Business Process Modeling Language

Working Draft June 24, 2002

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 abstract and 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 the seventh working draft of the BPML specification submitted for comments by members of the BPMI initiative on June 24, 2002. It has been produced by members of the BPML working group.

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

This is a draft document and may be updated, replaced, or made obsolete by other documents at any time. It is inappropriate to refer to this document as other than “work in progress”.

Table of Contents

1. Introduction

The BPML specification provides an abstract model and XML syntax for expressing 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; rather it defines an abstract model and grammar for expressing generic 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. Conventions

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

1.1.1. 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.

1.1.2. 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 XSDL 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 may be 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.

1.1.3. Use of namespaces

The BPML schema contains element and attribute declarations, and simple type and complex type definitions defined in the **BPML namespace** http://www.bpmi.org/2002/6/bpml.

The following namespace prefixes are used throughout this document:

Prefix	Namespace URI	Definition
bpml	http://www.bpmi.org/2002/6/bpml	BPML namespace for BPML definitions
wsci	http://openuri.org/WSCI/2002/wsci10	WSCI namespaces for WSCI definitions
wsdl	http://schemas.xmlsoap.org/wsdl	WSDL namespace for WSDL definitions
xsd	http://www.w3.org/2001/XMLSchema	XSDL namespace for XSDL 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.

1.2. Dependency on Other Specifications

The BPML specification depends on the following specifications: XML 1.0, XML-Namespaces, XML-Schema and XPath. In addition, support for the following specifications is a normative part of the BPML specification: WSCI, WSDL and XQuery.

The following abbreviations are used throughout this document:

This abbreviation	Refers to

WSCI	Web Services Choreography Interface (see WSCI). This abbreviation refers specifically to version 1.0 of the specification, but is intended to support future versions of the WSCI specification.
WSDL	Web Service Description Language (see WSDL). This abbreviation refers specifically to the W3C Technical Note, 15 March 2001, but is intended to support future versions of the WSDL specification.
XPath	XML Path Language (see XPath). This abbreviation refers specifically to the W3C Recommendation, 16 November 1999, but is intended to support future versions of the XPath specification.
XQuery	XML Query Language (see XQuery). This abbreviation refers specifically to the W3C Working Draft, 20 December 2001, but is intended to support future versions of the XQuery specification.
XSDL	XML Schema structures and data types (see XML-Schema). This abbreviation refers specifically to the W3C Recommendation, 2 May 2001, but is intended to support future versions of the XML Schema specification.

1.3. Terminology

Constructs are the base parts that comprise the BPML abstract model.

Definitions are named constructs that can be referenced.

The **abstract model** defines the information that is used to express BPML definitions and explains their semantics.

The **BPML schema** is an XML grammar for representing BPML constructs in the form of an XML document.

A **BPML package** is a collection of definitions, including both BPML definitions and other definitions that are referenced by or necessary for the purpose of interpreting the BPML definitions.

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

A **BPML processor** is responsible to process XML documents that conform to the BPML schema and the rules set forth in this specification.

A **BPML implementation** is responsible to perform one or more duties based on the semantics conveyed by BPML definitions.

A **process** is a progressively continuing procedure consists of a series of controlled activities that are systematically directed toward a particular result or end.

An **activity** is a component that performs a specific function within the process, such as invoking a service or another process. Activities can be as simple as sending or receiving a message or as complex as coordinating the execution of other processes and activities.

An **atomic activity** is an elementary unit of work that cannot be further decomposed. Atomic activities can be used to start other processes, perform calculations, or perform operations.

An **operation** directs work done by a service by establishing communication with that service. The atomic activity that performs an operation is called an **action**.

Activities are related to each other by means of composition and contexts.

A **complex activity** is composed of other activities, and directs the execution of these activities. The complex activity instructs these activities to execute in sequential order or in parallel, to execute once or repeatedly, or even whether to execute or not conditionally. A process is a type of complex activity.

Activities always execute within a **context**. The context retains an association between the activities and information and services they utilize during execution, for example, properties that they can access, security credentials, transaction, exception handling, etc.

An **activity set** is a collection of one or more activities that execute in the same context. A complex activity, then, is an activity that comprises of one or more activity sets. Activity sets are used in other places as well, such as exception event handlers and transaction compensation.

A **flow** is the series of executing activities resulting from the execution of an activity set.

A **property** is a named value. Activities can access a property’s value or establish a new value. Properties are access as part of the context in which an activity is executed, also known as its **current context**. Properties are communicated between loosely coupled processes by performing operations and mapping properties to the message exchanged in these operations.

A **transaction** is a logical unit of work that must be executed in an all-or-nothing manner. Once the transaction completes, its effects can be reverted by performing **compensation**.

Exception handling defines activities that respond to an unexpected event. An exception could be the receipt of a message, a time-out, or a fault.

A **fault** is an error that occurs while executing an activity. Specifically, it is an error that occurs while executing an operation.

A **process definition** is used to define a process within a particular context. The context can be part of a larger process, in which case the definition is known as a **nested process**. The context can be that of a package, in which case the definition is known as a **top-level process**.

Process instance TBD

A **property definition** is used to define a property within a particular context. The context can be part of a process, or that of a package..

Instantiation is the act of creating a new instance of a definition. For example instantiating a process definition creates a new process instance.

A **selector** is used to instantiate a property from part of an input message.

Independent **services** are processes that are loosely coupled but interact with each other.

A **locator** is used by an action for the purpose of identifying a particular service against which the operation is performed.

A **connector** establishes communication between two operations performed by different services.

A **global model** is a composition of interacting processes and shows linkage between these processes through the exchange of messages.

Correlation TBD

1.4. Use of Documentation

Documentation of BPML documents and constructs with material for human or computer consumption is supported by allowing human and application information at the beginning of most BPML elements. The XML representation for this is the **documentation** element.

Syntax

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. The contents need not validate against any particular schema.

There are no requirements on the type of content that can be used. However, authors and tool providers may want to standardize on the following:

- Use of RDF for semantic meta-data intended for both human and computer consumption
- Use of XHTML for expression information intended for human consumption
- Use of Dublin Core Meta Data for additional meta-data and for the purpose of cataloging.

Example illustrating the use of the *documentation* element with XHTML and Dublin Core Meta Data elements:

```
<bpml:process name="example">
  <bpml:documentation>
    <rdf:label xml:lang="en">Example process</rdf:label>
    <xhtml:div xml:lang="en">
      Example for using the <code>documentation</code>
      element in a process definition.
    </xhtml:div>
    <dublin:creator>Arkin, Assaf</dublin:creator>
    <dublin:date>2001-01-29</dublin:date>
  </bpml:documentation>
  . . .
</bpml:process>
```

1.5. XML Values

Several BPML elements allow the expression of static values. Such is the case when instantiating the value of a property. The static value can be of a simple or complex type. The XML representation for a static value is the **value** element.

Syntax

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

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

The *value* element contains mixed content, including elements in any namespace. The contents need not validate against any particular schema.

2. Packages

BPML constructs are the base parts that comprise the BPML abstract model. The BPML specification provides the abstract model and XML Syntax for those constructs that are an essential part of an executable process definition.

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 definitions, including both BPML definitions and other definitions that are referenced by or necessary for the purpose of interpreting the BPML definitions. This would include XSDL definitions and declarations, WSCI and WSDL definitions, etc.

A **BPML document** is the XML representation of a BPML package based on the syntax given in this specification. At a minimum, a BPML document that conforms to version 1.0 of the BPML specification must validate against the BPML schema defined in version 1.0 of the BPML specification.

A BPML definition has a target namespace, which is a namespace name as defined by XML-Namespaces. The target namespace identifies the namespace within which names are associated with BPML definitions.

The name of a definition must be unique within the scope of all definitions of the same type, such that the definition can be unambiguously referenced by a combination of its name and type. This applies to both BPML definitions and other definitions that are referenced by BPML constructs, as defined in this specification. In its normative part, the BPML specification depends on other specifications that adhere to this requirement.

At the abstract level there is no requirement that definitions in the same package share the same target namespace, and it is allowed for the same definition to appear in more than one package. The XML syntax only allows a document to contribute definitions to a single target namespace. A BPML document uses the import mechanism to reference definitions from other namespaces as well as definitions from documents given in other languages.

BPML documents can be used for the purpose of exchanging BPML definitions between BPML processors. Where multiple documents are required, they can be aggregated by importing documents from known and accessible URLs.

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 may exist in a manner that is independent of any XML representation, and may be accessible when referenced, given its fully qualified name and type.

2.1. Package Syntax

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

```
<package
  targetNamespace = anyURI>
  Content: (documentation?, feature*, import*,
            (connect | model | process | property |
             {extension element})*)
</package>
```

The **targetNamespace** attribute provides the namespace name for all BPML definitions contained in this *package* element. The qualified name of each definition is a combination of the target namespace name and the name given to that definition.

The target namespace name may be a URL or a URN (see URI). There is no requirement for the namespace to be identical to the URL of the document containing this package element.

Feature

The **feature** element indicates to a BPML implementation that it may not be able to process all definitions contained in this *package* element unless it supports the named feature.

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

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

A feature is identified by its name and optional version number. For example, the following element declares that the definitions in a package require support for XQuery 1.0:

```
<bpml:feature name=”http://www.w3.org/TR/xquery”
              version=”1.0”/>
```

It is not necessary to specify features that are required to be supported by a conformant implementation (see conformance).

Import

An **import** statement is used to import definitions from a different namespace or a different document, including both BPML definitions and definitions given in other languages.

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

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

The **namespace** attribute provides the namespace from which definitions are to be imported. The value of the namespace indicates that the importing package may contain qualified references to definitions made in that namespace.

The **location** attribute provides the location of the package to import. It is expected to be a URL that points to a document. However, BPML processors may support other forms of referencing.

It is an error if both *namespace* and *location* attributes are absent.

The abstract package is a combination of all definitions appearing inside the *package* element and all definitions imported there.

The *package* element is an extensible element. It can contain definitions that are covered by other specifications using extension elements. Extension elements must use a namespace that is different from that of BPML.

Example for using the *package* and *import* elements:

```
<package targetNamespace="http://www.bpmi.org/examples/import"
         xmlns="http://www.bpmi.org/2002/6/bpml">

  <!—- Import BPML declarations from this document -->
  <import location="http://www.bpmi.org/examples/import/decls.bpml"/>

  <!—- Import WSDL definitions from this namespace -->
  <import namespace="http://www.bpmi.org/examples/import/service"/>
  <!—- Import BPML definitions from this namespace/document -->
  <import namespace="http://www.bpmi.org/examples/import/process"
          location="http://www.bpmi.org/examples/import/process.bpml"/>

  <!—- This package makes the following declarations/definitions -->
  <process name="ImportExample">
    . . .
  </process>

</package>
```

2.2. Conformance

Editor's note: We are soliciting input from members regarding the definition of conformance for incorporation into future working drafts of the BPML specification.

A **BPML processor** is responsible to process XML documents that conform to the BPML schema and the rules set forth in this specification, and any related specification that must be supported in order to fully conform to the requirements of the BPML specification.

A **BPML implementation** is responsible to perform 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.

A conformant implementation is required to read and process other type of documents and definitions that are supported as a normative part of the BPML specification.

A conformant implementation declares full conformance to a given specification by specifying the name and optional version that designates that specification. Full conformance is defined by the specification.

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

Specification		**Feature**
BPML	WD	http://www.bpmi.org/2002/6/bpml
WSCI	1.0	http://openuri.org/WSCI/2002/wsci10
WSDL	1.1	http://schemas.xmlsoap.org/wsdl
XPath	1.0	http://www.w3.org/TR/xpath
XSDL	1.0	http://www.w3.org/2001/XMLSchema

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 only appear where allowed.

3. Activities and Processes

A **process** is a progressively continuing procedure that consists of a series of controlled activities systematically directed toward a particular result of end. A process is defined as performing activities of varying complexity.

An **activity** is a component that performs a specific function within the process. For example, invoking another process. Activities can be as simple as sending or receiving a message, or as complex as coordinating the execution of other processes and activities.

Activities are either atomic or complex. An **atomic activity** is an elementary unit of work that cannot be further decomposed. A **complex activity** is composed of other activities, and directs the execution of these activities. A process is a type of complex activity.

Activities retain knowledge of their predecessors by executing within the same **context**.

This section describes the abstract model of activities and introduces the syntax elements that are common to all activity definitions. It then defines the abstract model of a process as a specialized type of a complex activity. Contexts are discussed in the next section.

3.1. Activities

3.1.1. Activity Type

All activities are defined based on a common activity type. The **activity type** consists of the following attributes:

Attribute	Description
name	The activity name.
other	Other attributes defined for the specific activity type.

The **name** attribute provides a name that can be used to distinguish the activity from all other activities within the same activity set, and ultimately within the process definition.

The activity name is optional, since it is possible to reference the activity by its ordinal position within the activity set.

Specific activity types may define other attributes that are specific to that type, for example, the condition used in the *while* activity, or the process name used by the *spawn* activity.

Other specifications may introduce additional activity types or utilize extension elements and attributes to provide additional semantics to activity types defined by the BPML specification.

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

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

Activities that are defined in other namespaces are supported by extending the complex type *bpml:activity* and using the abstract element *bpml:otherActivity* as a substitution group.

3.1.2. Activity Context

An activity is always executed within a context. The context in which an activity is executed is known as its **current context**.

When two or more activities are performed in the same context, they often use the context as a means for sharing properties. The activities can relate to each other by accessing properties within the same context.

For example, an activity that sends a request message can construct the output message based on a property whose value was established upon receipt of an input message by a prior activity.

Contexts are also used to distinguish between multiple instances of the same activity. Two instances of the same activity can only occur in separate instances of that context.

3.1.3. Activity Set

An **activity set** is a collection of one or more activities that execute in the same context, and the definition of the context in which they execute.

Attribute	Description
context	Context definition.
activities	One or more activities.

An activity set can contain any type of activity, including activity types defined by BPML and activity types defined by other specifications. A conformant implementation must recognize all activity types defined in this specification.

An activity must execute within a context. The activity set is a unit that comprises a context and all activities that execute in that context.

An activity set is used in a larger container that determines the rules for executing the activities in the activity set. Such containers include all complex activities, process definition, exception event handlers and other constructs such as transaction compensation.

The order of activities within an activity set is significant.

Although a process definition is a type of complex activity, a process definition cannot appear as an activity within an activity set. Nested processes are supported by associating the process definition within the context definition.

The syntax for an activity set is given as:

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

If the *context* element is absent, the context definition is treated as if empty (see contexts).

The XML syntax of an activity set is defined as the model group *bpml:activitySet*, which is used as the content for activity set container definitions.

3.1.4. Atomic and Complex

An atomic activity is an activity that cannot be further decomposed and is performed in an all-or-nothing manner.

The most common atomic activity is *action*. This activity performs a single operation such as sending a message (a *notification* operation) or performing a synchronous call to a service (a *solicit-response* operation).

Complex activities are used to coordinate the execution of multiple atomic and complex activities.

A complex activity is an activity that contains one or more activity sets. If more than one, it also defines the rules for selecting which activity set to execute. A complex activity also determines the number of times the activity set will execute and the order in which activities from the set will execute.

Unlike atomic activities, complex activities can be further decomposed into atomic activities and complex activities, recursively. An all-or-nothing behavior can be achieved by using transactions.

3.1.5. Activity Types

BPML defines the following type of activities:

Activity	Description
action	Atomic activity that performs an operation, in particular operations involving the exchange of messages with other processes and services.
all	Complex activity that executes all the activities within the activity set in non-sequential order.
assign	Atomic activity that assigns a new value to a property in the current context.
call	Atomic activity that instantiates a process and waits for it to complete.
choice	Complex activity that selects and executes one activity set in response to a triggered event.
compensate	Atomic activity that performs compensation for instances of the named transaction.
delay	Atomic activity that expresses the passage of time.
empty	Atomic activity that does nothing.
fault	Atomic activity that triggers a fault within the current context.
foreach	Complex activity that performs all the activities within the activity set repeatedly, once for each item in the list.
join	Atomic activity that waits for instances of process to complete.
sequence	Complex activity that performs all the activities within the activity set in sequential order.
spawn	Atomic activity that instantiates a process.
switch	Complex activity that selects and executes one activity set based on the evaluation of one or more conditions.
until	Complex activity that executes all the activities within the activity set repeatedly, one or more times, based on the truth value of a condition.
while	Complex activity that executes all the activities within the activity set repeatedly, zero or more times, based on the truth value of a condition.

3.1.6. Activity Instance

Each activity has a unique instance identifier. The instance identifier is held in a property that has the same name as the activity and is assigned in the activity's current context (see instance properties). This allows the activity's instance identifier to be referenced by other activities occurring in the same context.

An activity instance transitions through the following states:

- **Active** – The activity exists and is performing work specific to that activity type.
- **Completing** – The activity has performed all work specific to that activity type, and is now preparing to complete. This may involve additional work, such as transaction commit or context completion.

- **Complete** – The activity has performed all work required in order to complete successfully.
- **Aborting** – The activity has failed to complete successfully, and is now preparing to abort. This may involve additional work, such as transaction rollback or context completion.
- **Aborted** – The activity has failed to complete successfully and has performed all other work required in order to abort.

An activity instance always begins in the *active* state. Once the activity has completed successfully it transitions to the *complete* state. This is a terminal state. The activity transitions to the *complete* state through the *completing* state.

An activity will transition to the *aborted* state if it cannot complete successfully, for example, due to a fault or the triggering of an event handler that terminates the activity. This is a terminal state. The activity transitions to the *aborted* state through the *aborting* state. The activity may also transition to the aborting state from the *completing* state.

Although these states are defined for both atomic and complex activities, they are generally not visible for atomic activities. The instance property associated with an atomic activity is not accessible to other activities until the atomic activity reaches a terminal state.

An activity can be cancelled if it is currently in the *active* state, but its execution has caused no side effects or all side effects are completely reversible. Cancellation of an activity is akin to the state before the activity has started.

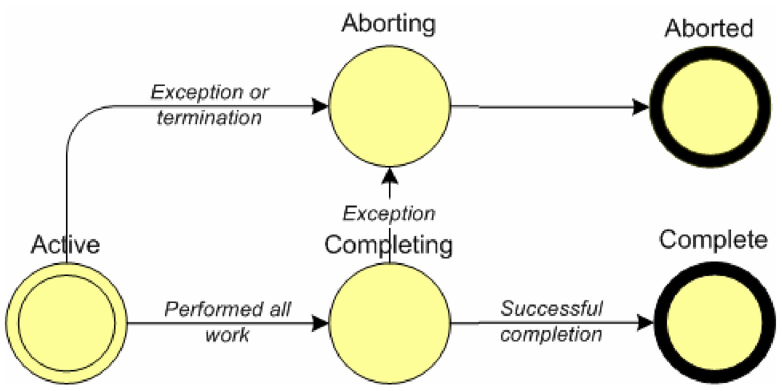

Figure 1: Transition diagram for states of an activity instance

3.2. Processes

A **process** is a special type of activity that establishes its own context of execution. As such, it can serve as a top-level definition, or as local definition inside a context. A process can be spawned and joined to allow for parallel execution and other complex flows. It can also be invoked from multiple activities and, as such, serves as a mechanism for reuse and composition.

3.2.1. Instantiation

A process can be instantiated from another process upon receipt of a **message**. This form of instantiation is used when the two processes are loosely coupled, can be defined independently of each other and can be distributed across heterogeneous systems.

For instantiation upon receipt of a message, the process definition must begin with an action or set of actions that respond to input messages. When using WSDL, these actions must perform either a *one-way* or a *request-response* operation.

If the process definition begins with a single action that responds to an input message, the process is instantiated upon receipt of that message.

If the process definition begins with an *all* activity that includes only such actions, the process is instantiated upon receipt of all such messages.

If the process definition begins with a *choice* activity that consists only of message event handlers, the process is instantiated upon receipt of any one message.

No other cases are supported for processes with instantiation type *message*.

A process can be instantiated by **other** means. For example, it can be spawned or called from another process. When used in this manner, the two processes are tightly coupled and cannot be distributed.

This form also allows the instantiation of the process in response to a system event, at a pre-defined schedule, or by other mechanisms not covered by the BPML specification.

The following table summarizes the relation between instantiation type, process definition, and triggering event.

Instantiation type	**First activity in process definition**	**Triggering event**
message	`action receiveMessage0`	Receipt of message0
message	`all`	Receipt of both message0 and message1
	`  action receiveMessage0`	
	`  action receiveMessage1`	
message	`choice`	Receipt of either message0 or message1
	`  onMessage`	
	`    action receiveMessage0`	
	`    . . .`	
	`  onMessage`	
	`    action receiveMessage1`	
	`    . . .`	
other	*`Any activity`*	*spawn*, *call*, other

When instantiating a process with a message, information required to instantiate the process is passed in the message. When instantiating a process by spawning or calling it, information required to instantiate the process is passed through parameters.

A process with instantiation type *other* can define input and output **parameters**. Parameters are always typed.

Required input parameters must be supplied by the spawning or calling process. A default value can be specified for optional parameters, and will be used if no value is supplied upon instantiation.

Values passed for input parameters are assigned to properties in the process context that have the same name as the input parameters.

When calling a process, output parameters can be passed from the called process back to the calling process. Values returned as output parameters are assigned to properties in the calling context with the same name as the output parameters.

3.2.2. Top-Level and Nested

A **top-level** process definition is performed in a context that is independent of any other process definition. The process is always available for instantiation.

A top-level process can be instantiated from other top-level processes defined in the same package, and from those defined in a different package, if the process definition scope is public.

A **nested process** is a process definition that is localized to a given context. Nested processes are used to constrain the availability of a process to a particular context, to re-define the behavior of a process within a specific context, or to enable property sharing between the process and other activities occurring in that context.

A nested process is instantiated in the context in which it is defined, and is available for instantiation only when that context instance exists.

A nested process can only be spawned or called by activities occurring within the same context or any of its child contexts. Its definition is always in a private scope.

A nested process can be instantiated upon receipt of a message only if the message is correlated to an instance of the context in which it is defined.

A nested process shares its context with other activities and nested processes defined in the same context and may access properties in that shared context. It may require local property definitions to isolate its properties from other activities and nested processes (see local definitions).

The lifetime of the nested process is generally independent from the lifetime of the parent process, or more specifically from the lifetime of the activity set of the context in which the nested process is defined.

However, there are means to establish a tight coupling between the lifetime of parent and nested processes, in particular using the *call* and *join* activities, using transactions, and using connectors to synchronize the processes through message exchange.

3.2.3. Process Definition

A process definition consists of the following attributes:

Attribute	**Description**
name	The process name.
instantiation	Either *message* or *other*.
scope	Either *public* or *private*.
parameters	Zero or more input and output parameters.
activity set	An activity set.

The **name** attribute provides a name that can be used to distinguish the process from all other process definitions. Unlike activities, a process definition requires this attribute to have a value.

A top-level process name must be different from any other process defined in the same package. A nested process name must be different from any other nested process defined in the same context, but may override a name defined in a parent context.

The **instantiation** attribute indicates whether the process is instantiated by a message or by other means. If the instantiation type is *message*, the process must begin with an activity that responds to an input message (or messages).

The **scope** attribute indicates whether this process definition can be called or spawned from processes defined in other packages. If the scope type is *private*, the process definition cannot be referenced from another package.

The scope type can be *public* only for top-level process definitions with instantiation type *other*.

The **parameters** collection consists of all input and output parameters of the process. Input parameters are passed to the process when the process is instantiated. Output parameters are passed back from the process upon completion of a call.

No properties can be specified for a process with instantiation type *message*.

The process definition consists of one top level **activity set**. The activities in the activity set execute in sequential order exactly once for each instance of the process.

The context defined for this activity set is also known as the **process context**. The process context is instantiated upon instantiation of the process. There is a one to one association between a process instance and the process context instance. While the process instance may be known outside the context of the process, the process context is accessible only to activities executing within the process.

The syntax for a process definition is given as:

```
<process
  name = NCName
  instantiation = (message | other) : message
  scope = (public | private) : private>
  Content: (documentation?, implements*, parameters*,
            context?, {any activity}+)
</process>
```

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

It is an error to use the *scope* attribute when the *instantiation* attribute is *message*, or the process definition is nested.

It is an error to use the *parameter* element when the *instantiation* attribute is *message*.

The syntax for an implementation construct is given as:

```
<implements
  interface = QName
  process = NCName
  {extension attribute}/>
```

Editor's note: This element will be defined in a future version of this specification.

3.2.4. Input/Output Parameter

A parameter construct consists of the following attribute:

Attribute	**Description**
name	The parameter name.
type	The parameter type.
direction	Input, output or input/output.
use	Either *required* or *optional*.
value	The default value for the parameter.

The **name** attribute provides a name that can be used to distinguish this parameter from all other parameters and properties defined in the process context.

Upon instantiation of the process, each parameter value (whether passed to the process or derived from the default value) will be used to instantiate a property with the same name in the process context.

A parameter definition is implicitly a local property definition in the process context. It is an error to use a local property definition in the process context with the same name as a parameter.

Strict typing allows any other process to determine the proper structure of the inputs and outputs that are supported by this process.

The **direction** attribute indicates whether the parameter is input, output or both input and output.

The **use** attribute indicates whether the parameter is required or optional. If the parameter is required, a value must be supplied when instantiating the process. The *use* attribute is applicable only to parameters of type input and input/output.

A default **value** can be specified only for input parameters with use type *optional*. The default value is used if no value is supplied when instantiating the process. If no default value is specified, the default value is empty or a default value that applies to the property type.

The syntax for a process parameter definition is given as:

```
<parameter
  name = QName
  type = QName
  element = QName
  use = (required | optional) : required
  input = boolean : true
  output = boolean : false
  {extension attribute}>
  Content: (documentation?, value?)
</parameter>
```

The parameter type is specified by referencing a type definition from some type system. BPML defines two such type referencing attributes for use with the XSDL type system:

- **element** – Refers to an XSDL element declaration using its QName
- **type** – Refers to an XSDL simple or complex type definition using its QName

Other type definitions can be referenced using extension attributes defined in a namespace different from that of BPML.

It is an error to use both *type* and *element* attributes, or use either attribute in combination with an extension attribute. If neither attribute is used, and no extension attribute is used, the parameter type is the XSDL type *anyType*.

It is an error for both *input* and *output* attributes to have the value *false*.

It is an error to use the *use* attribute or *value* element when the *input* attribute is *false*.

3.2.5. Process Instance

Each process has a unique instance identifier. The instance identifier is held in a property that has the same name as the process and is assigned in the process context (see instance properties). This allows it to be referenced by activities executing within the process.

For a process that is instantiated by spawning or calling, an instance list property is assigned in the current process of the *spawn* or *call* activity.

For a nested process with instantiation type *message*, an instance list property is assigned in a context of the parent process.

Beginning with the context in which the nested process is defined, that context is selected if a local property definition with the same name exists, and if no such definition exists and a parent context exists, this is repeated recursively up to top most context in that process (the process context).

The instance list property has the same name as the process but holds a list of all instances of that process instantiated in that context. As more instances are created the instance list property is modified by adding the new instance identifiers. Instance identifiers are never removed.

The instance list property can be used to join an instantiated process using the *join* activity.

Since the *call* activity is atomic, the instance list property is only updated upon completion of the called process; hence it has no affect on the execution of the *join* activity.

A process instance transitions through the following states:

- **Active** – The process exists and is performing all activities in the top most activity set
- **Suspended** – The process is active but is not performing any atomic activities until it is resumed.
- **Completing** – The process has performed all activities in the top most activity set, and is now preparing to complete. This may involve additional work, such as transaction commit or context completion.
- **Complete** – The process has performed all work required in order to complete successfully.
- **Aborting** – The process has failed to complete successfully, and is now preparing to abort. This may involve additional work, such as transaction rollback or context completion.
- **Aborted** – The process has failed to complete successfully and has performed all other work required in order to abort.

A process instance always begins in the *active* state. Once the process has completed successfully it transitions to the *complete* state. This is a terminal state. The process transitions to the *complete* state through the *completing* state.

A process will transition to the *aborted* state if it cannot complete successfully, in particular due to a fault that is not caught by any exception event handler. This is a terminal state. The process transitions to the *aborted* state through the *aborting* state. The process may also transition to the aborting state from the *completing* state.

A process in the *active* state can transition to the *suspended* state and back to the *active* state any number of times. When requested to transition to the *suspended* state, the process will not begin any new atomic activity or respond to events, and may cancel atomic activities if these activities can be retried successfully upon resuming (e.g. the *delay* activity).

Since a process in the *suspended* state is not executing any atomic activities or responding to events, it is also not performing any complex activity. However, complex activities that are in the *active* state will remain in that state.

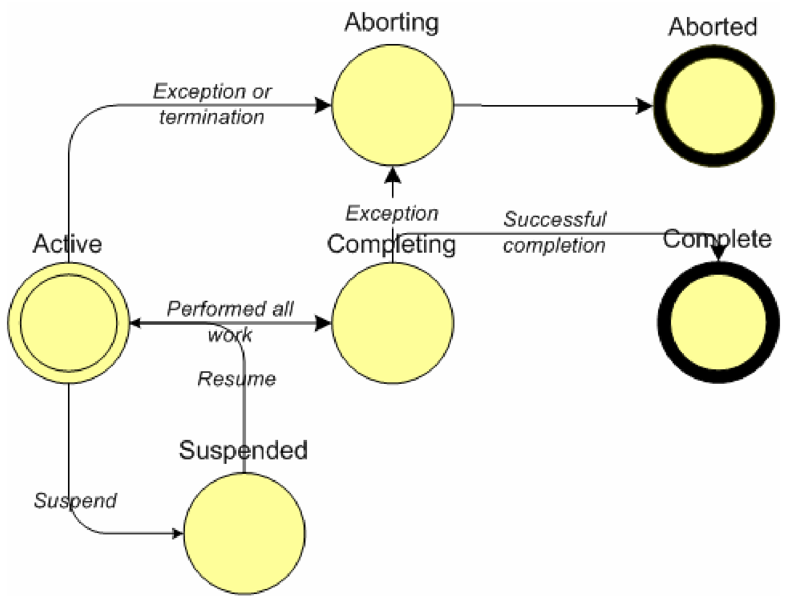

Figure 2: Transition diagram for states of a process instance

4. Contexts and Properties

4.1. Contexts

A context is used to establish an environment in which activities can execute. Processes and complex activities demarcate the context in which they execute other activities. Context instances form a distinction between multiple instances of the same activity or process.

Just like activities, contexts are composed hierarchically. An activity is performed in the current context, which is defined by the activity set in which it is contained. The activity set may be part of a complex activity or another construct with its own current context, which serves as the parent context for the activity set's context.

The current context inherits from the parent context and by recursion all its parent contexts up to top-most context (the process context) of a top-level process.

4.1.1. Local Definitions

The current context consists of the parent context and any **local definitions** made in that context. A local definition is a definition made in the current context, and is not available to any parent or sibling context. It will override a definition with the same name made in a parent context.

BPML defines five types of local definitions that may appear in a context:

- **Property** – Defining a property as local to a context assures that any changes made to that property's value are not visible in a parent or sibling context and do not affect a property with the same name that is used in a parent or sibling context.
- **Exception** – Defining exception handling as local to a context assures that the definition will not affect exception handling in a parent or sibling context and may override the behavior defined by a parent context for overlapping event.
- **Process** – Defining a process definition as local to a context assures that the process definition is not available for instantiation in any parent or sibling context. A local process definition will override any process definition with the same name appearing in any parent context. A local process definition is also known as a nested process.
- **Transaction** – Defining a transaction as local to a context ensures that only activities occurring in that context are performed as part of that transaction, and that the behavior of all activities performed in that context is transactional.
- **Connector** – Defining a connector as local to a context allows activities executing within that context to exchange messages with each other without exposing those messages outside this context.

The context of an activity set is instantiated before executing the first activity in the activity set. The context is instantiated only if it is determined that the activities in the activity set will be performed. For example, the process context is instantiated when instantiating a new process instance.

The context can be discarded only after performing the last activity in the activity set and all other work that requires the context instance to exist, such as, committing or aborting a transaction, context completing, event handlers, nested process instances, transaction compensation, etc.

The context in which a nested process is defined serves as the parent context for the process context of the nested process. Therefore a nested process can only be instantiated in the context in which it is defined. The context of a top-level process shares nothing with any other context.

Activity sets are used in places other than processes and complex activities, such as with exception event handlers and transaction compensation. In these cases, the behavior would follow the same guidelines.

4.1.2. Context

A context definition consists of the following attributes:

Attribute	**Description**
connectors	Zero or more connectors.
processes	Zero or more process definitions.
properties	Zero or more property definitions.
exception	Zero or more event handlers.
transaction	Transaction definition.
completion	Context completion.
others	Other local definitions of types defined by other specifications.

The **connectors** collection specifies one or more connectors that are applied only within this context. Operations performed within the same context may exchange messages using connectors defined in that context. Connectors are often used in this manner for inter-process communication.

The **processes** collection specifies one or more process definitions that are available for instantiation in this context. Such process definitions are known as nested processes and are not accessible from any parent or sibling context. A nested process can only be instantiated in an instance of the context in which it is defined.

The **properties** collection specifies one or more local property definitions. A local property definition distinguishes it from any property with the same name in a parent of sibling context. Changes made to the value of the property in this context are not visible in any parent or sibling context.

The **exception** collection specifies one or more event handlers that are specific to this context and may override overlapping event handlers in a parent context and add event handlers that do not exist in a parent context.

The **transaction** attribute specifies the transaction name and type. All activities performed in this context, including activities in nested processes, will be performed as part of that transaction. If no transaction is specified, the context is not transactional.

The **completion** attribute specifies an activity set that is executed upon completion of all activities in the activity set, and before the context is discarded. This activity set is also known as **context completion**.

The **others** collection specifies one or more local definitions of types that are defined by other specifications. These definitions are not a normative part of the BPML specification. A BPML implementation must be able to either process such definitions in conformance with the specification in which they are defined, or reject a BPML process definition that uses such local definitions.

The names of activities, nested processes and transactions that are defined in a context are used to name instance properties in that context. We recommend that distinct names be used such that the proper instance property can be accessed based on its name.

There are cases where a local property definition would have the same name as that of an activity, transaction or nested process appearing in that context. This practice is used to localize the instance property (or instance list property) to that particular context, such that it is not accessible in any parent or sibling context.

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

```
<context>
  Content: ((connect | process | property | {extension element})*,
            exception?, transaction?, completion?)
</context>
```

If the *context* element is absent or empty, is it treated as though the context definition has no local definitions of its own (an **empty context**).

The *context* element is an extensible element. Extension elements are used for local definition types that are covered by other specifications. Extension elements must use a namespace different from that of BPML.

Context completion is an activity set that is executed before the context is discarded. It is executed after successful completion of all activities in the activity set, after execution of any exception event handler and after completing or aborting the transaction associated with the context.

Context completion is particularly useful as a means to perform cleanup when an exception is handled by a parent context, of when cleanup is required after completion of nested process instances.

If context completion occurs in a context that specifies a transaction, the transaction state will be either *completing* or *aborting*. If context completion occurs in a context established by a complex activity or process, the activity or process state will be either *completing* or *aborting*.

The syntax for context completion is given as:

```
<completion>
  Content: (documentation?, {any activity}+)
</completion>
```

4.2. Properties

A **property definition** associates a type and initial value with all instances of the named property. A **property instance** is a named value that is specific to a given context instance.

A property instance that is derived from a property definition can only hold values of the specified type. Expressions that use such properties are subject to type checking, both static and dynamic (see XPath 2.0).

A **local property definition** is used to constrain a property to a given context. The property instance is not accessible from any parent or sibling context.

Every context instance requires a property instance for every local property definition made in its context definition. Depending on the property definition, it is either instantiated with the initial value provided in its definition, or with a value derived from properties in the parent context.

It is possible that another property with the same name and the same or different type will exist in a parent or sibling context, however, these are distinct property instances, and changing the value of one does not affect the other.

If no local property definition exists in the context, any access to the property will be made in the parent context and recursively up to the top-most context (the process context) or the top-level process.

A property that is not explicitly defined in a context is always implicitly defined in the process context of the process in which it is assigned, with the XSDL type *anyType*. No type checking is performed on properties of that type.

When a property definition appears at the package level, its initial value must be specified. The property definition is then inherited by all top-level process definitions in that package. The behavior is as if the property definition appeared in the process context of each top-level process definition.

A property name may be associated to more than one process; as such BPML allows process definitions to use properties from multiple namespaces, and for the same property name to be referenced in multiple process definitions.

4.2.1. Property Definition

A property definition consists of the following attributes:

Attribute	**Description**
name	The property name.
type	The property type.
value	The initial value.
expression	An expression for deriving the value.

The **name** attribute provides the property name.

The **value** attribute provides the initial value for the property. If specified, the property is instantiated with that initial value. This attribute must be used for property definitions at the package level.

The **expression** attribute provides a means for deriving the property value from the value of properties in the parent context. It is an expression that can refer to any (one or more) properties in the parent context. When used in the process context of a top-level process, it can refer only to properties defined at the package level.

The *selector* and *value* attributes are mutually exclusive.

The syntax for a property definition is given as:

```
<property
  name = QName
  type = QName
  element = QName
  xpath = XPath>
  Content: (documentation?, ({extension element} | value)?)
</property>
```

The property type is specified by referencing a type definition from some type system. BPML defines two such type referencing attributes for use with the XSDL type system:

- **element** – Refers to an XSDL element declaration using its QName
- **type** – Refers to an XSDL simple or complex type definition using its QName

Other type definitions can be referenced using extension attributes defined in a namespace different from that of BPML.

It is an error to use both *type* and *element* attributes, or use either attribute in combination with an extension attribute. If neither attribute is used, and no extension attribute is used, the property type is the XSDL type *anyType*.

The *xpath* attribute is an XPath expression. The expression is evaluated against the parent context to arrive at the value of the property.

The extension element allows other expression languages to be used, as long as they are defined in a namespace other than BPML.

The *xpath* attribute, *value* element and extension element are mutually exclusive. It is an error to use any two in the same property definition.

If the *xpath* attribute, *value* element and extension element are all missing, the default behavior is to inherit the property's value from a property with the same name in the parent context.

Only the *value* element may be used when a property definition appears at the package level.

Example illustrating a package level property definition for the "tns:someDate" property:

```
<property name="tns:someDate">
  <value>2001-01-29</value>
</property>
```

Example illustrating a local property definition for a counter:

```
<sequence>
  <context>
    <property name="tns:counter">
      <value>1</value>
    </property>
  </context>
  . . .
</sequence>
```

4.2.2. Assignments

Assignment is a change to a property's value that occurs as the direct or in-direct result of executing an activity. An assignment is always made in the activity's current context.

If a property with that name already exists in the current context, its value is changed to the new value. Otherwise, the property is instantiated in the current context with the new value.

Atomic activities always perform assignment in an atomic (all or nothing) manner.

The assign activity provides an explicit form of assignment by changing the value of a property to a fixed value or to a value derived from the value of other properties. This activity is used when a change to the property's value is intended to affect future activities.

The action activity performs an indirect assignment as the result of any operation that receives an input message. By default the entire message contents are assigned to a property with the name as given in the message definition.

Selectors perform explicit assignments as the result of an operation that receives an input message. The selector assigns part of the message to a property with the name specified by the selector.

When using WSDL, the message name is given in its definition and may be different from the name of the operation being performed.

The following example illustrates a WSDL message definition, an operation that uses that message as input, and a BPML action that perform that operation. Upon completion of the action, a property by the name *tns:myMessage* will be set to the contents of the input message.

```
<wsdl:message name=”myMessage”>
  <wsdl:part name=”contents” type=”tns:messageContentsType”/>
</wsdl:message>
<wsdl:portType name=”myPort”>
  <wsdl:operation name=”myOperation”>
    <wsdl:input message=”tns:myMessage”/>
  </wsdl:operation>
</wsdl:portType>
. . .
<bpml:action portType=”tns:myPort” operation=”myOperation”/>
```

4.2.3. Selectors

Selectors extract a specific value from the contents of an input message and assign it to a property.

A selector applies to a message definition if it references a part of the message directly or indirectly. A selector may reference the message part directly by its name and indirectly by its type. A selector will apply to all message definitions that contain a message part of the specified type.

If a selector applies to a message definition, then it applies to all operations that receive that message type as their input. For WSDL, the selector will only apply to the input message of *one-way*, *request-response* and *solicit-response* operations, and to fault messages return by the *solicit-response* operation.

If a selector applies to an operation, it is used to derive the value of the named property from the input message whenever that operation is performed. The property can be used as part of the process of receiving the message, in particular for the purpose of correlating the message to the proper context instance.

The *action* activity will assign all properties extracted by selectors as a result of receiving a message. The properties will be assigned in the current context and will have the same names as the properties extracted by the selectors. All properties will be assigned in the current context in an atomic (all or nothing) manner.

In addition, the *action* activity will assign the entire message contents to a property with the same name as the message type.

A selector definition consists of the following attributes:

Attribute	**Description**
name	The name of the property being instantiated.
part	Message part name.
type	Message part type.
expression	An expression for deriving the value.

The **name** attribute provides the name of the property extracted by this selector.

The **part** attribute provides a means to unambiguously identify the message part to which the selector applies based on its name.

The **type** attribute provides a means to unambiguously identify the message part to which the selector applies based on its type. This attribute must reference the same type definition that is used by the message part definition.

The selector must use either part or type identification but cannot use both at the same time.

The **expression** attribute provides an expression that can derive the value of the property from the message part. If absent, the entire message part contents is used.

The definition of selectors is covered by the WSCI specifications. Selectors are defined in a WSCI document and imported into a BPML package.

WSCI definition for a selector that extracts the *tns:poNumber* property from the messages *tns:poMessage* and *tns:poCancelMessage*, and a selector that calculates the *tns:orderTotal* property from the *tns:poMessage* message:

```
<wsci:selector property="tns:poNumber" type="tns:poNumberType"/>

<wsci:selector property="tns:orderTotal" element="tns:lineItems"
               select="sum(lineItems/lineItem/(quantity * price))"/>
```

WSDL definition for the *tns:poMessage* and *tns:poCancelMessage* messages:

```
<wsdl:message name="poMessage">
  <wsdl:part name="poNumber" type="tns:poNumberType"/>
  <wsdl:part name="lineItems" element="tns:lineItems"/>
</wsdl:message>

<wsdl:message name="poCancelMessage">
  <wsdl:part name="poNumber" type="tns:poNumberType"/>
</wsdl:message>
```

XSDL components for the *tns:poNumberType* simple type and *tns:lineItems* element:

```
<xsd:simpleType name="poNumberType" base="xsd:string"/>

<xsd:element name="lineItems">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="lineItem"
                   minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
             <xsd:element name="sku" type="xsd:string"
                          minOccurs="1"/>
             <xsd:element name="quantity" type="xsd:integer"
                          minOccurs="1"/>
             <xsd:element name="price"
                          type="xsd:float" minOccurs="1"/>
          </xsd:sequence>
        <xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

4.3. Expressions

Expressions are used to provide values that cannot be determined statically and must be evaluated at runtime based on the values of properties that are specific to a context instance. Expressions are used for assignments, constructing messages and parameters, conditions, etc.

Some expressions take the form of an XPath expression. Extension elements can be used to support other expression languages, such as XQuery or externally defined business rules. In some cases expressions take the simple form of naming a single property.

An expression is evaluated in the current context of an activity or other construct in which it appears. For example, an expression for constructing an output message would be evaluated in the context of the *action* activity that sends the message. An expression for a timeout event handler would be evaluated in the context in which the event handler is defined.

The context for evaluating an expression is identical, regardless of the language in which the expression is written. However, languages other than XPath and XQuery may add additional capabilities to the evaluation context.

The expression can access all properties that are available in that context, whether these properties are explicitly defined or implicitly available through prior assignment. An expression is invalid if it attempts to use a property that was not explicitly defined, or which is not known to be available in that context.

A property is implicitly available in a context if it can be determined that it might have been assigned by a preceding activity. The property is un-typed. If no value was assigned to the property, the property would have the empty value.

For XPath expressions, the value of a property is referenced as a variable using the syntax of the form `$QName`, where QName is the fully qualified name of the property. The value of a property is one of the allowed XPath types.

Lists are represented using a set of elements. For example, the first and last instance identifiers in an instance list property `ns1:proc1` could be accessed with the XPath expression `$ns1:proc1/[1]` and `$ns1:proc1/[last()]`.

For the purpose of evaluation the context node (as defined by XPath) is always a text node with character data of zero length.

All expressions are evaluated in an atomic fashion. The values of all properties used in the expression must be established at the same point in time, such that no change to values is visible while performing the expression. This must be consistent with other forms of indirect access to properties that are performed by the activity not necessarily as part of the expression.

A reusable process that adds the value of X and Y and returns the sum multiplied by two. The values X and Y are passed as input parameters, the result is returned as the output parameter Y. Parameters definitions result in explicitly defined properties with the same name.

This example shows both the long form to perform such a calculation, using a temporary property (implicitly available), and the short form to perform such a calculation in a single assignment.

```
<process name=”addXandYmultiplyByTwo”
         instantiation=”other” scope=”public”>
  <parameter name=”tns:X” type=”xsd:integer” input=”true”/>
  <parameter name=”tns:Y” type=”xsd:integer” input=”true”/>
  <parameter name=”tns:Z” type=”xsd:integer” input=”false”
                                             output=”true”/>

  <!--  Long form -->
  <assign property=”tns:tmp” xpath=”$tns:X + $tns:Y”/>
  <assign property=”tns:tmp” xpath=”$tns:tmp * 2”/>
  <assign property=”tns:Z” xpath=”$tns:tmp”/>

  <!--  Short form -->
  <assign property=”tns:Z” xpath=”($tns:X + $tns:Y) * 2”/>
</process>
```

4.4. Functions

xsd:dateTime bpml:currentTime()

This function returns the current time.

The return value is of type *xsd:dateTime*.

Multiple calls to this function within the same expression or atomic activity are guaranteed to return the same time instant.

4.5. Instance Properties

An **instance property** provides the instance identifier of the activity, transaction or process that it represents. The instance property is implicitly defined and has the same name as the activity, transaction or process. It takes the form of an element called *bpml:instance* with the value of the instance identifier as its contents.

The activity, transaction or process instance is unambiguously identified through an instance identifier that must be unique across time and space. The use of UUID identifiers is recommended (see UUID and GUID).

Like all other properties, instance properties are assigned in the current context. A local property definition can be used to differentiate that instance property from any property with the same name in a parent or sibling context.

Instance properties for activities are instantiated in the current context of the activity. Instance properties for processes are instantiated in the process context. In addition, instance list properties are managed for nested processes and processes instantiated using the *spawn* and *call* activities (see process instance).

The instance property for a transaction is instantiated in the context in which the transaction is defined. In addition, an instance list property is managed in a parent context.

Changes to an instance property occurring within an atomic transaction or atomic activity are made visible only when the transaction or activity complete or abort.

The syntax for an instance property is given as:

```
<instance>
  Content: {instance identifier}
<instance/>
```

4.6. Instance Functions

BPML defines several XPath functions that take an instance identifier as input and provide information about that activity, transaction or process instance. These functions can be used to check the status of an instance, determine the duration of execution, etc.

All these functions operate on a single instance identifier that must be associated with an instance existing in the same context in which the function is evaluated. If a list of instance identifiers is passed as input, these functions will operate on the first instance identifier in that list.

bpml:getType(instance)

This function returns the instance type based on its instance identifier. The instance type returned is one of the following:

- **activity** – The instance identifier is of an activity
- **transaction** – The instance identifier is of a transaction
- **process** – The instance identifier is of a process

The instance types are enumerated in the simple type definition bpml*:instancePropertyType*.

If the instance is not recognized in the context in which this function is used, the function returns the empty value.

xsd:dateTime bpml:getStart(instance)

This function returns the start time of instance based on its instance identifier.

The return value is of type *xsd:dateTime*.

If the instance is not recognized in the context in which this function is used, the function returns the empty value.

xsd:dateTime bpml:getEnd(instance)

This function returns the end time of instance based on its instance identifier.

The return value is of type *xsd:dateTime*.

If the instance has not transitioned to the *complete* or *abort* state yet, or the instance is not recognized in the context in which this function is used, the function returns the empty value.

xsd:dateTime bpml:getDuration(instance)

This function returns the duration of the instance based on its instance identifier.

The return value is of type *xsd:duration*.

If the instance has not transitioned to the *complete* or *abort* state yet, the function returns the difference between the current time (as returned by *bpml:currentTime*) and the start time (as returned by *bpml:getStart*).

If the instance is not recognized in the context in which this function is used, the function returns the empty value.

xsd:instanceStateType bpml:getState(instance)

This function returns the instance state based on its instance identifier. The instance states returned is one of the following:

- **active**
- **completing**
- **complete**
- **aborting**
- **aborted**
- **compensating**
- **compensated**
- **suspended**

The instance types are enumerated in the simple type definition bpml*:instanceStateType*.

If the instance is not recognized in the context in which this function is used, the function returns the empty value.

xsd:boolean bpml:isActive(instance)

This function returns true if it can determine that the instance is active based on its instance identifier.

The return value is of type *xsd:boolean*.

The instance is active if the function *bpml:getState* returns *active* or *suspended*.

Otherwise, the function returns false.

xsd:boolean bpml:isComplete(instance)

This function returns true if it can determine that the instance has completed based on its instance identifier.

The return value is of type *xsd:boolean*.

The instance has completed if the function *bpml:getState* returns *complete*, *aborted*, *compensating* or *compensated*.

Otherwise, the function returns false.

xsd:boolean bpml:isFailed(instance)

This function returns true if it can determine that the instance has failed based on its instance identifier.

The return value is of type *xsd:boolean*.

The instance has failed if the function *bpml:getState* returns *aborting* or *aborted*.

Otherwise, the function returns false.

xsd:string bpml:getFault(instance)

This function returns the fault associated with an instance based on its instance identifier.

The return value is of type *xsd:string*.

A fault is associated with an instance if the instance has aborted due to a fault.

If the instance is not recognized in the context in which this function is used, or is not associated with any fault, the function returns the empty value.

This example sends a report message with the identifier, status and duration of a transaction:

```
<action portType=”tns:reportPort” operation=”sendTxReport”>
  <output part=”identifier” xpath=”$tns:myTx”/>
  <output part=”status” xpath=”bpml:getStatus($tns:myTx)”/>
  <output part=”duration” xpath=”bpm:getDuration($tns:myTx)”/>
</action>
```

This example illustrates a condition based on the successful or unsuccessful completion of a transaction. It is assumed that this activity will occur whether the transaction has completed or aborted:

```
<choice>
  <case>
    <condition>bpml:isComplete( $tns:myTx )</condition>
    . . .
  </case>
  <default>
    . . .
  </default>
</choice>
```

5. Exceptions and Transactions

Exception handling defines how the process deals with unexpected occurrences. Transactions make it possible to treat multiple activities as a single unit of work, providing a guarantee of consistency and reliability.

Together, exception handling and transactions allow a process to react to unexpected conditions, attempt to recover and proceed past a point of failure and, if necessary, revert to a previous state.

5.1. Exception Handling

Exception handling defines activities that deal with unexpected occurrences and the events that signify such occurrences.

An **event handler** defines the triggering event and the activity set the process will perform when that event occurs. BPML requires a distinct event handler for each exception that could occur in a particular context.

There are three types of events and corresponding event handlers:

- **Message** – An input message is received by the process. The event handler identifies the input message by means of an *action* activity. The event is triggered when the process receives the input message on which it is able to perform the action.
- **Time-out** – A time-out occurs. The event handler identifies the time-out by specifying the time instant directly or as a time duration that is relative to a reference time. The event is triggered at the specified time instant.
- **Fault** – A fault occurs. The event handler identifies the fault(s) to which it responds. The event is triggered when a specified fault occurs.

Event handlers are associated with a context by aggregating them within the *exception* element.

```
<exception>
  Content: (onMessage | onTimeout | onFault)+
</exception>
```

5.1.1. Behavior

An event is always triggered within a context; the context instance is required for the event handler to respond to an event. Exception event handlers are defined as part of the context definition to which they apply.

The event handler becomes operational when the context is instantiated. It responds to events occurring while activities in the activity set are executed, and ceases to take effect once activities in the activity set have completed executing or were terminated. The event handler is not operative during execution of the context completion activity set.

An event handler is in effect for all child contexts as well. If the child context declares an event handler for an overlapping event, the child context’s event handler takes precedence over the event handler specified in the parent context.

Two events handlers are overlapping if triggering one also triggers the other: *message* event handlers are overlapping if they respond to the same input message; *time-out* event handlers overlap if they occur at the same time instant; f*ault* event handlers overlap if they respond to the same fault code.

It is an error if an exception event handler responds to the same input message or fault as another activity in the same context, in particular, the *action* and *choice* activities.

Only one time-out event handler can be specified in an exception handling of a given context, but a time-out event handler is allowed for both exception handling and *choice* activity defined in the same context.

This behavior is not extended to nested processes. An event handler that is specified for a context will not affect any nested processes instantiated in that context. Similarly, the event handler will not respond to any events that target an instance of the nested process.

The context in which the event handler is defined serves as the parent context for the activity set specified in the event handler and is not discarded until the event handler completes. Context completion occurs after executing activities in the event handler's activity set.

Once one event handler is triggered, all other event handlers specified in that context are inoperative. If an event handler is triggered for a parent context, all event handlers specified in child contexts become inoperative. In addition, there will be no instantiation of any nested process defined in these contexts.

Event handlers specified in a parent context are still operative and can event respond to events occurring while executing activities in the event handler of a child context.

When an event handler is triggered, it terminates execution of all activities in that context's activity set, any child context and any called process, excluding the contexts of nested processes (unless called). No new activities are executed in these contexts. The event handler itself is put on hold until all activities terminate.

Atomic activities are terminated by either completion or cancellation. An atomic activity can be canceled if there is no side effect resulting from such cancellation, or if it is possible to reverse all such side effects. For example, the *delay* activity can be cancelled at any time. Complex activities are terminated by first terminating all in-progress activities in their activity set.

Once the event handler completes successfully, the parent activity, process or construct is allowed to complete successfully. The event handler may fail to complete successfully if either an event handler in a parent context is triggered or it causes the propagation of a fault to the parent context.

If an event handler is triggered for a parent context, the child context will not be able to complete the activity set or respond with a specific event handler. To perform cleanup and any other work, the child context can use context completion to specify an activity set that is always executed upon completion of the activity set.

In order to provide a precise behavior, event handlers defined in the same context must not specify overlapping events. A BPML implementation can warn about overlapping events such as event handlers that respond to the same message or fault code.

5.1.2. Faults

Faults can occur when performing activities. Common cases where faults occur include:

- An action performs an operation that fails to complete, resulting in a fault. When using WSDL, faults can occur when performing a *solicit-respond* operation, as specified by the operation definition.
- An activity results in a fault. Specifically, the *call* activity faults if the process it calls faults.
- The *fault* activity can cause a specific fault to occur, as would be the case if it were necessary to abort a transaction, activity or process.
- A fault occurs while attempting to complete an atomic transaction. This type of fault could occur as part of the two-phase commit protocol.

If the fault is not handled by an event handler specified in that context, the fault is propagated to the parent context, and recursively up to the top-most context of the process. If no event handler responds to the fault, the process terminates with a fault.

A fault occurring in a nested process is not automatically propagated to the parent context. However, a fault in one process can affect another process, for example, when the process is instantiated by the *call* activity, or the two processes execute in the same transaction and the fault causes the transaction to abort.

An event handler can cause a fault to occur. In this case, the event handler is treated like any other activity set and, if no event handler is specified, the fault is propagated to the parent context.

When using WSDL, the fault could occur while processing an input message as part of a *request-response* operation. In this case, a fault in the called process is propagated to the *action* activity, causing it to return a fault to the sender.

5.1.3. Message Event Handler

A message event handler is a composition of the following attributes:

Attribute	**Description**
action	Action receiving an input message.
activity set	An activity set.

This event handler identifies the input message by means of the *action* activity. That activity is not part of the event handler's activity set, rather, it always executes in the context in which the event handler is defined.

The event is triggered when it is able to perform the action by receiving the input message. If the action is not able to complete successfully and cannot be canceled, the event handler propagates a fault to the parent context.

When using WSDL, the action must perform either a *one-way* or *request-response* operation. The action is allowed to call another process when performing a *request-response* operation. In this case, the action must be executed if the process is called.

Two event handlers overlap if they are triggered by the same input message. When using WSDL, two event handlers overlap if they use the same operation, but not if they use two different operations, even if both receive an input message of the same type.

The context instance is identified by correlating the input message.

The syntax for an *onMessage* event handler is given as:

```
<onMessage>
  Content: (documentation?, action, context?, {any activity}+)
</onMessage>
```

5.1.4. Time-out Event Handler

A time-out event handler is a composition of the following attributes:

Attribute	**Description**
dateTime	Name of property providing the time-out time instant.
duration	Name of property providing the time-out duration.
reference	Name of property providing the instance identifier.
start/end	Indicates whether reference time is start time or end time of instance.
activity set	An activity set.

The event handler identifies the time instant at which the event is triggered using the following attributes:

- **dateTime** – The name of a property. The property's value provides the specific time instant at which the time-out event will be triggered.
- **duration** – The name of a property. The property's value provides a time duration. The specific time instant is determined by adding the time duration to a reference time.
- **reference** – The name of a property. Used in combination with the *duration* attribute to provide the reference time. The property's value provides the instance identifier of an activity, transaction or process that is accessible in that context. The reference time is either the start or end time of that activity, transaction or process.
- **start/end** – Used in combination with the *reference* attribute to indicate whether the reference time is the start time or end time of the activity, transaction or process being referenced.

The *dateTime* attribute cannot be used in combination with any other attribute. The *duration* attribute can be used alone, or in combination with the *reference* and *start/end* attributes.

The value of the *dateTime* and *duration* attributes are property names. The value of the named property is used as either the time-out time instant or duration. If either value cannot be used to establish the time instant, the event will not be triggered.

If the *duration* attribute is used alone, the reference time is derived from the start time of the context, the time at which it was instantiated. This will be identical to the time in which the parent complex activity or process was instantiated, the time at which the event was triggered, compensation requested, etc.

If the *reference* attribute is used along with the *duration* attribute, it provides the name of a property. The value of the named property must be an instance identifier. An activity, transaction or process instance is identified based on that instance identifier.

If the value of the named property is an instance list, only the first instance identifier is used. If the instance is not recognized in the context in which the event handler is used, the reference time is undetermined and the event will not be triggered.

If the *start/end* attribute is *start*, the start time of the activity, transaction or process instance is used; otherwise the end time of the instance is used. If the instance has not completed or aborted, the event will not be triggered.

The value of the time instant, duration and reference time are all obtained from the context in which the event handler is defined, when the context is instantiated with all properties that are locally defined in that context. The time instant at which the event is triggered is determined when the context is instantiated and cannot be changed afterwards.

Two time-out event handlers overlap if they are triggered at the same time instant. If the more specific event handler occurs at the same time or prior to an event handler in any parent context, it will take effect. It is therefore not possible for a context to extend a time-out specified by a parent context.

The syntax for an *onTimeout* event handler is given as:

```
<onTimeout
  property = QName
  type = (duration | dateTime) : duration
  reference = (start QName | end QName)>
  Content: (documentation?, context?, {any activity}+)
</onTimeout>
```

The **property** attribute is the name of a property. The **type** attribute specifies the property type:

- **dateTime** – The value of the named property is the time-out time instant. The property value must be of the XSDL type *dateTime* or convertible to that type.

- **duration** – The value of the named property is the time-out duration. The property value must be of the XSDL type *duration* or convertible to that type.

The **reference** attribute is used to specify a reference time. It can be used only with the type *duration*. The reference attribute consists of the word *start* or *end* followed by the name of a property. The property value must be an instance identifier or instance list.

5.1.5. Fault Event Handler

A fault event handler is a composition of the following attributes:

Property	**Description**
code	Fault code.
activity set	An activity set.

A fault event handler identifies the applicable faults by means of their code. If no code is specified, any fault will trigger the event handler.

A fault is always generated in a particular context, typically as a result of performing an activity or some other work in that context. A fault can be generated in a child context for which no applicable event handler is specified or when a fault is propagated from the event handler of a child context.

If two event handlers are specified in the same context, and a fault that can be handled by both is triggered, the more specific one will take effect. This allows an event handler with no fault code to respond to all faults not handled by more specific event handlers.

Two event handlers are overlapping if they respond to the same fault code and neither is more specific, e.g. if both event handlers respond to all faults.

The syntax for an *onFault* event handler is given as:

```
<onFault
  code = QName>
  Content: (documentation?, context?, {any activity}+)
</onFault>
```

The **code** attribute specifies the fault code. If this attribute is used, the event handler responds only to a fault with this code. If this attribute is absent, the event handler responds to all fault codes.

5.2. Transactions

Transactions allow multiple activities to be treated as a single unit of work, providing a guarantee of consistency and reliability. A transaction is associated with an activity set by defining the activity set context as transactional. BPML supports two transaction models: atomic and open nested.

5.2.1. Atomic Transactions

Atomic transactions ensure that all activities performed as part of the transaction behave as a single unit of work. If the transaction cannot complete successfully, it will rollback to the state before the beginning of the transaction.

Individual activities can be atomic. However, there is no guarantee that a group of atomic activities will all complete successfully or rollback. The atomic transaction gives an all-or-nothing guarantee to any collection of activities that are executed as part of the transaction.

In order to provide an all-or-nothing guarantee, an atomic transaction must exhibit the following characteristics:

- **Atomic** – Either all changes performed by the transaction take effect or none of them take effect.
- **Consistent** – In order to complete, the transaction must preserve data integrity. One example of preserving data integrity is to cause the transaction to abort if it cannot complete successfully. Otherwise, it violates an integrity constraint.
- **Isolation** – Changes performed by the transaction are not visible to other activities occurring outside the transaction until the transaction completes. Isolated transaction instances that execute concurrently behave as if they execute serially.
- **Durable** – All changes performed by the transaction are permanent and persist even if a system failure occurs.

These four attributes are collectively known as **ACID**. Atomic transactions achieve ACID behavior by allowing all participants in the transaction to vote whether the transaction can complete or abort.

By participating in the completion process, the participants guarantee that the transaction can either complete successfully or can be canceled and returned to the state that existed before the beginning of the transactions. The completion process is also called **commit**, since it renders the effects of the transaction permanent. The cancellation process is also known as a **rollback** or **back-out**.

Atomic transactions are useful for activity sets that manipulate data, transform data from one or more sources to one or more targets, or coordinate multiple participants. The all-or-nothing guarantee ensures that all data changes and messages exchanged in the context of the transaction retain their consistency, regardless of how many steps are required in order to complete the transaction.

Where one or more participants are involved in the transaction, this act of coordination requires the use of a protocol or service that supports two-phase commit. Protocols that support two-phase commit include BTP (OASIS), CORBA OTS (OMG), DTC (Microsoft) and X/Open XA (OpenGroup).

Atomic transactions require resource locking. Resource locking can be disruptive if locks are maintained for a long period of time, decrease the concurrency of the system (its ability to handle multiple transactions at the same time), and may even result in deadlocks. As such, atomic transactions are only recommended for short-lived transactions.

In addition, in order to retain the isolation attribute, atomic transactions do not allow for transaction interleaving that occurs in complex processes. Transaction interleaving and long-lived transactions demand that the isolation requirement be relaxed, using alternative models such as open transactions.

5.2.2. Open Transactions

Open transactions relax the isolation requirement of atomic transactions and allow arbitrary levels of nesting. In literature, these are often referred to as Open Nested Transactions (see Activity Service) and reflect the relaxation of isolation and the ability to nest transactions.

With open transactions, resources are acquired for short periods of time and then released, typically by using a combination of open and atomic transactions. As such, open transactions can be used for long-lived transactions that cannot complete in a short time span.

Open transactions allow activities to progress from one consistent state to another, making each change permanent and durable immediately upon completion of the activity. As a result, open transactions are more resilient to system failures and are useful in supporting long–lived transactions where the possibility of temporary failure is too high to tolerate an automatic rollback.

Without isolation, data changes and message exchanges that performed in the context of the transaction are visible to other activities, whether these activities occur in the same transaction, a different transaction or outside of any transaction context.

One benefit is that the open transaction model allows transactions that span difference contexts and time spans and do not for a parent-child relationship to interact with each other (also known as **transaction interleaving)**.

Open transactions require additional work in order to perform backward recovery, since the effects of the transaction cannot be reverted automatically. Often, this is accomplished by compensating for a child transaction in order to rollback its parent transaction.

5.2.3. Aborting and Recovering

Backward recovery guarantees that in the event of the transaction aborting, the process will return to the consistent state that existed prior to the beginning of the transaction.

Atomic transactions provide automatic backward recovery. If the transaction has to abort, all data changes and message exchanges made by the transaction are discarded. Such backward recovery relates only to activities that are performed within the context of the atomic transaction (including any child contexts).

Open transactions cannot automatically revert data changes and message exchanges, since their effect is made visible to other participants and transactions. The transaction must define the proper logic in order to perform backward recovery.

An open transaction may not be able to guarantee complete backward recovery, and may include activities with side effects that cannot be reverted. For example, an open transaction may perform backward recovery by returning goods to the supplier but will still incur shipping and restocking fees.

Forward recovery guarantees that in the event of system failure, the transaction state can be restored to a consistent state and its execution can continue reliably past the point of failure. Forward recovery only applies to open transactions. Atomic activities and atomic transactions will always perform backward recovery in the event of system failure.

Forward recovery is guaranteed only for those activities that execute within the context of an open transaction, including any child contexts. Atomic activities and transactions contained in an open transaction are guaranteed forward recovery upon successful completion, and will perform backward recovery in case of failure.

A transaction can be retried if it has been aborted and was able to perform backward recovery successfully. A transaction can be retried as many times as necessary, however the use of low retry counts is recommended to prevent infinite loops in the event of a failure that cannot be overcome.

5.2.4. Compensation

In order to perform backward recovery, an open transaction that is made of multiple activities and transactions must revert their effects explicitly. **Compensation** is the logic for reverting the effects of a completed activity or transaction.

The relation between recovery and compensation is as follows:

- Forward recovery occurs before the transaction completes in order for it to proceed towards completion.
- Backward recovery occurs while the transaction aborts (in lieu of successful completion) in order to cancel the effects of the transaction.
- Compensation occurs after the transaction completes in order to revert the effects of the completed transaction.
- During backward recovery, a parent transaction will compensate for the child transactions that it performed by using the *compensate* activity.
- A transaction can specify its compensation logic as part of its definition, if that logic depends on the activities that the transaction performs. This logic is specified as an activity set by the *compensation* construct.
- The logic is invoked when applicable using the *compensate* activity.

- The logic that compensates for the transaction after its completion is defined separately from the logic that performs backward recovery in order to abort the transaction.

Both atomic and open transactions can define compensation logic as part of their definition. Activities that follow the transaction, typically as part of a parent transaction, can ask the transaction to engage in compensation.

The separation between the compensation logic of a transaction and the activities that invoke the compensation as part of a larger context allows for different activities, performed in different contexts or flows, to compensate the same transaction using the same logic.

The *compensate* activity will compensate for a transaction instance only once regardless of how many times the *compensate* activity is invoked for that transaction.

5.2.5. Behavior

A transaction is associated with an activity set by associating a transaction attribute to the activity set's context. All activities executed in that context, including child contexts will be part of that transaction. We refer to such a context as the **transaction context**.

The transaction attribute identifies the transaction name, type (atomic or open), retry count and compensation logic.

A complex activity that contains a transactional context is not by itself part of that transaction. However, all the work performed by the complex activity (meaning the activity set) is transactional.

A **nested transaction** is a transaction that executes in the context of a larger transaction. It is defined by a context that defines its own transaction and has a parent context that defines the parent transaction.

The parent-child relationship of the contexts transfers to a parent-child relationship between the transactions. The parent-child relationship of transactions is also carried to contexts defined for nested processes.

Both atomic and open transactions can be nested in an open transaction. It is an error to nest any transaction within an atomic transaction; however specific rules apply for nested processes (see Transactional Processes).

A nested transaction can complete or abort independent of the parent transaction. If the nested transaction has completed and the parent transaction aborts, it is the responsibility of the parent transaction to compensate for the nested transaction.

A parent transaction can complete only after all nested transactions complete or abort. If nested transactions are still active, the parent transaction must hold until they either complete or abort.

A **transaction instance** is instantiated each time the process instantiates a transactional context, specifically for the purpose of performing activity sets in that context. Two instances of the same transaction can execute concurrently and will be treated as separate transactions.

A transaction instance is unambiguously identified through an instance identifier. Two processes that participate in the same atomic transaction will use the same instance identifier.

Aborting and Event Handling

An exceptional event triggered in the context in which a transaction is defined or a parent context will cause the transaction to abort. The transaction initiates recovery by catching the event using an exception event handler. The **transaction event handler** can perform any set of activities that is required to assure complete backward recovery of the transaction.

An exceptional event caught in a child context will not cause the transaction to abort. Such exception handlers are used to recover from temporal errors that do not affect the transaction's ability to complete.

An exceptional event caught in the same context that defines the transaction will cause the transaction to abort. The event handler allows the transaction to perform backward recovery.

If the exception event handler is able to execute successfully, the transaction is considered recovered. A transaction that has been recovered can be repeated up to the retry count specified for that transaction. Each reiteration of the transaction is considered a distinct instance with its own unique instance identifier.

An exceptional event handler caught in a parent context will cause the transaction to abort. The transaction is not considered recovered and will not be repeated. Context completion allows the aborting transaction to perform necessary cleanup.

If the transaction has aborted and recovered up to the maximum specified in the retry count, it cannot be repeated anymore. This cases a fault to occur in the transaction context with the fault code **bpml:abortFinalAttempt** (a subtype of **bpml:abort**). This fault informs the parent context that the transaction cannot be completed successfully.

Transaction Instances

Each transaction has a unique instance identifier. The instance identifier is held in a property that has the same name as the transaction and is assigned in the transaction context and a parent context (see instance properties for more details). This allows it to be referenced by activities executing after completion of the transaction, in particular in order to compensate for the transaction.

The instance property is assigned in the context in which the transaction is defined. For atomic transactions, all transaction states are visible from that context.

In addition an instance list property is assigned in a parent context to contain all instances of the transaction that were instantiated in a child context. The instance list property has the same name as the transaction. As more instances are created the instance list property is modified by adding the new instance identifiers. Instance identifiers are never removed.

Beginning with the parent context of the context in which the transaction is defined, that context is selected if a local property definition with the same name exists, and if no such definition exists and a parent context exists, this is repeated recursively up to the top most context in that process (the process context).

If the transaction context is the process context of a nested process, the instance list will be set in a context of the parent process.

Transaction States

A transaction instance transitions through the following states:

- **Active** – The transaction exists and activities are performed in that transaction's context.
- **Completing** – The transaction has performed all activities in the activity set, and is now preparing to complete. This may involve additional work, such as persisting any data changes, performing two-phase commit (atomic transactions only) and context completion.
- **Complete** – The transaction has performed all work required in order to complete successfully.
- **Aborting** – The transaction has failed to complete successfully, and is now preparing to abort. This may involve additional work, such as reverting data changes, communicating outcome to participants (atomic transactions only) and context completion.
- **Aborted** – The process has failed to complete successfully and has performed all other work required in order to abort.
- **Compensating** – The transaction is now executing activities from the compensation activity set.
- **Compensated** – The transaction has completed executing activities from the compensation activity set.

A transaction instance always begins in the *active* state. Once the transaction successfully completes all activities in the activity set, it transitions to the *completing* state and performs all other work required to complete. If it completes this work successfully, the transaction transitions to the *complete* state.

A transaction will transition to the *aborting* state when one of the transaction's event handlers is triggered, or one of the parent context event handlers is triggered, or the transaction cannot complete successfully.

The transaction then performs all other work required to abort before moving into the *aborted* state, in particular the completion of any activities in its event handlers. This is a terminal state.

It is possible to compensate for a transaction once it is in the *complete* state. The first attempt to compensate a transaction will transition it to the *compensating* state. The transaction then performs all of the activities in the compensation activity set before moving into the *compensated* state. This is a terminal state.

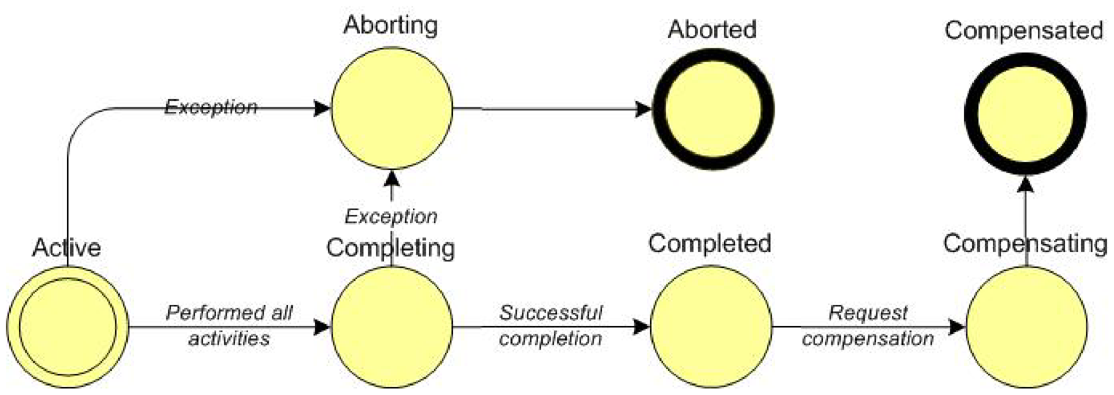

Figure 3: Transition diagram for states of transaction instance

Compensation

The compensate activity is used to compensate for a transaction. This activity must reference the transaction instance by its instance identifier; as such it must be executed in a context in which the instance identifier has been assigned.

The compensation activity set does not execute as part of the transaction, but after completion of the transaction. If compensation must provide a level of transactional guarantee, the compensation activity set should define its own transaction and event handlers.

During transition to the *complete* state, the transaction stores all information required in order to establish a context for the purpose of compensation. This is required only if the compensation activity set contains any activities that will be executed as part of compensation and will access any properties that are specific to the transaction context..

Once the transaction is in the *complete* state, it can be compensated exactly once. The first attempt to compensate the transaction will transition it to the *compensating* state and from there to the *compensated* state.

If the transaction is in the *active* or *completing* states, the *compensate* activity will hold until the transaction transitions to a different state. If the transaction is in the *aborting*, *aborted*, *compensating* or *compensated* state, no work will be done and the *compensate* activity will complete immediately.

It is an error to compensate for a transaction from any activity that executes in the context in which the transaction is defined, or one of its child contexts.

The compensation context is instantiated when the transaction performs compensation. It can be discarded after the transaction moves into the *compensated* state.

A transaction can be compensated from different contexts by passing input parameters to the compensation activity set. This allows different behavior, depending on the context in which the *compensate* activity is called

The compensation activity set may refer to local declarations made in the compensation context, local declarations made in the transaction context, and input parameters. Using any other property will change the value of that property in a parent context of the transaction.

If a fault occurs while executing the compensation activity set and is not caught by an event handler, that fault will propagate to the *compensate* activity and cause it to fault. The transaction will still transition to the *compensated* state.

When compensating multiple transaction instances, compensation occurs in the reverse chronological order in which these instances transitioned to the *complete* state.

Atomic Transactions

Changes made to the value of properties from any activity performed as part of an atomic transaction are visible only from other activities occurring in the same transaction.

Changes that affect a parent context will take effect only when the transaction completes and will not take effect if the transaction aborts. All such changes are applied collectively upon completion of the transaction. Partial changes are never visible.

Changes that affect properties declared as local in the transaction context are discarded along with that context and are not visible after the transaction completes.

This behavior affects both properties that are handled explicitly by activities occurring in that transaction, as well as all instance properties that are modified in that context.

All atomic activities behave in the same manner when they are not executing in an atomic transaction context.

Synchronous communication performed by activities in the transaction context, such as with the WSDL *request-response* and *solicit-response* operations, will include the transaction instance identifier. This allows other services to participate in the transaction.

The manner in which transaction context propagation is achieved depends on the messaging protocol used by the activity.

Asynchronous communication performed by activities in the transaction context, such as with the WSDL *one-way* and *notification* operations, are effective only if the transaction completes.

A message received by such an activity is permanently consumed only if the transaction completes. If the transaction aborts, the message is retained and may be consumed by other activities.

A message sent by such an activity is retained until the transaction completes. If the transaction aborts, the message is discarded.

While the transaction is in progress, all messages asynchronously sent and received are not visible to other activities in the same or different process or transaction.

Open Transactions

Open transactions do not provide isolation for data changes and messages exchanged. All data changes are visible in any context upon completion of the atomic activity or atomic transaction that performed them.

Synchronous communication performed by activities in the transaction context, as with the WSDL *request-response* and *solicit-response* operations, may include the transaction instance identifier. This allows other services to identify the transaction.

The manner in which transaction context propagation is achieved depends on the messaging protocol used by the activity.

Asynchronous communication performed by activities in the transaction context, as with the WSDL *one-way* and *notification* operations, behave in the same manner as activities performing synchronous communication. Messages sent are made available and messages received are consumed immediately upon completion of the action.

Other processes that interact with the transaction by sending and receiving messages are not necessarily part of the transaction. They may complete their work before or after the transaction has completed and independent of whether the transaction has completed, aborted, or was compensated.

5.2.6. Transactional Processes

A process can be instantiated from within the context of an atomic or open transaction. The rules differ depending on the type of transaction from which the process is instantiated and whether the process is defined in that transaction context or in some other context.

If the process is called from within an atomic transaction, the process is required to participate in that transaction. This is the case when using the *call* or *action* activities. The process indicates its ability to participate in the transaction by defining its context as transactional. The transaction is propagated to the process upon its instantiation.

A process can indicate that it *supports* instantiation from within an atomic transaction or that it must *always* be instantiated from within an atomic transaction. A process that requires instantiation from within an atomic transaction cannot be instantiated from outside an atomic transaction context.

If a process indicates that it *never* supports instantiation from within an atomic transaction, the process cannot be defined in, or called from within, the context of an atomic transaction.

A process that is defined within the context of an atomic transaction indicates that it will always be instantiated from within that transaction (*always*).

A process that is spawned from within an atomic transaction does not need to participate in the transaction. In this case, the process is instantiated only if the atomic transaction from which it was spawned completes. The process is not instantiated if that transaction aborts.

If the process indicates participation in the atomic transaction, the behavior for spawning is the same as for calling. Note that the transaction will not be able to complete until the process has completed. Likewise, the transaction will always abort if the process aborts.

If the process is instantiated from a message and is defined within an atomic transaction context, the process is instantiated in an atomic transaction context. The transaction instance identifier must be propagated to the process as part of the input message.

The manner in which transaction context propagation is achieved depends on the messaging protocol used by the activity.

The process is able to see data changes and message exchanges that occur in an open transaction without having to participate in that transaction. As a result, a process does not declare its participation in an open transaction.

A process that is defined within an open transaction context or one of its child contexts will execute in that transaction context. Calling, spawning or otherwise instantiating a process from an open transaction context does not propagate that transaction to the called/spawned process.

If the process defines an atomic transaction and is instantiated as part of another atomic transaction, both transactions have different instance properties, but are in fact the same transaction.

The transaction instance property of the nested process is a local declaration in the parent process's transaction context. The parent process can compensate for the nested process using the *compensate* activity and the name of the nested process transaction, but no other context can compensate for work performed by the nested process in this manner.

5.2.7. Transaction

A transaction definition is a composition of the following attributes:

Attribute	**Description**
name	The transaction name.
type	Either *atomic* or *open*.
participation	Either *supports*, *always* or *never*.
retries	Maximum number of retries.
compensation	Compensation activity set and parameters.

The **name** attribute provides the transaction name.

The **participation** attribute is applicable only if the transaction is defined in the process context and the transaction type is *atomic*.

Participation of type *supports* allows the instantiation of the process from an atomic transaction and the propagation of that transaction to the process. The process can be instantiated from any context, but cannot be defined in a context that is part of an atomic transaction.

Participation of type *always* requires the instantiation of the process from an atomic transaction and the propagation of that transaction to the process. It is an error to call or spawn the process from an activity that is not part of an atomic transaction. This value is implied if the process is defined in a context that is part of an atomic transaction.

Participation of type *never* precludes the instantiation of the process from an atomic transaction. It is an error to call or spawn the process from an activity that is part of an atomic transaction or to define the process in a context that is part of an atomic transaction. This is the default type for a process that is defined in a context that is not part of an atomic transaction, or any top-level definition.

The **retries** attribute is a positive integer that specifies the maximum number of times the transaction can be repeated until it completes. The value zero specifies that the transaction will execute at most once.

For the compensation activity set, both input and output parameters can be specified. The rules for specifying parameters are the same as for a process definition, the rules for passing parameters are the same as for the *call* activity.

Since the *compensate* activity is atomic, if multiple instances are compensated, the properties will be set from the value of output parameters of the last transaction instance that was compensated.

The syntax for a *transaction* definition is:

```
<transaction
  name = NCName
  type = (atomic | open) : atomic
  participation = (supports | always | never) : never
  retries = nonNegativeInteger : 0>
  Content: (compensation?)
</transaction>
```

The fully qualified transaction name is constructed by combining the *name* attribute with the target namespace attribute of the package.

It is an error to use the participation attribute if the transaction type is not atomic, or the transaction is not defined as part of the context of a process.

The syntax for the transaction *compensation* is:

```
<compensation>
  Content: (documentation?, parameter*, context?, {any activity}+)
</compensation>
```

The *parameter* elements are the same as used by a process definition (see there).

This process receives an order request from the user, submits the order to the supplier and waits for confirmation and notice of shipment before reporting completion to the user.

This process allows the user to cancel the transaction before it completes and will inform the supplier. The process also notifies the user if the transaction fails to complete.

The user sends a request order to start the process. The process then performs the *submitOrder* transaction so it can to send the order to the supplier.

The *submitOrder* transaction is a nested transaction that sends an order and waits for confirmation. The confirmation is expected within a certain duration. If no confirmation is received, the nested transaction is aborted and repeated up to three times.

To cancel the parent transaction, the *submitOrder* transaction is compensated by sending a cancellation message to the supplier. Cancellation occurs by terminating the parent transaction upon receipt of a cancel message from the user.

When the parent transaction completes successfuly, it sends a confirmation notice to the user. If the nested transaction fails repeatedly, the parent transaction is unable to complete and the user is sent a notice of failure.

```
<process name=”processOrder”>
  <context>
    <exception>
      <onMessage>
        <documentation>
          User askes to cancel the parent transaction. At this point,
          the submitOrder transaction is aborted. If it has completed,
          we need to compensate and inform the supplier.
        </documentation>
        <action portType=”tns:userPort” operation=”receiveCancel”>
          . . .
        </action>
        <compensate transaction=”submitOrder”/>
      </onMessage>
      <onFault code=”bpml:completion”>
        <documentation>
          The submitOrder nested transaction has failed repeatedly.
          Notify the user that the parent transaction has aborted.
        </documentation>
        <action portType=”tns:userPort” operation=”informFailure”>
          . . .
        </action>
      </onFault>
    </exception>
    <transaction name=”completeOrder” type=”open”/>
  </context>

  <action portType=”tns:userPort” operation=”receiveRequest”>
    . . .
  </action>

  <sequence>
    <context>
      <exception>
        <onTimeout property=”tns:timeToProcessOrder” type=”duration”>
          <documentation>
            Abort the transaction on time-out. Nothing special to do.
          </documentation>
          <empty/>
        </onTimeout>
      </exception>
      <transaction name=”submitOrder” type=”open” retries=”2”>
        <compensation>
          <documentation>
            Compensate for a completed submitOrder transaction by
            sending a cancellation message to the supplier.
          </documentation>
          <action portType=”tns:buyerPort” operation=”sendCancellation”>
            . . .
          </action>
        </compensation>
      </transaction>
    </context>
    <action portType=”tns:buyerPort” operation=”sendOrder”>
      . . .
    </action>
    <action portType=”tns:buyerPort” operation=”receiveConfirmation”>
      . . .
    </action>
  </sequence>

  <action portType=”tns:buyerPort” operation=”receiveShipNotice”>
    . . .
  </action>
  <action portType=”tns:userPort” operation=”informCompletion”>
    . . .
  </action>
```

```
</process>
```

6. Activities

6.1. Action

An **action** is an atomic activity. It provides the context for performing an operation, in particular, operations involving the exchange of messages with other processes and services.

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

Attribute	**Description**
name	The activity name.
operation	The operation being performed.
correlation	Zero or more correlations.
locator	Locator mechanism and zero or more properties passed to it.
call	Call activity.
output	Zero or more constructs for constructing parts of the output message.

An action does not define the operation it performs, but indicates which **operation** will be performed and provides the execution context for performing that operation. An action is atomic and so can only refer to a single atomic operation.

Correlation is a fairly simple mechanism used to identify a process instance based on data provided in the input message. An action may involve no correlations, or one or more correlations.

An action that is performed against a particular service must first identify the service. A **locator** can be used for the purpose of identifying the service, and properties may be passed by the action to that locator.

An action that receives an input message and must provide a response, can perform an arbitrary set of activities to process the input message by embedding a **call** activity.

An action may need to construct an **output** message. The output message parts are constructed individually from the values of properties available in the current context.

Support for WSDL operations is a normative part of the BPML specification. Actions may refer to the following WSDL operations:

- **One-way** – The process receives an input message. The input message may be correlated.
- **Request-response** – The process receives an input message, constructs and sends an output message back to the sender. Any work done between to process the input message is performed by calling another process. The input message may be correlated.
- **Solicit-response** – The process constructs and sends a message and waits for a response from the recipient. The recipient may be identified using a locator. The input message may be correlated with instantiation type *true*.
- **Notification** – The process constructs and sends a message. The recipient may be identified using a locator.

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

```
<action
```

```
    name = NCName
    portType = QName
    operation = NCName
    {extension attribute}>
    Content: (documentation?, correlate*, locate?, call?, output*)
  </action>
```

Operations are defined by other specifications. An operation definition can be imported into a BPML package using the import construct.

WSDL operations are referenced using the **portType** and **operation** attributes. The *portType* attribute references the WSDL port type definition, while the *operation* attribute references the particular operation of that port type definition.

6.1.1. Correlation

Correlating an action establishes a relation between the context in which the action occurs and the message received by the action. The relation is established through properties that are carried in the input message and are matched by value to properties contained in the context of the action.

An action must be correlated if it is required to provide a means for identifying the context instance in which it executes based on the message it receives, in particular when performing the WSDL *one-way* and *request-response* operations. Correlation is not allowed for the WSDL *notification* operation.

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

```
  <correlate
    correlation = QName
    instantiation = boolean : false/>
```

The correlate element references a correlation definition by its name. The correlation definition indicates which properties are used for the purpose of correlation. Correlation definitions typically use a single property, but multiple property correlations are allowed.

Correlation definitions are covered by the WSCI specification and require the use of selectors that retrieve the property values from the input message. Correlations and selectors are defined in a WSCI document and imported into a BPML package.

When instantiating a correlation, the correlation properties are instantiated in the context based on their value in the input message. The context is not required to have an established value for these properties.

When not instantiating a correlation, the correlation properties are used to identify the context in which the action should execute. The context must have an established value for these properties.

The context instance is identified by matching the set of property values in the message to the set of property values in the context, matching properties by their names. The input message can only be delivered to such a context instance, and only to one context instance.

It is possible for an action to be simultaneously associated with two different correlations, one of which is not instantiated one that is instantiated. It is also possible that one correlation extends another correlation and that both will be associated with the same action.

6.1.2. Locator

An action that is performed against a particular service must first identify the service.

A locator is required if the action must identify the service, in particular when performing the WSDL *notification* and *solicit-response* operations. Locators are not used with other WSDL operations.

Locators are mechanisms for identifying a service given a set or one or more properties or statically given no properties. Locator definitions are covered by the WSCI specification. Locators are defined in a WSCI document and imported into a BPML package.

The default locator takes a single property and uses that property to identify the service. The property value must be in the form of a URI which designates the service end-point. The default locator is unnamed, is not explicitly defined and is available to all process definitions.

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

```
<locate
  property = list of QName
  locator = QName/>
```

A service can be located in one of three ways:

- **Dynamically by URI** – Through a single property that provides its end-point. The property name is given by the *property* attribute. The *locator* attribute is absent (the default locator is used).
- **Dynamically by lookup** – Through one or more properties that provide all the information required to look up the service. The property names are given by the *property* attribute. The lookup mechanism is specified by name using the *locator* attribute.
- **Statically** – The *property* attribute is absent. The lookup mechanism is specified by name using the *locator* attribute.

6.1.3. Call

An action can perform an arbitrary set of activities before it completes by calling a process. This is done by embedding the call activity.

An action can perform an arbitrary set of activities only if its semantics require that these activities be performed in order for the action to complete, specifically when performing the WSDL *request-response* operation. The *call* activity is not allowed for other WSDL operations.

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

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

The *name* attribute cannot be used when the *call* element appears inside an action. Aside from that restriction, the *call* element has the same syntax and behavior as the *call* activity.

An action is an atomic activity. As such, the called process is invoked and completes before the action itself completes. If the called process faults, the action faults with the same fault code. If the action has to be terminated, the called process is also terminated.

Changes to the context of the *action* activity that are performed by the called process are not visible to other activities executing in the same context until the *action* activity completes.

6.1.4. Output

Constructing output messages is necessary only for actions that involve sending a message, specifically when performing the WSDL *request-response*, *notification* and *solicit-response* operations. Constructing output messages is not allowed for the WSDL *one-way* operation.

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

```
<output
  part = NCName
```

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

The **part** attribute names the message part being constructed. This must be a part defined in the message definition of the message that is used as output in the operation being performed, or for any other part allowed as an output by that operation (e.g. generic headers).

If the message consists of a single unnamed message part, this attribute may be omitted.

The contents of a message part are constructed using one of the following means:

- **value** – Uses an XML value that is statically provided in the content of that element
- **xpath** – Uses an XPath expression that is evaluated in the current context
- **extension element** – Supports other mechanisms by which the value is constructed (e.g. an XQuery expression)

The three uses are mutually exclusive and cannot be combined in the same element.

Expressions are evaluated in the context of the *action* activity, allowing them to access values of properties that are part of the current context, as well as values of properties that were changed by the *action* activity, but not yet available in the current context.

The following example is a process that consists of three actions.

The first action receives an order request that targets a new process instance. The correlation is instantiated by this action given the order identifier.

The second action sends an acceptance signal, and uses the same order identifier. The third action sends an invoice using the same order identifier. The invoice details message part is filled with the value of the *tns:invoiceDetails* property.

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

```
<process name=”example”>
  <action name=”receiveRequest”
          portType=”tns:supplierPort” operation=”receiveRequest”>
    <correlate name=”tns:orderID” instantiation=”true”/>
  </action>
  <action name=”sendAcceptance”
          portType=”tns:supplierPort” operation=”sendAccept”>
    <output part=”orderID” xpath=”$tns:orderID”/>
  </action>
  . . .
  <action name=”sendInvoice”
          portType=”tns:supplierPort” operation=”sendInvoice”>
    <output part=”orderID” xpath=”$tns:orderID”/>
    <output part=”details” xpath=”$tns:invoiceDetails”/>
  </action>
</process>
```

6.2. All

The **all** activity is a complex activity. It executes all the activities within the activity set in non-sequential order.

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

Attribute	**Description**
name	The activity name.
activity set	An activity set.

Activities are executed in non-sequential order. Activities may be executed concurrently or serially, but no particular order is specified.

The *all* activity completes only after all activities in the set have executed, regardless of order.

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

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

6.3. Assign

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

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

Attribute	**Description**
name	The activity name.
property	The name of the property being assigned.
expression	Expression to evaluate.
value	Fixed value.

The **property** attribute provides the property name.

The property is either assigned a static value specified by the **value** attribute, or a value derived from the result of an expression evaluated in the current context of the *assign* activity.

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

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

The value is constructed using one of the following three means:

- **value** – Uses an XML value that is statically provided in the content of that element

- **xpath** – Uses an XPath expression that is evaluated in the current context
- **extension element** – Supports other mechanisms by which the value is constructed (e.g. an XQuery expression)

The three uses are mutually exclusive and cannot be combined in the same element.

6.4. Call

The **call** activity is an atomic activity. It instantiates a process and waits for it to complete.

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

Attribute	Description
name	The activity name.
process	The process being instantiated.
output	Zero or more constructs for constructing call parameters.

The **process** attribute names the called process.

The *call* activity can instantiate any processes whose definition is accessible in its current context. This include any process defined in the same context as the *call* activity, or in any parent context, or any top-level process defined in the same package or imported.

The process is instantiated in the same context in which it is defined, which may not be the same as the context in which the *call* activity is executed. The activity may pass parameters to and from the process.

The activity may need to construct **output** values to pass to input parameters of the instantiated process. Output values are required for input parameters that are defined as required, and optional for input parameters that are defined as optional.

If the process defines output parameters, the value of these parameters are assigned to properties with the same name in the context of the *call* activity upon successful completion (transition to the *complete* state).

The activity waits until the instantiated process either completes successfully, or aborts with a fault. If the called process aborts, the *call* activity aborts with the same fault code. If the action has to be terminated, the called process is also terminated.

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

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

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

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

The **parameter** attribute names the parameter being constructed. This must be a parameter that is defined as an input parameter in the instantiated process.

The value of a parameter is constructed using one of the following means:

- **value** – Uses an XML value that is statically provided in the content of that element
- **xpath** – Uses an XPath expression that is evaluated in the current context
- **extension element** – Supports other mechanisms by which the value is constructed (e.g. an XQuery expression)

The three uses are mutually exclusive and cannot be combined in the same element.

Expressions are evaluated in the context of the *call* activity, allowing them to access values of properties that are part of the current context, as well as values of properties that were changed by the *call* activity, but not yet available in the current context.

6.5. Choice

The **choice** activity is a complex activity. It selects and executes one activity set in response to a triggered event.

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

Attribute	**Description**
name	The activity name.
events	Two or more event handlers.

The *choice* activity must specify two or more event handlers.

The *message*, *time-out*, and *fault* event handlers are mutually exclusive. The first event to occur will trigger the corresponding event handler and perform the specified activity set, after which the activity will complete.

It is an error to specify two overlapping event handlers, and at most one *time-out* event handler is allowed.

Events handlers are specified in the section dealing with exception handling. The three types of permissible event handlers are: message event handler, time-out event handler and fault event handler.

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

```
<choice
  name = NCName>
  Content: (documentation?,
            (onMessage | onTimeout | onFault){2,*})
</choice>
```

6.6. Compensate

The *compensate* activity is an atomic activity. It performs compensation for all instances of the named transaction.

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

Attribute	**Description**
name	The activity name.

transaction	The transaction to compensate.

The **transaction** attribute specifies the transaction to compensate. The name must match an instance list property that was assigned in the current context of the *compensate* activity, or indicate compensation for all transactions instantiated in the current context.

The activity performs compensation for all instances of the named transaction obtained from the instance list, or for all transactions instantiated in the current context, that are in the *complete* state. No work is done for transaction instances that are in any other state.

If multiple transaction instances are to be compensated, they are compensated for sequentially in the reverse of the chronological order in which they completed (i.e. transitioned to the *complete* state). The activity completes only after all such instances have transitioned to the *compensated* state.

The activity may need to construct **output** values to pass to input parameters of the compensation activity set. Output values are required for input parameters that are defined as required, and optional for input parameters that are defined as optional.

Output parameters are only allowed when compensating for a transaction by its instance list property, and must match the definition of the *compensation* construct. The same values are passed to all instances that are being compensated. Output parameters are not supported.

If compensation of any transaction instance aborts with a fault, the *compensate* activity will abort with the same fault, but only after attempting to compensate for all transaction instances.

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

```
<compensate
  name = NCName
  transaction = (QName | #all)>
  Content: (documentation?, output*)
</compensate>
```

The special value *#all* indicates that the activity will compensate all transaction instantiated in the current context.

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

6.7. Delay

The *delay* activity is an atomic activity. It expresses the passage of time.

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

Attribute	**Description**
name	The activity name.
dateTime	Name of property providing the time-out time instant.
duration	Name of property providing the time-out duration.
reference	Name of property providing the instance identifier.
start/end	Indicates whether reference time is start time or end time of instance.

The *delay* activity identifies the time instant at which the activity will complete. The activity can be cancelled and suspended, e.g. when an exceptional event handler is triggered in the same context.

The time instant is identified using the same attribute and syntax as for the time-out event handler.

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

```
<delay
  name = NCName
  property = QName
  type = (duration | dateTime) : duration
  reference = ($QName | bpml:start($QName) | bpml:end($QName))>
  Content: (documentation?)
</delay>
```

6.8. Empty

The *empty* activity is an atomic activity. It does no work.

The *empty* activity is a composition of the following attributes:

Attribute	Description
name	The activity name.

This activity can be used in places where an activity is expected, but no work is to be performed.

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

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

6.9. Fault

The *fault* activity is an atomic activity. It triggers a fault within the current context.

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

Attribute	Description
name	The activity name.
code	The fault code.

The fault code is specified using the **code** attribute. The fault occurs immediately in the current context, see the definition of exception handling for how faults and other exceptions are handled.

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

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

6.10. Foreach

The *foreach* activity is a complex activity. It performs all the activities within the activity set repeatedly, once for each item in the list.

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

Attribute	**Description**
name	The activity name.
expression	Expression to evaluate.
activity set	An activity set.

The expression is evaluated once against the parent context of the *foreach* activity, resulting in an item list. The activity set is repeated once for each item in the item list. Activities are executed in sequential order.

The *forach* activity must instantiate a new context instance for each iteration. The current value of the item for that iteration is assigned to the property *bpml:current* in that context.

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. If the result of evaluating the XPath expression is a node set, then each node is an item in the item list. If the result of evaluating the XPath expression is any other type, than that value is the only item in the item list.

6.11. Join

The *join* activity is an atomic activity. It waits for instances of process to complete.

The *join* activity is a composition of the following attributes:

Attribute	**Description**
name	The activity name.
process	The name of the process being joined.
count	The name of a property with a count value.

The **process** attribute names an instance list property. The value of this property is a list of process instances that are accessible in the current context.

The *join* activity waits for a specified number of instances that are currently active to complete, that is, transition to either *complete* or *aborted* states. This may include both instances that were instantiated by the *spawn* and *call* activities, as well as instances instantiated upon receipt of a message.

The **count** attribute is the name of the property. The value of that property is the count, or number of instances for which to wait. If absent, the *join* activity waits for all instances to complete.

If the number of active instances is higher than the count, the *join* activity will wait until as many instances as specified by the count have completed, in any order.

If the number of active instances is lower than or equal to the count, the *join* activity will wait until all instances that are currently active will complete.

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

```
<join
  name = NCName
  process = QName
  count = QName>
  Content: (documentation?)
</join>
```

6.12. Sequence

The *sequence* activity is a complex activity. It performs all the activities within the activity set in sequential order.

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

Attribute	**Description**
name	The activity name.
activity set	An activity set.

Activities are executed in sequential order. The *sequence* activity completes only after all activities in the set have executed.

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

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

6.13. Spawn

The *spawn* activity is an atomic activity. It instantiates a process.

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

Attribute	**Description**
name	The activity name.
process	The name of the process being instantiated.
output	Set of zero or more constructs for constructing call parameters.

The **process** attribute names the spawned process.

The *spawn* activity can instantiate any processes whose definition is accessible in its current context. This include any process defined in the same context as the *spawn* activity, or any parent context, or any top-level process defined in the same package or imported.

The process is instantiated in the same context in which it is defined, which may not be the same as the context in which the *spawn* activity is executed.

The activity may need to construct **output** values to pass to input parameters of the instantiated process. Output values are required for input parameters that are defined as required, and optional for input parameters that are defined as optional.

The activity does not wait for the instantiated process to complete, but returns immediately.

If the *spawn* activity is performed in an atomic transaction and the instantiated process does not participate in that transaction, the process will be instantiated only if and when the transaction completes and will not be instantiated if the transaction aborts.

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.

6.14. Switch

The *switch* activity is a complex activity. It selects and executes one activity set based on the evaluation of one or more conditions.

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

Attribute	Description
name	The activity name.
cases	One or more conditional cases.
default	Activity set.

A **conditional case** selects an activity set based on the truth value of a condition. This construct is a composition of the following attributes:

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

For each conditional case, the condition is evaluated in the parent context of the *switch* activity. The order for evaluating conditions is the same as the order in which the conditional cases are specified.

If a condition evaluates to *true*, the activity set associated with that conditional case will be selected and no other conditions will be evaluated. Activities are executed in sequential order.

If no condition evaluates to *true*, the activity set for the default case is selected, if specified. Activities are executed in sequential order.

The *switch* activity completes after executing all the activities in the selected activity set, or immediately if no activity set has been selected.

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

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

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

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

The *condition* element specifies an expression that evaluates to a Boolean value.

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

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

The expression is provided as the character data for this element and cannot be an empty string. A condition is always an XPath expression if no extension attribute is used.

The *condition* element is an extension element and can incorporate any number of extension attributes, as long as these attributes are defined in a namespace other than the BPML core namespace. If extension attributes are used, the type of expression and manner in which the expression is evaluated depends on these extension attributes.

6.15. Until

The *until* activity is a complex activity. It executes all the activities within the activity set repeatedly, one or more times, based on the truth value of a condition.

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

Attribute	**Description**
name	The activity name.
condition	Condition to evaluate.
activity set	An activity set.

The activity set is executed at least once. Activities are executed in sequential order, and after completion, the condition is evaluated. These two steps are repeated until the condition evaluates to *false*, at which point the *until* activity completes.

The *until* activity must instantiate a new context instance for each iteration. The condition is always evaluated in the parent context of the activity.

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.

6.16. While

The *while* activity is a complex activity. It executes all the activities within the activity set repeatedly, zero or more times, based on the truth value of a condition.

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

Attribute	**Description**
name	The activity name.
condition	Condition to evaluate.
activity set	An activity set.

If the condition evaluates to *true*, the activities are executed in sequential order. These two steps are repeated until the condition evaluates to *false*, at which point the *while* activity completes.

The *while* activity must instantiate a new context instance for each iteration. The condition is always evaluated in the parent context of the activity.

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.

7. Connecting Services

7.1. Global Model

A **global model** provides a view of how multiple processes interact. The processes that constitute a global model are loosely coupled and interact through the exchange of messages.

Connectors express the interaction between processes. These connectors provide a link between the operation performed by a process and the reciprocal operation performed by another process. The global model does not use facilities that connect tightly bound processes, such as the *spawn* and *call* activities.

Connectors are not used to express interactions with processes that are not part of the global model. However, interactions can be used to model operations performed against a service for which no process definition is given.

A given process can be used as part of multiple compositions. The global model definition does not include any process definition, but references them by name. This allows multiple compositions to reuse the same set of processes or to show different links between the processes.

A global model does not include process definitions, but references all applicable process definitions. The process definitions must be accessible, for example, by importing them into the same package in which the global model is defined. A global model includes declarations of all connectors.

A global model defines a closed system if all operations performed in the global model are connected to each other. If the global model defines a closed system, but includes operations which are not associated with any particular process, it is assumed that these operations are performed by a service for which no process definition is required (a stateless service).

The *global model* construct is a composition of the following attributes:

Attribute	**Description**
name	The name of this global model.
processes	Reference to two or more processes that are part of the global model.
connectors	One or more connector declarations.

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

```
<model
  name = NCName>
  Content: (documentation?, uses+, connect+)
</model>

<uses
  model = QName
  interface = QName
  package = anyURI
  process = NCName/>
```

7.2. Connector

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

Attribute	**Description**
name	The name of this connector. Optional.
type	The connector type, either *direct* or *broadcast*.
operation	Reference to two operations on opposing ports.
mapping	Mapping between the connected operations. Optional.

The syntax for a *connect* declaration is given as:

```
<connect
  name = NCName
  type = (direct | broadcast) : direct>
  Content: (documentation?, operation{2}, {extension element}?)
</connect>

<operation
  portType = QName
  name = NCName
  {extension attribute}/>
```

8. References

8.1. 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

Web Services Choreography Interface (WSCI) 1.0, BEA, Intalio, Sun, SAP et al, June 2002

http://www.intalio.com/wsci/

WSDL

Web Services Description Language (WSDL) 1.1, W3C Note, 15 March 2001

http://www.w3.org/TR/wsdl.html

XML 1.0 (Second Edition)

Extensible Markup Language (XML) 1.0, Second Edition, Tim Bray et al., eds., W3C, 6 October 2000

http://www.w3.org/TR/REC-xml

XML-Namespaces

Namespaces in XML, Tim Bray et al., eds., W3C, 14 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, 2 May 2001

http://www.w3.org/TR/xmlschema-1//

XML Schema Part 2: Datatypes, Paul V. Biron and Ashok Malhotra, eds., W3C, 2 May 2001

http://www.w3.org/TR/xmlschema-2/

XPath

XML Path Language (XPath) 1.0, James Clark and Steve DeRose, eds., W3C, 16 November 1999 http://www.w3.org/TR/xpath

8.2. 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

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

OMG UML

Unified Modeling Language Specification, OMG, June 1999

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 Part21: Adjuncts, W3C Working Draft

http://www.w3.org/TR/soap12-part2/

UDDI

Universal Description, Discovery and Integration, Ariba, IBM and Microsoft, UDDI.org.

http://www.uddi.org

UUID and GUID

UUIDs and GUIDs, Network Working Draft, February 4 1998

http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt

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

XQuery 1.0: An XML Query Language, W3C Working Draft

http://www.w3.org/TR/xquery/

XML Syntax for XQuery 1.0 (XQueryX), W3C Working Draft

http://www.w3.org/TR/xqueryx

XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Working Draft

http://www.w3.org/TR/xquery-operators/