Business Process Modeling Language

BPMI Proposed Recommendation
January 24, 2003

Authors:

Assaf Arkin, Intalio

Abstract

The Business Process Modeling Language (BPML) specification provides an abstract model for expressing business processes and supporting entities. BPML defines a formal model for expressing executable processes that address all aspects of enterprise business processes, including activities of varying complexity, transactions and their compensation, data management, concurrency, exception handling and operational semantics. BPML also provides a grammar in the form of an XML Schema for enabling the persistence and interchange of definitions across heterogeneous systems and modeling tools.

Status of this Document

This document is a Proposed Recommendation of the BPML specification.

Comments on this document should be sent to bpmi-dev@bpmi.org.

This is still a draft document and may be updated, replaced, or made obsolete by other documents at any time.

Acknowledgements

This specification was edited by:

Michael Dailey	mjpdailey@comcast.net

Contributions to the BPML specification were made by:

David Blondeau	Intalio
Ismael Ghalimi	Intalio
Wolfgang Jekeli	SAP
Stefano Pogliani	Sun
Matthew Pryor	Versata
Karsten Riemer	Sun
Howard Smith	CSC
Ivana Trickovic	SAP
Stephen A. White	SeeBeyond

Notice of BPMI.org Policies on Intellectual Property Rights & Copyright

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

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

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to BPMI.org, except as needed for the purpose of developing BPMI.org specifications, in which case the procedures for copyrights defined in the BPMI.org Intellectual Property Rights document must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by BPMI.org or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and BPMI.org DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

List of all Figures

List of all Tables

List of all Examples

1 Introduction

The BPML specification provides an abstract model and XML syntax for expressing executable business processes and supporting entities. BPML itself does not define any application semantics such as particular processes or application of processes in a specific domain. Instead, it defines an abstract model and grammar for expressing executable processes. This allows BPML to be used for a variety of purposes that include, but are not limited to, the definition of enterprise business processes, the definition of complex Web services, and the definition of multi-party collaborations.

1.1 Dependency on Other Specifications

The BPML specification depends on the following specifications: XML 1.0, XML-Namespaces, XML-Schema 1.0 and XPath 1.0. In addition, support for importing and referencing service definitions given in WSDL 1.1 is a normative part of the BPML specification.

1.2 Conventions

The section introduces the conventions used in this document. This includes notational conventions, notations for schema components and designated namespace definitions.

Notational Conventions

This specification incorporates the following notational conventions:

- The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.
- A **term** is a word or phrase that has a special meaning. When a term is defined, the term name is highlighted in **bold** typeface.
- A reference to another definition, section, or specification is highlighted with underlined typeface and provides a link to the relevant location in this specification.
- A reference to an element, attribute or BPML construct is highlighted with *italic* typeface.
- Non-normative examples are set of in boxes and accompanied by a brief explanation.
- XML and pseudo text is highlighted with `mono-spaced` typeface.

Notations for schema components

- The definition of each kind of schema component is given in XML-like grammar using the `mono-spaced` typeface. The definition of an element is shown with the element name enclosed in angle brackets.
- Notations for attributes are as follows:
  - Required attributes appear in **bold** typeface.
  - Where the attribute type has an enumerated type definition, the values are shown separated by vertical bars.

  - Where the attribute type is given by a simple type definition, the type definition name from either XML Schema or the BPML schema is used.
  - Where the attribute is optional and has a default value, it is shown following a colon.

- Support for **extension attributes** is shown by *{extension attribute}*. Where used in the grammar, it indicates support for any number of attributes defined in a namespace other than the BPML namespace.
- The allowed content of the schema component is shown using a simple grammar.
  - An element name is used for any content part that must be an element of that type.
  - A name enclosed in curly braces and appearing in italic typeface refers to a contents part of some other type. For example, *{any activity}* refers to any element that defines an activity.
- The cardinality of any content part is specified using the following operators:

Operator	Value
?	zero or one
*	zero or more
+	one or more

  If no operator is used, the content part must appear exactly once. Cardinality that cannot be expressed using any of these operators is shown using curly braces, with the minimum and maximum values separated by comma. For example, *{2,*}* denotes two or more.

- Groups of content parts are notated as follows:
  - A choice group consists of all consecutive content parts, separated by a vertical bar.
  - A sequence group consists of all consecutive content parts that are separated by a comma.
  - Content parts are grouped together using parentheses to form a new content part.
- Support for **extension elements** is shown by *{extension element}*. Where used in the grammar, the content part may be any element defined in a namespace other than the BPML namespace.
- Support for mixed content is shown by *{mixed}*. Where used in the grammar, the allowed content is a mix of character data and of elements defined in any namespace.

Use of namespaces

The following namespace prefixes are used throughout this document:

Prefix	Namespace URI	Definition
bpml	http://www.bpmi.org/2003/BPML/process	BPML namespace for BPML process definitions
inst	http://www.bpmi.org/2003/BPML/instance	BPML namespace for BPML instance definitions
func	http://www.bpmi.org/2003/BPML/function	BPML namespace for BPML function definitions.
wsdl	http://schemas.xmlsoap.org/wsdl	WSDL namespace for WSDL definitions
wsci	http://www.w3.org/2002/07/wsci10	WSCI namespace for WSCI definitions
xsd	http://www.w3.org/2001/XMLSchema	XML Schema namespace for XML Schema definitions and declarations
tns	(various)	The “this namespace” prefix is used as a convention in order to refer to the current document
(other)	(various)	All other namespace prefixes are samples only and represent some application-dependent namespace as per the example in which they are used.

Table 1 Namespace prefixes used in this document

2 Definitions

BPML constructs are the base parts that comprise the BPML abstract model. The BPML specification provides the abstract model and XML Syntax for these constructs.

BPML definitions are named constructs that can be referenced. A BPML process definition is by itself a construct and an assembly of multiple constructs.

A **BPML package** is a collection of BPML definitions and can import definitions made in other languages, such as XML Schema or WSDL.

A **BPML document** is the XML representation of a BPML package that is based on the syntax given in this specification.

BPML documents can be used for the purpose of exchanging BPML definitions between BPML processors. There is no requirement that a BPML definition must exist within a BPML document or that a BPML document be accessible from a know URL. A definition can exist in a manner that is independent of any XML representation, and can be accessible when referenced, given its fully qualified name and type.

2.1 The Package

The package construct is a composition of the following attributes:

Attribute	Description
namespace	The target namespace.
documentation	Documentation. (Optional)
features	Zero or more features.
imports	Zero or more imports.
processes	Zero or more process definitions.
properties	Zero or more property definitions.
schedules	Zero or more schedule definitions.

The *namespace* attribute provides the namespace associated with all definitions contained in this package. Definitions imported to the package may be associated with other namespaces.

The *features* attribute indicates to a BPML implementation that it will be able to process all definitions contained in this package element only if it supports all the named features.

The *imports* attribute provides the namespace names of namespaces that contain additional definitions that must be imported into the package. The imported definitions may be defined in languages other than BPML. A BPML implementation must be able to import XML Schema and WSDL definitions and documents.

We say that a definition exists in the package if the definition is contained in that package or is imported from another package or namespace.

The qualified name of any definition must be unique within the scope of all definitions of that type. This ensures that the definition can be unambiguously referenced by a combination of its qualified name and type. It is an error when two definitions that have the same type and name exist in the package.

The *package* element and top-level BPML definition elements are defined in the namespace *http://www.bpmi.org/2003/BPML/process*.

The syntax for the *package* element is given as:

```
<package
  targetNamespace = anyURI>
  Content: (documentation?, feature*, import*,
            (process | property | schedule)+)
</package>
```

The *targetNamespace* attribute provides the namespace name for all definitions contained in this *package* element.

Feature

The syntax for the *feature* element is given as:

```
<feature
  name = anyURI
  version = NMTOKEN/>
```

The *name* attribute provides the name of the feature. This should be a namespace that is associated with the feature definition.

The *version* attribute is used when multiple versions of the feature exist and all have the same name.

Import

The syntax for the *import* element is given as:

```
<import
  namespace = anyURI
  location = anyURI/>
```

The *namespace* attribute provides the namespace associated with all imported definitions. Only definitions defined in this namespace are imported.

The *location* attribute provides the location of a document containing these definitions. If this attribute is missing, a BPML implementation will use other mechanisms to locate the imported definitions.

Importing definitions from another document is not recursive. Only definitions may in that document or included in that document through an inclusion mechanism are imported. Definitions imported to that document are not imported.

2.2 Conformance

A **BPML implementation** performs one or more duties based on the semantics conveyed by BPML definitions. A BPML implementation must understand the semantics of BPML definitions as set forth in this specification.

A **conformant implementation** is any BPML implementation that can process BPML documents and perform one or more duties based on the semantics conveyed in BPML definitions, as set forth in this specification.

At the minimum, a fully conformant implementation of version 1.0 of the BPML specification must support for the features listed below. There is no need to specify these features in a BPML package.

Specification	Feature
BPML 1.0	http://www.bpmi.org/2003/BPML
WSDL 1.1	http://schemas.xmlsoap.org/wsdl
XPath 1.0	http://www.w3.org/TR/xpath
XML Schema 1.0	http://www.w3.org/2001/XMLSchema

Table 2 Features supported by BPML 1.0

A conformant implementation is not required to process any extension elements or attributes, or any BPML document that contains them. Extension elements and attributes are specified in a namespace that is other than the BPML namespace and may appear only where allowed.

2.3 Use of Documentation

BPML definitions use the *documentation* attribute to provide additional information that cannot be expressed using other attributes. The value of the *documentation* attribute can include human readable descriptions as well as meta-data for application processing.

The information provided by the *documentation* attribute does not affect the semantics of the definition. A BPML implementation must ensure that two definitions that vary only in the contents of the documentation attribute are semantically equivalent.

The syntax for the *documentation* element is given as:

```
<documentation>
  Content: {mixed}
</documentation>
```

The *documentation* element can contain mixed content, including elements in any namespace. It is not necessary for the contents to be validated against a particular schema.

Authors of BPML documents and tool providers may want to standardize on the following:

- Use of RDF for semantic meta-data that may be human readable or that is intended for application processing
- Use of XHTML for expressing information in human readable form
- Use of Dublin Core Meta Data

3 Activities

An **activity** is a component that performs a specific function. Complex activities are composed of other activities and direct the execution of those activities. A process is such a composition and may itself be an activity within a larger process. The semantics of an activity definition apply to a process definition with a few noted exceptions.

3.1 Activity Types

An activity definition specifies the manner in which a given activity will execute. The behavior is defined by specifying the values of the activity's attributes. An activity type definition specifies the attributes that are used in the definition of an activity of that type and how the values of these attributes affect the execution of that activity.

The BPML specification defines 17 activity types and three process types. All activity types are derived from a common base type. The base type defines the following attributes:

Attribute	Description
name	The activity name. (Optional)
documentation	Documentation. (Optional)
atomic	Atomic attribute. (Optional)
other	Other attributes defined for the specific activity type.

The *name* attribute provides a name that can be used to reference the activity definition or activity instance. Two activity definitions are distinct even if they have the same name. It is not an error if, within a given context, the same name references both activity definitions.

With the exception of process definitions, all activity definitions have an ordinal position within an activity list. If the *name* attribute is unspecified, the activity name is its ordinal position. For example, "1" for the first activity in the activity list, "2" for the second activity, and so forth. The *name* attribute is optional for all but process definitions.

The *atomic* attribute specifies whether the activity is atomic or not. Most simple activities, with the exception of the *action*, *call* and *compensate* activities, are atomic by default. Complex activities are not atomic by default. These activities are atomic only if the *atomic* attribute is specified with the value *true*. This attribute is not supported for other activity types. Atomic activities are covered in the section 10.1 Atomic Activity.

An activity type will define additional attributes that are specific to that type. For example, the *operation* attribute of the *action* activity, or the *condition* attribute of the *while* activity. Other specifications may introduce additional activity types that are derived from this base type.

The syntax for the base type *bpml:activity* is given as:

```
<{activity type}
  name = NCName
  {other attributes}>
  Content: (documentation?, {other element}*)
</{activity type}>
```

Each activity type defines a syntax that specifies additional XML attributes and XML elements that represent values of the abstract model attributes.

Other specifications may introduce additional activity types. The XML elements for these activity types are derived from the type *bpml:activity* and use the substitution group *bpml:otherActivity*. They must be defined in a namespace other than the BPML namespace.

The BPML specification defines the following activity types:

Simple	**Description**
action	Performs or invokes an operation involving the exchange of input and output messages.
assign	Assigns a new value to a property.
call	Instantiates a process and waits for it to complete.
compensate	Invokes compensation for the named processes.
delay	Expresses the passage of time.
empty	Does nothing.
fault	Throws a fault in the current context.
raise	Raises a signal.
spawn	Instantiates a process without waiting for it to complete.
synch	Synchronizes on a signal.

Complex	**Description**
all	Executes activities in parallel.
choice	Executes activities from one of multiple sets, selected in response to an event.
foreach	Executes activities once for each item in an item list.
sequence	Executes activities in sequential order.
switch	Executes activities from one of multiple sets, selected based on the truth value of a condition.
until	Executes activities once or more based on the truth value of a condition.
while	Executes activities zero or more times based on the truth value of a condition.

Table 3 Activity types defined by the BPML specification

3.2 The Activity Context

Activities that execute in the same context use that context to exchange information through properties that are defined in that context. For example, an activity that receives an input message sets the value of a property from the contents of the input message. A subsequent activity uses the value of that property to construct and send an output message.

The context defines common behavior for all activities executing in that context, such as handling of exceptional conditions and faults, providing atomic semantics, defining a time constraint, and so forth.

The context in which an activity executes is referred to its *current context*. Activities and contexts are composed hierarchically. The current context of an activity may be the child context of some other context, and the parent of multiple child contexts. Contexts are covered in section 5 Contexts.

The term *downstream activity* refers to an activity that executes following the current activity. The downstream activity may depend on the value of properties set by the current activity, a signal raised by the current activity, or the instantiation of another activity from the current activity.

Activities that execute in the same context are grouped together into an *activity set*. The activity set is a composition of one or more activity definitions and the definition of the context in which these activities execute – their current context.

The activity set contains an *activity list*, which is an ordered list of activity definitions. Generally, activities from the activity list are executed in sequential order. This means that an activity must complete before the next activity in the list is executed. The BPML specification defines one activity that causes activities from the activity list to execute in parallel.

The activity set can define activities that can be executed multiple times in parallel with other activities defined in the activity set. These activities are modeled as process definitions and are contained in the activity set's context definition. We refer to them as *nested processes*. Nested processes are covered in section 4.2 Nested Processes.

The activity set can define activities that execute in response to exceptional condition and interrupt the execution of all other activities defined in the activity set. These activities are defined in a similar manner to nested processes and are referred to as *exception processes*. Exception processes are covered in section 9.1 Exception Processes.

The activity set construct is a composition of the following attributes:

Attribute	Description
context	A context definition. (Optional)
activities	One or more activity definitions. (Ordered)

The syntax for the activity set is given as:

```
Content: (context?, {any activity}+)
```

The *context* element is absent if the context definition contains no local definitions (an *empty context*).

The activity list must contain at least one activity definition. With the exception of process definitions, any activity type can be used in the activity list. This includes activity types defined in other specification. Nested process definitions appear inside the *context* element.

The occurrence of the *bpml:activitySet* model group in the content of an XML element indicates that it contains an activity set.

3.3 Simple and Complex Activities

A **complex activity** is an activity that contains one or more child activities. It establishes a context for their execution and directs their execution.

Complex activities provide a form of hierarchical composition. It may be as simple as repetitive execution of a single activity, or a means to establish a nested context for the execution of multiple activities.

The BPML specification supports other forms of composition, including cyclic graphs and recursive composition. Complex activities are used when hierarchical composition is required, in particular to establish a new context for the execution of child activities.

A **simple activity** is any activity that is not a complex activity. A simple activity may lead to the execution of multiple activities, specifically the *action*, *call*, *compensate* and *spawn* activities. However, a simple activity does not by itself define the context for the execution of other activities.

A complex activity that contains multiple activity sets must select which one to use. The *choice* activity waits for an event to be triggered and selects the activity set associated with that event handler. The *switch* activity evaluates conditions and selects the activity set associated with a condition that evaluates to true. All other complex activities defined in the BPML specification contain a single activity set.

A complex activity determines the number of times to execute activities from the activity set. The *until* activity repeats executing activities until a condition evaluates to true. The *while* activity executes activities repeatedly while the condition evaluates to true. The *foreach* activity repeats executing activities, once for each item in the item list. All other complex activities defined in the BPML specification execute activities from the activity set exactly once.

A complex activity determines the order in which activities are executed. The *sequence* activity executes all activities from the activity set's list in sequential order. The *all* activity executes all activities from the activity set's list in parallel. All other complex activities defined in the BPML specification execute activities in sequential order.

The complex activity completes after it has completed executing all activities from the activity set. This includes all activities that are defined in the activity list and all processes instantiated from a definition made in the activity set's context. Nested processes and exception processes are considered activities of the activity set.

Simple activities abort and throw a fault if they cannot complete due to an unexpected error. Complex activities abort and throw a fault if one of their activities throws a fault from which they cannot recover.

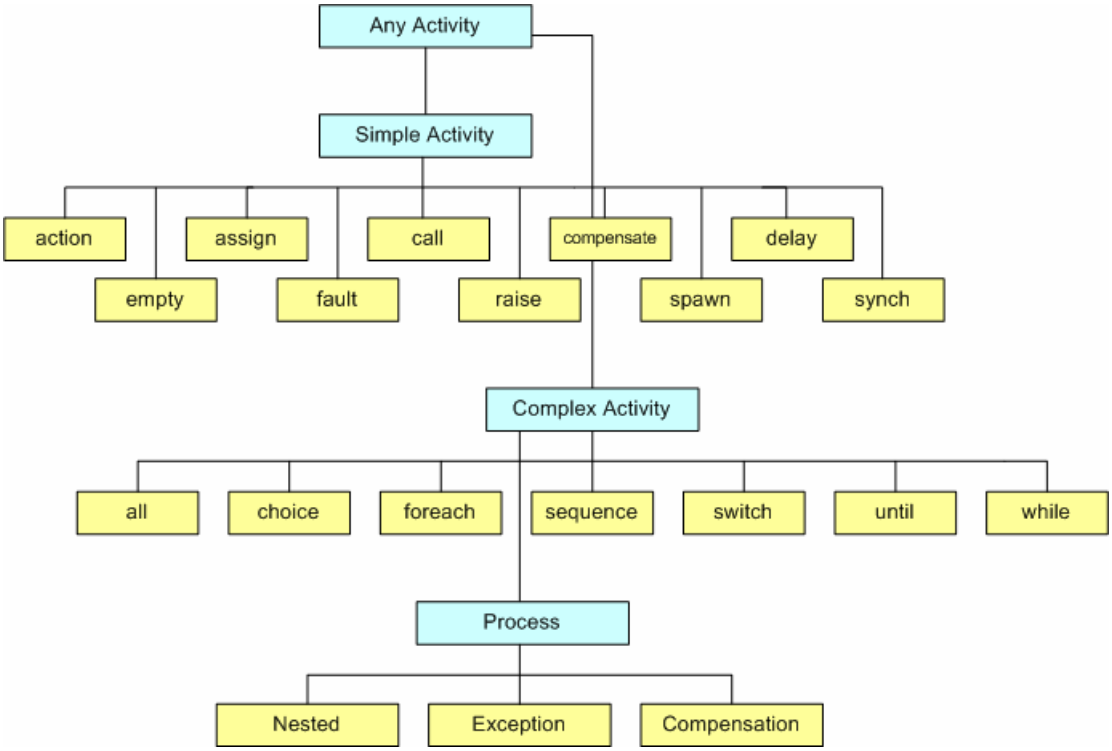


Figure 1 Activity types defined in the BPML specification

3.4 Activity Instance

The activity definition specifies how an activity instance will execute. Multiple instances of that activity definition may execute in parallel. Although these instances are created from the same activity definition, they are distinct and not directly related to each other.

An activity instance transitions through the following states:

- **ready** – The activity instance is not performing any work
- **active** – The activity instance is performing work specific to its definition
- **completing** – The activity instance has performed all work specific to its definition, and is now performing all work required to complete
- **completed** – The activity instance has performed all work required in order to complete
- **aborting** – The activity instance cannot complete and is now performing all work required to abort
- **aborted** – The activity instance has performed all work required in order to abort

An activity instance always begins in the *ready* state. In order to perform any work, the activity instance transitions to the *active* state. The activity instance remains in this state as long as required. It may be the duration required to send and receive a message or to execute all activities from the activity set.

Once the activity instance is ready to complete, it transitions first to the *completing* state and then to the *completed* state. The activity is completed when the activity instance transitions to the terminal *completed* state. The activity instance uses the *completing* state to perform additional work that is required to complete, such as persisting data, waiting for the transaction outcome and so forth.

The activity instance transitions first to the *aborting* state and then to the *aborted* state if it cannot complete due to a fault. The activity instance may transition to this state from the *active* or *completing* states. The activity is aborted when the activity instance transitions to the terminal *aborted* state. The activity instance uses the *aborting* state to perform additional work that is required to abort, such as aborting a transaction, reverting data changes and so forth.

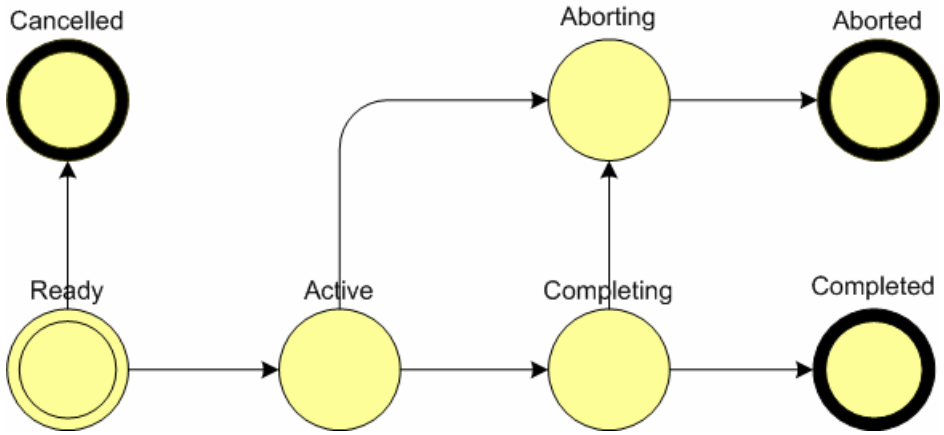


Figure 2 Transition diagram for activity instance states

The *ready* state is a special state. An activity instance that is in the *ready* state is not performing any work and can be terminated without repercussion. When the activity instance terminates it does not transition to the *aborted* state. Instead it is discarded. We then say that the activity instance has been canceled.

The *action* activity is in the *ready* state while it waits for an input message to be received. The *delay* activity is in the ready state while waiting for the specified time instant. Likewise, a *sequence* activity that starts with an *action* activity will stay in the *ready* state until the *action* activity transitions to a different state.

Since an activity instance that is in the *ready* state can cease to exist, a BPML implementation should not provide access to any instance that is in this state. For example, a BPML implementation that provides monitoring for executing process instances should not show activity and process instances that are in the *ready* state.

The state transitions of a complex activity affect the execution of activities from its activity set, and are affected by their state transitions. The complex activity instance must be in the *active* state before it can execute any activities from its activity set.

While in the *active* state, nested processes and exception processes defined in the activity set's context will instantiate in response to an input message or a raised signal. While in this state, schedules defined in the context will fire events. Nested processes and exception processes do not respond to input messages and raised signals while the parent activity is in any other state.

The complex activity instance transitions to the *completing* state after all executing activity instances, including any instantiated nested processes, have completed. If a fault is thrown in the context, the complex activity will attempt to execute a fault handler. Before doing so it will terminate all child activities that are currently executing or wait for them to complete.

If it is able to execute a fault handler successfully, it will then transition to the *completing* state. If no fault handler is defined for that fault code, or if a fault handler aborts, the complex activity will transition to the *aborting* state.

4 Processes

A **process** is a type of complex activity that defines its own context for execution. Like other complex activity types, it is a composition of activities and it directs the execution of these activities. A process can also serve as an activity within a larger composition, either by defining it as part of a parent process or by invoking it from another process. Processes are often defined as reusable units of work.

A process that is defined independently of other processes is called a *top-level process*, since its definition is found at the package level. A process that is defined to execute within a specific context is called a *nested process*, since its definition is part of that context's definition.

An *exception process* is defined as part of a parent process to handle exceptional conditions that may interrupt activities executing in that process. A *compensation process* provides the compensation logic for its parent process. Exception processes and compensation processes are specific type of process definitions and are covered in section 9.1 Exception and section 9.4 Compensation respectively.

A process can be instantiated from the *call*, *compensate,* and *spawn* activities and from a schedule. Alternatively, it can define an instantiation event that responds to an input message, or instantiation event that responds to a raised signal.

A BPML implementation that detects a process definition that violates one of the constraints defined in the BPML specification should flag the process definition as erroneous. A process definition is also erroneous if it references an erroneous process definition from the *call* and *spawn* activities. A BPML implementation must not create instances from an erroneous process definition.

4.1 Instantiation

A process can be instantiated in one of three ways: in response to an input message, in response to a raised signal, or invoked from an activity or schedule. A process definition is allowed to have one means of instantiation. Depending on the process definition, we say that the process has instantiation type *activity* or *event*.

Activity

The *call*, *compensate,* and *spawn* activities are used to instantiate a process definition that has instantiation type *activity*. The instantiating and instantiated processes must execute in the same context or a related context. The definition of the instantiating process is dependent on the definition of the instantiated process. The two processes are tightly coupled.

This form of process definition is commonly used when the process is an activity of a larger composition or a reusable activity in multiple compositions. Such processes are often defined as nested processes or as sub-processes of other processes defined in the same package.

During instantiation, the process can receive input values that are mapped to properties defined in its context. Upon completion, the process can map properties defined in its context to output values. The process definition uses parameters to specify the name and type of input and output values.

Event

A process that is instantiated in response to an input message can be deployed as an independent service and can be instantiated from processes deployed in other systems and networks. Such processes are loosely coupled and can be deployed and executed in heterogeneous environments. These processes can evolve independently from each other.

A process can be instantiated in response to signals that are raised by other activities executing in the same context. This form of instantiation is only possible for processes defined as part of a larger process, such as nested processes and exception processes.

The process definition uses the *event* attribute to indicate the instantiation event (or events). This attribute names one or more event-triggered activities and indicates whether or not these activities are mutually exclusive. The process is instantiated each time an event occurs and the event-triggered activity/activities execute. The event-triggered activities must be the first activities in the process definition's activity list.

Event-triggered activities can be either *action* activities or *synch* activities, or a combination of both. Event-triggered *action* activities must respond to an input message that is received unsolicited. When using WSDL 1.1, they must perform the *one-way* or *request-response* operations or invoke the *notification* or *solicit-response* operations.

If the process definition has one such event-triggered activity, the process is instantiated each time the activity receives an input message or synchronizes on a raised signal.

If the process definition has multiple event-triggered activities that are mutually exclusive, the process is instantiated each time one of these activities receives an input message or synchronizes on a raised signal. All other event-triggered activities of that process instance are canceled.

If the process definition has multiple event-triggered activities that are not exclusive, the process is instantiated each time all of these activities are able to receive an input message or synchronize on a raised signal at the same time.

The following table illustrates some forms of process instantiation by showing the relation between the event-triggered activities, process inputs and instantiating activities.

Instantiation	Definition	Inputs	Instantiating activity
activity	*no event*	parameters	*call, compensate, spawn, schedule*

event (message)	`event activity=a`
	`action name=a`
	`  receive messageA`
		messageA	Send messageA

event (messages)	`event activity=a b`
	`  exclusive=true`
	`action name=a`
	`  receive messageA`
	`action name=b`
	`  receive messageB`
		messageA or messageB	Send messageA or messageB

Instantiation	Definition	Inputs	Instantiating activity

event (messages)	`event activity=a b`
	
	`exclusive=false`
	`action name=a`
	`  receive messageA`
	`action name=b`
	`  receive messageB`
		messageA and messageB	Send messageA and messageB

event (signal)	`event activity=a`
	`synch name=a`
	`  signalA condA`
		signalA	Raise signalA

event (signals)	`event activity=a b`
	`  exclusive=true`
	`synch name=a`
	`  signalA condA`
	`synch name=b`
	`  signalB condB`
		signalA or signalB	Raise signalA or signalB

Table 4 Process instantiation examples

The event-triggered activity can depend on the values of properties defined in the process context in order to identify the input message or raised signal. In order to execute these activities, a BPML implementation can create a process instance in the *ready* state. It can cancel and discard the process instance if it has determined that the process will not be instantiated (that is, transition to the *active* state). If the BPML implementation provides monitoring for executing process instances, it should not show process instances when they are in the *ready* state.

Process definitions should use messages for instantiation, unless they are defined to interact with other activities executing in the same context or are instantiated only by processes defined in the same package. Using message exchange allows processes to be defined, deployed, and managed independently. It also provides greater flexibility and improved change management.

4.2 Nested Processes

A **nested process** is a process definition that is localized to a given context. Nested processes are used as activities in the composition of larger processes, to localize the process definition to the context in which it is used, and to constrain the instantiation and availability of the process to a particular context.

A nested process is instantiated in the context in which it is defined. The nested process can access properties defined in that context, instantiate other processes defined in that context, use signals to interact with other activities defined in that context, and utilize the exception processes and fault handler defined by the context.

A nested process is considered an activity in the activity set. Its lifetime is demarcated by the parent activity or process that contains that activity set. The parent activity does not complete until it completes executing all activity instances from the activity set. That includes any nested process instantiated in that context. The parent activity terminates all nested process instances when it aborts. The parent activity aborts if any nested process instance aborts.

A nested process with instantiation type *event* constrains the instantiation of that process to a particular context. The nested process cannot be instantiated until the context instance exists or until after the parent activity/process completes or aborts. Since multiple context instances may exist, the input message must be correlated to the proper context instance.

This form of instantiation is commonly used to implement an operation that can be performed multiple times in a particular state, but only in that particular state of the parent process. The state is demarcated by the parent activity of the nested process definition.

A **top-level process** is a process definition that is not localized to any particular context. A top-level process is defined at the package level and does not share its context with any other top-level process. A top-level process can be instantiated at any time.

If the top-level process has instantiation type *activity*, then *call* and *spawn* activities from other process definitions can reference it. If the process has instantiation type *event*, the process is instantiated in response to an input message that may be received at any time, or a signal that can be raised in that context. A top-level process cannot be instantiated in response to a signal.

It is an error for multiple top-level process definitions to have an event-triggered activity that responds to the same input message. They may responds to an input message of the same type only if they distinguish one from the other by performing different operations or by using correlation.

An exception process is a special type of nested process. When an exception process is instantiated, it causes the parent activity to terminate all child activities that are currently executing, execute the exception process and complete. Whereas a nested process can be instantiated multiple times in a context instance, only one exception process can be instantiated in a given context instance. Exception processes are covered in section 9.1 Exception.

A compensation process is a special type of nested process. A compensation process is instantiated after the completion of the parent process instance in order to compensate for any activities it performed. A compensation process can be instantiated only if the parent process instance has completed, and can be completed only once. Compensation processes are covered in section 9.4 Compensation.

4.3 Process Definition

A process definition is a composition of the following attributes:

Attribute	Description
name	The process name.
documentation	Documentation. (Optional)
correlation	One or more correlation references. (Optional)
persistent	The persistence attribute. (Optional)
event	The instantiation event. (Optional)
parameters	Zero or more input parameters. (Optional)
activity set	An activity set.
compensation	The compensation process. (Optional)

The *name* attribute provides a name that can be used to reference the process definition. The *name* attribute must be specified. A context definition must not contain two process definitions with the same name. It is an error when two process definitions with the same name exist in the package.

The *correlation* attribute provides the name of one or more correlation definitions. A correlation definition provides the name of one or more properties that identify the process instance.

The *persistent* attribute specifies whether or not a process instance created from this process definition is persistent. The process instance is persistent if this attribute is specified with the value *true* and not persistent if this attribute is specified with the value *false*. If the *persistent* attribute is unspecified, the process instance is persistent if the parent process is persistent or if instantiated from a persistent process. Persistent processes are covered in section 10.2 Persistent Processes.

The *event* attribute provides the instantiation event. It names one or more event-triggered activities and indicates whether or not these activities are mutually exclusive. The event-triggered activities must be the first activities in the process definition's activity list. They must all be *action* activities that respond to an input message that is received unsolicited or they must all be *synch* activities.

The *parameters* attribute provides the names of all input and output parameters. It may be specified only if the process has instantiation type *activity*. If this attribute is unspecified, the process has no input or output parameters.

The *activity set* attribute provides the activity set that is executed by the process. After instantiation, the process instance executes activities from the activity set list in sequential order, excluding canceled event-triggered activities. The context defined by this activity set is also referred to as the *process context*.

The *compensation* attribute provides the compensation process. If this attribute is specified, the *persistent* attribute must not be specified with the value *false*. Compensation processes are covered in section 9.4 Compensation.

The syntax for a process definition is given as:

```
<process
  name = NCName
  correlation  = list of QName
  persistent = boolean : false>
  Content: (documentation?, (event | parameters?),
            context?, {any activity}+, compensation?)
</process>

<event
  activity = list of NCName
  exclusive = boolean : false/>
```

The fully qualified process name is constructed by combining the *name* attribute with the *targetNamespace* attribute of the package.

If the *event* element is absent, the process has instantiation type *activity*.

4.4 Parameters

A parameter construct consists of the following attributes:

Attribute	Description
name	The parameter name.
direction	Input, output.
required	True if required, false if optional. (Input only)
code	The fault code. (Output only, optional)

The *name* attribute provides the name of the parameter. It must have the same name as a property defined in the process context. The property definition provides the parameter type.

The *direction* attribute specifies whether the parameter is an input parameter or an output parameter.

- **input** – When the process is instantiated, an input value for that parameter is assigned to the named property.
- **output** – When the process completes or aborts, the value of the named property instance is used as the output value for that parameter.

The *required* attribute indicates whether an input parameter is required or optional. The attribute must be specified for an input parameter and must not be specified for an output parameter. If the value of the *required* attribute is *true*, an input value for that parameter must be provided when instantiating the process.

The *code* attribute provides the fault code. This attribute may be specified for an output parameter and must not be specified for an input parameter. If this attribute is unspecified, an output value for that parameter is provided only if the process completes. If this attribute is specified, an output value for that parameter is provided only if the process aborts with that fault code.

A process can define an input parameter with the same name as an output parameter, but must not define two input parameters with the same name. A process can define two output parameters with the same name if they use different values for the *code* attribute or if one parameter specifies the *code* attribute and the other does not.

The syntax for a process parameter definition is given as:

```
<parameters>
  Content: (input*, output*)
</parameters>

<input
  name = QName
  required = boolean : true/>

<output
  name = QName
  code = QName/>
```

4.5 Correlation

Correlation serves as a means to identify a specific process instance from a collection of processes instantiated from the same definition. It also identifies related process instances in a collection of processes instantiated from different definitions.

The process identifier is a unique identifier that exists in every process instance so as to distinguish it from any other process instance. The process identifier is accessed from the implicit property *inst:identifier*. The process identifier is universally unique across space and time. BPML implementations are encouraged to generate process identifiers that conform to the UUID scheme.

Correlation allows multiple process instances to use the same identity value and allow any property to be used. The identity value can serve as a thread across multiple processes instantiated from different definitions or relate the process instance to a particular entity. Correlations are generally used to associate input messages to the process instance (see the action activity) and to allow queries and analysis.

The correlation definition names one or more properties that together define the range of correlation values. Two correlation values are equivalent if, for all properties named in the correlation definition, the value of a property in one correlation value is equal to the value of a property with the same name in the other correlation value.

The process context must define all properties named in all referenced correlation definitions. The process correlation must reference fixed properties and the property values must be established when the process is instantiated. For example, by receiving an input message or from inputs parameters.

The correlation definition exists at the package level. A process definition can reference one or more correlation definitions using the *correlation* attribute. In addition, the *action* activity references specific correlations that apply to its operation. It uses the correlation to match an input message to an existing context instance that has properties with the same values as in the input message.

A nested process has all the correlations of the parent process, but can define additional correlations. Exception processes and compensation processes have the same correlations as the parent process.

A BPML process definition can import correlation definitions from a WSDL document. The correlations are defined using the WSCI *correlation* and *selector* elements.

5 Contexts

Contexts define an environment for the execution of related activities. Activities that execute within the same context use it to exchange information and coordinate their execution. An activity can access and modify the value of properties defined within that context, instantiate a nested process, raise and synchronize signals, and so forth.

The context definition specifies common behavior for all activities executing within that context. It defines how exceptional conditions and faults are handled, provides atomic semantics for accessing properties and exchanging messages, and so forth.

A context definition contains *local definitions* for entities that are accessible only from that context. These local definitions are not accessible outside the context. Activities that are defined for execution within that context can reference the local definitions and instantiate new instances of these definitions. Local definitions include properties, nested processes and signals, among others.

The hierarchical composition of activities creates a relationship between contexts. When a context definition encapsulates another context definition, we refer to the encapsulating definition as the *parent context* and to the encapsulated definition as the *child context*. The child context shares all the definitions of the parent context and, recursively, any of its parent contexts. It can also add local definitions of its own. The child context does not share any of its local definitions with the parent context and, by definition, with any sibling context.

The context in which an activity is defined is called its **current context**. The current context is a composition of the context that contains the activity definition and all parent contexts. When we say that an activity executes in a specific context, that statement is valid whether we refer to the specific context that contains the activity definition or to any of its parent contexts.

The hierarchical composition of contexts allows complex activities to establish a secure environment for the execution of activities. A local property definition allows these activities to access and modify the property value without affecting or being affected by activities executing in other contexts. These activities are still able to access and modify the value of any property defined within a parent context.

A process definition includes one context definition that is shared by all activities that execute as part of that process. We refer to it as the **process context**. Nested processes and exception processes are defined within the context of a parent process. That context serves as the parent for the nested and exception process context. This allows the nested process to access properties and to interact with other activities that are defined within the parent process. A compensation process is part of the parent process context.

The process context of a top-level process definition is called the *root context* since it is not a child of any other parent context.

A context instance provides an environment for the execution of activity instances that is distinct from any other instance of the same context. An executing activity must be able to reference properties that are not shared with any instance of the same activity that executes in a different context instance.

A context instance is instantiated from a context definition. Activities are created and executed within an instance of the context in which they are defined. A child context instance is always created within an instance of the parent context. Through its current context, an activity can access properties and other instances that were created in any parent context instance.

Context instances that are created from the same context definition are distinct and do not share any instances that were created from local definitions. Thus, a property instance or nested process instance that is created in one context instance is distinct from one that is created in a different context instance. Specifically, two top-level process instances can not share any property, activity or nested process instance.

5.1 Local Definitions

A context definition can include any number of local definitions, any of which can be instantiated within an instance of that context. The BPML specification defines six types of local definitions:

- **Exception** – Defining an exception process as local to a context ensures that the process will respond only to events from an instance of that context and that any such event would interrupt all activities executing in that context instance. Exception processes are covered in section 9.1 Exception.
- **Process** – Defining a process as local to a context ensures that the process can be instantiated only within an instance of that context. Such a process is called a *nested process*. Nested processes are covered in section 4.2 Nested Processes.

  Nested processes are used to restrict the availability of a process to a particular context. They also allow the process to access properties defined within that context and interact with other activities executing within the same context. A local process definition will hide another process definition with the same name that exists in a parent context or is defined at the package level.
- **Property** – Defining a property as local to a context ensures that the value of that property instance can only be accessed and modified from activities executing in an instance of that context. Properties are covered in section 6 Properties.

  Local property definitions are used to restrict access to a property. This access is limited to activities that execute within that context instance. Local property definitions allow different contexts to refer to different properties that have the same name.
- **Schedule** – Defining a schedule as local to a context ensures that the schedule will fire time events while activities are executing within an instance of that context. Schedules are covered in section 8 Schedules.

  Local schedule definitions are used to invoke processes while performing other work in that context. A local schedule definition can also place a time constraint on the execution of activities in that context by throwing a fault.
- **Signal** – Defining a signal as local to a context ensures that the signal can be raised and synchronized only by activities executing within that context. Signals are covered in section 7 Signals.

  A signal is used to synchronize between activities executing within the same context and does not cross the boundary of that context.

5.2 Activities and Processes

A complex activity instantiates a context when it attempts to execute activities from its activity set. Most complex activities have a single activity set and instantiate the context immediately. The *choice* activity waits for an event-triggered activity to complete before instantiating a new context instance and executing the associated activity set.

While the activity instance is in the *active* state, it allows the instantiation of nested processes and exception processes in response to an input message or to a raised signal. Schedules instantiated in the context instance are allowed to fire time events. The activity instance can discard the context instance when it completes and transitions to the *completed* state or when it aborts and transitions to the *aborted* state.

A complex activity must not continue executing activities from its activity set if one of those activities aborts. The complex activity can recover from the situation by responding to the fault through one of its fault handlers. These fault handlers are defined as part of the context definition and respond to faults generated by activities executing within that context.

When a fault occurs while executing an activity from the activity set, we say that a fault is *thrown* in the context. The related fault code is assigned to the *inst:fault* property in the context instance. That property is implicitly defined for every context definition.

When the fault is thrown, all activity instances executing in that context instance are terminated if they cannot complete. If a fault handler is associated with the applicable fault code, activities specified by that fault handler executed next. Faults and fault handlers are covered in section 9.2 Faults.

An exception process is instantiated in response to an input message or raised signal. When an exception process is instantiated, all other activity instances executing within that context instance are terminated if they cannot complete. A context instance can instantiate only one exception process. Exception processes are covered in section 9.1 Exception.

The process context is instantiated with the process instance. The process context instance is retained after completion of the process instance to allow instantiation of the compensation process. The process context can be discarded if no compensation process is defined, if the process instance aborts, after completion of the compensation process, or when it is no longer possible to instantiate the compensation process. Compensation processes are covered in section 9.4 Compensation.

The process context instance can be retained for an indefinite period of time. A BPML implementation may provide mechanisms for performing queries on historical data that require access to the property values of completed process instances. The BPQL specification covers the means by which such mechanisms are defined.

5.3 Context Definition

A context definition consists of the following attributes:

Attribute	Description
exception	Zero or more exception processes.
processes	Zero or more process definitions.
properties	Zero or more property definitions.
schedule	Zero or more schedule definitions.
signals	Zero or more signal definitions.
faults	Zero or more fault handlers.

All local definitions have a name by which they can be referenced. A context definition must not contain two local definitions of the same type that have the same name.

The syntax for a *context* definition is given as:

```
<context>
  Content: ((exception | process | property |
             schedule | signal)*, faults?)
</context>
```

The syntax of the *faults* element is given in section 9.2 Faults.

6 Properties

A **property definition** declares a property name, associates a type to that name, and provides an optional initial value. A **property instance** holds the property value. The property name is used to reference the property instance and the property type defines the range of values that it can hold.

A property instance can exist only within a context instance. The context definition contains property definitions for all property instances that are created within an instance of that context. A context instance cannot have two property instances with the same name. Two property instances with the same name are distinct if they belong to different context instances.

A property instance is created in the context that contains the property definition. The property instance is created when one of the following happens:

- The property definition specifies an initial value – when the context instance is created.
- The property definition specified the *reference* attribute and a property instance with the same definition exists in a parent context instance – when the context instance is created. The property instance will have the same value as the referenced property instance.
- A value is first assigned to that property from any activity that executes within that context instance or within a child context.

The value of the property instance is modified when a new value is assigned to an existing property instance. An activity can access the value of a property instance only if the property instance exists. If the property instance does not exist, the activity throws the *bpml:noSuchInstance* fault.

6.1 Property Types

A property may have one of the following XML types:

- **Simple type** – The property type is one of the simple types defined by XML Schema or a type derived from it. The property value is an atomic value of that type or a derived type and can be empty.
- **Complex type** – The property type is a complex type defined by an XML Schema definition. The property value is a complex value of that type or of a derived type. The property value can be empty only if the complex type definition allows an empty sequence.
- **Element** – The property type is an element defined by an XML Schema declaration. The property value is an element of that type or of a derived type and cannot be empty.

If the property definition specifies an initial, the XML content of the initial value must match the production rules defined by the property type.

6.2 Fixed and Implicit

The value of a fixed property instance cannot be modified. Once a fixed property instance is created, its value remains constant. If a fixed property definition does not specify an initial value, a value can be assigned to that property once to create a new property instance.

Fixed properties should be used to represent values that do not change during the lifetime of a process instance. They can be used for static values, to correlate input messages, to hold the location of the service implemented by the process, and so forth. Properties that are used for process correlation must be fixed.

The process instance changes the values of implicit property instances. They are not defined explicitly and it is an error to define a property with the same name as an implicit property. Implicit properties are used to hold the process instance identifier (*inst:identifier*) and start time (*inst:startTime*), the iteration count (*inst:iteration*) and branch number (*inst:branch*) of a complex activity, the fault code (*inst:fault*), and so forth. Implicit properties are listed in Appendix A: Implicit Properties.

It is an error when a value is assigned to a fixed property or an implicit property. An activity that attempts to assign a value to a fixed property or an implicit property will throw the *bpml:readOnly* fault. A BPML implementation should detect such activities and flag the processes definition as erroneous.

6.3 Expressions

Expressions are used to establish values dynamically and can be derived from the values of one or more properties. They are used when assigning a value to a property, mapping an output value to an output message, and so forth. Conditions are expressions that result in a Boolean value. They are used for branching and iteration.

A BPML implementation must support the use of XPath 1.0 expressions. A BPML implementation may also support additional expression languages, such as XSTL and XQuery. While expression languages may differ in their capabilities and syntax, they all must access property instances in the same manner.

Evaluating an expression does not by itself cause a change to the value of any property. The expression may access any property instance that exists in the context, including fixed and implicit properties. If the expression attempts to access a property instance that does not exist, it throws the *bpml:noSuchInstance* fault. A BPML implementation can detect such an expression and flag the process definition as erroneous.

For XPath, XQuery and XSLT expressions, the value of the property instance is accessed from a variable with the same name. The qualified name is preceded with a dollar sign, for example, *$tns:myValue* to access the value of the property instance with the local name *myValue* and namespace name that is mapped to the prefix *tns*.

Typically the context node for evaluating an expression has the empty value. One exception is the *source* attribute (see the *assign* activity), which sets the context node to the value of a single property instance and allows only fixed property instances to be referenced from the expression.

All expressions are evaluated in an atomic manner. When multiple expressions are evaluated at once by a single activity, such as is the case with the conditions of the *switch* activity, this rule is further extended to cover them all.

The BPML specification supports the following type conversions:

- All type conversions that are covered in the XML Schema and XPath 1.0 specifications
- An element derived from a simple type can be converted to an atomic value
- An atomic value can be converted to an element derived from a simple type
- An element derived from a complex type can be converted to that complex type

- A complex type can be converted to an element derived from that complex type, if there are no additional restrictions in the element definition
- A sequence of elements can be converted to a complex type that has a matching content model

A BPML implementation may support additional type conversions, such as defined in XPath 2.0 and XQuery 1.0. Expression can be used to perform complex conversions. For example, an expression can extract an atomic value from the attribute of an element or to transform the value of one element into another element of an unrelated type.

A type error occurs if a type conversion is not possible between a value used in the expression and an operand or function that operates on that value, or if a type conversion is not possible between the result of the expression and the expected result type. Static type checking uses type information to identify such expressions and flag the process definition as erroneous. It is possible that static type checking would not detect the type errors and an attempt to evaluate the expression would throw the *bpml:typeMismatch* fault.

6.4 Property Definition

A property definition consists of the following attributes:

Attribute	Description
name	The property name.
type	The property type.
value	The initial value. (Optional)
fixed	True if fixed.
documentation	Documentation. (Optional)

The *name* attribute provides the property name. A context definition must not contain two property definitions with the same name. It is an error for two property definitions with the same name to exist in the package.

The *type* attribute specifies the property type. The BPML specification supports referencing an XML schema type definition or element declaration.

The *value* attribute provides the initial value for the property. If specified, the property is instantiated with that initial value.

The syntax for a property definition is given as:

```
<property
  name = NCName
  type = QName
  element = QName
  fixed = boolean>
  Content: (documentation?, value?)
</property>
```

The fully qualified property name is constructed by combining the *name* attribute with the *targetNamespace* attribute of the package.

A property definition must use one of the following attributes to specify the property type:

- **type** – References an XML Schema simple type or complex type definition using its qualified name
- **element** – References an XML Schema element declaration using its qualified name

If the *fixed* attribute is absent, the default value is *false*.

Referenced Definition

A reference property definition consists of the following attributes:

Attribute	Description
reference	The referenced definition name.
value	The initial value. (Optional)
fixed	True if fixed. (Optional)
documentation	Documentation. (Optional)

The *reference* attribute provides the name of another property definition. Only a context property definition may use this attribute. It must reference a property definition that exists in a parent context or at the package level. The property definition is said to be the same as the referenced property definition.

If the *fixed* attribute is unspecified, it has the same value as specified in the referenced definition.

If the *initial* attribute is unspecified, it has the same value as specified in the referenced definition, if specified there.

The syntax for a reference property definition is given as:

```
<property
  ref = QName
  fixed = boolean>
  Content: (documentation?, value?)
</property>
```

Initial Value

The *value* element can be used to specify an initial value for an XML type property definition.

The syntax for the *value* element is given as:

```
<value>
  Content: {mixed}
</value>
```

Service Reference

The *service* element can be used to specify an initial value that references a service definition. The *service* element is defined in the namespace http://www.bpmi.org/2003/BPML/instance.

The syntax for the *service* element is given as:

```
<service
  name = QName
  location = URI/>
```

The *name* attribute identifies the service definition by its qualified name.

The *location* attribute identifies the location of a document that contains the service definition. The document may contain any number of service definitions but must contain the named service definition. If the *location* attribute is absent, a BPML implementation must use other mechanisms to obtain the service definition.

The initial value indicates that the service definition is required in order to instantiate the process. For example, when the process requires the service definition in order to locate the service or to communicate the service endpoint to other processes.

The following example illustrates a property definition at the package level, a context property definition referencing it, another property defined in that context and an assignment that establishes the value of that property.

```
<property name="timeToPerform" type=”xsd:duration”>
  <value>PT24H</value>
</property>

<process name=”calculateEndTime”>
  <context>
    <property ref=”tns:timeToPerform”/>
    <property name=”endTime” type=”xsd:dateTime”/>
  </context>
  <assign property=”endTime”
          xpath=”$tns:timeToPerform + func:currentDateTime()”/>
</process>
```

Example 1 Package-level and local property definitions

7 Signals

Signals are used to coordinate the execution of activities executing in the same context. For example, to synchronize the start of one activity with the completion of another activity. Signals are also used to reflect conditions that arise from the execution of activities, and to allow other activities executing in that context to detect and react to these conditions. A signal does not cross the boundary of the context in which it is defined.

The *raise* activity "raises" the signal by creating a new instance of the signal. The *synch* activity "synchronizes" on the signal by waiting for the signal to be raised, and then "lowers" the signal by discarding one signal instance. The *raise* and *synch* activities can exchange information by passing values through the signal instance.

Signal Instance

The *raise* activity raises a signal by creating a new instance of that signal. The signal instance is created in an instance of the context that contains its definition. The *raise* activity can raise a signal defined in a parent context, but not a signal defined in a child or sibling context.

The signal is "raised" in a context if one or more instance of that signal exist in that context instance. The signal is "lowered" when no instances of that signal exist. The signal's raise count is the number of single instances that exist in the context instance.

The signal definition indicates whether the signal is single-raise or multi-raise. A context instance may have at most one instance of a single-raised signal, but can have zero or more instances of a multi-raise signal. If the signal is single-raise and is already raised in the context, the *raise* activity throws the *bpml:signalRaised* fault. If the *raise* activity must wait until the signal is lowered and then raise it, it should specify the *fault* attribute with the value *false*.

The *synch* activity synchronizes on a signal by looking for a matching signal instance in the current context. When it finds a matching signal instance, it discards it and completes. The *synch* activity can match and discard one signal instance each time it is executed.

If the *synch* activity determines that the signal will not be raised in the context, it throws the *bpml:noSuchSignal* fault. If the *synch* activity anticipates that the signal may not be raised and wants to abort, it should specify the *fault* attribute with the value *false*.

The *synch* activity determines that the signal will not be raised if the signal's source count is zero. The signal's source count is the number of sources that may raise the signal before the current activity instance completes. The source count is zero if the signal is only raised by the next activity in a sequence, since that activity will not execute until after the current activity completes. The source count is calculated from all activities that are executing or may execute in parallel with the current activity.

Each signal instance holds one value. The signal source can use that value to pass information to the synchronizing activity. The *raise* activity maps output values to the signal value and the *synch* activity assigns these values to properties in its current context.

The *synch* activity can match a particular signal instance using a synchronization condition. The synchronization condition may reference properties that are mapped from the signal. It can also determine the status of another signal by inspecting its raise count and source count.

The synch activity executes by retrieving all instances of the signal in the order in which they were created (raised). It evaluates the synchronization condition against each instance until it finds a match. If the signal is not raised or no match is found, it waits for the signal to be raised and evaluates the synchronization condition against the new instance. It completes only once a match is found, and discards the matching signal instance.

If the signal definition specifies an initial value, the signal starts in the raised state. When the context is instantiated, it creates one instance of the signal using the initial value.

The *spawn* activity implicitly defines a signal when it instantiates a new process. The signal source is the process instance and the signal is raised when the process instance completes or aborts. The signal is defined in the process context of the *spawn* activity and has the same name as the spawned process. The signal is multi-raise since the process can be instantiated multiple times. It is an error to define a signal with the same name in that context.

If the signal source is a property, the signal is raised when a value is assigned to the property and the condition evaluates to true. The signal value is set from the property value.

Signal Definition

A signal definition is a composition of the following attributes:

Attribute	Description
name	The signal name.
documentation	Documentation. (Optional)
type	The value type. (Optional)
multi	Multi- or single-raise.
source	The signal source as a property. (Optional)
value	Initial value. (Optional)

The *name* attribute provides a name that can be used to reference the signal. A context definition must not contain two signal definitions with the same name.

The *type* attribute specifies the signal value type. If specified, every signal instance will hold a value of that type. Otherwise, every signal instance will hold the empty value.

The *multi* attribute specifies whether the signal is multi-raise or single-raise. If the value of this attribute is *false*, the context instance may contain at most one instance of that signal.

The *source* attribute specifies a property that serves as the signal source. The signal is raised when a value is assigned to that property and the condition evaluates to true. The *source* attribute must name a property definition that exists in the same context or a parent context, and must not name an implicit or a fixed property. The signal value type is the same as the property type. A signal definition must not specify the *source* attribute in combination with the *type* or *value* attributes.

The *value* attribute provides an initial value for the signal. If specified, each context instance creates one signal instance using that initial value. If the *value* attribute is specified and the *type* attribute is unspecified it must contain an empty value.

The syntax for a signal definition is given as:

```
<signal
  name = NCName
  type = QName
  element = QName
  multi = boolean : false>
  Content: (documentation?, (value | source)?)
</signal>

<source
  property = QName>
  Content: (condition?)
</source>
```

The *type* and *element* attributes are the same as for the *properties* element. Only one of these attributes may be used. If both attributes are absent, all signal instances will contain an empty value.

The *value* element specifies an initial value. The syntax for this element is the same as for the *properties* element.

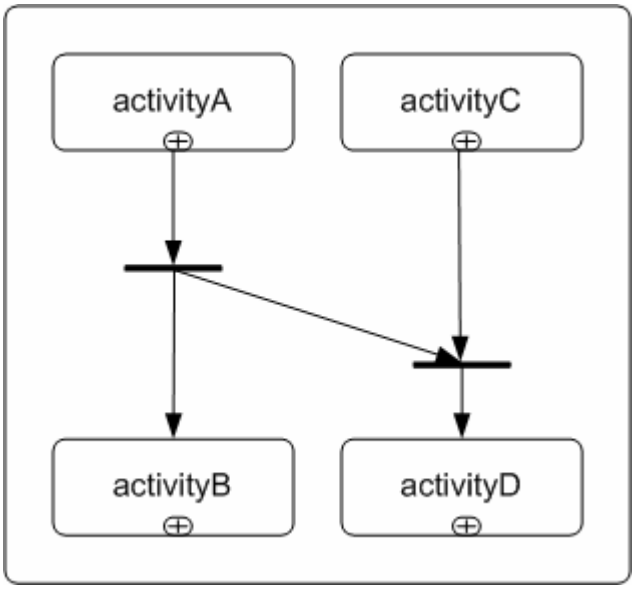
The syntax for the *condition* element is the same as for the *switch* activity. If the *condition* element is absent, the condition always evaluates to true.

Signals are often used to synchronize between parallel activities. In that capacity, the *raise* activity signifies the completion of an activity, and the *synch* activity signifies a synchronization barrier that waits for the completion of that activity.

This classical example involves four dependent activities. Activity B is dependent on the completion of activity A, but is not dependent on activities C or D. Activity D is dependent on the completion of both activity A and activity C, but not dependent on activity B. Activity A and C are not dependent on any other activity.

There are several ways to model this example using signals. Activity A may be the source of a signal on which both activities B and D synchronize. Activity C may be the source of a signal on which only activity D synchronizes. Activities A, B, C and D are started in parallel and use signals to ensure a proper order of execution.

In this example we have chosen to use two parallel sequences. One sequences activity A followed by activity B. The other sequences activity C followed by activity D. The signal is raised from one sequence and synchronized from the other.




```
<all>
  <context>
    <signal name=”tns:completedA”/>
  </context>
  <sequence>
    <call process=”activityA”/>
    <raise signal=”completedA”/>
    <call process=”activityB”/>
  </sequence>
  <sequence>
    <call process=”activityC”/>
    <synch signal=”completedA”/>
    </call process=”activityD”/>
  </sequence>
</all>
```

Example 2 Using signals to synchronize activities executing in parallel

This example illustrates an activity that obtains multiple quotes from different suppliers. For efficiency, the activity requests all the quotes simultaneously. It does so by spawning multiple instances of the same process.

Before proceeding to select the cheapest supplier, the activity must wait for all executing process instances to complete. It does so by synchronizing on a signal that is raised each time one of the spawned process instances completes.

```
<foreach select=”$tns:suppliers/supplier”>
  <spawn process=”tns:requestQuote”>
    <output parameter=”tns:supplier”>
      <source property=”inst:current”/>
    </output>
    <output parameter=”tns:orer”>
      <source property=”tns:order”/>
    </output>
  </spawn>
</foreach>

<assign property=”tns:count”
        xpath=”func:countInstances(’tns:requestQuote’)”/>

<while>
  <condition>$inst:iteration <= $tns:count</condition>
  <synch signal=”requestQuote”/>
</while>
```

Example 3 Using signals to detect completion of an asynchronous process

8 Schedules

A schedule represents a series of time events. The schedule fires at the specified time events. If the time event is calculated from property instances, the time event can be changed by assigning new values to these properties. The schedule can be modified, canceled or made to fire another time event.

At the specified time event the schedule fires by invoking a process. The schedule can also represent a time constraint by throwing a fault that causes the related activity or process to abort.

Schedule Instance

A schedule instance is created from a schedule definition. The schedule instance keeps track of the next time event and fires at the specified time instant. If the time event is in the past, the schedule instance fires when it is instantiated. If the time event is unspecified, the schedule instance does not fire.

The schedule instance calculates the next time event using the values of the *duration*, *instant* and *repeat* properties. The *duration* and *instant* properties are mutually exclusive. The schedule definition supports three attributes that provide the name of these properties.

The schedule instance calculates the next time event in the following manner:

- The value of the *instant* property provides the next time event. If the value is not of type *xsd:dateTime* or convertible to that type, the next time event is unspecified.
- The value of the *duration* property is added to the time instant at which the schedule was instantiated. The result provides the next time event. If the value is not of type *xsd:duration* or convertible to that type, the next time event is unspecified.
- The value of the *repeat* property is added to the current time instant. The result provides the next time event. If the value is not of type *xsd:duration* or convertible to that type, the next time event is unspecified.

When a new schedule instance is created it calculates the next time event using the *duration* or *instant* properties. The schedule instance re-calculates the next time event when a new value is assigned to either property.

When the schedule instance fires, it calculates a new time event using the current value of the *repeat* property. If the time event is unspecified the schedule will not fire again unless an assignment of the *duration* or *instant* properties causes it to calculate a new time event.

A schedule defined at the package level is instantiated using the initial values specified in the definition of these properties. A schedule defined within a context is instantiated using values from instances of these properties and tracks changes to their values.

A schedule defined within a context is instantiated when the parent activity instance transitions to the *active* state. It will use the *duration* property to calculate the next time event relative to the activity's start time.

If the schedule definition names a process, the schedule instance fires by invoking an instance of the named process. If the invoked process instance aborts, the schedule instance throws a fault with the same code in the context.

A schedule instance may invoke only one process instance at any given time. If the schedule is fired while the invoked process instance is still executing, it skips that invocation but calculates the next time event. Multiple schedule instances that reference the same process may invoke multiple instances of that process in parallel.

If the schedule definition specifies a fault code, the schedule instance fires by throwing that fault in the context. If the context defines a fault handler for that fault code, this causes the fault handler to execute next. Otherwise, the parent activity aborts. This type of schedule is used to express a time constraint on the completion of the parent activity.

Schedule Definition

A schedule definition is a composition of the following attributes:

Attribute	Description
name	The schedule name.
documentation	Documentation. (Optional)
process	The name of a process.
code	A fault code.
duration	The name of the *duration* property.
instant	The name of the *instant* property.
repeat	The name of the *repeat* property. (Optional)
other	Other mechanism.

The *name* attribute provides a name that can be used to reference the schedule definition. A context definition must not contain two schedule definitions with the same name. It is an error for two schedule definitions with the same name to exist in the same package.

The *process* attribute provides the name of a process. A schedule defined at the package level must name a process definition that exists in the package. A schedule defined within a context must name a process definition that exists in the same context, a parent context or the package. The named process definition must have instantiation type *activity* and must not specify any required input parameters.

The *code* attribute specifies a fault code. The *code* attribute is mutually exclusive with the *process* attribute. A schedule defined within a context must specify either attribute. A schedule defined at the package level must specify the *process* attribute.

The *instant*, *duration* and *repeat* attributes provide the name of the *instant*, *duration* and *repeat* properties, respectively. The *instant* and *duration* attributes are mutually exclusive. The *repeat* attribute is optional and may be used with both *instant* and *duration* attributes.

A schedule defined at the package level must name property definitions that exist in the package. A schedule defined within a context must name property definitions that exist in the same context or a parent context.

A schedule definition can use other mechanisms to specify the series of time events. For example, it can specify a particular time instant, a recurrence rule, or reference a calendar. A schedule definition that uses the *other* attribute must not use the *instant*, *duration* or *repeat* attributes.

The syntax for a schedule definition is given as:

```
<schedule
  name = NCName
  process = QName
  code = QName
  duration = QName
  instant = QName
  repeat = QName>
  Content: (documentation?, {extension element}?)
</schedule>
```

Extension elements can be used to specify a series of time instants using a mechanism defined in some other specification. Extensions elements are defined in a namespace other than the BPML namespace.

9 Exceptions

A process must be able to detect exceptional conditions, whether they are communicated by other processes using messages, signified by signals, or are the result of activities that fail to complete. The process reacts to these exceptional conditions using exception processes and fault handlers. If the process elects to back-out from a completed activity or select an alternative path for execution, it may need to utilize compensation processes.

9.1 Exception Processes

Exceptional events may occur that do not allow activities and processes to complete successfully. An exceptional event could be triggered from an input message, asking the process to terminate all work and complete immediately. It can also be triggered from a source that raises a signal to signify the exceptional condition.

Exception processes react to exceptional events that are signified by an input message or raised signal. An exception process is similar to a nested process, but can be instantiated only once within a context instance and only in response to an input message or a raised signal. When the exception process is instantiated, all other activities executing in that context are terminated.

An exception process interrupts the execution of a parent activity. The activity defines all the exception processes that interrupt it in a child context. Interruption can only occur while the activity instance is in the *active* state. Exception processes are not instantiated while their parent activity is in any other state.

When the activity instance is interrupted, it terminates all activities executing in its child context before responding to the exceptional event. Activities that cannot terminate are allowed to complete or abort. Termination of activities is covered in section 9.3 Terminating Activities.

From that point on the interrupted activity will not instantiate any nested process or exception process in response to an input message or a raised signal. Once all activities that are executing in the context instance have completed or aborted, the exception process can proceed to deal with the exception by executing activities from its activity set.

After the exception process instance completes the parent activity transitions to the *completed* state. If the exception process aborts, that fault is thrown in the context that defines the exception process. It will cause the parent activity to execute a fault handler or to abort.

An exception process instance is not terminated by throwing the *bpml:terminate* fault in its context. If a parent context responds to an exception while an exception process is executing in a child context, it must wait for the later to complete or abort.

An exception process definition is similar to a nested process definition, with the following differences:

- Exception processes do not support instantiation type *activity*, and cannot be instantiated from the *call* and *spawn* activities. The *parameters* attribute is not supported.
- Since only one exception process may be instantiated in a context instance, it has the same correlation as the parent process. The *correlation* attribute is not supported.
- Exception processes cannot be compensated. The *compensation* attribute is not supported.
- An exception process instance is persistent if the parent process instance is persistent. The *persistent* attribute is not supported.

An exception process is defined using the *exception* element. The syntax for the *exception* element is given as:

```
<exception
  name = NCName>
  Content: (documentation?, event, context?, {any activity}+)
</exception>
```

9.2 Faults and Fault Handlers

A **fault** indicates an erroneous condition that prevents an activity from completing successfully. The activity aborts to indicate that it cannot deal with the erroneous condition and to prevent the execution of downstream activities that may depend on it. The fault code is communicated to parent activities that can react to it.

The *delay* and *empty* activities never abort. The *fault* activity always aborts with the specified fault code. The *assign*, *raise*, *spawn* and *synch* activities will abort due to a type mismatch, erroneous expression, no signal source, and so forth.

The *call* and *compensate* activities will abort for the same reason. In addition, if the invoked process aborts, they will abort with the same fault code. If the *action* activity performs a synchronous operation that receives a fault message, it will abort with a fault code that identifies that fault message.

Complex activities recover from a fault by executing a **fault handler**. Successful completion of the fault handler signifies that the activity has recovered from the fault, allowing the activity to complete. If the fault handler aborts, or no suitable fault handler is defined, the activity must abort.

Fault handlers execute after exception processes. This allows the parent activity to react to an exception process that aborts.

When an activity aborts, that fault code is set in the context instance in which the activity executed. We say that the fault is *thrown* in that context. This behavior applies to simple and complex activities alike. It also applies to any nested processes or exception processes instantiated in that context and to every schedule instantiated in that context.

The fault code is assigned to the implicit property *inst:fault*. This property is implicitly defined as part of every context definition. It is an error to define a property with this name, or to assign a value to this property. A context instance can respond to only one fault. As such, the *inst:fault* property can be assigned once in a given context instance. If a second fault occurs it will not change the value of the *inst:fault* property.

The fault code is a qualified name that is used to identify the fault, allowing the process to react differently to each fault using specific fault handlers. Fault codes are not explicitly defined. The BPML specification defines some fault codes and indicates which activities throw these faults and for what reason. Process definitions will add additional fault codes to indicate erroneous conditions that are specific to their behavior.

When a fault is thrown in a context instance, the parent activity terminates all activities executing in its child context. Activities that cannot terminate are allowed to complete or abort. Termination of activities is covered in section 9.3 Terminating Activities.

From that point on the parent activity will not instantiate any nested process or exception process in response to an input message or a raised signal.

Once all activities executing in the context instance have completed or aborted, a fault handler is executed. Fault handlers are part of a context definition, and execute in the context instance in which the fault was thrown.

If the fault handler completes successfully, the activity transitions to the *completing* state and from there to the *completed* state. If the fault handler aborts or if no fault handler is associated with the fault code, the activity transitions to the *aborting* state and from there to the *aborted* state.

A fault handler specifies a set of fault codes and an activity list that is executed if the thrown fault is identified by that set. The context can define multiple fault handlers but must not define two fault handlers that respond to the same fault code. For convenience, the context may define one fault handler that responds to all fault codes that are not specified in any other fault handler.

The fault handler executes all activities from its activity list in sequential order. If any of these activities aborts, the parent activity aborts with the same fault code.

A process definition should include suitable fault handlers if it must perform any activities or invoke any compensation processes before aborting.

Fault Handler

The fault handler construct is a composition of the following attributes:

Attribute	Description
name	The handler name. (Optional)
code	One or more fault codes. (Optional)
documentation	Documentation. (Optional)
activities	One or more activities. (Ordered)

The *name* attribute provides a name that can be used to reference the fault handler. A context definition must not contain two fault handlers with the same name. If the *name* attribute is unspecified, the fault handler should be referenced by its ordinal position.

The *code* attribute specifies one or more fault codes. A context definition must not contain two fault handlers that specify the same fault code. A context definition may contain one fault handler that does not specify any fault codes.

The *activities* attribute provides a list of activities to execute.

A context definition groups all its fault handlers under a single *faults* element. The syntax for the *faults* element is given as:

```
<faults>
  Content: ((case+, default?) | default)
</faults>

<case
  name = NCName
  code = list of QName>
  Content: (documentation?, {any activity}+)
</case>

<default
  name = NCName>
  Content: (documentation?, {any activity}+)
</default>
```

The *case* element is used for a fault handler that specifies one or more fault codes. The *default* element is used for the one fault handler that does not specify any fault codes.

9.3 Terminating Activities

An activity can terminate without repercussion only when it is in the *ready* state. The activity in this case is canceled. If the activity is in any other state and it is not possible to terminate the activity, it must be allowed to complete or abort.

The *assign*, *empty*, *fault* and *spawn* activities take an insignificant amount of time to complete and are never terminated. The *delay* activity terminates immediately.

The *action* activity is in the *active* state when it performs or invokes an operation. Operations require only a short amount of time to complete and terminating an in-progress operation is disruptive. For that reason the *action* activity is allowed to complete the operation. The *action* activity is in the *ready* state when it is waiting to receive the first input message of the operation. In this state it terminates immediately.

The *synch* activity is in the *ready* state when it is waiting for a signal to be raised. The *raise* activity is in the *ready* state when it is waiting for a single-raise signal to be synchronized. When in this state, they terminate immediately.

The *call* and *compensate* activities must terminate the invoked process instance before they can terminate. They do so by throwing the *bpml:terminate* fault in the context of the invoked process instance.

Complex activities can terminate when they are in the *ready* or *active* state. In the *active* state they execute activities that are defined in a child context and terminate by throwing the *bpml:terminate* fault in that context instance. The complex activity reacts to the *bpml:terminate* fault in the same manner in which it will react to all other faults.

The *choice* activity is in the *ready* state when it is waiting for an event and in the *active* state while executing an event-triggered activity. Event-triggered activities that are in the *ready* state can terminate. If an event-triggered activity is already in the *active* state, it completes and the *choice* activity then completes or aborts.

A process terminates in the same manner as a complex activity. Complex activities nested within other complex activities and nested processes are terminated recursively by throwing the *bpml:terminate* fault in their context instance.

Throwing the *bpml:terminate* fault in a context instance causes the context instance to execute a fault handler. The fault handler should perform any activities that are required for recovery before the parent activity completes. The fault handler may throw a fault with a different code. Since a fault is thrown in the parent context before the child activity is terminated, the fault code from the child activity is ignored.

Terminating a parent activity does not terminate exception processes or fault handler executed by any child activity. If the child activity is executing an exception process, the *bpml:terminate* fault is thrown in its context only if the exception process completes. If it executes a fault handler then the *inst:fault* property is already associated with a fault code and the activity cannot react to another fault.

9.4 Compensation

When a process encounters an exceptional condition that prevents it from completing it should back-out from any steps it has already completed. In order to back-out the process would attempt to reverse the effects of completed activities.

We refer to the act of reversing the effects of a completed activity as *compensation*. The process should compensate for completed activities before it aborts, in order to recover from a fault and as part of handling an exception.

A *compensation process* is a process defined as part of a parent process for the purpose of reversing the effects of the parent process. In other words, the compensation process has the logic to undo what the parent process has done. When the parent process is used in a larger composition that must back-out, its compensation process is invoked to carry out that logic.

A process definition can define one compensation process. The compensation process can be instantiated only after the parent process instance completes. If the parent process encounters an error, it should use fault handlers to recover and back-out.

The compensation process context is a child of the parent process context. Although the compensation process executes when the parent process instance is in the *completed* state, it may access and modify the value of property instances in that context and may instantiate nested processes defined there.

The compensation process should use the same instantiation type as the parent process, allowing both processes to be instantiated from the same source. It should also use the same correlation as the parent process.

Once the parent process instance transitions to the *completed* state the compensation process starts responding to input messages or raised signals. The compensation process may be instantiated and completed only once for a given parent process instance. A new instance is not instantiated until an existing instance aborts, and must not be instantiated after an existing instance has completed.

The *call* and *spawn* activities cannot reference a compensation process. Instead, the *compensate* activity invokes a compensation process that has instantiation type *activity* by referencing the name of its parent process. It can reference a process instance that was instantiated using the *call* and *spawn* activities or a nested process that was instantiated in the same process instance.

The *compensate* activity ignores process instances that are in the *aborted* state, but throws the *bpml:compensation* fault if the named process instance is in the *active* state. It also ignores any process instance for which a compensation process was already instantiated and completed.

A compensation process definition is similar to a nested process definition, with the following differences:

- Since only one compensation process may be instantiated from a process instance, it has the same correlation as the parent process. The *correlation* attribute is not supported.
- Compensation processes cannot be compensated. The *compensation* attribute is not supported.
- A compensation process instance is persistent if the parent process instance is persistent. The *persistent* attribute is not supported.

A compensation process is defined using the *compensation* element.

The syntax for the *compensation* element is given as:

```
<compensation
  name = NCName>
  Content: (documentation?, (event | parameters?),
            context?, {any activity}+)
</compensation>
```

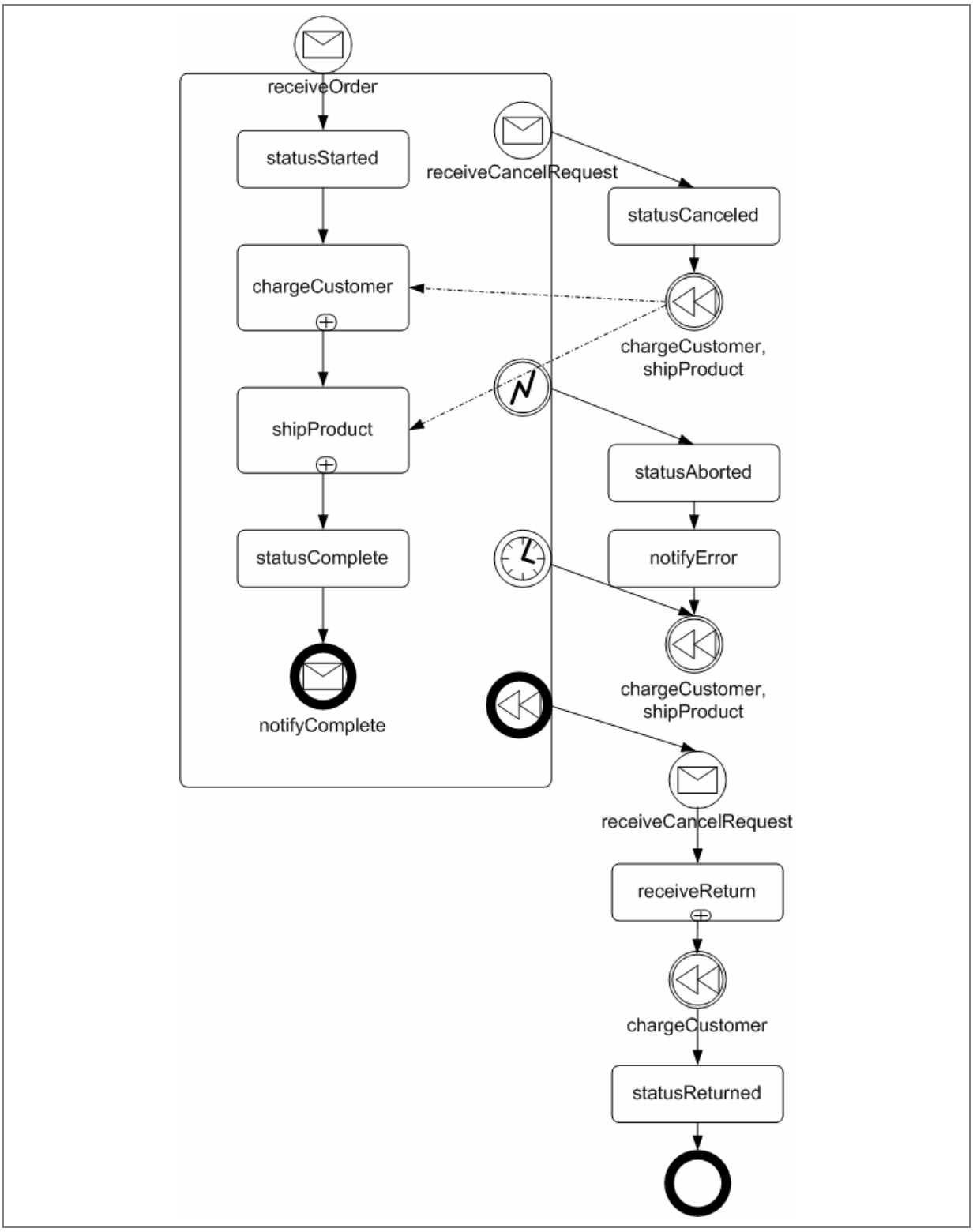
The following example illustrates the use of exceptions, fault handlers and compensation to implement a two-step order process.

The process starts in response to an order request that provides the order details and the expected time to complete. The operation is synchronous and returns the order identifier that is used for correlating subsequent messages received or sent by this process.

The process performs two steps by invoking the processes '*chargeCustomer*' and '*shipProduct,*' and then sends a notification message upon completion.

The process can be terminated at any point by sending a cancellation request that will instantiate the exception process '*cancelRequest*'.

If the process does not complete within the specified time limit, the fault '*tns:timeout*' will cause it to terminate.



If a fault occurs that prevents it from completing, the process sends a suitable notification message before aborting. The notification is not required if the process aborts due to a cancellation request or timeout, since the user is aware of these conditions.

Since the process may have completed '*chargeCustomer*' and/or '*shipProduct*' when a fault or exception occur, it compensates for them by invoking their compensation process (one issues a refund, the other cancels the shipment).

Once the process has completed, it is possible to compensate for the process by sending a cancellation request. This is the same operation as before, but occurs in a different context. In this state the order has already shipped and the process waits for the order to be returned before issuing a refund.

```
<correlation name="order" property="tns:orderID"/>

<property name="orderID" type="tns:orderID"/>

<process name="twoStepOrder"
         correlation="tns:order">

  <event activity="receiveOrder"/>

  <context>

    <property ref="tns:orderID" fixed="true"/>
    <property name="details" element="tns:orderDetailsType"/>
    <property name="tns:timeLimit" type="xsd:duration"/>

    <schedule name="timeToComplete" code="tns:timeout"
              duration="tns:timeLimit"/>

    <exception name="cancelRequest" >
      <event activity="receiveCancelRequest"/>

      <action name="receiveCancelRequest"
              portType="orderService" operation="cancelRequest"
              correlate="tns:orderID">
        . . .
      </action>

      <assign name="statusCanceled" property="status">
        <value>canceled</value>
      </assign>
```

```
      <compensate process="tns:activityA tns:activityB"/>
    </exception>

    <faults>
      <case code="tns:timeout">
        <assign property="status">
          <value>timeout</value>
        </assign>

        <compensate process="tns:activityA tns:activityB"/>
      </case>

      <default>
        <assign name="statusAborted" property="status">
          <value>aborted</value>
        </assign>

        <action name="notifyError"
                portType="orderService" operation="notifyError">
          <output element="tns:orderID">
            <source property="tns:orderID"/>
          </output>
          <output element="tns:reason">
            <source property="inst:fault"/>
          </output>
        </action>

        <compensate process="tns:activityA tns:activityB"/>
      </default>
    </faults>

  </context>

  <action name="receiveOrder"
          portType="orderService" operation="order">
    <input element="tns:details"
           property="tns:orderDetails"/>
    <input element="tns:timeToComplete"
```

```
            property="tns:timeLimit"/>
    <output element="tns:orderID">
      <source property="tns:orderID"/>
    </output>
    <assign property="tns:orderID"
            xpath="func:newIdentifier('tns:orderID')"/>
  </action>

  <assign name="statusStarted" property="status">
    <value>started</value>
  </assign>

  <call process="tns:chargeCustomer">
    <output parameter="tns:details">
      <source property="tns:details"/>
    </output>
  </call>

  <call process="tns:shipProduct">
    <output parameter="tns:details">
      <source property="tns:details"/>
    </output>
  </call>

  <assign name="statusComplete" property="status">
    <value>complete</value>
  </assign>

  <action name="notifyComplete"
          portType="orderService" operation="notifyCompletion">
    <output element="tns:orderID">
      <source property="tns:orderID"/>
    </output>
  </action>

  <compensation name="cancelRequest">
    <event activity="receiveCancelRequest"/>

    <action name="receiveCancelRequest"
```

```
            portType="orderService" operation="cancelRequest"
            correlate="tns:orderID">
      . . .
    </action>

    <call process="tns:receiveReturn">
      <output parameter="tns:details">
        <source property="tns:details"/>
      </output>
    </call>

    <compensate process="tns:chargeCustomer"/>

    <assign name="statusReturned" property="status">
      <value>returned</value>
    </assign>
  </compensation>

</process>
```

Example 4 Using exceptions, fault handlers and compensations to implement a two-step order

10 Transactions

This section explores the relation between atomic activities, persistent processes and transaction protocols.

It defines the behavior of an atomic activity that executes as a single unit of work. It also defines how two atomic activities that interact through message exchange execute in the context of an atomic transaction that applies atomic behavior to both activities.

It defines the behavior of persistent processes and how a BPML implementation provides a failure resilient environment for the execution of such processes.

Last, it explores the relationship between the BPML specification and existing transaction protocols such as BTP and WS-Transactions.

10.1 Atomic Activity

An **atomic activity** is an activity that executes as a single unit of work. An atomic activity ensures that other activities will not see partial results while the activity is executing. Instead, the other activities see only the result of its execution once it completes. Should the activity abort, any partial work is discarded.

Consider an *action* activity that assigns values to two properties from a single input message. If an error occurs while attempting to assign a value to the second property, the activity must abort. A fault handler can recover from the error and elect to execute a different set of activities. These activities may expect that the second property holds a previous value, or if the property is fixed holds no value. The *action* activity must behave in an atomic manner: it must either assign a value to both properties and complete, or abort without assigning a value to either property.

Most simple activities are atomic by default. The exception are the *action*, *call* and *compensate* activities. Complex activities are not atomic by default.

While atomic behavior is desirable in many situations, it is not practical in all situations. Atomic behavior would prevent two complex activities from executing in parallel and sharing information through properties or from coordinating their work through messages and signals.

The only way in which two atomic complex activities can interact is by executing in the same atomic context. Otherwise, one atomic activity may abort while the other one may complete. Having gained information from the aborting activity through the interaction, the completed activity has seen the partial results of its execution. We can no longer say that the aborted activity is atomic.

The fact that two atomic activities can interact only when executing in the same atomic context places a limitation on the utility of atomic activities. It prevents one activity from completing before all other activities it interacts with have completed, or from completing even though some activities have aborted. Complex processes, especially those executing over a long period of time and interacting with multiple participants should not be defined as atomic activities.

Atomic activities should be defined to execute over a relatively short period of time. An activity that can not complete in a short time frame should not be defined as an atomic activity. In particular activities that involve request/response interactions with users or over low-latency protocols such as SMTP.

Long-lived processes can and often are composed of multiple atomic activities, and the ability to model some complex activities as atomic simplifies the definition of these processes.

Behavior

An activity is defined as atomic by specifying the *atomic* attribute with the value *true*. This attribute is supported for all complex activity types and for the *action*, *call* and *compensate* activities.

If an activity is atomic than all its child activities execute in an atomic context. The atomic activity ensures that all activities from the activity set execute as one unit of work. It is an error if any child activity of the atomic activity specifies the *atomic* attribute with the value *false*.

An atomic activity must ensure that property assignments are not reflected outside the atomic context until the activity completes. The atomic activity must also ensure that the assignments do not take place if the activity aborts.

If any activity executing in the atomic context assigns a value to a property defined in a non-atomic context, no activity executing outside the atomic context is allowed to access that value until the atomic activity completes or aborts. The assignment takes place only if the atomic activity completes.

The atomic activity may need to assign the value of a property when it aborts. It does so using the *property* attribute of the *fault* activity. If that *fault* activity is the last activity executed by the atomic activity, an assignment to the named property would take place when the atomic activity aborts.

The process instance keeps track of all activity instances and makes their status available through instance functions. This allows one activity to determine the status of another activity instance. Instance functions executing in a non-atomic context can determine the status of an activity executing in an atomic context only after the parent atomic activity completes. Instance functions can determine the status of the atomic activity itself. Instance functions are covered in section 12.2 Instances.

If the atomic activity performs an asynchronous operation that sends an output message, the output message may be received after the atomic activity completes or aborts. To prevent inconsistencies, the output message is sent only if the atomic activity completes, and is not sent if it aborts. The atomic activity must not expect to receive an input message in response to an output message that was sent in this manner.

If the atomic activity performs an asynchronous operation that receives an input message and aborts, it provides no indication whether or not the message was received. To prevent inconsistency, the input message is not considered received until the atomic activity completes.

If the atomic activity performs a synchronous operation that sends an input message and receives an output message, it may introduce a dependency with an activity executing in a different process. To prevent inconsistency, both activities must complete or abort together. They should use a communication protocol that allows this form of coordination. Such communication protocols are discussed in section 10.3 Transactions.

If the atomic activity raises a signal that is defined in a non-atomic context, the signal instance is not available to any activity executing outside the atomic context until the atomic activity completes. The signal instance is discarded if the atomic activity aborts.

If the atomic activity synchronizes on a signal that is raised in a non-atomic context, the signal instance is discarded only if the atomic activity completes. Activities executing outside the atomic context must wait for the atomic activity to complete before they can synchronize on that signal.

If the atomic activity invokes a process using the *call* or *compensate* activities or using a schedule, the atomic behavior is extended to include the invoked process instance. This applies whether or not that process is defined as an atomic activity. However, it is an error if the process specifies the atomic *attribute* with the value *false*.

If the atomic activity instantiates a process using the *spawn* activity and that process is not defined in the same atomic context, the process is instantiated only when the atomic activity completes and is not instantiated if the atomic activity aborts.

10.2 Persistent Processes

An activity that executes over a long period of time must execute in a failure resilient environment. If a system failure occurs, the activity must be allowed to continue executing from a consistent state.

Consider a process that orders a product, submits payment and waits for the product to arrive when a system failure occurs. If the process is discarded, we have no recollection that the product was ordered and payment was made. Attempting to execute a new process instance will result in two orders, two payments and two shipments. After system failure, the process should proceed from the same point it was at before the failure – waiting for the shipment of the product.

A persistent process is failure resilient. Persistence is defined as the ability to recall a process instance for an indefinite amount of time after the process has completed or aborted and the ability to recall a consistent process instance after system failure.

A *consistent process instance* is defined as the state of the process instance that exists after the execution of atomic activities. The process instance must recall all atomic activities that have completed or aborted. The process instance must not reflect the partial execution of any atomic activity. The activity behaves as if it aborted, but the process instance has no recollection that it ever executed.

A BPML implementation provides a failure resilient environment for the execution of processes by implementing backward and forward recovery. Forward recovery is the ability to recall a process instance after system failure, bring it to a consistent state and then continuing executing it. Backward recovery is the ability to abort and discard an atomic activity that did not complete or abort before system failure occurred.

Business intelligence and performance analysis require access to both current and historical data. Aggregate queries are performed over process instances that have executed over a significant period of time that can range in months or years, and often the majority of these process instances are no longer active.

A persistent process instance can be retained for an indefinite period of time to allow such queries to access its properties. The definition of such queries and the means for specifying the desired period of time are covered in the BPQL specification.

Behavior

A process instance is persistent if the process definition specifies the *persistent* attribute with the value *true*. A process instance is not persistent if the process definition specifies the *persistent* attribute with the value *false*. A process definition must not define a compensation process and specify the *persistent* attribute with the value *false*. A process instance is also persistent if some other mechanism indicates that it is persistent.

A nested process instance is persistent if it is instantiated in a persistent process instance and does not specify the *persistent* attribute with the value *false*. Exception and compensation process instances are persistent if the parent process instance is persistent.

A process instance is persistent if it is instantiated from a persistent process instance using the *call* or *spawn* activities or using a schedule and it does not specify the *persistent* attribute with the value *false*.

The *call* and *compensate* activities are not atomic by default. If they are not defined as atomic, or if they instantiate a process that is not atomic, they will continue executing after recovery. If a system failure occurs while the *call* activity invokes a non-persistent process instance, the activity will invoke the process instance again after recovery.

The *action* activity is not atomic by default. If it is not defined as atomic, after recovery it must abort with the fault *bpml:terminate*.

A persistent process instance is retained after the process completes if the process definition specifies a compensation process. The process instance can be discarded after the compensation process has completed, or when the compensation process can no longer be invoked.

10.3 Transactions

Transaction protocols allow two processes that interact through message exchange to coordinate the completion of their activities. Transaction protocols are used when two processes must assure that their activities complete or abort together. We cover two types of transactions and related protocols: atomic transactions and open nested transactions.

Atomic Transactions

Atomic transactions allow two systems to execute atomic activities in a shared atomic context, such that both activities complete or abort together. The atomic transaction provides the context that spans the execution of the two atomic activities. The outcome of the transaction is a combination of the outcome of all activities that execute as part of its context. The outcome of the transaction is success if all activities are able to complete. The outcome is failure if any activity is not able to complete or if the state of that activity cannot be determined.

Atomic transactions are realized through transaction protocols that allow two systems to coordinate the execution of activities and communicate the outcome of the transaction. Transaction managers are used to maintain the transaction context and track the outcome of the transaction.

The transaction context is created before executing the first activity in the transaction. The transaction context is communicated as part of the message exchange to indicate to activities that they must execute as part of that transaction.

An atomic activity that executes in a transaction context does not complete until the transaction outcome is determined. It will remain in the *completing* state, awaiting a resolution of the transaction outcome. The activity transitions to the *completed* state only if the outcome of the transaction is success. If the outcome of the transaction is failure, the *bpml:rollback* fault is thrown in the context of the atomic activity.

If the outcome of the transaction is failure, all of its activities are asked to abort. An atomic activity that aborts forces the transaction outcome to become failure. If the outcome of the transaction is success, than all activities executing in that transaction are able to complete.

An atomic activity demarcates the boundary of participation in the transaction context. The activity requires a transaction context in order to communicate it as part of messages exchanged with other services. An atomic activity instance cannot participate in two unrelated transaction contexts.

If the activity starts by performing a synchronous operation that responds to an input message, it obtains the transaction context from the input message.

To execute activities as part of the same transaction, two systems must use a common transaction protocol. A single transaction can involve the use of multiple transaction protocols, but any two systems interacting with each other must agree to use the same protocol.

The BPML specification does not mandate the use of any particular transaction protocol. A BPML implementation should support existing atomic transaction protocols such as BTP, OTS, WS-Transaction and X/Open XA. The BPML specification was designed to leverage these transaction protocols.

Open Nested Transactions

Non-atomic activities that need to coordinate with each other use open nested transactions. While atomic transactions are the domain of short-lived transactions, open nested transactions are suitable for short- and long-lived transactions. An open nested transaction can involve activities that perform request/response interactions using asynchronous operations and activities that execute over a long period of time.

An open nested transaction can encompass one or more open nested and atomic transactions. Transactions can be nested within each other to any arbitrary level. A non-atomic activity cannot participate in two unrelated transaction. However, it can execute atomic and non-atomic activities in unrelated transaction contexts, allowing transactions to be interleaved.

Non-atomic activities communicate the transaction context and transaction outcome in a similar manner to atomic activities. To participate in an open nested transaction, the non-atomic activity must execute as a persistent process instance or as an activity in the context of one. A process instance that does not specify the *persistent* attribute becomes persistent if it participates in an open nested transaction.

The BPML specification does not mandate the use of any particular transaction protocol. A BPML implementation should support existing transaction protocols such as BTP and WS-Transaction. The BPML specification was designed to leverage these transaction protocols.

11 Activity Types

This section defines all the activity types that are a normative part of the BPML specification.

11.1 Action

The *action* activity performs or invokes a single operation that involves the exchange of input and output messages.

The *action* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
operation	The operation name.
input	Zero or more mappings from the input message to properties.
output	Zero or more mappings from output values to the output message. (Ordered)
locator	The locator. (Optional)
correlate	One or more correlations. (Optional)
activities	One or more activities. (Optional, Ordered)

The *action* activity is a simple activity. It performs or invokes a single operation and completes after the operation completes or aborts if the operation fails.

The *operation* attribute provides the name of the operation. The activity does not define the operation. Rather it provides the context for execution. The semantics of the activity depend on the definition of the operation and the manner in which the operation is used (see semantics).

The activity passes values in the output message by mapping output values to the message. The *output* attribute is an ordered collection of zero or more mappings. Each mapping produces one output value. The activity throws the *bpml:typeMismatch* fault if an output value does not match the type declared by its mapping. The result of all mappings is an ordered sequence of elements. The activity throws the *bpml:typeMismatch* fault if it cannot map the result to the output message.

The activity can map values from the input message to properties in the current context. The *input* attribute is a collection of zero or more mappings. The source value is an ordered sequence of elements derived from the message contents. Each mapping produces one input value that is assigned to a property defined in the current context. Information that is not mapped is lost.

It is an error for two mappings to exist for the same property, if that property cannot be assigned in the current context or if the input value and property are of incompatible types. The activity throws the *bpml:typeMismatch* fault if it cannot map the input value to the property.

When using WSDL 1.1, the abstract message definition specifies the message content model. If the message definition consists of a single message part of a complex type, the message content model is the same as the complex type content model. If the message definition consists of zero or more message parts, the message content model is an ordered sequence of elements as given in the message definition.

The *locator* attributes provides the location of a service. When the activity invokes an operation it must provide the location of the service that performs that operation. The use of the *locator* attribute identifies whether the activity performs or invokes the operation.

The *correlate* attribute provides an association between the input message and the context instance in which the activity is executed. The correlation refers to properties mapped from the input message using selectors. It identifies the context instance in which the activity is performed by matching property instances with the same name and value.

The *activities* attribute provides a means for performing activities in response to an input message and before sending an output message. This attribute may be used when the activity responds to an input message by performing or invoking a synchronous operation.

If the activity responds to an input message, it remains in the *ready* state until the input message is received and transitions to the *active* state in order to receive the message.

If the activity encounters a communication error when attempting to send an output message or receive an input message, it throws the *bpml:communication* fault.

The syntax for the *action* element is given as:

```
<action
  name = NCName
  portType = QName
  operation = NCName
  correlate = list of QName
  locate = QName
  role = QName
  {extension attribute}>
  Content: (documentation?, (input | output)*, {any activity}*)
</action>
```

When the activity responds to an input message, the *input* elements should precede the *output* elements. When the activity initiates the operation with an output message, the *output* elements should precede the *input* elements.

When using WSDL 1.1, operations are referenced using the *portType* and *operation* attributes. Both attributes must be used in order to reference the operation definition. Using these attributes indicates that the operation definition adheres to the semantics defined in the WSDL specification.

Extension attributes can be used to reference operations defined in other languages. Extensions attributes are defined in a namespace other than the BPML namespace.

Semantics

An operation is *asynchronous* when it involves a single input or output message. The operation completes after it receives the input message, or sends the output message. When sending a message, the operation does not wait for an indication that the message was received by its destination.

An operation is *synchronous* when it involves both an input and an output messages. The operation completes only after it sends an output message and receives an input message or after it receives an input message and sends an output message.

The activity *initiates* an operation when it begins by sending an output message. The activity *responds* to an operation when it begins by receiving an input message.

The activity *performs* an operation when it is not aware of the source of the input message and/or the destination of the output message. The operation definition specifies the input message (or messages) that the activity receives as its input and the output message (or messages) that the activity sends as its output.

The activity *invokes* an operation when it identifies the service that is the source of the input message and/or the destination of the output message. The operation definition specifies the input message (or messages) that the activity sends as its output and the output message (or messages) that the activity receives as its input.

Input

The *input* construct is a composition of the following attributes:

Attribute	Description
property	The property name.
element	An element name. (Optional)
expression	An expression. (Optional)

The *property* attribute provides the name of a property. The input value is assigned to that property. It must name a property that can be assigned in the current context. It is an error when an activity maps two input values to the same property. The activity throws the *bpml:typeMismatch* fault if it cannot map the input value to the property.

The input construct operates on a source value that is either a sequence of elements or an atomic value. If the *element* and *expression* attributes are unspecified, the input value is the same as the source value.

The *element* attribute provides the name of an element declaration. If specified, the source value must be a sequence of elements and the input value is the named element from that sequence. The activity throws the *bpml:typeMismatch* exception if the source value is not a sequence of elements, the sequence does not contain the named element, or the sequence contains multiple instances of the named element.

The *expression* attribute provides an expression that is evaluated to determine the input value. If specified alone, the expression is evaluated on the source value. If specified in combination with the *element* attribute, the expression is evaluated on that single element. The expression must not reference any properties.

The syntax for the *input* element is given as:

```
<input
  property = QName
  element = QName
  xpath = XPath/>
```

The *xpath* attribute specifies the expression in the form of an XPath expression. It must not reference any properties.

Output

The *output* construct is a composition of the following attributes:

Attribute	Description
type	The output value type.
values	A collection of one or more source properties and static values. (Optional)
expression	An expression to evaluate. (Optional)

The *type* attribute provides the type of the output value. The activity throws the *bpml:typeMismatch* fault if the output value does not match the declared type.

The *values* attribute is a collection of one or more source properties and static values. If the collection has more than one item, each item must provide a sequence of elements. The output value is a combination of these sequences.

The *expression* attribute provides the result of an expression. The expression is evaluated in the current context.

The *values* and *expression* attributes are mutually exclusive. If neither attribute is used, the output value is an empty value.

The syntax for the *output* element is given as:

```
<output
  type = QName
  element = QName
  xpath = XPath>
  Content: (((source | value)+ | {extension element})?)
</output>
```

The output value type is identified using one of the following attributes:

- *type* – Identifies an XML Schema simple or complex type definition using its qualified name
- *element* – Identifies an XML Schema element declaration using its qualified name

One of these attributes must be used.

The output value is defined using the following attributes and elements:

- *value* – A static value.
- *source* – A source property.
- *xpath* – An XPath expression.
- *extension* – An expression in any other language.

The *xpath* attribute and *extension* element are mutually exclusive and cannot be used together. They must not be used in combination with the *value* and *source* elements.

The definition of these elements is the same as for the *assign* activity.

Locator

In order to invoke an operation the activity must locate a service that is able to perform that operation. The *locate* attribute provides the name of a property, and the property's value provides the location of the service. Typically, it is a URI that identifies the service endpoint.

If the property value is of type *inst:service* or is convertible to that type, it references a service definition that may specify multiple endpoints and protocol bindings. The manner in which a BPML implementation selects a particular endpoint and protocol binding is undefined.

A service definition indicates which operations are supported by the service. The activity throws the *bpml:serviceType* fault if it detects that the invoked operation is not support by the specified service.

The activity throws the *bpml:noSuchInstance* fault if the property instance does not exist or has an empty value. It throws the *bpml:typeMismatch* fault if it cannot convert the property instance value to a service location.

Correlation

A correlation associates an input message to an activity instance. A correlation is required only if the activity responds to an input message and that input message can be received by multiple context instances

When using WSDL 1.1, the *correlation* attribute may be used when performing a *one-way* or *request-response* operation and when invoking a *notification* or *solicit-response* operation. It must not be used in any other case.

The correlation establishes a relation between an input message and the context instance of the activity. It does so by matching the values of properties that are mapped from the input message to property instances with the same name and value in the current context.

The context instance must have property instances with existing values before a correlation can be used. However, event-triggered activities that are non-exclusive can use correlations to instantiate a process upon receipt of multiple messages with the same correlation value.

The *correlate* attribute names one or more correlation definitions. Each correlation definition names one or more properties defined in that context. Selectors are used to extract the values of these properties from the input message. It is possible for the properties to be defined in different contexts, for the value of property instances to change between operations, and for the properties to be read-only.

Correlations can be used to distinguish between context instances of the same context definition, as well as between context instances of different context definitions. When two activity instances respond to an input message of the same type, it is a conflict only if the message can be associated with more than one of these activity instances.

Activities

The *activities* attribute allows the *action* activity to execute one or more activities after receiving an input message and before sending an output message as part of a synchronous operation.

When using a WSDL 1.1, the *activities* attribute may be used when performing a *request-response* operation and when invoking a *solicit-response* operation. It must not be used in any other case.

After the input message is received, input values are assigned to properties in the current context. Next, activities from the activity list are executed in sequential order. After executing all activities from the activity list, the output message is sent.

These activities may access the input values from the assigned properties and may assign values to properties that are then mapped to the output message.

If any activity in the activity list aborts, the action activity identifies a suitable output message based on the fault code, sends the output message and aborts with the same fault.

The following example illustrates a simple process that receives an order request and processes the order, replying with an acceptance of the order followed by an invoice. The process can be canceled by the customer before the invoice is submitted. For brevity, other messages and activities are not shown here.

We define a service type for this example that supports two operations: receiving an order request from the customer and receiving a request to cancel the pending order. These operations are performed by the example process given here.

The order message identifies the order using the order identifier that is assigned to a property with that name. The order identifier is used to correlate the cancellation request to that particular process instance.

We define the service type for the customer that supports two operations: receiving an acceptance message and receiving an invoice message. These operations are invoked against the customer service. The customer service is identified by a property that is assigned from the order request message. The order is identified by mapping the order identifier value to the two output messages.

This example does not show the activities that are used to construct the value of the '*invoiceDetails*' property.

```
<wsdl:message name=”requestMessage”>
  <wsdl:part name=”orderID” element=”type:orderID”/>
  <wsdl:part name=”sender” element=”type:senderService”/>
  <wsdl:part name=”details” element=”type:orderDetails”/>
</wsdl:message>

<wsdl:message name=”cancelMessage”>
  <wsdl:part name=”orderID” element=”type:orderID”/>
</wsdl:message>

<wsdl:portType name=”exampleServiceType”>
  <wsdl:operation name=”request”>
    <wsdl:input message=”srv:requestMessage”/>
  </wsdl:operation>

  <wsdl:operation name=”cancel”>
    <wsdl:input message=”srv:cancelMessage”/>
  </wsdl:operation>
</wsdl:portType>
```

```
<wsdl:message name="acceptMessage">
  <wsdl:part name="orderID" element="type:orderID"/>
</wsdl:message>

<wsdl:message name="invoiceMessage">
  <wsdl:part name="orderID" element="type:orderID"/>
  <wsdl:part name="details" element="type:invoiceDetails"/>
</wsdl:message>

<wsdl:portType name="customerServiceType">
  <wsdl:operation name="accept">
    <wsdl:input message="srv:acceptMessage"/>
  </wsdl:operation>

  <wsdl:operation name="invoice">
    <wsdl:input message="srv:invoiceMessage"/>
  </wsdl:operation>
</wsdl:portType>

<wsci:correlation name="orderID" property="tns:orderID"/>

<wsci:selector property="tns:orderID" element="type:orderID"/>

<bpml:process name="example">
  <bpml:event activity="receiveRequest"/>

  <bpml:context>
    <bpml:property name="orderID" type="type:identifier"/>
    <bpml:property name="customerService" type="inst:service"/>
    <bpml:property name="orderDetails" element="type:orderDetails"/>
    <bpml:property name="invoiceDetails" element="type:invoiceDetails"/>
  </bpml:context>

  <bpml:action name="receiveRequest"
               portType="srv:exampleServiceType" operation="request">
    <bpml:input element="type:orderID"
                property="bp:orderID"/>
    <bpml:input element="type:senderService"
```

```
                property="bp:customerService"/>
    <bpml:input element="type:orderDetails"
                property="bp:orderDetails"/>
  </bpml:action>

  <bpml:sequence>
    <bpml:context>

      <bpml:exception name="cancel">
        <bpml:event activity="receiveCancelRequest"/>

        <bpml:action name="receiveCancelRequest"
                     portType="srv:exampleServiceType"
                     operation="cancel" correlate="bp:orderID">
          . . .
        </bpml:action>
      </bpml:exception>

    </bpml:context>

    <bpml:action name="sendAcceptance"
                 portType="srv:customerServiceType" operation="accept"
                 locate="bp:customerService">
      <bpml:output element="type:orderID">
         <bpml:source property="bp:orderID"/>
      </bpml:output>
    </bpml:action>

    . . .

  </bpml:sequence>

  <bpml:action name="sendInvoice"
               portType="srv:customerServiceType" operation="invoice"
               locate="bp:customerService">
    <bpml:output element="type:orderID">
      <bpml:source property="bp:orderID"/>
    </bpml:output>
    <bpml:output element="type:invoiceDetails">
```

```
      <bpml:source property="bp:invoiceDetails"/>
    </bpml:output>
  </bpml:action>

</bpml:process>
```

Example 5 Sending and receiving messages

11.2 All

The *all* activity executes activities in parallel.

The *all* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
activity set	An activity set.

The *all* activity is a complex activity. It contains a single activity set and executes it exactly once. It executes all activities from the activity set in parallel.

The syntax for the *all* element is given as:

```
<all
  name = NCName>
  Content: (documentation?, context?, {any activity}+)
</all>
```

11.3 Assign

The *assign* activity assigns a new value to a property.

The *assign* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
property	The property name.
values	A collection of one or more source properties and static values. (Optional)
expression	An expression to evaluate. (Optional)

The *assign* activity is a simple activity. It assigns a new value to a property in the current context.

The *property* attribute provides the name of the property. It must reference a property that can be assigned in the current context.

The *values* attribute is a collection of one or more source properties and static values. If the collection has more than one item, each item must provide a sequence of elements and the assigned value is a combination of these sequences.

The *expression* attribute provides the result of an expression. The expression is evaluated in the current context.

The *values* and *expression* attributes are mutually exclusive. If neither attribute is used, the property is assigned an empty value.

The syntax for the *assign* element is given as:

```
<assign
  name = NCName
  property = QName
  xpath = XPath
  {extension attribute}>
  Content: (documentation?, ((source | value)+ |
                             {extension element})?)
</assign>
```

The value is defined using the following attributes and elements:

- *value* – A static value.
- *source* – A source property.
- *xpath* – An XPath expression.
- *extension* – An expression in any other language.

The *value* and *source* element can be used in combination to create a sequence of elements. Only one element may be used if the result of that element is an atomic value. The definition of the *value* element is the same as for a property definition

The *xpath* attribute and *extension* element are mutually exclusive and cannot be used together. They must not be used in combination with the *value* and *source* elements.

Extension elements are used to support other expression languages. Extension elements are defined in a namespace other than the BPML namespace. If an extension element is used, the manner in which the expression is evaluated is dependent on the specification that covers that particular extension element.

Source Attribute

The *source* attribute extracts a value from a single property instance.

The *source* attribute is a composition of the following attributes:

Attribute	Description
property	The source property.
expression	An expression to evaluate. (Optional)

The *property* attribute provides the name of the property. It must reference a property in the current context.

The *expression* attribute provides the result of an expression. The expression is evaluated on the value of the property instance. The expression may only reference fixed properties with an existing value.

The syntax for the *source* element is given as:

```
<source
  property = QName
  xpath = XPath>
  Content: ({extension element})?)
</source>
```

The *xpath* attribute provides an XPath expression that is evaluated with the value of the property instance as the context node.

Alternatively, an extension element can be used to support other expression languages. Extension elements are defined in a namespace other than the BPML namespace. If an extension element is used, the manner in which the expression is evaluated is dependent on the specification that covers that particular extension element.

11.4 Call

The **call** activity instantiates a process and waits for it to complete.

The *call* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
process	The process being instantiated.
input	Zero or more mappings from output parameters to properties.
output	Zero or more mappings from output values to input parameters.

The *call* activity is a simple activity. It instantiates a process and waits for it to complete. The process is instantiated in the same context in which it is defined. It can be different from the context in which the *call* activity is executed.

The *process* attribute names the called process. The process definition must have instantiation type *activity*. The process definition must exist in the current context. Starting with the current context and traversing up to the root context, the first process definition with that name is instantiated. Otherwise, a top-level process definition with that name is instantiated.

The activity passes parameter values to the process by mapping output values to input parameters. The process definition specifies the name and type of all input parameters. The *output* attribute is a collection of zero or more mappings. An output value must be mapped for every input parameter that is defined as required.

It is an error when two mappings exist for the same input parameter. It is also an error when the output value and input parameter are of incompatible types. The activity throws the *bpml:typeMismatch* fault if it fails to map the output value to the input parameter.

The process instance constructs output parameters that are passed back to the activity. The process definition specifies the name and type of all output parameters. The *input* attribute is a collection of zero or more mappings. An output parameter is mapped to a property defined in the current context. If an output parameter is not mapped, the value of that output parameter is lost.

It is an error when two mappings exist for the same property. It is also an error when the property cannot be assigned in the current context or when the value of the output parameter and property are of incompatible types. The activity throws the *bpml:typeMismatch* fault if it fails to assign the value of the output parameter to the property.

The activity creates an association between the two processes:

- If the called process instance aborts, the activity aborts with the same fault.
- If the called process defined a compensation process, the calling process can use the *compensate* activity with the process name to invoke the compensation process.
- In order for the activity to terminate, the called process instance is terminated first.
- Activities of the calling process instance can reference the called process instance using one of the instance functions defined in section 12.2 Instances.
- For information about atomic activities and persistent processes refer to section 10 Transactions.

The syntax for the *call* element is given as:

```
<call
  name = NCName
  process = QName>
  Content: (documentation?, output*, input*)
</call>
```

Input

The *input* construct is a composition of the following attributes:

Attribute	Description
parameter	The output parameter.
property	A property name.

The *parameter* attribute provides the name of the output parameter. It must reference the name of an output parameter from the called process definition.

The *property* attribute provides the name of a property. The value of the output parameter is assigned to that property. It must reference a property that can be assigned in the current context.

The output parameter and property must have the same or compatible types. The activity throws the *bpml:typeMismatch* fault if the output parameter cannot be assigned to the property.

The syntax for the *input* element is given as:

```
<input
  property = QName
  parameter = QName/>
```

If the *parameter* attribute is absent, it has the same value as the *property* attribute.

Output

The *output* construct is a composition of the following attributes:

Attribute	Description
parameter	The input parameter.
values	A collection of one or more source properties and static values. (Optional)
expression	An expression to evaluate. (Optional)

The *parameter* attribute provides the name of the input parameter. It must reference the name of an input parameter from the called process definition. The activity throws the *bpml:typeMismatch* fault if it cannot map the output value to the input parameter.

The *values* attribute is a collection of one or more source properties and static values. If the collection has more than one item, each item must provide a sequence of elements and the assigned value is a combination of these sequences.

The *expression* attribute provides the result of an expression. The expression is evaluated in the current context.

The *values* and *expression* attributes are mutually exclusive. If neither attribute is used, the output value is an empty value.

The syntax for the *output* element is given as:

```
<output
  parameter = QName
  xpath = XPath>
  Content: (((source | value)+ | {extension element})?)
</output>
```

The output value is defined using the following attributes and elements:

- *value* – A static value.
- *source* – A source property.
- *xpath* – An XPath expression.
- *extension* – An expression in any other language.

The *xpath* attribute and *extension* element are mutually exclusive and cannot be used together. They must not be used in combination with the *value* and *source* elements.

The definition of these elements is the same as for the *assign* activity.

11.5 Choice

The *choice* activity executes one of multiple activity sets in response to a triggered event.

The *choice* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
event handlers	Two or more event handlers. (Ordered)

The *choice* activity is a complex activity. It contains two or more event handlers that associate an event with an activity set. The *choice* activity executes a single activity set. It executes all activities from that activity sets in sequential order.

The *event handlers* attribute is a collection of two or more event handlers. An event handler associates an event with an activity set that is executed when the event occurs. An event is defined by an event-triggered activity that completes when the event is triggered.

The *choice* activity waits for the first event-triggered activity to complete. Only one event-triggered activity is allowed to complete. The *choice* activity starts in the *ready* state. Once an event-triggered activity transitions to the *active* state, the *choice* activity also transitions to the *active* state and all other event-triggered activities are terminated. Their associated activity sets will not execute.

Two event-triggered activities are not allowed to reach the *active* state together. The *choice* activity does not specify order of precedence for its event-triggered activities.

Event handlers are mutually exclusive and the *choice* activity must not have two event-triggered activities that respond to the same event.

An event handler construct is a composition of the following attributes:

Attribute	Description
documentation	Documentation. (Optional)
activity	The event-triggered activity.
activity set	An activity set.

Three event-triggered activity types are supported:

- **action** – An *action* activity that responds to an input message. The event is triggered if the input message is received. The *action* activity may generate an output message in response to the input message.
- **synch** – A *synch* activity that responds to a raised signal. The event is triggered if the signal is raised and is matched by the synchronization condition.
- **delay** – A *delay* activity that represents the passage of time. The event is triggered when the time instant is reached.

The event-triggered activity is executed in the current context of the *choice* activity. If a fault occurs while executing an event-triggered activity, the *choice* activity aborts with the same fault.

The branch number is the ordinal position of the selected event handler, starting with one for the first event handler in the set. The branch number is assigned to the property *inst:branch* in the context of the activity set.

The syntax for the *choice* element is given as:

```
<choice
  name = NCName>
  Content: (documentation?, event{2,*})
</choice>

<event>
  Content: (documentation?, (action | synch | delay),
            context?, {any activity}+)
</event>
```

In this example we use the choice activity to determine the status of order based on the response message and assign it the appropriate status code.

After sending the order we wait for a message indicating that the order was accepted or for a message indicating the order was rejected.

If the supplier has not responded within the specified time frame, we identify the order has not been processed and assign it the status code '*noResponse*'.

```
<action portType=”tns:supplier” operation=”submitOrder”/>
<choice>
  <event>
    <action portType=”tns:supplier” operation=”orderAccepted”/>
    <assign property=”tns:status”>
      <value>accepted</value>
    </assign>
  </event>
  <event>
    <action portType=”tns:supplier” operation=”orderRejected”/>
    <assign property=”tns:status”>
      <value>rejected</value>
    </assign>
  </event>
  <event>
    <delay duration=”tns:maxResponseTime”/>
    <assign property=”tns:status”>
      <value>noResponse</value>
    </assign>
  </event>
```

```
</choice>
```

Example 6 Waiting for one of two input messages or a time-out

11.6 Compensate

The *compensate* activity performs compensation for the named processes.

The *compensate* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
process	The process names.
output	Zero or more mapping from output values to input parameters.

The *compensate* activity is a simple activity. It invokes the compensation process of the named process instances.

The *process* attribute provides the names of one or more processes.

The activity obtains all instances of the named processes that were instantiated in the same process instance as the *compensate* activity or that were instantiated from the same process instance using the *call* and *spawn* activities. The activity does not obtain any instance of a process that was instantiated from a nested process instance.

All process instances must be in the *completed* or *aborted* state and must define a compensation process with instantiation type *activity*. The activity throws the *bpml:compensation* fault if any process instance does not meet these requirements.

The activity ignores any process instance that is in the *aborted* state, or if its compensation process was instantiated and completed. If the compensation process is executing, the activity will wait to determine whether it completes or aborts. The activity will instantiate a new instance of the compensation process only if the last instance aborted.

The activity passes parameter values to the compensation process by mapping output values to input parameters. The process definition specifies the name and type of all input parameters. The *output* attribute is a collection of zero or more mappings. An output value must be mapped for every input parameter that is defined as required by any compensation process.

It is an error when two mappings exist for the same input parameter. It is also an error when the output value and input parameter are of incompatible types. The activity throws the *bpml:typeMismatch* fault if it fails to map the output value to the input parameter.

The activity invokes compensation processes sequentially, in reverse to the chronological order in which their associated process instances have completed. It waits for each compensation process instance to complete before proceeding to the next compensation process.

The activity creates an association between the two processes:

- If any compensation process instance aborts, the activity aborts with the same fault.
- In order for the activity to terminate, the invoked compensation process instance is terminated first.

The syntax for the *compensate* element is given as:

```
<compensate
  name = NCName
  process = list of QName>
  Content: (documentation?, output*)
</compensate>
```

The syntax for the *output* element is the same as for the *call* activity.

This example uses the *compensate* activity to perform compensation for multiple nested activities.

The process '*compound*' completes after executing the nested processes '*activityX*' and '*activityY*'. If a fault occurs in the process, it recovers by compensating for either '*activityX*' *or* '*activityY*'.

The compensation process for '*compound*' compensates for both '*activityX*' and '*activityY*' in the proper order.

```
<process name="compound">
  <context>

    <process name="activityX">
      . . .

      <compensation name="compensateX">
        . . .
      </compensation>
    </process>

    <process name="activityY">
      . . .

      <compensation name="compensateY">
        . . .
      </compensation>
    </process>

    <faults>
      <default>
        <compensate process="tns:activityX tns:activityY"/>
      </default>
    </faults>
```



```
  </context>

  <spawn name="tns:activityX"/>
  <spawn name="tns:activityY"/>

  <compensation name="compensateCompound">
    . . .
    <compensate process="tns:activityX tns:activityY"/>
  </compensation>
</process>
```

Example 7 Using the compensate *activity to compensate for nested processes*

11.7 Delay

The *delay* activity expresses the passage of time.

The *delay* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
duration	The name of the *duration* property.
instant	The name of the *instant* property.

The *delay* activity is a simple activity. The *delay* activity remains in the *ready* state until the specified time instant, at which point it completes.

The time instant at which the activity completes is specified using one of the following attributes:

- **instant** – The attribute provides the name of a property. The value of the property is the time instant. If the value is not of type *xsd:dateTime* or convertible to that type, the activity completes immediately.
- **duration** – The attribute provides the name of a property. The value of the property is added to the current time instant. If the value is not of type *xsd:duration* or convertible to that type, the activity completes immediately..

The syntax for the *delay* element is given as:

```
<delay
  name = NCName
  duration = QName
  instant = QName>
  Content: (documentation?)
</delay>
```

11.8 Empty

The *empty* activity does nothing.

The *empty* activity is a simple activity. It can be used in places where an activity is expected, but no work is required.

The syntax for the *empty* element is given as:

```
<empty
  name = NCName/>
  Content: (documentation?)
</empty>
```

11.9 Fault

The *fault* activity throws a fault in the current context.

The *fault* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
code	The fault code. (Optional)
property	The fault property. (Optional)

The *fault* activity is a simple activity. It throws a fault in the current context and completes immediately.

The *code* attribute specifies the fault code. The value of the *code* attribute is assigned to the property *inst:fault*. If this attribute is unspecified the existing value of the *inst:fault* property is used. The *code* attribute may be unspecified when the *fault* activity appears in a fault handler and must be specified in all other cases.

The *property* attribute specifies a property that is modified when the parent atomic activity aborts. The *property* attribute is used when the *fault* activity appears in an atomic context and the property is defined in a non-atomic context.

The syntax for the *fault* element is given as:

```
<fault
  name = NCName
  code = QName
  property = QName>
  Content: (documentation?)
</fault>
```

11.10 Foreach

The *foreach* activity repeats the activity set once for each item in the item list.

The *foreach* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
select	A selection expression.
activity set	An activity set.

The *foreach* activity is a complex activity. It contains a single activity set that is executed zero or more times. It executes all activities from the activity list in sequential order.

The activity creates a new context instance and uses that instance for each iteration of the activity list.

The activity evaluates a selection expression in the current context. The result of the expression is a sequence of zero or more items. The activity iterates once for each item in the sequence. The value of the item for that iteration is assigned to the property *inst:current* in the context of the activity set.

The property *inst:current* is implicitly defined in the context of the activity set. It has the type *xsd:anyType*. It is an error when a property is defined with the same name or when a value is assigned to that property. That property is not accessible from a parent or sibling context.

The iteration count starts at one and is incremented once after each iteration. The iteration count is assigned to the property *inst:iteration* in the context of the activity set.

The syntax for the *foreach* element is given as:

```
<foreach
  name = NCName
  select = XPath>
  Content: (documentation?, context?, {any activity}+)
</foreach>
```

The *select* attribute is an XPath expression.

11.11 Raise

The *raise* activity raises a signal.

The *raise* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
signal	The signal name.
output	Zero or more mappings from output values to the signal value. (Ordered)
abort	Abort or wait.

The *raise* activity is a simple activity. It raises a signal but does not wait for the signal to be synchronized and completes immediately.

The *signal* attribute provides the signal name. The signal definition must exist in the current context. Starting with the current context and traversing up to the root context, the first signal definition with that name is used.

The activity passes information in the signal instance by mapping output values to the signal value. The *output* attribute is an ordered collection of zero or more mappings. Each mapping produces one output value. The activity throws the *bpml:typeMismatch* fault if an output value does not match the type declared by its mapping. The definition of the *output* construct is the same as for the *action* activity. The result of all mappings is an ordered sequence of elements. The activity throws the *bpml:typeMismatch* fault if it cannot map the result to the signal value.

If the signal is defined as single-raised and is already raised in the context, the activity can wait for the signal to be lowered or abort. If the value of the *abort* attribute is *true*, the activity throws the *bpml:signalRaised* fault. Otherwise, it waits until it can raise the signal and complete.

The syntax for the *raise* element is given as:

```
<raise
  name = NCName
  signal = NCName
  abort = boolean : true>
  Content: (documentation?, output*)
</raise>
```

The syntax for the *output* element is the same as for the *action* activity.

11.12 Sequence

The *sequence* activity executes activities in sequential order.

The *sequence* activity is a composition of the activity attributes and the following attribute:

Attribute	Description
activity set	An activity set.

The *sequence* activity is a complex activity. It contains a signal activity set and executes it exactly once. It executes all activities from the activity set in sequential order.

The syntax for the *sequence* element is given as:

```
<sequence
  name = NCName>
  Content: (documentation?, context?, {any activity}+)
</sequence>
```

11.13 Spawn

The *spawn* activity instantiates a process without waiting for it to complete.

The *spawn* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
process	The process name.
output	Zero or more mappings from output values to input parameters.

The *spawn* activity is a simple activity. It instantiates a process but does not wait for the process to perform any activity. The process is instantiated in the same context in which it is defined. It can be different from the context in which the *spawn* activity is executed.

The *process* attribute names the spawned process. The process definition must have instantiation type *activity*. The process definition must exist in the current context. Starting with the current context and traversing up to the root context, the first process definition with that name is instantiated. Otherwise, a top-level process definition with that name is instantiated.

The activity passes parameter values to the process by mapping output values to input parameters. The process definition specifies the name and type of all input parameters. The *output* attribute is a collection of zero or more mappings. An output value must be mapped for every input parameter that is defined as required.

It is an error for two mappings to exist for the same input parameter. It is also an error if the output value and input parameter are of incompatible types. The activity throws the *bpml:typeMismatch* fault if it fails to map the output value to the input parameter.

The activity creates an association between the two processes:

- A signal is implicitly defined in the spawning process context. The signal has the same name as the spawned process. The signal is raised when the process instance completes or aborts.
- If the spawned process defines a compensation process, the spawning process can use the *compensate* activity with the process name to invoke the compensation process.
- Activities of the spawning process instance can reference the spawned process instance using one of the instance functions defined in section 12.2 Instances.
- For information about atomic activities and persistent processes refer to section 10 Transactions.

The syntax for the *spawn* element is given as:

```
<spawn
  name = NCName
  process = QName>
  Content: (documentation?, output*)
</spawn>
```

The syntax for the *output* element is the same as for the *call* activity.

11.14 Switch

The *switch* activity executes one of multiple activity sets based on the truth value of a condition.

The *switch* activity is a composition of the activity attributes and the following attribute:

Attribute	Description
cases	One or more conditional cases. (Ordered)

The *switch* activity is a complex activity. It contains one or more activity sets and executes a single activity set based on the truth value of a condition. It executes all activities from that activity set in sequential order.

The activity selects the first conditional case with a condition that evaluates to *true.* After selecting a single conditional case, the associated activity set is executed in a new context instance. If no condition evaluates to true, it completes immediately.

The conditional case construct is a composition of the following attributes:

Attribute	Description
name	The conditional case name. (Optional)
documentation	Documentation. (Optional)
condition	The condition to evaluate.
activity set	An activity set.

The *condition* attribute specifies a condition to evaluate. The *condition* is evaluated in the current context. The order in which conditions are evaluated is not specified, only the order in which one activity set is selected. It is an error to depend on a particular order of evaluation.

The branch number is the ordinal position of the selected conditional case, starting with one for the first conditional case in the set. The branch number is assigned to the property *inst:branch* in the context of the activity set.

The syntax for the *switch* element is given as:

```
<switch
  name = NCName>
  Content: (documentation?, case+, default?)
</switch>

<case
  name = NCName>
  Content: (documentation?, condition,
            context?, {any activity}+)
</case>

<default
  name = NCName>
  Content: (documentation?, context?, {any activity}+)
</default>
```

The *default* element is used for the default case. The default case is always the last conditional case and its condition always evaluates to true.

Condition

The *condition* attribute is an expression that evaluates to a Boolean value.

The syntax for the *condition* element is given as:

```
<condition
  {extension attribute}>
  Content: {expression}
</condition>
```

An XPath expression is given in the content of the *condition* element.

Extension attributes are used to support other expression languages. Extension attributes are defined in a namespace other than the BPML namespace. If extension attributes are used, the manner in which the expression is evaluated is dependent on the specification that covers these extension attributes.

The *switch* activity is used to determine whether an order should be approved based on the total value of the order.

If the total value is over 5,000, the approval must be performed by a manager. This task is performed by the '*managerApproval*' process.

If the total value is over 500, only credit authorization is required. This task is performed by the '*creditAuthorization*' process.

All orders below 500 are always approved.

```
<switch>
```

```
  <case>
    <condition>$tns:totalValue > 5000</condition>
    <call process="tns:managerApproval">
      <input property="tns:approved" parameter="tns:approved"/>
    </call>
  </case>
  <case>
    <condition>$tns:totalValuel > 500</condition>
    <call process="tns:creditAuthorization">
      <input property="tns:approved" parameter="tns:authorized"/>
    </call>
  </case>
  <default>
    <assign property="tns:approved">
      <value>true</value>
    </assign>
  </default>
</switch>
```

Example 8 Determining whether an order should be approved based on its total value

11.15 Synch

The *synch* activity synchronizes on a signal.

The *synch* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
signal	The signal name.
condition	The condition to evaluate. (Optional)
abort	Abort or terminate.
input	Zero or more mappings from signal value to properties.

The *synch* activity is a simple activity. It waits in the *ready* state until it finds a matching signal instance, discards it and completes.

The activity executes by retrieving all instances of the signal in the order in which they were created (raised). It evaluates the synchronization condition against each signal instance until it finds a match. If no match is found, the activity waits for the signal to be raised again. It evaluates the synchronization condition against each new signal instance until a match is found.

The *signal* attribute provides the signal name. The signal definition must exist in the current context. Starting with the current context and traversing up to the root context, the first signal definition with that name is used.

The activity can map values from the signal value to properties in the current context. The *input* attribute is a collection of zero or more mappings. Each mapping produces one input value that is assigned to a property defined in the current context. Information that is not mapped is lost.

It is an error when two mappings exist for the same property, if the property cannot be assigned in the current context, or if the input value and property are of incompatible types. The activity throws the *bpml:typeMismatch* fault if it cannot map the input value to the property.

The *condition* attribute provides the synchronization condition. The synchronization condition is evaluated for each signal instance using properties mapped from the message. The condition may access other properties defined in the current context. A match is found if the condition evaluates to true. If the *condition* attribute is unspecified, the condition always evaluates to *true*.

If the activity detects that a signal will not be raised in the current context, it can terminate or abort. If the value of the *abort* attribute is *true*, the activity throws the *bpml:noSignalSource* fault. Otherwise, it terminates immediately.

The syntax for the *synch* element is given as:

```
<synch
  name = NCName
  signal = NCName
  abort = boolean : true>
  Content: (documentation?, condition?, input*)
</synch>
```

If the *condition* element is absent, the condition always evaluates to true. The syntax for the *condition* element is the same as for the *switch* activity.

The syntax for the *input* element is the same as for the *action* activity.

11.16 Until

The *until* activity repeats the activity set once or more based on the truth value of a condition.

The *until* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
condition	The condition to evaluate.
activity set	An activity set.

The *until* activity is a complex activity. It contains a single activity set that is executed one or more times. It executes all activities from the activity list in sequential order.

The activity creates a new context instance and uses that instance for each iteration of the activity list.

The activity starts by performing the first iteration. After each iteration, the condition is evaluated. If the condition evaluates to *true*, the activity completes. Otherwise, it performs another iteration.

The iteration count starts at one and is incremented once after each iteration. The iteration count is assigned to the property *inst:iteration* in the context of the activity set. It can also be referenced from the condition.

The syntax for the *until* element is given as:

```
<until
  name = NCName>
  Content: (documentation?, condition,
            context?, {any activity}+)
</until>
```

The syntax for the *condition* element is the same as for the *switch* activity.

The *until* activity and *inst:iteration* property are used to repeat a task until successful completion, but no more than a specified number of times (5). We know the task performed successfully when the output parameter '*completed*' has the value *true*.

```
<process name="performWorkAndRetry" >
  <parameters>
    <input name="tns:details" use="required"/>
    <output name="tns:success"/>
  </parameters>

  <context>
    <property name="success" type="xsd:boolean"/>
    <property name="details" element="tns:workDetails"/>
  </context>

  <until>
    <condition>$tns:success and $inst:iteration <= 5</condition>

    <call process="tns:acquireResources">
      <output parameter="tns:details">
        <source property="tns:details">
      </output>
    </call>

    <call process="tns:performWork">
      <output parameter="tns:details">
        <source property="tns:details">
      </output>
      <input property="tns:success" parameter="tns:completed"/>
    </call>

  </until>
```

```
</process>
```

Example 9 Repeating a task until completion, but no more than a specified number of times

11.17 While

The *while* activity repeats the activity set zero or more times based on the truth value of a condition.

The *while* activity is a composition of the activity attributes and the following attributes:

Attribute	Description
condition	The condition to evaluate.
activity set	An activity set.

The *while* activity is a complex activity. It contains a single activity set that is executed zero or more times. It executes all activities from the activity list in sequential order.

The activity creates a new context instance and uses that instance for each iteration of the activity list.

Before each iteration, the condition is evaluated. If the condition evaluates to *false*, the activity completes. Otherwise, the activity list is executed again.

The iteration count starts at one and is incremented once after each iteration. The iteration count is assigned to the property *inst:iteration* in the context of the activity set. It can also be referenced from the condition.

The syntax for the *while* element is given as:

```
<while
  name = NCName>
  Content: (documentation?, condition,
            context?, {any activity}+)
</while>
```

The syntax for the *condition* element is the same as for the *switch* activity.

12 Functions

The BPML specification defines a number of functions that are required for the definition of executable processes. A process definition can use any of these functions in its expressions.

Functions are defined in the namespace *http://www.bpmi.org/2003/BPML/function.*

Each function's signature is presented in this form:

```
returnType funcName( paramType paramName, ... )
```

The type name of the return value is specified before the function name. If the function has no return value, the type name is omitted. If the function may return an empty value, the type name is followed by a question mark. If the function may return a sequence, the type name is followed by an asterisk.

The function name is followed by a parenthesized list of parameters. Two or more parameters are separated by commas. Each parameter identifies the type name and parameter name. The parameter name has no significance other than referencing that parameter in the definition of the function.

The parameter type name indicates that the function accepts an argument of that type or a compatible type in that position. If the function accepts an empty value for a given parameter, the type name is followed by a question mark. If the function accepts a sequence, the type name is followed by an asterisk.

All type name refer to types of XPath values. A type can be a specific atomic type, an attribute, a sequence, and so forth. The following type names are used:

- *'node'* indicates any XML node
- *'item'* indicates any item
- *'QName'* indicates a fully qualified name consisting of a namespace name and a local name
- a qualified name indicates any atomic type of the named type, or a derived type

Some functions take an argument that provides the name of a definition. The parameter type is either a '*QName*' or an '*item*'. If the argument type is not a '*QName*', its string value provides the qualified name. The fully qualified name is determined as described below.

The string value can take the form "prefix:local-part", or it can take the form "local-part". In the latter case, the namespace prefix is the empty string. The value of the namespace name is determined by searching the namespace nodes of the element containing the function. There must be at least one such namespace node whose namespace prefix matches the namespace prefix from the string value. If no such namespace node exists, the function throws the *bpml:noSuchInstance* fault.

12.1 Generic

xsd:dateTime currentTime()
xsd:dateTime currentDate ()
xsd:dateTime currentDateTime()

Returns the current date/time instant.

When invoked multiple times in the same expression, these functions return the same time instant. When multiple expressions are evaluated at once by a single activity, such as is the case with the conditions of the *switch* activity, this rule is further extended to cover them all.

item newIdentifier(QName type)
item newIdentifier(item type)

Returns a new value for use in an identifier of the specified type.

The *type* parameter provides the type name. It references one of the simple types defined by XML Schema or a type derived from it. The function returns a new identifier that is most suitable for use with a property of that type.

This function should not return the same value twice when called with the same type name, or attempt to minimize the frequency with which multiple values are returned.

xsd:dateTime? getNextInstant (QName schedule)
xsd:dateTime? getNextInstant (item schedule)

Returns the next time event associated with the named schedule instance, or the empty value if the next time event is unspecified.

The *schedule* parameter provides the schedule name. If the schedule is not defined in the same context, a parent context or at the package level, the function throws the *bpml:noSuchInstance* fault.

xsd:boolean getSourceCount(QName signal)
xsd:boolean getSourceCount(item signal)

Returns the signal source count as a Boolean value. The source count is *true* if the signal may be raised before the current activity instance completes, *false* otherwise.

The *signal* parameter provides the signal name. If the signal is not defined in the same context or a parent context, the function throws the *bpml:noSuchInstance* fault.

xsd:integer getRaiseCount(QName signal)
xsd:integer getRaiseCount(item signal)

Returns the signal raise count. The raise count is the number of instances of that signal that exist in the context instance.

The *signal* parameter provides the signal name. If the signal is not defined in the same context or a parent context, the function throws the *bpml:noSuchInstance* fault.

12.2 Instances

Each process instance keeps track of all activity instances that were instantiated during its execution. Instance functions are used to retrieve state information about these instances. For example, they can be used to determine if a nested process completed or aborted, or to obtain the start time of an activity.

An instance is only accessible from the process context of that process in which it was instantiated. An instance that was instantiated in a nested process is not accessible from a context of the parent context. The following instances are accessible:

- Any simple or complex activity instantiated in the process, if the activity definition specifies the *name* attribute.
- Any nested process or exception process instantiated in the process.
- Any process instantiated from the process using the *call* and *spawn* activities.
- The process itself.

Instance functions ignore any instances that are in the *ready* state.

xsd:integer countInstances(QName name)
xsd:integer countInstances(item name)

Returns a count of all named instances.

This is the only instance function that does not return a sequence.

xsd:dateTime* getStartTime(QName name)
xsd:dateTime* getStartTime(item name)

Returns the start time of all named instances.

The start time of an instance is the time instant at which it transitioned to the *active* state.

xsd:dateTime* getEndTime(QName name)
xsd:dateTime* getEndTime(item name)

Returns the end time of all named instances.

The end time of an instance is the time instant at which it transitioned to the *completed* or *aborted* state. A value is returned only for an instance that is in the *completed* or *aborted* state.

xsd:duration* getDuration(QName name)
xsd:duration* getDuration(item name)

Returns the execution duration of all named instances.

The execution duration of an instance is the difference between its end time and start time. If the instance is still executing, it is the difference between the current time and its start time.

inst:state* getState(QName name)
inst:state* getState(item name)

Returns the state of all named instances.

The state of an instance is one of: *active*, *completing*, *completed*, *aborting* or *aborted*.

xsd:QName* getAbortFault(QName name)
xsd:QName* getAbortFault(item name)

Returns the abort fault code of all named instances.

A value is returned only for an instance that aborted.

inst:state * getCompensationState(QName name)
inst:state * getCompensationState(item name)

Returns the compensation state of all named instances.

The compensation state of a process instance is the state of the last compensation process instance instantiated. The state is one of: *active*, *completing*, *completed*, *aborting* or *aborted*. If the compensation process was not instantiated, its state is not returned.

Appendix A: Implicit Properties

Implicit properties are defined in the namespace *http://www.bpmi.org/2003/BPML/instance*.

Name	Type	Description
branch	xsd:positiveInteger	The ordinal number of the activity set selected by the *choice* or *switch* activity.
current	xsd:anyType	The value of the item for the current iteration of the *foreach* activity.
endTime	xsd:dateTime	The time instant at which the process instance completed or aborted.
fault	xsd:QName	The fault code associated with the context instance.
identifier	inst:instance	The process instance identifier.
iteration	xsd:positiveInteger	The current iteration number of the *foreach*, *until* or *while* activity.
startTime	xsd:dateTime	The time instant at which the process instance started executing.
state	inst:state	The process instance state. One of: *active*, *completing*, *completed*, *aborting* or *aborted*.

Table 5 Implicit properties

The properties *endTime*, *identifier*, *startTime* and *state* are implicitly defined in every process context. With the exception of *endTime*, they are all instantiated when the process instance transitions to the *active* state.

The property *fault* is implicitly defined in every context and is instantiated when a fault is thrown in the context.

The properties *branch*, *current and iteration* are implicitly defined in the context of the activity set of some complex activities. The table above indicates which property is defined by which complex activity.

Appendix B: References

B.1. Normative

RFC-2119

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

URI

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

WSCI 1.0

Web Services Choreography Interface (WSCI) 1.0, BEA, Intalio, Sun, SAP et al, June 2002
http://www.intalio.com/wsci/

WSDL 1.1

Web Services Description Language (WSDL) 1.1, W3C Note, March 2001
http://www.w3.org/TR/wsdl.html

XML 1.0

Extensible Markup Language (XML) 1.0, Second Edition, Tim Bray et al., eds., W3C, October 2000
http://www.w3.org/TR/REC-xml

XML-Namespaces

Namespaces in XML, Tim Bray et al., eds., W3C, January 1999
http://www.w3.org/TR/REC-xml-names

XML-Schema

XML Schema Part 1: Structures, Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, W3C, May 2001
http://www.w3.org/TR/xmlschema-1//

XML Schema Part 2: Datatypes, Paul V. Biron and Ashok Malhotra, eds., W3C, May 2001
http://www.w3.org/TR/xmlschema-2/

XPath

XML Path Language (XPath) 1.0, James Clark and Steve DeRose, eds., W3C, November 1999
http://www.w3.org/TR/xpath

B.2. Non-Normative

Activity Service

Additional Structuring Mechanism for the OTS specification, OMG, June 1999
http://www.omg.org

J2EE Activity Service for Extended Transactions (JSR 95), JCP
http://www.jcp.org/jsr/detail/95.jsp

BPMN

Business Process Modeling Notation, BPMI, 2002
http://bpmi-notation-wg.netfirms.com

BPQL

Business Process Query Language, BPMI
http://www.bpmi.org

BTP

Business Transaction Protocol, OASIS, May 2002
http://www.oasis-open.org/committees/business-transactions/

Dublin Core Meta Data

Dublin Core Metadata Element Set, Dublin Core Metadata Initiative
http://dublincore.org/documents/dces/

OMG OTS

Transaction Service, OMG, November 1997
http://www.omg.org

Open Nested Transactions

Concepts and Applications of Multilevel Transactions and Open Nested Transactions, Gerhard Weikum, Hans-J. Schek, 1992
http://citeseer.nj.nec.com/weikum92concepts.html

RDF

RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft
http://www.w3.org/TR/rdf-schema/

SOAP 1.2

SOAP Version 1.2 Part 1: Messaging Framework, W3C Working Draft
http://www.w3.org/TR/soap12-part1/

SOAP Version 1.2 Part 2: Adjuncts, W3C Working Draft
http://www.w3.org/TR/soap12-part2/

UUID and GUID

UUIDs and GUIDs, Network Working Draft, February 1998
http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt

WS-Transactions

Web Services Transactions, IBM/Microsoft, August 2002
http://www-106.ibm.com/developerworks/library/ws-transpec/

XHTML

XHTML[tm] 1.0: The Extensible HyperText Markup Language, W3C Working Draft
http://www.w3.org/TR/xhtml1/

XPath 2.0

XML Path Language (XPath) 2.0, W3C Working Draft
http://www.w3.org/TR/xpath20

XQuery 1.0

XQuery 1.0: An XML Query Language, W3C Working Draft
http://www.w3.org/TR/xquery/

XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Working Draft
http://www.w3.org/TR/xquery-operators/

XLST 1.0

XSL Transformations (XSLT) Version 1.0, W3C, November 1999
http://www.w3.org/TR/xslt