its exception handling activity has faulted.

## 5.4 Processes

BPML allows processes to be modeled and instantiated directly, or in response to a message. Nested processes can be defined to allow processes to be bound to the state of a larger process. Abstracts enable the definition of a participant's view of a process.

### Process

A process has a single top-level activity whose execution spans the lifetime of the process.

A process is instantiated by spawning a new instance, or by receiving a message if the process begins with an activity that consumes a message. In the later case, a process will be instantiated for every suitable message.

If the first message consuming activities of a process are contained in a choice, the process will be instantiated for any available message that targets one of these activities.

If the first message consuming activities of a process are contained in an all, the process will be instantiated for any set of available messages that targets all of these activities.

A process will execute until its top-level activity has completed, until it completes explicitly using the `complete` activity, or until its top-level activity ends with an error. In the latter case, the process is said to have aborted.

If the process is engaged in multiple flows of control, it will complete only after all flows of control have completed, whether explicitly or by an exception occurring.

It is permissible for a process to never complete by repetition of a complex activity.

Process data is a context that exists for each process instance and can be used to hold or reference information that is accumulated during the life of the process.

### Nested Process

A nested process can be instantiated from a process instance, while the parent process is in a state in which the nested process is available.

The nested process becomes available if it is defined or referenced in a complex activity, and that complex activity is executing.

When available, the nested process is subject to the same instantiation rules as a top-level process.

Upon instantiation, the nested process obtains a copy of the parent process' process data at the time of instantiation, and may further access and modify the parent's process data.

A nested process is instantiated in the same transaction from which it was spawned, or the transaction communicated to it by the participant.

A nested process completes independently of the parent process. Faults occurring in the nested process do not affect the parent process and vice versa.

A parent process may wait for a nested process to complete before proceeding.

### Abstract

An abstract is the definition of a process that cannot be executed.

Abstracts are used to model processes supported by participants, or to define views of a process with regards to one or more participants.

An abstract needs not be complete and may omit information that is not relevant outside the context of interaction with its participants.

A process that interacts with a participant can be validated for completeness against the abstract of that participant.

A process can declare support for one or more abstracts.

Section 6

# Process Data

The term process data defines a context that exists for each process instance and can be used to hold or reference information that is accumulated during the life of the process.

## 6.1 Definition

The process data is subject to transactional control that depends on the transaction in which an assignment occurs. Nested processes inherit a copy of the parent process' process data, but may perform explicit assignment to the parent process data.

### Structure

Each process instance has a process data context that is independent of any other process instance.

The process data is described as a tree of information, where each item can be referenced by its location within the tree. This model is similar to the tree structured presentation of data in XML documents.

In particular, the process data is a set of zero or more process data elements that can be accessed directly by name.

BPML benefits from the power of the XPath expression language to extract information, perform calculations, and evaluate conditions.

### Transactions

The process data is subject to transactional control.

Coordinated transactions provide isolated serializable transaction isolation, and revert all assignments if the transaction aborts.

Extended transactions and unreliable contexts allow visibility to assignments from unrelated transactions.

Nested transactions make changes visible to the parent transaction after completion.

### Nested Processes

A nested process has a process data that is independent of the parent process, but is allowed access to the parent's process data.

When the nested process is instantiated, its process data is initiated with a copy of the process data of the parent process at the time of instantiation.

Assignments performed within the nested process are not visible to the parent process.

The nested process may perform assignments to the parent process using a prefix of the form `{parent}/` where `parent` is the name of the parent process.

Such an assignment is subject to the transaction context of the nested process.

If it is in the interest of the parent process to be informed of or reject assignments, such assignments should be performed by explicit message passing between the nested process and the parent process.

A nested process cannot perform assignment to the process data of a sibling nested process. A process cannot perform assignment to the process data of a nested process.

*Editor Note: We plan to revise the relation between the nested and parent process's data in a future revision of the specification.*

## 6.2 Assignment

We use the term assignment to describe how data items are mapped to and from the process data. From the process perspective, assignments can be used to accumulate, replace, or emit data.

### Assignment Contexts

An assignment occurs between a source and target contexts. Data from the source context is extracted and manipulated to create data in the target context.

The source context may consist of process data or message contents. The target context may consist of process data or message contents.

### Usage

When assignment elements occur within the `input` element, they perform assignment from the incoming message to the process data. The source context consists of the message contents, and the target context consists of the process data.

When the assignment elements appear within the `output` element, they perform assignment from the process data to the outgoing message. The source context consists of the process data, and the target context constructs the message contents.

When the assignment elements appear as an activity in the process, they perform assignment within the process data. The process data is used for both the source and target contexts. This allows data to be created and replaced.

## Message Assignment

Process inputs are a common case and for convenience we allow a simpler form of assignment to occur. Whenever the `input` element is used without any assign elements, an automatic assignment of the message occurs.

Default message assignment eliminates the need to define explicit assignments for the purpose of accessing data previously communicated to the process.

Once assigned, the message can be referenced by name from the process data. If a previous message with the same name has been assigned, it will be replaced. To override the default assignment, use the assignment elements explicitly.

## Participant Assignment

Participants are considered part of the process data, allowing assignments to be used for communicating participants, and allowing participants to be selected from previous communications.

The `participant` element can reference a participant from the process data using the `select` attribute. Such a reference can yield multiple participants, in which case communication will occur with all the selected participants.

Whenever a nested process is instantiated using the `spawn` activity, it is added as a participant and can be referenced from the process data by its name. A process can always communicate itself (as an instance) using the process name in the `select` attribute.

## Releasing Data

As the process executes, it accumulates data, some of which is only used temporarily. For long-running processes, this might result in a performance penalty.

While data that will not be needed for future activities can be discarded automatically, some processes need an explicit form of control.

The `release` activity can be used to explicitly release data.

## 6.3 Activity Context

*Editor's Note: This section has been removed we plan to revise it and introduce activity context in a future version of the specification.*

Section 7

# Transaction And Exceptions

This section describes the transactional models supported by BPML, the manner in which transactions are declared, controlled, and recovered from, and the manner in which exceptions are handled in an executing process.

## 7.1 Transaction Models

BPML supports two transactional models: coordinated transactions (as known as ACID, X/A, or closed transactions) and extended transactions (as known as open nested transactions or Sagas). In addition, BPML allows processes to execute unreliably outside of any transaction context.

### Coordinated

The term coordinated is used to describe transactions that rely on coordination across all their participants in order to determine whether the transaction can complete or abort.

Coordinated transactions are supported by protocols such as CORBA OTS, X/Open X/A, DTC, and XAML. They usually involve a form of two-phase commit in order to allow their participants to vote whether a transaction can commit or must abort.

Coordinated transactions are susceptible to escalated resource locking, deadlocks, communication failures, and change visibility. Hence, they are typically used for short-lived transactions.

Coordinated transactions are also known as closed or ACID transactions by virtue of preventing changes occurring within one transaction to be visible to other transactions (isolation).

### Extended

The term extended is used to describe transactions that relax the isolation requirements of ACID transactions and render changes visible to other transactions.

Extended transactions can be nested to arbitrary levels, and often are composed of multiple coordinated transactions acting as leafs in a transaction tree.

Extended transactions allow resources to be acquired for short periods of time. Relaxing the isolation requirement allows the interleaving of transactions. The all-or-nothing nature of a transaction is accomplished by means of compensating transactions.

Extended transactions are also known as open nested transactions (ONT) and are used to support long-lived business transactions and interleaving transactions that are an essential part of complex processes.

### Unreliable

The term unreliable denotes a context that makes no guarantee with regards to the reliable completion and the all-or-nothing nature of a transaction.

Processes and activities that do not impose such requirements will execute more efficiently outside the context of a coordinated or extended transaction.

### Transactional Guarantee

Coordinated and extended transactions are guaranteed to execute as a single unit of work, with prescribed forward-recovery, backward-recovery, and retries to deal with failures occurring while executing a transaction.

### All-or-Nothing

All-or-nothing behavior implies that the transaction is atomic, consistent, and durable. The transaction executes as a single unit of work.

The isolation nature of ACID transactions, while advantageous in short-lived transactions, is not desirable in long-lived transactions, or in complex processes that require transaction interleaving.

The BPML transactional model guarantees all-or-nothing behavior for both coordinated and extended transactions.

Isolation is allowed only for coordinated transactions, although not all participants can guarantee the same level of transaction isolation.

BPML processes assure serializable transaction isolation with regards to process data (context maintained by the process instance).

### Forward-Recovery

Forward-recovery guarantees that in case of failure, the process state is restored and its execution continues reliably.

Forward-recovery is required only for those activities that execute within an extended transaction.

Activities that execute within a coordinated transaction will automatically rollback in the event of a failure (backward-recovery) and may be retried.

Activities that execute in an unreliable context are not guaranteed to proceed past a point of failure.

### Backward-Recovery

If the transaction aborts, backward-recovery allows the process state to be restored to a point prior to the beginning of the transaction.

Backward-recovery is automatic for activities that execute within a coordinated transaction.

Backward-recovery relies on compensating transactions for activities that execute within an extended transaction.

Backward-recovery is not provided for activities that execute within an unreliable context.

### Retries

A transaction that fails may be repeated any number of times, if backward-recovery has been successful.

BPML allows transactions to specify a maximum number of retries, although there is no guarantee that a transaction will be retried that number of times.

An idempotent activity is an activity that has no effect on the state of the process and may be repeated any number of times with the same result.

Idempotent activities may be retried if it was impossible to determine whether the activity completed or not (e.g. due to a communication failure).

## 7.2 Transaction Context

Transaction context propagation allows a process to communicate a transactional context to its participants, and activities to be initiated within the transaction of a participant.

The participation of an activity in a transaction depends on the specified transaction type and determines the boundaries of the transaction.

## Transaction Context Propagation

An activity that executes within the context of a transaction will communicate the transaction to its transactional-aware participants.

In reverse, a participant may communicate its transaction context to an activity it initiates.

An activity that supports transactions (supported, required, new, or nested) will execute in the same transaction as the one communicated to it by the participant that initiated the activity.

An activity that requires an associated or dissociated transaction (new or nested) cannot be initiated by a participant that communicates a different transaction than the one that the activity requires.

An activity that supports an existing transaction (required or supported) may execute in the same transaction as its parent activity or a new transaction (required), if not initiated by a transactional-aware participant.

A nested process will be spawned in the transaction context of the activity that spawned it. A nested process will be instantiated in the transaction context of the participant that initiated it.

## Transaction Types

A complex activity may decide to participate in a given transaction, or demarcate its own transaction that will begin when the activity is initiated and complete with the activity.

The activity declares its transaction participation by specifying a transaction type. If no participation type is specified, the default is supported.

The participation type `supported` indicates that the activity supports a transaction, but does not require one. If the activity is initiated within a transaction, it will execute within that transaction.

The participation type `none` indicates that the activity does not support transactions. If the activity is initiated within a transaction, it will suspend from that transaction.

The participation type `required` indicates that the activity requires a transaction. If the activity is initiated by a transactional-aware participant, it will execute within the transaction communicated to it by the participant. Otherwise, the activity will execute within the transaction of its parent activity, or if none is available, execute within a new transaction.

The participation type `new` indicates that the activity requires a new transaction that is different than the one of its parent activity. If the activity is initiated by a transactional-aware participant, it will execute within the transaction communicated to it by the participant. Otherwise, the activity will execute within a new transaction.

The participation type `nested` indicates that the activity requires a new transaction that is nested within the transaction of its parent activity. If the activity is initiated by a transactional-aware participant, the participant must communicate a nested transaction. Otherwise, the activity will execute within a new transaction that is nested within the transaction of its parent activity.

A transaction model (coordinated or extended) must be specified for transactions of type `required`, `new`, and `nested`. No transaction model can be specified for transactions of type `supported` and `none`.

If no transaction type is declared, the default is always `supported`.

### Transaction Boundaries

If a transaction is created for an activity (required, new, or nested), the transaction will begin when the activity is initiated and complete with the activity.

In a coordinated transaction, the transaction will complete after all participants have communicated their agreement to complete the transaction.

A coordinated transaction may abort even after all activities within the transaction have completed, if any participant requests that the transaction abort.

A nested transaction will abort if the parent transaction aborts, even if all activities within the transaction have completed.

If the transaction is not associated with any other transaction, its boundaries are demarcated by the activity. An extended transaction will complete once all activities within the transaction have completed.

Repeating a complex activity for which the transaction type is `required`, `new`, or `nested` does not cause a new transaction to occur.

### Activity Control

Activities occurring within the context of a coordinated transaction against a transactional-aware participant obey the following rules:

- All messages consumed become available for subsequent consumption if the transaction aborts.
- All messages produced are discarded if the transaction aborts.
- All operations performed by the participant will be reverted if the transaction aborts.
- All assignments are discarded if the transaction aborts.

Activities occurring within the context of an extended transaction, or in an unreliable context obey the following rules:

- Messages are consumed and produced immediately.
- All operations performed by the participant are permanent.
- All assignments are permanent.

### Assignments

An assignment performed within the context of a coordinated transaction will only be visible within that transaction. When a nested transaction completes, assignments made by that transaction are visible to the parent transaction.

An assignment performed within the context of an extended transaction or within an unreliable context is visible to other transactions.

This model applies equally to replacement and appending assignments, as well as to releases of process data and default assignments.

## 7.3 Compensating Transactions

The all-or-nothing guarantee might require compensating activities to be prescribed for the purpose of backward-recovery. Each activity that completes may prescribe a compensating activity in order to revert any changes made permanent by the activity.

Compensating activities are guaranteed an order of execution and forward-recovery, and may utilize nested transactions for the purpose of all-or-nothing backward-recovery.

### Backward-Recovery

A transaction that aborts must engage in backward-recovery in order to revert any changes made by the transaction.

Coordinated transactions limited to transactional-aware participants can engage in automatic recovery since no durable changes occur until the transaction completes.

Extended transactions allow durable changes to occur, and must revert such changes in order to restore the process to a state prior to the beginning of the transaction.

Such backward-recovery is aided by compensating activities. Compensating activities are supported by both coordinated and extended transactions, allowing extended transactions to be nested within coordinated transactions.

### Compensating Activities

An activity may prescribe a compensating activity to revert any permanent changes introduced by its completion.

When such an activity completes, the compensating activity is remembered in the context of the transaction.

When the transaction aborts, it will execute all compensating activities in the same transaction context. Compensating activities will be executed in reverse to the order in which they were prescribed, even across nested transactions.

A complex activity may prescribe a compensating activity that will override any compensating activities prescribed by its sub-activities executing within the same transaction.

Compensating activities overridden by parent compensating activities, or compensating activities occurring within other transactions will not execute when the transaction aborts.

Compensating activities are not subject to the time constraints of the activity that prescribed them (but can introduce their own time constraints).

Compensating activities have access to process data available immediately after completion of the activity that prescribed the compensating activity.

The only exception is the transaction context that changes to indicate the transaction has been aborted.

### Transactional Compensation

An aborted transaction will provide forward-recovery guarantee to all compensating activities executing while engaged in backward-recovery.

An aborted transaction cannot provide backward-recovery for compensating activities. If an all-or-nothing behavior is required, compensating activities are allowed to create new transactions.

## 7.4 Exception Handling

Errors may occur at every step of the process. Exception handling allows the process to recover and proceed with execution. Exceptions that are not handled will propagate upward and will cause the transaction to abort. In extended transactions, participants will communicate exceptions to each other.

### Exceptions

Exception handling is engaged if an error occurs while initiating or executing a complex activity, prior to its completion.

An activity can prescribe an exception handling activity, which will be executed if a exception occurs. The exception handling activity might itself cause an exception to occur.

If an exception occurs in an activity and the exception handling activity completes successfully, the activity is considered to have completed successfully.

Exception handling activities are not subject to the time constraints of the exceptioning activity.

## Exception Propagation

Exceptions are propagated upwards from an exceptioning activity to its parent activity within the same flow of control.

When multiple activities are executed in parallel, an exception in one activity will not cause a parallel executing activity to exception, but will cause the parent activity to exception once all parallel activities have completed.

When a top-level activity within a transaction faults, the transaction will be aborted. The exceptions will propagate to the parent activity where it can be handled. The exception will propagate after the transaction has completed backward-recovery.

## Exception Handling vs. Transaction Recovery

Exception handling is used to assure proper execution within a flow of control, while transactions assure reliable completion across multiple flows of control and across participants.

Exception handling allows the process to recover from failure to complete an activity, and recover from non-transactional faults. Compensating activities are used for backward-recovery after an activity has completed.

Exceptions that are not handled will cause the transaction to abort. Exceptions that are handled allow the transaction to complete successfully. Aborting a transaction will prevent the transaction from completing successfully.

While it is possible to use nested transactions and compensating activities to implement exception handling, such a model would demand excessive use of nested transaction and constructs that would hide the explicit modeling of exception handling.

## Transaction Exceptions

If an exception is communicated to a transaction, it causes the transaction to abort. Activities associated with the transaction (or nested transactions) are allowed to complete, but the transaction will invariably abort.

In coordinated transactions, participants generally do not communicate exceptions, but communicate inability to complete to the transaction coordinator.

Extended transactions may involve participants to whom exceptions are communicated automatically and participants to which exceptions must be communicated explicitly.

Compensating activities are often used to communicate exceptions to participants using the `exception` activity.

In some cases, a compensating activity must occur between the process and the participant in order to perform backward-recovery, such as when involving participants that are not transactional-aware (e.g. e-mail messages).

# Section 8 BPML Elements

This section provides a reference of all the BPML elements defined in the BPML XML Schema. Elements are listed in alphabetical order.

For convenience in reading, the BNF notation is used to illustrate the schema of each element. The conformant XML Schema is included in the appendixes.

BPML elements are categorized in five groups:

Activity - An element representing a process activity, simple activity or complex activity.

Core - An element used within a larger context, such as the participant of an activity, the output message, etc.

Definition - An element used for defining a process, declaring a participant, or a message.

Meta-data - An element that provides meta-data information. Meta-data does not affect the manner in which a process is interpreted or executed.

Type - An element that defines a base type from which other elements are derived.

## 8.1 Abstract (definition)

Defines an abstract process.

### Schema

```
<abstract name>
  <annotation/>*
  <meta/>?
  <import/>*
  <schema/>?
  <message/>*
  <participant name/>*
  <ruleSet/>*
  <activity name/>*
  ( simpleActivity | complexActivity )
</abstract>
```

### Usage

The `abstract` element is used to define an abstract process.

The `import` element is used to import definition from a different BPML document. The abstract can then reference participants, messages and processes appearing in any imported document.

The `schema` element is used to define elements, simple types and complex types that will be used in messages and assignments. It is an element of type xsd:schema.

The `message` element defines a message that is specific to the abstract. Messages used in the abstract that are not imported must be declared explicitly.

The `participant` element defines a participant that is specific to the abstract. Participants that are not imported or communicated to the process must be declared explicitly.

The `ruleSet` element defines a rule set that will be referenced in a *switch* activity, used in an input element, or used as dependency to another rule set. Dependencies are not affected by the order in which rule sets are defined.

The `activity` element defines activities that will be referenced within the body of the process.

An abstract consists of exactly one simple or complex activity. This is the top-level activity that will execute for each process instance. The process will complete once this activity completes.

## See Also

- activity (65)
- annotation (68)
- complexActivity (80)
- import (89)
- message (92)
- meta (94)
- participant (102)
- process (104)
- ruleSet (113)
- simpleActivity (119)

## 8.2 Activity (core)

Defines or references an activity.

### Schema

```
<activity ref/>
<activity name>
  ( simpleActivity | complexActivity )
</activity>
```

### Usage

A complex `activity` which completes after all its sub-activities have completed. Sub-activities may execute in parallel.

### Schema

```
<all>
  activity
  activity+
</all>
```

### Usage

The `all` element is a complex activity and extends the base type complexActivity. It can be used any place where an activity can appear.

It models a compound state that includes multiple sub-states, one for each sub-activity, and completes once all its sub-states have completed.

Two or more activities must be specified. Sub-activities might execute in parallel.

### Example

Notify the buyer that an order is ready to ship and ask carrier to ship the order.

```
<all>
  <produce>
    <participant select="buyer" />
    <output message="orderShipping" />
  </produce>

  <produce>
    <participant select="carrier" />
```

```
    <output message="shipOrder" />
  </produce>
</all>
```

## See Also

- complexActivity (80)

## 8.3 All

A complex activity that completes after all its sub-activities have completed. Sub-activities may execute in parallel.

### Schema

```
<all>
  activity
  activity+
</all>
```

### Usage

The `all` element is a complex activity and extends the base type complexActivity. It can be used any place where an activity can appear.

It models a compound state that includes multiple sub-states, one for each sub-activity, and completes once all its sub-states have completed.

Two or more activities must be specified. Sub-activities might execute in parallel.

### Example

Notify the buyer that an order is ready to ship and ask the carrier to ship the order.

```
<all>
  <produce>
    <participant name="buyer"/>
    <output message="orderShipping"/>
  </produce>

  <produce>
    <participant name="carrier"/>
    <output message="shipOrder"/>
  </produce>
</all>
```

### See Also

- complexActivity (80)

## 8.4 Annotation (meta-data)

Provides human readable annotation.

### Schema

```
<annotation xml:lang?>
  mixed content
</annotation>
```

### Usage

The annotation element provides human readable annotation to the BPML element in which it appears.

Multiple annotations in different languages are supported using the xml:lang attribute. Each annotation must specify a different language, and only one annotation is allowed to omit the xml:lang attribute.

Annotations support mixed content that need not validate against any known schema. We recommend the use of XHTML 1.0.

### Example

```
<annotation xml:lang="en">
  Annotation in English using <code>XHTML</code>
</annotation>
```

### See Also

- meta (94)

## 8.5 Assign (core)

Performs assignment.

### Schema

```
<assign select/>

<assign target select/>

<assign target>
  ( <assign/> | <foreach/> | <variable/> )+
</assign>

<variable name select/>

<foreach name select>
  ( <assign/> | <foreach/> | <variable/> )+
</foreach>

<variable name>
  ( <assign/> | <foreach/> | <variable/> )+
</variable>
```

### Description

The assign, foreach and variable elements are used to perform assignment between a source and target context. The assignment elements are used in the *assign* activity, input and output elements, and rule sets.

An assignment occurs between a source context and a target context. The source and target contexts depend on the location in which an assignment element is used.

The source context is equivalent to the definition given by the XPath specification. The select attribute is an XPath expression that operates on the source context and returns an XPath result.

### Assign Element

The `target` attribute specifies a new target context within an existing target context. When this attribute is used, a new target context is created and assigned to the existing target context, and all assignments are performed to the new target context.

A simple assignment evaluates the XPath expression against the source context and assigns the result to an existing target context, or if the `target` attribute is used, to a new target context.

A complex assignment defines a target context and performs a series of nested assignments to that target context.

The `target` attribute takes the form of an element or an attribute name. An attribute name is prefixed with `@`.

## Foreach Element

The `foreach` element evaluates the XPath expression into a set of zero or more values, and iterates the nested assignments over that value set.

Each value in the set is bound to the named variable, and the nested assignments are repeated with the same source and target context, and the new variable bindings.

A bound variable has no namespace associated with it, and is referenced from an XPath expression using the prefix `$` and the same name as specified by the `name` attribute.

## Variable Element

The `variable` element assigns the result of a simple or complex assignment into a bound variable. The bound variable is in scope only for assignments that follow the `variable` element within the same parent assignment.

The variable element creates a new target context and performs all simple and complex assignments to that target context. The target context is bound to the named variable, without affecting any existing target context.

A bound variable has no namespace associated with it, and is referenced from an XPath expression using the prefix `$` and the same name as specified by the `name` attribute.

## Example

Reorder a list of books into a bibliography of authors.

```
<assign target="biblio">
  <foreach name="author"
           select="distinct(library/book/author)">
    <assign target="author">
      <assign target="name" select="$author/text()" />
      <assign select="library/book[author=$author]/title" />
    </assign>
  </foreach>
</assign>
```

The input to the assignment:

```
<library>
  <book>
    <title>Moby Dick</title>
    <author>Herman Melville</author>
   </book>
</library>
```

The output from the assignment:

```
<biblio>
  <author>
    <name>Herman Melville</name>
    <title>Moby Dick</title>
  </author>
</biblio>
```

## See Also

- assign (72)
- input (90)
- output (99)
- processActivity (106)

## 8.6 Assign (activity)

Performs assignment.

### Schema

```
<assign target select append?/>

<assign target append?>
  ( <assign/> | <foreach/> | <variable/> )+
</assign>
```

### Description

The `assign` element is a process activity and extends the base type `processActivity`. It can be used any place where an activity can appear.

The *assign* activity performs assignment from the process data to the process data. Changes to the target context are not reflected until the assignment concludes. Assignments are subject to ACID properties when performed within a coordinated transaction.

The `target` attribute specifies the name of the process data element to which the result is assigned.

If the `append` attribute is true, the assignment will append the new process data element to any existing list of elements with the same name. If the `append` attribute is false (the default), the assignment will replace any process data element(s) with the same name.

A simple assignment uses the `select` attribute to specify an XPath expression, evaluated against the process data. The result is assigned to the target context.

A complex assignment creates a new target context with the specified name, and performs a sequence of nested assignments to that target context, before assigning the target context to the process data.

The `assign`, `foreach` and `variable` elements can be used as nested assignments within the *assign* activity.

### Example

Calculate the total of a purchase order and assign it to the process data element total:

```
<assign target="total"
        select="sum(purchaseOrder/lineItem/price)" />
```

Assigns the SKU's of all line items priced over $500 under the process data element over500:

```
<assign target="over500">
  <foreach name="item"
           select="purchaseOrder/lineItem[price > '500']">
    <assign select="$item/sku" />
  </foreach>
</assign>
```

## See Also

- assign(core) (69)
- processActivity (106)

## 8.7 Choice (activity)

A complex activity which completes after one of its sub-activities have completed, subject to message consumption. Models participant branching.

### Schema

```
<choice>
  activity
  activity+
</choice>
```

### Usage

The `choice` element is a complex activity and extends the base type `complexActivity`. It can be used any place where an activity can appear.

The *choice* activity is used to model participant branching. Participant branching depends on a decision taken by a participant and communicated to the process. Two or more activities must be specified, each must begin with the consumption of a message.

Participant branching models a compound state with multiple sub-states. Once a message has been consumed by any one activity, all other states are discarded immediately.

Multiple activities may accept a message with the same name, but must reference different participants or rules in order to discriminate between messages with the same name.

The order in which activities appear cannot be used to infer priority for the purpose of message consumption.

### Example

On the buyer side a participant branch waits for the order to be accepted and delivered, or completes if the order has been rejected.

```
<choice>
  <sequence>
    <consume>
      <input message="orderAccepted" />
    </consume>
    <!-- Wait for order to be delivered -->
    . . .
  </sequence>
```

```
  <sequence>
    <consume>
      <input message="orderRejected" />
    </consume>
    <complete />
  </sequence>
</choice>
```

## See Also

- complexActivity (80)
- consume (82)
- switch (122)

## 8.8 Compensate (core)

Associates a compensating activity with a simple or complex activity.

### Schema

```
<compensate>
  activity+
</compensate>
```

### Usage

The `compensate` element is used to associate a compensating activity with a simple or complex activity. The compensating activity is used to revert any changes resulting from the completion of the activity in the event of backward-recovery.

The compensating activity will be executed if the activity has completed, but the transaction in which the activity executed has been aborted. Compensating activities will be executed in the reverse order to that in which they were defined.

The compensating activity can reference process data available immediately after completion of the activity.

If a compensating activity is defined for a complex activity, it overrides any compensating activities defined for any of its sub-activities, once the complex activity has completed.

### Example

The inventory participant allows an item to be created or deleted, but is not transactional-aware. If the transaction aborts, the create operation must be compensated for explicitly.

```
<operation>
  <participant select="inventory" />
  <output message="createItemInput" />
  <input message="createItemOutput" />

  <compensate>
    <participant select="inventory" />
    <output message="deleteItemInput">
      <assign select="createItemOutput/sku" />
    </output>
    <input message="deleteItemOutput" />
  </compensate>
</operation>
```

## See Also

- complexActivity (80)
- simpleActivity (119)

## 8.9 Complete (activity)

Completes the process.

### Schema

```
<complete/>
```

### Usage

The `complete` element is a process activity and extends the base type `processActivity`. It can appear as the last activity in a *sequence* or once in a *choice* or *switch*.

The *complete* activity causes the process to complete immediately. It does not cause nested or parent processes to complete.

### See Also

- choice (74)
- sequence (117)
- switch (122)

## 8.10 CompleteBy (core)

Places a time constraint on an activity.

### Schema

```
<completeBy duration>
 <relativeTo ref end?/>?
</completeBy>
```

### Usage

The `completeBy` element may appear in a simple or complex activity to place a time constraint on the completion of the activity.

If a time constraint is placed on an activity and the activity does not complete within the specified time, an exception with the code `bpml:timeout` will occur.

An activity is always constrained by any time constraint placed on a parent activity, and the most restrictive time constraint takes precedence.

An activity's time constraint does not apply to exception handling or compensating activities associated with it.

The `duration` attribute is an XPath expression that evaluates to a time duration, as specified by the XML Schema `timeDuration` data type.

The `relativeTo` element is used to specify a time constraint relative to a previous or parent activity.

If the `end` attribute is true, the time constraint is relative to the completion of the referenced activity and the referenced activity must be a previous activity in a sequence.

If the `end` attribute is false (the default), the time constraint is relative to the beginning of the referenced activity and the referenced activity must be a parent activity, previous activity in a *sequence*, or a parallel activity in an *all*.

### Example

See the definition of `produce` (107).

### See Also

- complexActivity (80)
- schedule (115)
- simpleActivity (119)

## 8.11 ComplexActivity (type)

Base type for the elements `all`, `choice`, `foreach`, `sequence` and `switch` that model complex activities.

### Schema

```
<complexActivity name?>
  <annotation/>*
  <meta/>?
  <completeBy/>?
  <schedule/>?
  <transaction/>?
  <compensate/>?
  <onException/>*
  ( process | abstract | processRef )*
</complexActivity>
```

### Usage

A complex activity is a composition of multiple sub-activities and depends on one or more of its sub-activities to complete.

Each type of complex activity defines a different rule for completion, expressed in the form of an element that extends the base type `complexActivity`.

The *sequence* and *all* activities complete once all their sub-activities have completed, while the *choice*/*switch* activities complete after some of their sub-activities have completed. The *foreach* activity is similar to the *sequence* activity, but iterates over a set of zero or more values.

The optional `transaction` element declares the transactional behavior expected from this activity. If this element is missing, the default type is `supported`.

The optional `completeBy` element places a time constraint on the completion of the activity. The optional `schedule` element schedules the execution of this activity.

The optional `compensate` element prescribes a compensating activity, to be executed for the purpose of backward-recovery if the transaction aborts.

If a compensating activity is prescribed for a complex activity, it overrides any compensating activities prescribed for any of its sub-activities.

The optional `onException` element provides an exception handling activity that will be executed if the complex activity ends with an exception.

If a nested process (or abstract) is defined or referenced, it becomes available while the process is in the state demarcated by this activity. A message targeting this nested process

or the *spawn* activity will instantiate a new instance of the nested process while this activity is executing.

A complex activity can be repeated multiple times without recursion using the *repeat* activity.

## See Also

- all (67)
- choice (74)
- compensate (76)
- completeBy (79)
- foreach (87)
- onException (95)
- repeat (110)
- schedule (115)
- sequence (117)
- switch (122)
- transaction (125)

## 8.12 Consume (activity)

Consumes a message from a participant.

### Schema

```
<consume>
  <participant/>?
  <input/>
  extension
</consume>
```

### Usage

The `consume` element is a simple activity and extends the base type `simpleActivity`. It can be used any place where an activity can appear.

This activity consumes a message provided by a participant. It will wait until a suitable message is available before completing.

If a time constraint is placed on this activity, failure to consume the message in the specified time will cause an exception with the code `bpml:timeout` to occur.

The consumed message is assigned to the process data either explicitly, using the assignment elements, or by default replacement. See the `input` element for more information.

When the *consume* activity appears in a coordinated transaction and references a transactional aware participant, the message is regarded as consumed only if the transaction completes, and is available for subsequent consumption if the transaction aborts.

The participant element restricts the activity to consume a message only from the referenced participant (s). When absent, a message will be accepted from any participant.

If one or more rules are specified, they are evaluated against each available message. A message will be consumed only if all rules evaluate to true.

The manner in which a message is consumed is left to the implementation. Extension elements can be used to provide implementation specific details.

### Example

Wait for one of several buyers to send a purchase order totaling $5000 or more. Ignore all messages from unknown buyers or totaling less than $5000.

```
<consume>
  <participant select="buyers" />
```

```
  <input name="purchaseOrder">
    <rule condition="total >= '5000'"/>
  </input>
</consume>
```

## See Also

- input (90)
- participant (102)
- produce (107)
- simpleActivity (119)

## 8.13 Empty (activity)

An activity that does nothing.

### Schema

```
<empty/>
```

### Usage

The `empty` element is a simple activity and extends the base type `simpleActivity`. It can be used any place where an activity can appear.

An *empty* activity is used to model an activity that occurs outside the process, i.e. the process has no visibility to its execution, nor does it interact with the participant during this activity.

As such, the *empty* activity is not required to reference a participant or specify a time constraint.

### Example

Model a meeting that takes up to four hours to complete, before proceeding with the next activity:

```
<empty name="meeting">
  <completeBy duration="PT4H" />
</empty>
```

### See Also

- simpleActivity (119)

## 8.14 Exception (activity)

Causes an exception to occur in the process, or to be communicated to a participant.

### Schema

```
<exception code?>
  <participant/>?
 ( <reason> any content </reason> |
   <reason select/> )?
</exception>
```

### Usage

The `exception` element is a simple activity and extends the base type `simpleActivity`. It can be used any place where an activity can appear.

When the `participant` element is absent, an exception occurs in the parent complex activity.

The `participant` element causes the exception to be communicated to that participant in the same transaction context. No behavior is prescribed if a failure occurs while communicating the exception.

The `code` attribute provides a global identifier of the exception type, in the form of a URI reference. The exception code can be used to define handling specific to the exception type.

The `reason` element provides a human readable explanation that might be logged or presented visually. It can be specified statically or by referencing process data.

The `completeBy`, `onException` and `compensate` elements have no effect when used within the `exception` element.

The manner in which an exception is communicated to a participant is left to the implementation. Extension elements can be used to provide implementation specific details.

### Example

While attempting to negotiate a trade between a buyer and a seller, an error is detected and the transaction must be aborted. The buyer and seller are immediately notified.

```
<exception code="someErrorCode">
  <participant select="buyer" />
</exception>

<exception code="someErrorCode">
```

```
  <participant select="seller" />
</exception>
```

## See Also

- simpleActivity (119)

## 8.15 Foreach (activity)

A complex activity which repeats a sequence of sub-activities over a set of values.

### Schema

```
<foreach name select>
  activity+
</foreach>
```

### Description

The `foreach` element is a complex activity and extends the base type `complexActivity`. It can be used any place where an activity can appear.

The *foreach* activity evaluates an expression against the process data and repeats its sub-activities once for each value in the set. The sub-activities transition as for the *sequence* activity. This activity completes once the last sub-activity has completed over the last value in the set.

The `select` attribute specifies an XPath expression evaluated against the process data, resulting in a set of zero or more values.

The resulting set is guaranteed to have some internal order, and its values will be accessed by that order. XPath extension functions must be used to guarantee distinct values or a particular order.

The *foreach* activity will repeat all its sub-activities in sequence in the same manner as for the *sequence* activity. Each sequence will be repeated once for each value in the set. The current value can be referenced from an XPath variable with the same name as the *foreach* activity (prefixed with $).

### Example

Process each line item in the purchase order and send a message containing just that one line item:

```
<foreach name="item"
         select="purchaseOrder/lineItem">
  <produce>
    <output message="processLineItem">
      <assign select="$item" />
    </output>
  </produce>
</foreach>
```

Simulate a *for* loop by creating a set of values from 1 to 10:

```
<foreach name="index"
         select="set(1,10)">
  . . .
</foreach>
```

## See Also

- assign (core) (69)
- complexActivity (80)
- sequence (117)

## 8.16 Import (definition)

Imports a process (or abstract) definition.

### Schema

```
<import href/>
```

### Usage

Allows a process to reference schema, message and participant definitions appearing in another (the imported) process definition.

The `href` attribute specifies a URI for the location of a BPML package to import. The fragment identifier can be used to reference a particular process or abstract definition within that package.

### See Also

- abstract (63)
- process (104)

## 8.17 Input (core)

Accepts a message delivered to the process.

### Schema

```
<input message>
  <rule/>*
  ( <assign/> | <foreach/> | <variable/> )*
</input>
```

### Usage

The `input` element is used in the *consume* and *operation* activities to accept a message delivered from a participant to the process (process input).

The `message` attribute references the relevant message definition.

The assignment elements are used to perform assignment from the message contents to the process data. Multiple assignments are used for multi-part messages.

If no assignments are specified, a default assignment by replacement will occur and the message may be referenced from the process data by its name.

If rules are specified, they are evaluated against each available message. A message will be accepted only if all rules evaluate to true. The rule conditions can reference the message contents as well as the process data.

### Example

See the definition of `consume` (82).

### See Also

- assign (72)
- consume (82)
- operation (96)
- rule (112)

## 8.18 Join (activity)

Waits for a nested process to complete before proceeding.

### Schema

```
<join select/>
```

### Usage

The `join` element is a process activity and extends the base type `processActivity`. It can be used any place where an activity can appear.

This activity completes after all nested processes by this name have completed executing. This activity will wait only for nested processes instantiated from the same process in which the activity is executed.

If no nested processes by this name are executing when this activity occurs, it completes immediately.

The `select` attribute is an XPath expression that evaluates to zero or more participants that represent nested processes. The *join* activity will ignore any participants that are not nested processes instantiated from the same process.

### Example

Spawn three nested processes for posting a bid and wait until all three bids have been accepted or rejected before proceeding.

```
<spawn ref="postBid" />
<spawn ref="postBid" />
<spawn ref="postBid" />
<join select="postBid" />
```

### See Also

- processActivity (104)

## 8.19 Message (definition)

Defines a message that can be communicated between a process and a participant.

### Schema

```
<message name type?>
 <annotation/>*
 <meta/>?
 ( element | complexType )*
 extension
</message>
```

### Usage

The `message` element defines a message used by the process to communicate with participants.

The message definition is referenced by name from the `input` and `output` elements using their `message` attribute.

Messages of simple composition are defined using one or more elements (`xsd:element`), while messages of complex body composition are defined using complex types (`xsd:complexType`).

The message type can be `data`, `request` or `response`, with the default being `data`. There is no need to define messages for exceptions.

The manner in which messages are represented, constructed and delivered is left to the implementation. Extension elements can be used to provide implementation specific details.

### Example

Defines two messages, a request comprising a list of one or more products or services, and a result providing the total price.

```
<message name="calcTotalInput" type="request">
  <xsd:complexType>
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element ref="product" />
      <xsd:element ref="service" />
    </xsd:choice>
  </xsd:complexType>
</message>
```

```
<message name="calcTotalOutput">
  <xsd:element name="totalPrice" type="xsd:double" />
</message>
```

## See Also

- input (90)
- output (99)

## 8.20 Meta (meta-data)

Provides meta data about an element.

### Schema

```
<meta>
  any element*
</meta>
```

### Usage

The `meta` element provide additional information about a BPML element for the purpose of advertising, searching and categorizing.

It contains a list of elements in various namespaces. Each element constitutes a name-value pair, with the fully qualified element name serving as the pair name, and its contents as the pair value. Such elements need not validate against any known schema.

### Example

Meta data for a purchase order process using Dublin Core and CVS revision numbering.

```
<meta>
  <dc:title>Manage Purchase Order</dc:title>
  <dc:creator>Arkin, Assaf</dc:creator>
  <dc:date>11-01-2000</dc:date>
  <cvs:revision>1.2</cvs:revision>
</meta>
```

### See Also

- annotation (68)

## 8.21 OnException (core)

Associates an exception handling activity with a complex activity.

### Schema

```
<onException code?>
  activity+
</onException>
```

### Usage

The onException element is used to associate an exception handling activity with a complex activity. It is used to recover from exceptions and allow the process to continue.

The exception handling activity will be executed if the activity ends with an exception. If the exception handling activity completes successfully, then the failed activity also completes successfully.

Multiple onException elements may be used to handle different exception codes. A set of onException elements must use unique codes and at most on element may omit the code attribute.

### Example

See the definition of produce (107).

### See Also

- complexActivity (80)
- exception (85)

## 8.22 Operation (activity)

A synchronous operation.

### Schema

```
<operation>
  <participant/>
  <output/>
  <input/>
  extension
</operation>

<operation>
  <participant/>?
  <input/>
  <output/>
  <exception code/>*
  extension
</operation>
```

### Usage

The `operation` element is a simple activity and extends the base type `simpleActivity`. It can be used any place where an activity can appear.

An operation involves a synchronous request/response message exchange with a possible exception message. It is shorthand for a transactional-bound *consume*/*produce* pair of activities.

When an operation is invoked, it delivers a request message and waits for a response message. If an exception is communicated, the operation will end with an exception.

If a time constraint is specified, failure to complete the operation in the specified time will cause an exception with the code `bpml:timeout` to occur.

When the *operation* activity appears in a coordinated transaction, the operation is regarded as permanent only if the transaction completes and is retracted if the transaction aborts.

The manner in which the process communicates with a participant is left to the implementation. Extension elements can be used to provide implementation specific details.

The `operation` element can be used in two contexts, either to invoke an operation or to implement an operation.

### Invoking

In this context the `operation` element defines an activity that invokes an operation against the participant.

It will produce the outgoing message through assignment, and wait for the incoming message in order for the activity to complete.

The `participant` element is mandatory. The `output` element must precede the `input` element.

### Implementing

In this context the `operation` element defines an activity that implements an operation invoked by a participant.

It will consume the incoming message and produce an outgoing message through assignment.

The `participant` element is optional and can be used to restrict the participants that may invoke the operation. The `input` element must precede the `output` element.

The operation may declare possible exceptions that might occur using the `exception` element. Each exception declaration must use a unique code.

### Example

Defines an operation that deposits funds in an account. The operation is performed against an account process using two messages defined within that process. An exception occurs if the account is closed.

```
<operation>
  <input message="depositFundsInput" />
  <output message="depositFundsOutput" />
  <exception code="accountClosed" />
</operation>
```

Invokes an operation to deposit funds. The account is used as the participant, a message is constructed and the operation completes once the funds have been deposited.

```
<operation>
  <participant select="account" />
  <output message="depositFundsInput">
    <assign select="amount" />
  </output>
  <input message="depostFundsOutput" />
</operation>
```

## See Also

- exception (85)
- input (90)
- output (99)
- participant (102)

## 8.23 Output (core)

Constructs a message delivered by the process.

### Schema

```
<output message>
  ( <assign/> | <foreach/> | <variable/> )*
</output>
```

### Usage

The `output` element is used in the *produce* and *operation* activities to construct a message delivered by the process to a participant (process output).

The `message` attribute references the relevant message definition.

The assignment elements are used to construct the contents of the message by assignment from the process data.

Multiple assignments are used for multi-part messages, no assignments are required for empty messages.

### See Also

- assign (69)
- operation (96)
- produce (107)

## 8.24 Package (definition)

The root element of every BPML document.

### Schema

```
<package namespace>
  <annotation/>*
  <meta/>?
  ( <process/> | <abstract/> ) +
</package>
```

### Usage

The package element is the root element of every BPML document. It is used to group together one or more executable or abstract process definitions for the purpose of interchange, deployment or modeling.

All process definitions contained within the package element share the same namespace URI, as declared by the namespace attribute. The namespace URI is used for all message, participant and activity names declared within these process definitions.

### Example

A package for a purchase order process:

```
<package namespace="http://mycommerce.com"
         xmlns="http://www.bpmi.org/BPML"
         xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

  <process name="purchaseOrder">
    . . .
    <activity name="sendPO">
      . . .
    </activity>
  </process>

</package>
```

In the above package the elements `package` and `process` are defined in the namespace `http://www.bpmi.org/BPML`, used for all BPML elements.

The process `purchaseOrder` and the activity `sendPO` are defined in the namespace `http://mycommerce.com`.

## See Also

- abstract (63)
- process (104)

## 8.25 Participant (definition, core)

Defines or references a participant of the process.

### Schema

```
<participant name>
  <annotation/>*
  <meta/>?
  extension
</participant>

<participant select>
  extension
</participant>
```

### Usage

The participant element is used to define a participant in the process (or abstract), or reference a participant within a simple activity.

When the participant is defined in the process or abstract, it defines a static participant, and must be named using the name attribute. Such a definition may include annotations, meta-data and extension elements.

When referencing a defined participant from a simple activity, the select attribute is be used with the same name as that used in the participant definition.

When referencing a dynamic participant from the process data, the select attribute is used. It takes the form of an XPath expression that results in one or more participants.

Using the select attribute, two participant elements may resolve to the same participant. The process maintains a conversational state across all interactions with the same participant.

The manner by which the process establishes communication with the participant and applies the proper security restrictions is left to the implementation. Extension elements can be used to provide implementation specific details.

### Example

The following example illustrates three uses of the participant element. The first example defines a static participant. The second example consumes a message from that static participant. The last example produces a message to a participant named in a previously consumed message.

```
<participant name="manager">
  . . .
</participant>

<consume>
  <participant select="manager" />
  <input message="confirmVacation" />
</consume>

<produce>
  <participant select="vacationRequest/employee" />
  <output message="vacationConfirmed" />
</produce>
```

## See Also

- abstract (63)
- consume (82)
- exception (85)
- operation (96)
- process (104)
- produce (107)

## 8.26 Process (definition)

Defines a process.

### Schema

```
<process name>
  <supports abstract/>*
  <annotation/>*
  <meta/>?
  <import/>*
  <schema/>?
  <message/>*
  <participant name/>*
  <ruleSet/>*
  <activity name/>*
  ( simpleActivity | complexActivity )
  extension
</process>

<processRef ref/>
```

### Usage

The `process` element is used to define a top-level or nested process.

The `import` element is used to import definition from a different BPML document. The process can then reference participants, messages and processes appearing in any imported document.

The `schema` element is used to define elements, simple types and complex types that will be used in messages and assignments. It is an element of type `xsd:schema`. It validates against the XML Schema.

The `message` element defines a message that is specific to the process. Messages used in the process that are not imported must be declared explicitly.

The `participant` element defines a participant that is specific to the process. Participants that are not imported or communicated to the process must be declared explicitly.

The `ruleSet` element defines a rule set that will be referenced in a *switch* activity, used in an `input` element, or used as dependency to another rule set. Dependencies are not affected by the order in which rule sets are defined.

The `activity` element defines activities that will be referenced within the body of the process.

A process consists of exactly one simple or complex activity. This is the top-level activity that will execute for each process instance. The process will complete once this activity completes.

The `processRef` element is used to reference a process definition within a complex activity, for the purpose of declaring a nested process.

Extension elements can be used to provide implementation specific details.

## See Also

- abstract (63)
- activity (65)
- annotation (68)
- complexActivity (80)
- import (89)
- message (92)
- meta (94)
- participant (102)
- ruleSet (113)
- simpleActivity (119)

## 8.27 ProcessActivity (type)

Base type for the elements `assign`, `complete`, `join`, `release`, `repeat` and `spawn` that model process activities.

### Schema

```
<processActivity name?>
  <annotation/>*
</processActivity>
```

### Usage

A *process* activity is an activity that is internal to the process and does not involve any participant.

Each type of process activity has a different effect on the process, expressed in the form of an element that extends the base type `processActivity`.

A compensating activity is automatically prescribed for the assign and release activities.

It is an error to reference a process activity using the `activity` element. A process activity may be referenced by name from a `case` element.

### See Also

- assign (69)
- complete (78)
- join (91)
- release (109)
- repeat (110)
- spawn (121)

## 8.28 Produce (activity)

Produces a message to a participant.

### Schema

```
<produce>
  <participant/>
  <output/>
  extension
</produce>
```

### Usage

The `produce` element is a simple activity and extends the base type `simpleActivity`. It can be used any place where an activity can appear.

This activity produces a message and delivers it to the participant. The message contents is constructed using assignment (see the `output` element).

If a time constraint is specified, the message will be available to the participant only for the specified time duration, and this activity will complete only after the message has been consumed.

Failure to consume the message in the specified time will cause the message to be retracted and an exception with the code `bpml:timeout` to occur.

If no time constraint is specified, the message will be available indefinitely and this activity will complete immediately after the message has been produced.

When the *produce* activity appears in a coordinated transaction and references a transactional aware participant, the message is regarded as permanent only if the transaction completes, and is discarded if the transaction aborts.

The manner in which a message is produced and communicated to the participant is left to the implementation. Extension elements can be used to provide implementation specific details.

### Example

Send a bid to a marketplace and wait for 20 minutes for the message to be consumed. If the message was not consumed within 20 minutes, notify the user.

```
<sequence>

  <produce>
    <completeBy duration="PT20M" />
```

```
    <participant select="marketPlace" />
    <output message="postBid" />
  </produce>

  <onException code="bpml:timeout">
    <produce>
      <participant select="operator" />
      <output message="errorPlacingBid" />
    </produce>
  </onException>

</sequence>
```

## See Also

- consume (82)
- output (99)
- participant (102)
- simpleActivity (119)

## 8.29 Release (activity)

Releases an assignment from the process data.

### Schema

```
<release target/>
```

### Usage

The `release` element is a process activity and extends the base type `processActivity`. It can be used any place where an activity can appear.

The `target` attribute refers to a previous assignment in the process data that will be discarded once this activity completes.

The *release* activity can be used to discard previous assignments made by the *assign* activity, or made by default message and participant assignments. It is not an error to release an assignment that did not occur.

Like assignments, releases are subject to ACID properties when performed within a coordinated transaction.

### See Also

- assign (69)
- processActivity (106)

## 8.30 Repeat (activity)

Repeats a parent activity.

### Schema

```
<repeat ref/>
```

### Usage

The `repeat` element is a process activity and extends the base type `processActivity`. It can appear as the last activity in a *sequence* or once in a *choice* or *switch*.

The *repeat* activity is used to repeat the referenced complex activity without recursion. It models a transition to the beginning of a state or a parent state, or the repetition of an executing activity.

It must be used as a sub-activity of the complex activity which it references.

### Example

Perform once and repeat performing while condition is met (do .. while loop).

```
<sequence name="onceOrMore">

  <!-- Body of loop -->
  . . .

  <switch>
    <case condition="...someRule...">
      <repeat ref="onceOrMore" />
    </case>
  </switch>
</sequence>
```

Perform only while condition is met (while loop).

```
<switch name="zeroOrMore">
  <case condition="...someRule...">
    <sequence name="body">

      <!-- Body of loop -->
      . . .

      <repeat ref="zeroOrMore" />
    </sequence>
```

```
  </case>
</switch>
```

## See Also

- processActivity (106)
- sequence (117)
- switch (122)

## 8.31 Rule (core)

Evaluates a rule.

### Schema

```
<rule condition/>

<rule ruleSet negative?/>
```

### Usage

The `rule` element is used to evaluate a rule. A rule is a condition, or dependency on a rule set, that can be used in the `input` element, or in a rule set.

When referenced by the `input` element, rules are evaluated against the contents of a message. All rules must evaluate to true for the message to be consumed.

When referenced by the `case` element, rules are evaluated against the process data. The evaluation of a rule determines which activities are candidate for execution.

Simple rules are expressed in BPML using the `rule` element. The `rule` element uses the `condition` attribute to specify an XPath expression that evaluates to a boolean value.

Alternatively, the `rule` element can use the `ruleSet` attribute to reference a previously defined rule set. In this case, the result of the rule is identical to the result of the rule set, or contrary if the `negative` attribute is true. If the referenced rule set cannot be evaluated, the result of the rule is always false.

### Example

See `ruleSet` (113).

### See Also

- input (90)
- ruleSet (113)
- switch (122)

## 8.32 RuleSet (definition)

Defines a rule set.

### Schema

```
<ruleSet name>
  <annotation/>*
  <meta/>?
  ( <variable/> | <rule/> )*
  extension
</ruleSet>
```

### Usage

The `ruleSet` element is used to define a rule set. A rule set is a collection of rules that can be referenced by another rule and by the `case` element. Rule sets are declared as part of a process or abstract definition.

Rule sets can be defined in BPML using the `ruleSet` element. Multiple rules appearing inside a rule set imply logical *and*. This notation is for convenience in reading, other forms of logical composition are fully supported by XPath. In addition variables can be used to retain the result of temporary calculations over consecutive `rule` elements.

Rule sets can be defined in other means, or reference an external rule engine, or another mechanism for evaluating rules. For this purpose the `ruleSet` element supports extension elements that can be used to define external rules in an implementation specific manner.

### Example

- A repeating customer is a customer that visited us more than once. A new customer is not a repeating customer.

- A valued customer is a customer that bought goods worth $500 or more, but must be a repeating customer.

```
<ruleSet name="repeatingCustomer">
  <rule condition="visits > '1'" />
</ruleSet>

<ruleSet name="newCustomer">
  <rule ruleSet="repeatingCustomer" negative="true" />
</ruleSet>
```

```
<ruleSet name="valuedCustomer">
  <rule ruleSet="repeatingCustomer" />
  <rule condition="totalPurchase >= '500'" />
</ruleSet>
```

## See Also

- abstract (63)
- process (104)
- rule (112)

## 8.33 Schedule (core)

Schedules the execution of an activity.

### Schema

```
<schedule duration>
  <relativeTo ref end?/>
</schedule>

<schedule instant/>
```

### Usage

The `schedule` element may appear in a simple or complex activity to schedule the execution of that activity.

If a schedule is associated with an activity, the activity will begin not before the next scheduled time instant. It is possible that an activity will be forced to complete before its scheduled time due to a time constraint placed on the activity.

The `duration` attribute is an XPath expression that evaluates to a time duration, as specified by the XML Schema `timeDuration` data type.

The `relativeTo` element is used in combination with the `duration` attribute to specify a schedule relative to a previous or parent activity.

If the `end` attribute is true, the schedule is relative to the completion of the referenced activity and the referenced activity must be a previous activity in a *sequence*.

If the `end` attribute is false (the default), the schedule is relative to the beginning of the referenced activity and the referenced activity must be a parent activity, previous activity in a *sequence*, or a parallel activity in an *all*.

The `instant` attribute is an XPath expression that evaluates to a recurring time, as specified by the XML Schema `timeDuration` data type. The `instant` attribute specifies a schedule that is independent of any other activity.

The `duration` and `instant` attributes are mutually exclusive.

### Example

Schedule a sequence of activities to occur at the end of the business day (after 5PM):

```
<sequence>
  <schedule instant="'PT17H'" />
  . . .
</sequence>
```

## See Also

- complexActivity (80)
- completeBy (79)
- simpleActivity(119)

## 8.34 Sequence (activity)

A complex activity which completes after all its sub-activities have completed. All sub-activities are executed in sequence.

### Schema

```
<sequence>
  activity+
</sequence>
```

### Usage

The sequence element is a complex activity and extends the base type complexActivity. It can be used any place where an activity can appear.

A sequence models a compound state that transitions through a series of sub-states, one for each sub-activity. A transition is triggered upon the completion of a previous activity. This activity completes once the last sub-activity has completed.

It is not uncommon to find a sequence nested within another sequence for the purpose of modeling nested transactions.

### Example

Open a bank account and deposit money in it.

```
<sequence>
  <operation>
    <participant select="bank" />
    <output message="openAccountInput" />
    <input message="openAccountOutput" />
  </operation>

  <operation>
    <participant select="openAccountOutput/account" />
    <output message="depositInput" />
    <input message="depositOutput" />
  </operation>
</sequence>
```

## See Also

- complexActivity (80)

## 8.35 SimpleActivity (type)

Base type for the elements `consume`, `empty`, `exception`, `operation` and `produce` that model simple activities.

### Schema

```
<simpleActivity name? idempotent?>
  <annotation/>*
  <meta/>?
  <completeBy/>?
  <schedule/>?
  <compensate/>?
</simpleActivity>
```

### Usage

A simple activity is an activity that involves a participant of the process. It communicates with the participant by consuming a message, producing a message, performing an operation, or communicating an exception.

Each type of simple activity defines a different interaction, expressed in the form of an element that extends the base type `simpleActivity`.

The `completeBy` element places a time constraint on the completion of the activity. The `schedule` element can be used to schedule the execution of this activity.

The `compensate` element prescribes a compensating activity, to be executed for the purpose of backward-recovery if the transaction aborts.

The `participant` element is required only if the activity produces a message or performs an operation, communicates an exception to a participant, or consumes a message from a specific participant.

The `idempotent` attribute declares whether an activity is idempotent and can be retried if it was impossible to determine whether the activity executed. In its absence, the default (false) is assumed.

The *empty* activity is a simple activity that allows the modeling of an activity performed by a participant outside the scope of the process.

### See Also

- compensate (76)
- completeBy (79)
- consume (82)

- empty (84)
- exception (85)
- operation (96)
- produce (107)
- schedule (115)

## 8.36 Spawn

Spawns a nested process.

### Schema

```
<spawn ref/>
```

### Usage

The `spawn` element is a process activity and extends the base type `processActivity`. It can be used any place where an activity can appear.

The *spawn* activity causes a new instance of the referenced nested process to be instantiated. The nested process must be available in the context in which this activity occurs.

The nested process is instantiated in the same transaction as the spawn activity, and maintains a copy of the process data at the time it was instantiated.

Once instantiated, the nested process can be referenced as a participant from the process data of the parent process.

### Example

See the definition of `join` (91).

### See Also

- process (104)
- simpleActivity (119)

## 8.37 Switch (activity)

A complex activity which completes after zero or more of its sub-activities have completed, subject to one or more conditions. Models process branching.

### Schema

```
<switch exclusive?>
 <case/>+
 <otherwise/>?
</switch>

<case condition probability?>
  activity
</case>

<case ruleSet negative? probability?>
  activity
</case>

<otherwise>
  activity
</otherwise>
```

### Usage

The `switch` element is a complex activity and extends the base type `complexActivity`. It can be used any place where an activity can appear.

The *switch* activity models process branching. One or more rules are evaluated to determine which activity(s) to execute next.

The `case` element associates an activity with the outcome of a condition or rule set. That activity will be candidate for execution if the condition or rule set evaluates to true. At least one `case` element must exist.

The `condition` attribute specifies a rule in the form of an XPath expression. The XPath expression evaluates to a boolean value.

The `ruleSet` attribute references a rule set definition. The rule set is evaluated to select zero or more activities as candidate for execution.

If the `exclusive` attribute is true (the default), and multiple activities are candidate for execution, the candidate list will be narrowed down to the first candidate activity, based on the order of `case` elements.

If no *case* activity is candidate for execution, and the `otherwise` element is used, the activity defined in that element becomes candidate for execution. Otherwise, no activity is candidate for execution.

If it is determined that no activity is candidate for execution, the *switch* activity completes immediately. This is a shorthand for including an additional *empty* activity for the default case. A *switch* using a single activity can be used to model deferred activities.

If it is determined that one or more activities are candidate for execution, the *switch* activity completes after all activities have completed. Multiple activities are executed in the same manner as for the *all* activity.

The order in which `case` elements appear does not affect the order in which rules are evaluated, and cannot be used to infer priority for rules.

The `probability` attribute can be used to specify the probability of a rule. The probability is expressed as a numeric value in the range 0.0 to 1.0 (exclusive) and is used for the purpose of modeling and analysis.

## Example

- On the seller side a process branch uses a rule to determine whether an order can be accepted. It informs the buyer of the decision, and proceeds to deliver the order if it has been accepted.

```
<switch>

  <case condition="...someRule...">
    <sequence name="accept">
      <produce>
        <output message="orderAccepted" />
      </produce>
      <!-- Now deliver the order -->
      . . .
    </sequence>
  </case>
```

```
  <otherwise>
    <sequence name="reject">
       <produce>
        <output message="orderRejected" />
      </produce>
      <complete />
    </sequence>
  </otherwise>

</switch>
```

## See Also

- choice (74)
- complexActivity (80)
- rule (112)
- ruleSet (113)

## 8.38 Transaction (core)

Declares the transactional behavior of a complex activity.

### Schema

```
<transaction type model? repeat?/>
```

### Usage

The `transaction` element may appear in any complex activity to declare the expected transactional behavior.

If absent, the transactional behavior is inherited from the parent activity. The top most activity in a process defaults to the type `supported`.

The transactional behavior model and participation type are specified using the `model` and `type` attributes, respectively.

The following transactional models are supported:

- coordinated All-or-nothing behavior is guaranteed by coordinating transaction completion across all transactional-aware participants.
- extended All-or-nothing behavior is guaranteed through forward-recovery in the event of a failure, and backward-recovery in the event of transaction abortion.

The following types of transaction participation are supported:

- supported A transaction is supported. The activity will participate in a transaction if one is provided when the activity is initiated, but will not begin a new transaction.
- required A transaction is required. The activity will participate in a transaction if one is provided when the activity is initiated, or if no transaction exists, demarcate a new transaction for the duration of the activity.
- new A new transaction is required. The activity requires a new transaction that is independent of any transaction associated with the parent activity.
- nested A nested transaction is required. The activity requires a new transaction that is nested within the transaction associated with the parent activity.
- none Transactions are not supported. The activity will suspend from any transaction provided when the activity is initiated.

The transaction types `required`,`new` and `nested` must specify which transaction model to use.

The transaction type `supported` inherits the transaction from the parent activity or initiating participant, and is not allowed to declare a transaction model. The transaction type `none` is not allowed to declare a transaction model either.

The `repeat` attribute specifies the maximum number of times a transaction can be repeated until successful completion. If this value is higher than one, the transaction will be repeated if it failed to complete, but succeeded in backward recovery.

The value of the `repeat` attribute is an XPath expression which must evaluate to a positive integer. In its absence the default (one) is assumed and the transaction will occur at most once.

## See Also

- compensate (76)
- complexActivity (80)
- exception (85)
- onException (95)

# Appendix A: BPML XML Schema

The BPML XML Schema is defined in the XML schema document located at http://www.bpmi.org/2001/3/BPML.xsd.

We recommend the use of the namespace prefix `bpml` to denote the namespace URI `http://www.bpmi.org/BPML`.

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
            xmlns:bpml="http://www.bpmi.org/BPML"
            targetNamespace="http://www.bpmi.org/BPML">


  <!-- The package element is the root element of all BPML documents -->

  <xsd:element name="package">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        The package element is the root element of all BPML documents
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:group ref="metaData" />
      <xsd:choice minOccurs="1" maxOccurs="unbounded">
        <xsd:element ref="abstract" />
        <xsd:element ref="process" />
      </xsd:choice>
      <xsd:attribute name="namespace" type="uri"
                     use="required" />
    </xsd:complexType>
  </xsd:element>


  <!-- Process definition elements -->

  <!-- Process abstract definition -->
  <xsd:element name="abstract">
    <xsd:annotation>
```

```
    <xsd:documentation xml:lang="en">
      Process abstract definition
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="processDefinition">
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<!-- Message definition -->
<xsd:element name="message">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Message definition
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="metaData" />
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="xsd:element" />
        <xsd:element ref="xsd:complexType" />
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="name" type="name"
                   use="required" />
    <xsd:attribute name="type" type="messageType"
                   use="optional" value="data" />
  </xsd:complexType>
</xsd:element>

<!-- Participant definition -->
<xsd:complexType name="participantDef">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Participant definition
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:group ref="metaData" />
    <xsd:group ref="extension" />
```

```
  </xsd:sequence>
  <xsd:attribute name="name" type="name"
                 use="required" />
</xsd:complexType>

<!-- Process definition -->
<xsd:element name="process">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Process definition
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="processDefinition">
        <xsd:sequence>
          <xsd:group ref="extension" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<!-- Base type for process and abstract definitions -->
<xsd:complexType name="processDefinition">
  <xsd:sequence>
    <xsd:element name="supports"
                 minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:attribute name="abstract" type="name"
                       use="required" />
      </xsd:complexType>
    </xsd:element>
    <xsd:group ref="metaData" />
    <xsd:element name="import"
                 minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:attribute name="href" type="uri"
                       use="required" />
      </xsd:complexType>
    </xsd:element>
    <xsd:element ref="xsd:schema"
                 minOccurs="0" maxOccurs="1" />
    <xsd:element ref="message"
```

```
                minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="participant" ref="participantDef"
                minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="ruleSet"
                minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="activity"
                minOccurs="0" maxOccurs="unbounded" />
      <xsd:complexType>
        <xsd:complexContent>
          <xsd:choice>
            <xsd:group ref="anyComplexActivity" />
            <xsd:group ref="anySimpleActivity" />
          </xsd:choice>
        </xsd:complexContent>
        <xsd:attribute name="name" type="name"
                       use="required" />
      </xsd:complexType>
    <xsd:choice>
      <xsd:group ref="anyComplexActivity"/>
      <xsd:group ref="anySimpleActivity"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="name" type="name"
                use="required" />
</xsd:complexType>

<!-- Rule set definition -->
<xsd:element name="ruleSet">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Rule set definition
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="metaData" />
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="rule"/>
        <xsd:element name="variable" type="variableAssignment" />
      </xsd:choice>
      <xsd:group ref="extension" />
    </xsd:sequence>
    <xsd:attribute name="name" type="name"
                   use="required" />
```

```
  </xsd:complexType>
</xsd:element>


<!-- Meta data and annotation elements and metaData group -->

<!-- Element for an annotation -->
<xsd:element name="annotation" type="xsd:anyType">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Annotation
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<!-- Element for meta-data -->
<xsd:element name="meta">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Meta data
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:any processContents="skip"
           minOccurs="0" maxOccurs="unbounded" />
     </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!-- Meta data group includes zero or more annotation elements
     and zero or one meta element in that order -->
<xsd:group name="metaData">
  <xsd:sequence>
    <xsd:element ref="annotation"
                 minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="meta"
                 minOccurs="0" maxOccurs="1" />
  </xsd:sequence>
</xsd:group>


<!-- Base types for simple and complex activities -->
```

```
<!-- Base type for all complex activities -->
<xsd:complexType name="complexActivity">
  <xsd:sequence>
    <xsd:group ref="metaData" />
    <xsd:element ref="completeBy"
                 minOccurs="0" maxOccurs="1" />
    <xsd:element ref="schedule"
                 minOccurs="0" maxOccurs="1" />
    <xsd:element ref="transaction"
                 minOccurs="0" maxOccurs="1" />
    <xsd:element ref="compensate"
                 minOccurs="0" maxOccurs="1" />
    <xsd:element ref="onException"
                 minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="name"
                 use="optional" />
</xsd:complexType>

<!-- Group for nested process available in a complex activity -->
<xsd:group name="nestedProcess">
  <xsd:element ref="abstract" />
  <xsd:element ref="process" />
  <xsd:element name="processRef">
    <xsd:complexType>
      <xsd:attribute name="ref" type="name"
                     use="required" />
    </xsd:complexType>
  </xsd:element>
</xsd:group>

<!-- Group for selecting any one complex activity -->
<xsd:group name="anyComplexActivity">
  <xsd:choice>
    <xsd:element name="all" type="allActivity" />
    <xsd:element name="choice" type="choiceActivity" />
    <xsd:element name="foreach" type="foreachActivity" />
    <xsd:element name="sqeuence" type="sequenceActivity" />
    <xsd:element name="switch" type="switchActivity" />
  </xsd:choice>
</xsd:group>

<!-- Base type for all process activities -->
<xsd:complexType name="processActivity">
```

```
  <xsd:sequence>
    <xsd:element ref="annotation"
                 minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="name"
                 use="optional" />
</xsd:complexType>

<!-- Group for selecting any one process activity -->
<xsd:group name="anyProcessActivity">
  <xsd:choice>
    <xsd:element name="assign" type="assignActivity" />
    <xsd:element name="complete" type="completeActivity" />
    <xsd:element name="join" type="joinActivity" />
    <xsd:element name="release" type="releaseActivity" />
    <xsd:element name="repeat" type="repeatActivity" />
    <xsd:element name="spawn" type="spawnActivity" />
  </xsd:choice>
</xsd:group>

<!-- Base type for all simple activities -->
<xsd:complexType name="simpleActivity">
  <xsd:sequence>
    <xsd:group ref="metaData" />
    <xsd:element ref="completeBy"
                 minOccurs="0" maxOccurs="1" />
    <xsd:element ref="schedule"
                 minOccurs="0" maxOccurs="1" />
    <xsd:element ref="compensate"
                 minOccurs="0" maxOccurs="1" />
  </xsd:sequence>
  <xsd:attribute name="name" type="name"
                 use="optional" />
  <xsd:attribute name="idempotent" type="xsd:boolean"
                 use="optional" value="false" />
</xsd:complexType>

<!-- Group for selecting any one simple activity -->
<xsd:group name="anySimpleActivity">
  <xsd:choice>
    <xsd:element name="consume" type="consumeActivity" />
    <xsd:element name="empty" type="emptyActivity" />
    <xsd:element name="exception" type="exceptionActivity" />
    <xsd:element name="operation" type="operationActivity" />
```

```
    <xsd:element name="produce" type="produceActivity" />
  </xsd:choice>
</xsd:group>

<!-- Group for selecting any one activity -->
<xsd:group name="anyActivity">
  <xsd:choice>
    <xsd:group ref="anyComplexActivity" />
    <xsd:group ref="anyProcessActivity" />
    <xsd:group ref="anySimpleActivity" />
    <xsd:element name="activity" type="activityRef" />
  </xsd:choice>
</xsd:group>


<!-- Complex activities -->

<xsd:complexType name="allActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      All complex activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="complexActivity">
      <xsd:sequence>
        <xsd:group ref="anyActivity"
                   minOccurs="2" maxOccurs="unbounded" />
        <xsd:group ref="nestedProcess"
                   minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="choiceActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Choice complex activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="complexActivity">
      <xsd:sequence>
```

```
        <xsd:group ref="anyActivity"
                   minOccurs="2" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="switchActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Switch complex activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="complexActivity">
      <xsd:sequence>
        <xsd:element name="case"
                     minOccurs="1" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:group ref="anyActivity" />
            </xsd:sequence>
            <xsd:attribute name="condition" type="xpath"
                           use="optional" />
            <xsd:attribute name="ruleSet" type="name"
                           use="optional" />
            <xsd:attribute name="negative" type="xsd:boolean"
                           use="optional" value="false" />
            <xsd:attribute name="probability" type="probability"
                           use="optional" />
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="otherwise">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:group ref="anyActivity" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
  <xsd:attribute name="exclusive" type="xsd:boolean"
                 use="optional" value="true" />
```

```
</xsd:complexType>

<xsd:complexType name="foreachActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Foreach complex activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="complexActivity">
      <xsd:sequence>
        <xsd:group ref="anyActivity"
                   minOccurs="1" maxOccurs="unbounded" />
        <xsd:group ref="nestedProcess"
                   minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
  <xsd:attribute name="select" type="xpath"
                 use="required" />
</xsd:complexType>

<xsd:complexType name="sequenceActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Sequence complex activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="complexActivity">
      <xsd:sequence>
        <xsd:group ref="anyActivity"
                   minOccurs="1" maxOccurs="unbounded" />
        <xsd:group ref="nestedProcess"
                   minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>


<!-- Process activities -->

<xsd:complexType name="assignActivity">
```

```
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Assign process activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="processActivity">
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:group ref="assignment" />
      </xsd:sequence>
      <xsd:attribute name="target" type="name"
                     use="required" />
      <xsd:attribute name="select" type="xpath"
                     use="optional" />
      <xsd:attribute name="append" type="xsd:boolean"
                     use="optional" value="false" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="completeActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Complete process activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="processActivity" />
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="joinActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Join process activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="processActivity">
      <xsd:attribute name="select" type="xpath"
                     use="required" />
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

```
<xsd:complexType name="releaseActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Release process activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="processActivity">
      <xsd:attribute name="target" type="name"
                     use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="repeatActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Repeat process activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="processActivity">
      <xsd:attribute name="ref" type="name"
                     use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="spawnActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Spawn process activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="processActivity">
      <xsd:attribute name="ref" type="name"
                     use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

```
<!-- Simple activities -->

<xsd:complexType name="consumeActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Consume simple activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="simpleActivity">
      <xsd:sequence>
        <xsd:element ref="participant"
                     minOccurs="0" maxOccurs="1" />
        <xsd:element ref="input"
                     minOccurs="1" maxOccurs="1" />
        <xsd:group ref="extension" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="emptyActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Empty simple activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="simpleActivity">
      <xsd:sequence>
        <xsd:group ref="extension" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="exceptionActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Exception simple activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="simpleActivity">
```

```
      <xsd:sequence>
        <xsd:element ref="participant"
                     minOccurs="0" maxOccurs="1" />
        <xsd:element name="reason"
                     minOccurs="0" maxOccurs="1" >
           <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="xsd:anyType">
                <xsd:attribute name="select" type="xpath"
                               use="optional" />
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:group ref="extension" />
      </xsd:sequence>
      <xsd:attribute name="code" type="uri"
                     use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="operationActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Operation simple activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="simpleActivity">
      <xsd:sequence>
        <xsd:choice>
          <xsd:sequence>
            <xsd:element ref="participant"
                         minOccurs="1" maxOccurs="1" />
            <xsd:element ref="output"
                         minOccurs="1" maxOccurs="1" />
            <xsd:element ref="input"
                         minOccurs="1" maxOccurs="1" />
          </xsd:sequence>
          <xsd:sequence>
            <xsd:element ref="participant"
                         minOccurs="0" maxOccurs="1" />
            <xsd:element ref="input"
```

```
                        minOccurs="1" maxOccurs="1" />
            <xsd:element ref="output"
                        minOccurs="1" maxOccurs="1" />
            <xsd:element name="exception"
                        minOccurs="0" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:attribute name="code" type="uri"
                               use="required" />
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:choice>
        <xsd:group ref="extension" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="produceActivity">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Produce simple activity
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="simpleActivity">
      <xsd:sequence>
        <xsd:element ref="participant"
                    minOccurs="1" maxOccurs="1" />
        <xsd:element ref="output"
                    minOccurs="1" maxOccurs="1" />
        <xsd:group ref="extension" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>


<!-- Core elements -->

<!-- Activity element used to reference an activity -->
<xsd:complexType name="activityRef">
  <xsd:attribute name="ref" type="name"
                 use="required" />
```

```
</xsd:complexType>

<!-- Assign element used as part of assignment -->
<xsd:complexType name="assignAssignment">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:group ref="assignment" />
  </xsd:sequence>
  <xsd:attribute name="target" type="name"
                 use="optional" />
  <xsd:attribute name="select" type="xpath"
                 use="optional" />
</xsd:complexType>

<!-- Variable element used as part of assignment -->
<xsd:complexType name="variableAssignment">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:group ref="assignment" />
  </xsd:sequence>
  <xsd:attribute name="name" type="name"
                 use="required" />
  <xsd:attribute name="select" type="xpath"
                 use="optional" />
</xsd:complexType>

<!-- Foreach element used as part of assignment -->
<xsd:complexType name="foreachAssignment">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:group ref="assignment" />
  </xsd:sequence>
  <xsd:attribute name="name" type="name"
                 use="required" />
  <xsd:attribute name="select" type="xpath"
                 use="required" />
</xsd:complexType>

<!-- Group for selecting any assignment element -->
<xsd:group name="assignment">
  <xsd:choice>
    <xsd:element name="assign" type="assignAssignment" />
    <xsd:element name="foreach" type="foreachAssignment" />
    <xsd:element name="variable" type="variableAssignment" />
  </xsd:choice>
</xsd:group>
```

```
<!-- Compensate element used to define compensating activity -->
<xsd:element name="compensate">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="anyActivity"
                 minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!-- CompleteBy element used to define time constraint -->
<xsd:element name="completeBy">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="relativeTo"
                   minOccurs="0" maxOccurs="1">
        <xsd:complexType>
          <xsd:attribute name="ref" type="name"
                         use="required" />
          <xsd:attribute name="end" type="xsd:boolean"
                         use="optional" value="false" />
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="duration" type="xpath"
                   use="required" />
  </xsd:complexType>
</xsd:element>

<!-- Input element used to consume a message -->
<xsd:element name="input">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="rule"
                   minOccurs="0" maxOccurs="unbounded" />
      <xsd:group ref="assignment"
                 minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="message" type="name"
                   use="required" />
  </xsd:complexType>
</xsd:element>

<!-- OnException element used to define exception handling activity -->
```

```
<xsd:element name="onException">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="anyActivity"
                 minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="code" type="uri"
                   use="optional" />
  </xsd:complexType>
</xsd:element>

<!-- Output element used to produce a message -->
<xsd:element name="output">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="assignment"
                 minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attribute name="message" type="name"
                   use="required" />
  </xsd:complexType>
</xsd:element>

<!-- Participant element used in simple activities -->
<xsd:element name="participant">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="extension" />
    </xsd:sequence>
    <xsd:attribute name="select" type="xpath"
                   use="required" />
  </xsd:complexType>
</xsd:element>

<!-- Rule element used in input and ruleSet elements -->
<xsd:element name="rule">
  <xsd:complexType>
    <xsd:attribute name="condition" type="xpath"
                   use="optional" />
    <xsd:attribute name="ruleSet" type="name"
                   use="optional" />
    <xsd:attribute name="negative" type="xsd:boolean"
                   use="optional" default="false" />
  </xsd:complexType>
```

```
</xsd:element>

<!-- CompleteBy element used to schedule activity -->
<xsd:element name="schedule">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="relativeTo"
                   minOccurs="0" maxOccurs="1">
        <xsd:complexType>
          <xsd:attribute name="ref" type="name"
                         use="required" />
          <xsd:attribute name="end" type="xsd:boolean"
                         use="optional" value="false" />
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="duration" type="xpath"
                   use="optional" />
    <xsd:attribute name="instant" type="xpath"
                   use="optional" />
  </xsd:complexType>
</xsd:element>

<!-- Transaction element used to define transactional behavior -->
<xsd:element name="transaction">
  <xsd:complexType>
    <xsd:attribute name="type" type="transactionType"
                   use="required" />
    <xsd:attribute name="model" type="transactionModel"
                   use="optional" value="unreliable" />
    <xsd:attribute name="repeat" type="xpath"
                   use="optional" value="'1'" />
  </xsd:complexType>
</xsd:element>


<!-- Simple types -->

<xsd:simpleType name="uri">
  <xsd:restriction base="xsd:uriReference"/>
</xsd:simpleType>

<xsd:simpleType name="name">
  <xsd:restriction base="xsd:NCName"/>
```

```
</xsd:simpleType>

<xsd:simpleType name="xpath">
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="probability">
  <xsd:restriction base="xsd:float">
    <xsd:minExclusive value="0.0" />
    <xsd:maxExclusive value="1.0" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="messageType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="data"/>
    <xsd:enumeration value="request"/>
    <xsd:enumeration value="response"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="transactionType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="supported"/>
    <xsd:enumeration value="required"/>
    <xsd:enumeration value="nested"/>
    <xsd:enumeration value="new"/>
    <xsd:enumeration value="none"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="transactionModel">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="coordinated"/>
    <xsd:enumeration value="extended"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:group name="extension">
  <xsd:sequence>
     <xsd:any processContents="strict" namespace="##any"
              minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:group>
```

```
</xsd:schema>
```

# Appendix B: Glossary

### Abstract Process

The definition of a process void of any implementation details.

### Assignment

The mapping of data items to and from the process data.

### BPML Document

An XML document used to define a process or set of processes. A BPML document may be validated against the BPML XML Schema.

### BPML XML Schema

An XML Schema used to define the structure and types of BPML documents. The BPML XML Schema is defined in this document.

### Collaborative Business Process (CBP)

Defines the manner in which two business partners interact based on exchange of messages.

### Compensating Activity

An activity that offsets the result of a completed activity, used to perform backward-recovery.

### Enterprise Business Process (EBP)

A process that spans multiple enterprise applications, corporate departments and business partners.

### Exception

Any error or unexpected condition that occurs while executing the process.

### Participant

The participant of a process is any entity with which the process interacts by exchanging messages or performing operations.

### Process

A process involves interaction between participants and the execution of activities according to a defined set of rules in order to achieve a common goal.

### Process Data

A context that exists for each process instance and can be used to hold or reference information that is accumulated during the life of the process.

### Process Element

An XML element of a type defined by the BPML XML Schema and used in constructing BPML documents.

### Transaction

A unit of work treated in a coherent and reliable way independent of other transactions.

# Appendix C: References

## Normative

### XML

W3C (World Wide Web Consortium), Extensible Markup Language (XML) 1.0 (Second Edition), October 6 2000

http://www.w3.org/TR/2000/REC-xml-20001006

### XML Namespace

W3C (World Wide Web Consortium), *Namespaces in XML*, January 1999

http://www.w3.org/TR/1999/REC-xml-names-19990114

### XML Schema

W3C (World Wide Web Consortium), *XML Schema Part 1: Structures*, W3C Candidate Release, October 24 2000

http://www.w3.org/TR/2000/CR-xmlschema-1-20001024

W3C (World Wide Web Consortium), *XML Schema Part 2: Datatypes*, W3C Candidate Release, October 24 2000

http://www.w3.org/TR/2000/CR-xmlschema-2-20001024

### XPath

W3C (World Wide Web Consortium), *XML Path Language*, November 16 1999

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

### URI

IETF (Internet Engineering Task Force), *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*, eds. T. Berners-Lee, R. Fielding, L. Masinter, August 1998

http://www.ietf.org/rfc/rfc2396.txt

## Informative

### BRML

OASIS, Business Rules Markup Language (BRML), August 5 1999

http://www.oasis-open.org/cover/brml.html

### Dublin Core Metadata Element Set

Dublin Core Metadata Initiative, Dublin Core Metadata Element Set, Version 1.1: Reference Description

http://purl.org/DC/documents/rec-dces-19990702.htm

### ebXML

ebXML (Electronic Business XML Initiative), Messaging Services and Transport, Routing and Packaging

http://www.ebxml.org

### Functional Nets

Martin Odersky, *Functional Nets*, Colloque d'Informatique at EPFL, October 19 1999

http://lampwww.epfl.ch/slides/funnets/index.htm

### Join Calculus

Cédric Fournet and Luc Maranget, *The Join-Calculus language*, version 1.05, September 27 2000

http://join.inria.fr/

### NMF 601

NMF (Network Management Forum), Customer to Service Provider Trouble Administration Information Agreement , issue 1.0, March 1997

http://www.nmf.org

### OAMAS

OAG (Open Applications Group), *Open Applications Middleware API Specification*, version 1.0, August 15 1999

http://www.openapplications.org

### OMG Activity Service

OMG (Object Management Group), Additional Structuring Mechanism for the OTS specification, June 1999

http://www.omg.org

### OMG OTS

OMG (Object Management Group), *Transaction Service*, Version 1.1, November1997

http://www.omg.org

OMG UML

OMG (Object Management Group), *Unified Modeling Language Specification*, Version 1.3, June 1999

http://www.omg.org

Open Nested Transactions

Gerhard Weikum, Hans-J. Schek, Concepts and Applications of Multilevel Transactions and Open Nested Transactions,1992

http://citeseer.nj.nec.com/weikum92concepts.html

PetriNet

ISO JTC1/SC7, *High-level Petri Net Standard*, version 3.4, October 2 1997

http://saturne.info.uqam.ca/Labo_Recherche/Lrgl/sc7/

RosettaNet

RosettaNet Implementation Framework

http://www.rosettanet.org

SAGAS

Hector Garcia-Molina, Kenneth Salem, *Sagas*, SIGMOD Conference 1987: 249-259

http://www.acm.org/sigmod/dl/SIGMOD87/P249.PDF

SOAP

W3C (World Wide Web Consortium), *Simple Object Access Protocol (SOAP) 1.1*, W3C Note, May 8 2000

http://www.w3.org/TR/SOAP/

UDDI

UDDI.org, Universal Description, Discovery and Integration, Ariba, IBM and Microsoft

http://www.uddi.org

Workflow Reference Model

Workflow Management Coalition, *The Workflow Reference Model*, Issue 1.1, January 19, 1995

http://www.aiim.org/wfmc

WSDL

Microsoft Corporation, Web Services Description Language (WSDL) 1.0, September 25 2000

http://msdn.microsoft.com/xml/general/wsdl.asp

XAML

Transaction Authority Markup Language (XAML)

http://www.xaml.org

### XHTML

W3C (World Wide Web Consortium), *XHTML 1.0: The Extensible HyperText Markup Language*, W3C Recommendation 26 January 2000

http://www.w3.org/TR/xhtml1

### XML Query Algebra

W3C (World Wide Web Consortium), *The XML Query Algebra*, W3C Working Draft, December 4 2000

http://www.w3.org/TR/2000/WD-query-algebra-20001204

### XML Schema

W3C (World Wide Web Consortium), *XML Schema Part 0: Primer*, W3C Candidate Release, October 24 2000

http://www.w3.org/TR/2000/CR-xmlschema-0-20001024

# Appendix D: Document History

## August 15th 2000

First Working Draft submitted to the BPMI members for comment.

## September 21st 2000

Second Working Draft submitted to the BPMI members for comment. Major changes include:

- Change to definition of the meta data to include any element
- Change to state to include Transitions
- Definition of Mapping
- Definition of Errors

## November 30th 2000

Third Working Draft preliminary copy submitted to BPMI members for comments. Major changes include:

- Specification for an executable process
- Introduced model for defining, referencing and communicating participants
- Introduced model for assignment between process data and messages
- Added definition of dependent rules
- Modified model for activity sequencing to provide greater flexibility by explicitly modeling sequences, branching and parallel flows
- Allow process to declare the availability of sub-processes and reference them as participants
- Introduced distinction between short and long transactions and support for non-transactional processes
- Distinguished fault handling from normal flow logic and added compensating activities

- Support for nested processes

## March 8th 2001

- Redefined assignments to simplify syntax and add explicit elements for variables and for-each construct.
- Now using *exception* instead of *fault*.
- For modeling purposes, the *empty* activity is now a simple activity.
- The *choice* activity is now used for participant branching, process branching is supported by the new *switch* activity.
- Consolidated the attribute `completeBy@from` with `completeBy@duration`.
- Added *foreach* activity to enable iteration over a set of values.
- The attribute `join@ref` has been renamed `join@select` and is now an XPath expression that can select zero or more nested processes.
- Added root element `package`.
- The attribute `participant@ref` has been consolidated with the attribute `participant@select`, which replaces `participant@from`.
- The attribute `release@from` has been renamed `release@target` to comply with the *assign* activity.
- Separated rules from rule sets. The `rule` element is used by the input element, while the `ruleSet` element appears in the definition of a process or abstract.
- Added the `schedule` element to schedule the execution of an activity.
- The attribute `transaction@retry` has been renamed `transaction@repeat` and is now an XPath expression. To repeat a transaction up to three times (in case of failure), use `retry="'3'"`.
- Added informative reference to XML Query Algebra. The underlying model serves as the basis for BPML assignments.