Business Process Management Initiative (BPMI)

Business Process Modeling Notation

Working Draft (0.9) November 13, 2002

Abstract

The Business Process Modeling Notation (BPMN) specification provides a graphical notation for expressing business processes in a Business Process Diagram (BPD). The objective of BPMN is to support process management by both technical users and business users by providing a notation that is intuitive to business users yet able to represent complex process semantics. The BPMN specification also provides a mapping between the graphics of the notation to underlying the constructs of execution languages, such as BPEL4WS and BPML.

Status of this Document

This document is the first working draft of the BPMN specification submitted for comments from the public by members of the BPMI initiative on November 13, 2002. It has been produced based on the work of the members of the Notation Working Group. Comments on this document and discussions of this document should be sent to BPMN-PublicReview@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.”

Acknowledgements

The members of the BPMI Notation Working Group contributed to the development of this specification. The following were members of the committee for at least part of the time from August 2001 until this version of the specification:

Michael Anthony	International Performance Group
Ashish Agrawal	Intalio
Assaf Arkin	Intalio
Jeanne Baker	Sterling Commerce
Steve Ball	Sterling Commerce
Chuck Faris	Popkin Software
Ismael Ghalimi	Intalio
Jared Groth	Sterling Commerce
Paul Harmon	Individual Member
Brian James	Proforma Corporation
Simon Johnston	Rational Software
George Keeling	Casewise
Antoine Lonjon	MEGA International
Dave Madigan	Glue Limited
Mike Marin	FileNet
Martin Owen	Popkin Software
Matthew Pryor	Versata
Robert Shapiro	Cape Visions
Howard Smith	CSC
Don Stewart	Genient
Bala Suryanarayanan	Infosys Technologies
Stephen A. White	SeeBeyond
Paul Wuethrich	New Era of Networks (Sybase)

The primary author and editor of the main body of the specification was

Stephen A. White (swhite@seebeyond.com)

Additional contributions to its writing were made by

Ashish Agrawal (ashish@intalio.com)
Michael Anthony (manthony@ipgl.com)
Assaf Arkin (arkin@intalio.com)
George Keeling (george@casewise.co.uk)
Brian James (bjames@proformacorp.com)
Antoine Lonjon (alonjon@mega.com)
Martin Owen (martin.owen@popkin.co.uk)

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

Table of Contents

List of Figures

List of Tables

List of Examples

1. Introduction

The **Business Process Management Initiative** (BPMI) has developed a standard **Business Process Modeling Notation** (BPMN). The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes. Thus, BPMN creates a standardized bridge for the gap between the process analysis and process implementation.

Another goal, but no less important, is to ensure that XML languages designed for the execution of business processes, such as **BPEL4WS** (Business Process Execution Language for Web Services) and **BPML** (Business Process Modeling Language), can be visualized with a common notation. We will consider that each of these execution languages is equally relevant to BPMN. In the interest of consistency, however, they will be listed in alphabetical order when both are being discussed.

This specification defines the notation and semantics of a **Business Process Diagram** (BPD) and represents the amalgamation of best practices within the business modeling community. BPMN is the standardization of many different modeling notations and viewpoints and provides a simple means of communicating process information to other business users, process implementers, customers, and suppliers.

The BPMN specification defines a mapping from BPMN to BPEL4WS and BPML, and is comprised of the following topics:

BPMN Overview provides an introduction to BPMN, its requirements, and discusses the range of modeling purposes that BPMN can convey.

Business Process Diagram Concepts provides a summary of the BPMN graphical elements and their relationships.

Business Process Diagram Graphical Objects details the graphical representation and the semantics of the behavior of BPMN diagram elements.

Connecting Objects defines the graphical objects used to connect two objects together (i.e., the connecting lines of the diagram) and how flow progresses through a Process (i.e., through a straight sequence or through the creation of parallel or alternative paths).

BPMN by Example provides a walkthrough of a sample Process using BPMN.

Mapping to Execution Languages provides the formal mechanism for converting a BPMN diagram to a BPEL4WS or BPML document.

References provides a list of normative and non-normative references.

Open Issues provides a list of issues that will affect the future of the BPMN specification.

Appendix A: E-Mail Voting Process BPEL4WS provides a full sample of BPEL4WS code based on the example business process described in the “BPMN by Example” section.

Appendix B: E-Mail Voting Process BPML provides a full sample of BPML code based on the example business process described in the “BPMN by Example” section.

Appendix C: Glossary presents an alphabetical index of terms that are relevant to practitioners of BPMN.

1.1 Conventions

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

1.1.1 Typographical and Linguistic Conventions and Style

This specification incorporates the following 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 BPMN construct is highlighted with a capitalized word (e.g., Sub-Process).
- A reference to a BPEL4WS or BPML element, attribute, or construct is highlighted with an italic lower-case word, usually preceded by the word "BPEL4WS" (e.g., BPEL4WS *pick*) or "BPML" (e.g., BPML *choice*).
- Non-normative examples are set of in boxes and accompanied by a brief explanation.
- XML and pseudo text is highlighted with `mono-spaced` typeface.
- The cardinality of any content part is specified using the following operators:
    - (none) — exactly once
    - ? — 0 or 1
    - * — 0 or more
    - + — 1 or more
    - Properties separated by | and grouped within (and) — alternative values
    - : <value> — default value

1.2 Dependency on Other Specifications

The BPMN specification supports for the following specifications is a normative part of the BPMN specification: BPEL4WS and BPML.

The following abbreviations may be used throughout this document:

This abbreviation	**Refers to**
BPEL4WS	Business Process Execution Language for Web Services (see BPEL4WS). This abbreviation refers specifically to version 1.0 of the specification, but is intended to support future versions of the BPEL4WS specification.

BPML	Business Process Modeling Language (see BPML). This abbreviation refers specifically to version 1.0 of the specification, but is intended to support future versions of the BPML specification.
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 Conformance

A **BPMN processor** is responsible to process XML documents that conform to the BPMN 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 BPMN specification.

A **BPMN implementation** is responsible to perform one or more duties based on the semantics conveyed by BPMN definitions. A BPMN implementation must understand the semantics of BPMN definitions as set forth in this specification.

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

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

Specification Feature	
BPMN 0.9	http://www.bpmi.org/2002/11/bpmn

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

2. BPMN Overview

There has been much activity in the past two or three years in developing web service-based XML execution languages for BPM systems. Languages such as BPEL4WS and BPML provide a formal mechanism for BPM Systems to define and execute business processes and to interoperate with each other. The key element of these languages is that they are optimized for the operation and interoperation of BPM Systems. The optimization of these languages for software operations renders them less suited for direct use by humans to design and manage business processes. BPML is a block-structured language and BPEL4WS is a combination block- and graph-structured language. In addition, these languages define the behavior of a business process in a very compact and efficient manner. Given the nature of these languages, a complex business process will be organized in a potentially complex, disjointed, and unintuitive format that is handled very well by a software system (or a computer programmer), but would be hard to understand by the business analysts and managers tasked to develop and manage the business process. Thus, there is a human level of interoperability that is not addressed by these web service-based XML execution languages.

Humans tend to visualize business processes in a flow-chart format. There are thousands of business analysts studying the way companies work and defining business processes with simple flow charts. There is a technical gap between the format of the initial design of business processes and the format of the languages that will execute these business processes. This gap needs to be bridged with a formal mechanism that maps the appropriate visualization of the business processes (a notation) to the appropriate execution format (a BPM execution language) for these business processes.

Interoperation of business processes at the human level, rather than the software engine level, can be solved with standardization of the Business Process Modeling Notation (BPMN). BPMN provides a Business Process Diagram (BPD), which is a diagram designed for use by the people who design and manage business processes. BPMN also provides a formal mapping to execution languages of BPM Systems, such as BPEL4WS and BPML. Thus, BPMN would provide a standard visualization mechanism for business processes defined in an execution optimized business process language.

BPMN will provide businesses with the capability of understanding their internal business procedures in a graphical notation and will give organizations the ability to communicate these procedures in a standard manner. Furthermore, the graphical notation will facilitate the understanding of the performance collaborations and business transactions between the organizations. This will ensure that businesses will understand themselves and participants in their business and will enable organizations to adjust to new internal and B2B business circumstances quickly. To do this, BPMN will follow the tradition of flowcharting notations for readability; yet still provide the mapping to the executable constructs. BPMI is using the experience of the business process notations that have preceded BPMN to create the next generation notation that combines readability, flexibility, and expandability.

BPMN will also advance the capabilities of traditional business process notations by inherently handling B2B business process concepts, such as public and private processes and choreographies, as well as advanced modeling concepts, such as exception handling and transaction compensation.

2.1 BPMN Scope

BPMN will be constrained to support only the concepts of modeling that are applicable to business processes. This means that other types of modeling done by organizations for business purposes will be out of scope for BPMN. For example, the modeling of the following will not be a part of BPMN:

- Organizational structures
- Functional breakdowns
- Data models

In addition, while BPMN will show the flow of data, it is not a data flow diagram.

2.1.1 Uses of BPMN

Business process modeling is used to communicate a wide variety of information to a wide variety of audiences. BPMN is designed to cover this wide range of usage and allows modeling of end-to-end business processes to allow the viewer of the diagram to be able to easily differentiate between the sub-model sections of a BPMN diagram.

There are three basic types of sub-models within an end-to-end BPMN model:

- Private (internal) business processes
- Interface (public/abstract) processes
- Collaboration Processes

Some BPMN specification terms regarding the use of swimlanes (e.g., Pools and Lanes) are used in the descriptions below. Refer to the section entitled "Pools and Lanes" on page 64 for more details on how these elements are used in a BPD.

Private Business Processes

Private business processes are those internal to a specific organization and are the types of processes that have been generally called workflow or BPM processes. A single private business process will map to a single BPEL4WS or BPML document.

If swimlanes are used then a private business process will be contained within a single Pool. The Sequence Flow of the Process is therefore contained within the Pool and cannot cross the boundaries of the Pool. Message Flow can cross the Pool boundary to show the interactions that exist between separate private business processes. Thus, a single BPMN diagram may show multiple private business processes.

Interface Processes

This is also called an abstract process and this represents the interactions between a private business process and another process or participant. Only those activities that are used to communicate outside the private business process are included in the interface process. All other "internal" activities of the private business process are not shown in the interface process. Thus, the interface process shows to the outside world the sequence of messages that are required to interact with that business process. A single interface process may be mapped to a single WSCI document (however, this mapping will not be done in this specification).

Interface processes are contained within a Pool and can be modeled separately or within a larger BPMN diagram to show the Message Flow between the interface process activities and other entities. If the interface process is in the same diagram as its corresponding private business process, then the activities that are common to both processes can be linked together.

Note: The mechanisms for defining how the activities can be linked has not been defined and is an open issue. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN.

Collaboration Processes

A collaboration process depicts the interactions between two or more business entities. These interactions are defined as a sequence of activities that represent the messages being sent between the entities involved. A single collaboration process may be mapped to an ebXML, RosettaNet, or WSCI global model process (however, these mappings are outside the scope of this specification).

Collaboration processes are contained within a Pool and the different participant business roles are shown as Lanes within the Pool. These processes can be modeled separately or within a larger BPMN diagram to show the Message Flow between the collaboration process activities and other entities. If the collaboration process is in the same diagram as one of its corresponding private business process, then the activities that are common to both processes can be linked together.

Note: The mechanisms for defining how the activities can be linked has not been defined and is an open issue. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN.

Types of BPD Diagrams

Within and between these three BPMN sub-models, many types of diagrams can be created. The following are the types of business processes that can be modeled with BPMN (those with asterisks will not map to an executable language):

- High-level private process activities (not functional breakdown)*
- Detailed private business process
    - As-is or old business process*
    - To-be or new business process
- Detailed private business process with interactions to one or more external entities (or "Black Box" processes)
- Two or more detailed private business processes interacting
- Detailed private business process relationship to Interface Process

- Detailed private business process relationship to Collaboration Process
- Two or more Interface Processes—not executable
- Interface Process relationship to Collaboration Process*
- Collaboration Process only (e.g., ebXML BPSS or RosettaNet)*
- Two or more detailed private business processes interacting through their Interface Processes
- Two or more detailed private business processes interacting through a Collaboration Process
- Two or more detailed private business processes interacting through their Interface Processes and a Collaboration Process

BPMN is designed to allow all the above types of diagrams. However, it should be cautioned that if too many types of sub-models are combined, such as three or more private processes with message flow between each of them, then the diagram may become too hard for someone to understand. Thus, we recommend that the modeler pick a focused purpose for the BPD, such as a private process, or a collaboration process.

BPMN mappings

Since BPMN covers such a wide range of usage, it will map to more than one lower-level specification language:

- BPEL4WS and BPML are the primary languages that BPMN will map to, but they only cover a single executable private business process. If a BPMN diagram depicts more than one internal business process, then there will a separate mapping for each on the internal business processes.
- The interface sections of a BPMN diagram will be mapped to Web service interfaces specifications, such as the abstract processes of BPEL4WS and WSCI.
- The Collaboration model sections of a BPMN will be mapped Collaboration models such as ebXML BPSS and RosettaNet.

This specification will only cover the mappings to BPEL4WS and BPML. Mappings to other specifications will have to be a separate effort, or perhaps a future direction of BPMN (beyond Version 1.0 of the BPMN specification). It is hard to predict which mappings will be applied to BPMN at this point, since process language specifications is a volatile area of work, with many new offerings and mergings.

A BPD is not designed to graphically convey all the information required to execute a business process. Thus, the graphic elements of BPMN will be supported by properties that will supply the additional information required to enable a mapping to BPEL4WS and BPML.

2.1.2 Diagram Point of View

Since a BPMN diagram may depict the Processes of different Participants, each Participant may view the diagram differently. That is, the Participants have different points of view regarding how the Processes will behave. Some of the activities will be internal to the Participant (meaning performed by or under control of the Participant) and other activities will be external to the Participant. Each Participant will have a different perspective as to which are internal and external. At runtime, the difference is important in how a Participant can view the status of the activities or trouble-shoot any problems. However, the diagram itself remains the same.

Although the diagram point of view is important for a viewer of the diagram to understand how the behavior of the Process will relate to that viewer, BPMN will not currently specify any graphical mechanisms to highlight the point of view. It is open to the modeler or modeling tool vendor to provide any visual cues to emphasize this characteristic of a diagram.

2.1.3 Extensibility of BPMN and Vertical Domains

BPMN is intended to be extensible by modelers and modeling tools. This extensibility allows modelers to add non-standard elements or artifacts to satisfy a specific need, such as the unique requirements of a vertical domain. While extensible, BPMN diagrams should still have the basic look-and-feel so that a diagram by any modeler should be easily understood by any viewer of the diagram.

The graphical elements of BPMN are designed to be open to allow specialized markers to convey specialized information. For example, the three types of Events all have open centers for the markers that BPMN standardizes as well as user-defined markers.

3. Business Process Diagram Concepts

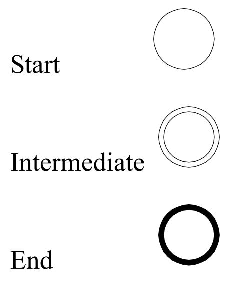
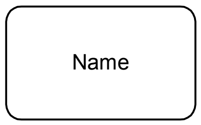
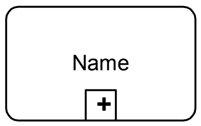
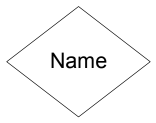
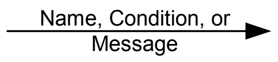
This section provides a summary of the BPMN graphical objects and their relationships. More details on the concepts will be provided in “Business Process Diagram Graphical Objects” on page 27 and “Connecting Objects” on page 73.

One of the goals of BPMN is that the notation be simple and adoptable by business analysts. Also, there is a conflicting requirement that BPMN provide the power to depict complex business processes and map to BPM execution languages. To help understand how BPMN can manage both requirements, the list of BPMN graphic elements is presented in two groups.

First, there is the list of core elements that will support the requirement of a simple notation. These are the elements that define the basic look-and-feel of BPMN. Most business processes will be modeled adequately with these elements. Second, there is the entire list of elements, including the core elements, which will help support requirement of a powerful notation to handle more advanced modeling situations.

3.1 BPD Core Element Set

Table 1 displays a list of the core business process concepts that are depicted through the notation:

Element	Description	Notation
Event (three types)	An event is something that “happens” during the course of a business process. These events affect the flow of the process and usually have a cause or an impact. There are three types of events in terms of how they affect the flow: start, intermediate, and end.	
Task (atomic)	A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail.	
Sub-Process (Compound)	A Sub-Process is a compound activity that is included within a Process. It is compound in that it is broken down into a finer level of detail through a set of sub-activities.	
Decision	Decisions are locations within a business process where the flow of control can take two or more alternative paths.	
Sequence Flow	A Sequence Flow is used to show the order that activities will be performed in a Process.	

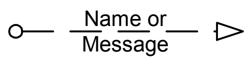
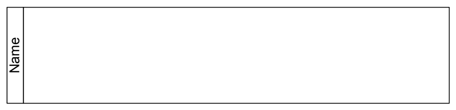
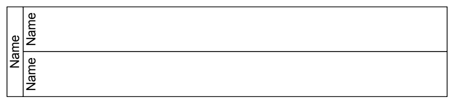
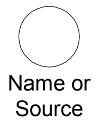
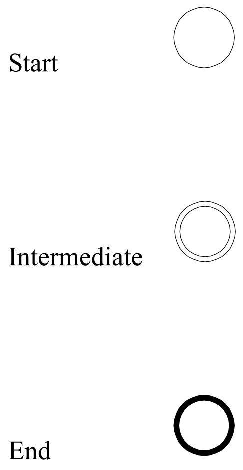
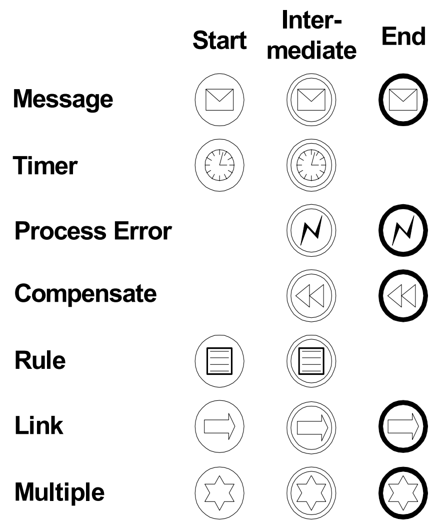
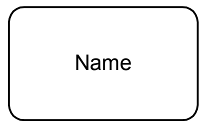
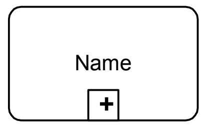
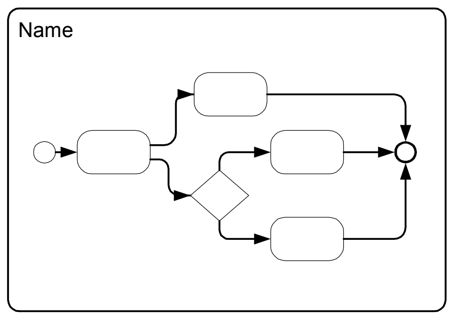
Message Flow	A Message Flow is used to show the flow of messages between two entities that are prepared to send and receive them. In BPMN, two separate Pools in the diagram will represent the two entities.	
Pool	A Pool is a "swimlane" and a graphical container for partitioning a set of activities from other Pools, usually in the context of B2B situations.	
Lanes	A Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally. Lanes are used to organize and categorize activities within a Pool.	

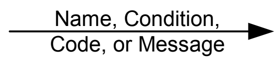
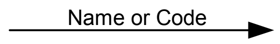
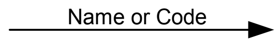
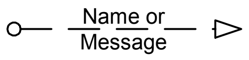
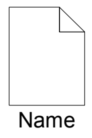
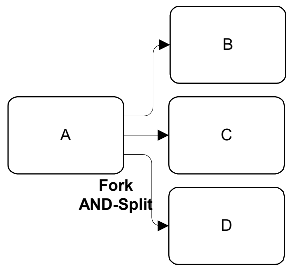
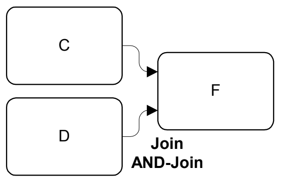
Table 1 BPD Core Element Set

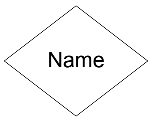
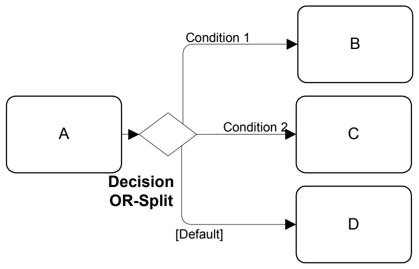
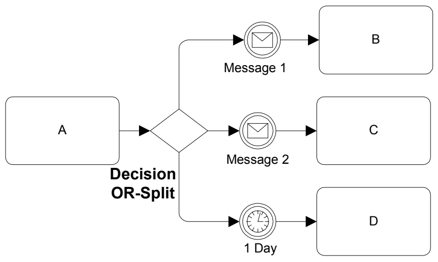
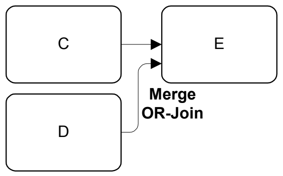
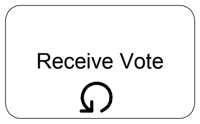
3.2 BPD Complete Set

Table 2 displays a more extensive list of the business process concepts that could be depicted through a business process modeling notation.

Element	Description	Notation
Event	An event is something that "happens" during the course of a business process. These events affect the flow of the process and usually have a cause or an impact. There are three types of events in terms of how they affect the flow: start, intermediate, and end.	
Flow Dimension (e.g., Start, Intermediate, End) Start (Message, Timer, Rule, Link, Multiple) Intermediate (Message, Timer, Process Error, Compensate, Rule, Link, Multiple) End (Message, Process Error, Compensate, Link, Multiple)	As the name implies, the Start Event indicates where a particular process will start. Intermediate Events occur between a Start Event and an End Event. This is an event that occurs after a Process has been started. It will affect the flow of the process, but will not start or (directly) terminate the process. As the name implies, the End Event indicates where a process will end.	

Type Dimension (e.g., Message, Timer, Process Error, Compensate, Rule, Link, Multiple)	Start and Intermediate Events have "Triggers" that define the cause for the event. There are multiple ways that these events can be triggered. End Events may define a "Result" that is a consequence of a Sequence Flow ending.	 Start Inter-mediate End Message Timer Process Error Compensate Rule Link Multiple
Task (Atomic)	A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail.	
Process/Sub-Process (non-atomic)	A Sub-Process is a compound activity that is included within a Process. It is compound in that it is broken down into a finer level of detail through a set of sub-activities.	See Next Two Figures
Collapsed Sub-Process	The details of the Sub-Process are not visible in the diagram.	
Expanded Sub-Process	The boundary of the Sub-Process is expanded and the details of the Sub-Process are visible within its boundary.	
Sequence Flow	A Sequence Flow is used to show the order that activities will be performed in a Process.	See next three figures

Normal flow	Normal sequence flow refers to the flow that originates from a Start Event and continues through activities via alternative and parallel paths until it ends at an End Event. Conditions (or guards) are only available for Flows exiting a Decision.	
Exception flow	Exception flow occurs outside the normal flow of the Process and is based upon an event (an Intermediate Event) that occurs during the performance of the Process.	
Transaction Compensation flow	Transaction Compensation Flow occurs outside the normal flow of the Process and is based upon an event (an Intermediate Event) that is triggered during the rolling back of a Process that has started, but is later cancelled.	
Message Flow	A Message Flow is used to show the flow of messages between two entities that are prepared to send and receive them. In BPMN, two separate Pools in the diagram will represent the two entities.	
Data Object	Data Objects are considered artifacts because they do not have any direct affect on the Sequence Flow or Message Flow of the Process, but they do provide information about what the Process does.	?
Fork (AND-Split)	BPMN uses the term forking to refer to the dividing of a path into two or more parallel paths (also known as an AND-Split). It is a place in the Process where activities can be performed concurrently, rather than serially.	
Join (AND-Join)	BPMN uses the term joining to refer to the combining of two or more parallel paths into one path (also known as an AND-Join). The Join mechanism is an Open Issue.	

Decision, Branching Point; (OR-Split)	Decisions are locations within a business process where the flow of control can take two or more alternative paths.	
Data-Based Exclusive Condition Default	The set of Decision Alternatives for Data-Based Exclusive Decisions are based on condition expressions. These expressions evaluate the current values of process data to determine which path should be taken. This means that if none of the other condition expressions is true at runtime, then the default expression will be chosen.	 Condition 1 B A Condition 2 C Decision OR-Split [Default] D
Event-Based Exclusive	This Decision represents a branching point in the process where the Alternatives are based on an Intermediate Event that occurs at that point in the Process. The specific Intermediate Event, usually a message type, determines which of the paths will be taken.	 B Message 1 A C Message 2 Decision OR-Split D 1 Day
Inclusive	An Inclusive Decision is a hybrid between a Fork (AND-Split) and a Decision (OR-Split). In some sense it is a grouping of related independent Binary (Yes/No) Decisions. Since each path is independent, all combinations of the paths may be taken, from one to all.	Notation TDB
Merging (OR-Join)	BPMN uses the term merging to refer to the combining of two or more alternative paths into one path (also known as an a OR-Join). The Merge mechanism is an Open Issue.	
Looping; Multiple Instances	BPMN provides 2 (two) mechanisms for looping within a Process.	See Next Two Figures
Activity Looping	The properties of Tasks and Sub-Processes will determine if they are repeated or performed once. There are two types of loops: Standard and ForEach.	

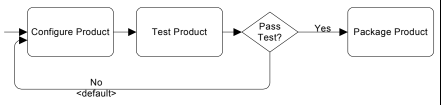
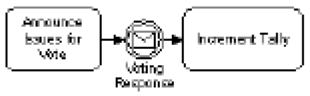
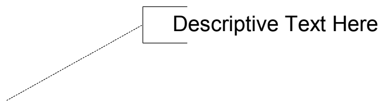
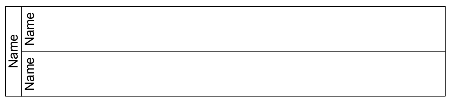
Sequence Flow Looping	Loops can be created by connecting a Sequence Flow to an “upstream” object. An object is considered to be upstream if that object has an outgoing Sequence Flow that leads to a series of other Sequence Flows, the last of which is an incoming Sequence Flow to the original object.	 Configure Product Test Product Pass Test? Yes Package Product No <default>
Process Break (something out of the control of the process makes the process pause)	A Process Break is a graphical marker that shows where an expected delay will occur within a Process.	 Announce Issues for Vote Voting Response Increment Tally Other Notation TBD
Transaction group/context		Notation TBD
Nested Process (Inline Block)		Notation TBD, if at all
Group (a box around a group of objects for documentation purposes)	A grouping of activities that does not affect the Sequence Flow. The grouping is generally for documentation or analysis purposes.	Notation TBD, if at all
Off-Page Connector (used within a page?)	Generally used for printing, this object will show where the Sequence Flow leaves one page and then restarts on the next page.	Notation TBD, if at all
Association	An Association is used to associate information with flow objects. Text and graphical non-flow objects can be associated with the flow objects.	------------------------- ----------------------->
Text Annotation (attached with an Association)	Text Annotations are a mechanism for a modeler to provide additional information for the reader of a BPMN diagram.	
Pool	A Pool is a “swimlane” and a graphical container for partitioning a set of activities from other Pools, usually in the context of B2B situations.	
Lanes	A Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally. Lanes are used to organize and categorize activities within a Pool.	

Table 2 BPD Complete Element Set

3.3 Use of Text, Color, and Lines in a Diagram

Flow objects and Flows can have labels (e.g., its name) placed inside the shape, or above or below the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor. Text Annotation objects can be used by the modeler to display additional information about a Process or properties of the objects within the Process.

3.4 Flow Object Connection Rules

An incoming Sequence Flow can connect to any location on a flow object (left, right, top, or bottom). Likewise, an outgoing Sequence Flow can connect from any location on a flow object (left, right, top, or bottom). Message Flows also have this capability. BPMN allows this flexibility, however, we also recommend that modelers use judgment in how flow objects should be connected so that readers of the diagrams will find the behavior clear and easy to follow. This is even more important when a diagram contains Sequence Flows and Message Flows. In these situations it is best to pick a direction of Sequence Flow, either left to right or top to bottom, and then direct the Message Flow at a 90° angle to the Sequence Flow. The resulting diagrams will be much easier to understand.

3.4.1 Sequence Flow Rules

Table 3 displays the BPMN flow objects and shows how these objects can connect to one another through Sequence Flows. The ↗ symbol indicates that the object listed in the row can connect to the object listed in the column. The quantity of connections into an object is specified in the column header with a code letter that precedes the graphical shape. The quantity of connections out of an object is specified in the row header with a code letter that follows the graphical shape. The code letters are: 0 (No Connections); 1 (One Connection); M (Multiple Connections); and M(E) (Multiple Exclusive Connections). *Note that if a sub-process has been expanded within a diagram, the objects within the sub-process cannot be connected to objects outside of the sub-process. Nor can Sequence Flows cross a Pool boundary.*

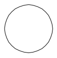
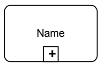
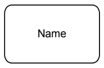
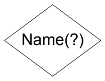
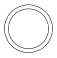
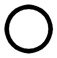
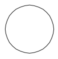
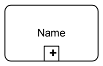
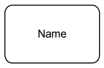
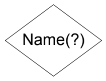
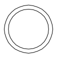
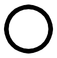
From\To	0 	M 	M 	M 	1 	M 
 M		↗	↗	↗	↗	
 M		↗	↗	↗	↗	↗
 M		↗	↗	↗	↗	↗
 M(E)		↗	↗	↗	↗	↗
 M		↗	↗	↗	↗	↗
 0						

Table 3 Sequence Flow Connection Rules

Note: Only those objects that can have incoming and/or outgoing Sequence Flow are shown in the table. Thus, Pool, Lane, Data Object, and Text Annotation are not listed in the table.

3.4.2 Message Flow Rules

Table 4 displays the BPMN modeling objects and shows how these objects can connect to one another through Message Flows. The ⬀ symbol indicates that the object listed in the row can connect to the object listed in the column. The quantity of connections into an object is specified in the column header with a code letter that precedes the graphical shape. The quantity of connections out of an object is specified in the row header with a code letter that follows the graphical shape. The code letters are: 0 (No Connections); 1 (One Connection in a single direction); M (Multiple Connections in a single direction). *Note that Message Flows cannot connect to objects that are within the same Participant Lane boundary.*

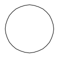
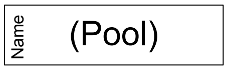
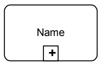
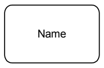
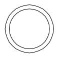
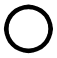
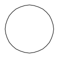
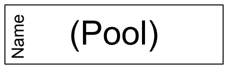
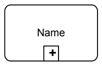
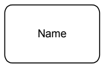
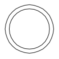
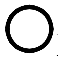
From\To	M 	M 	M 	0-1 	1 	0 
 0						
 M	⇗	⇗	⇗	⇗	⇗	
 M	⇗	⇗	⇗	⇗	⇗	
 0-1	⇗	⇗	⇗	⇗	⇗	
 0						
 M	⇗	⇗	⇗	⇗	⇗	

Table 4 Message Flow Connection Rules

Note: Only those objects that can have incoming and/or outgoing Message Flow are shown in the table. Thus, Lane, Decision, Data Object, and Text Annotation are not listed in the table.

4. Business Process Diagram Graphical Objects

This section details the graphical representation and the semantics of the behavior of BPD elements.

4.1 Events

An Event is something that “happens” during the course of a business process. These Events affect the flow of the Process and usually have a cause or an impact. The term “event” is general enough to cover many things in a business process. The start of an activity, the end of an activity, the change of state of a document, a message that arrives, etc., all could be considered events. However, BPMN has restricted the use of events to include only those types of events that will affect the sequence or timing of activities of a process. BPMN further categorizes Events into three main types: Start, Intermediate, and End.

Start and Intermediate Events have “Triggers” that define the cause for the event. There are multiple ways that these events can be triggered (refer to the section entitled “Start Event Triggers” on page 29 and “Intermediate Event Triggers” on page 39). End Events may define a “Result” that is a consequence of a Sequence Flow ending. There are multiple types of Results that can be defined (refer to the section entitled “End Event Results” on page 34).

All Events share the same shape footprint, a small circle. Different line styles, as shown below, distinguish the three types of flow Events. All Events also have an open center so that BPMN-defined and modeler-defined icons can be included within the shape to help identify the Trigger or Result of the Event.

4.1.1 Start

As the name implies, the Start Event indicates where a particular Process will start. In terms of sequence flow, the Start Event starts the flow of the Process, and thus, will not have any incoming Sequence Flows—no Sequence Flows can connect to a Start Event.

The Start Event shares the same basic shape of the Intermediate Event and End Event, a circle, but is drawn with a single thin line (see Figure 1). Text associated with the Start Event (e.g., its name) can be placed above or below the shape, in any direction or location, or on the outgoing Sequence Flow, depending on the preference of the modeler or modeling tool vendor.

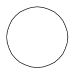


Figure 1 A Start Event

Throughout this document, we will discuss how Sequence Flow proceeds within a Process. To facillitate this discussion, we will employ the concept of a “**Token**” that will traverse the Sequence Flows and pass through the flow objects in the Process. The behavior of the Process can be described by tracking the path(s) of the Token through the Process. A

Token will have a unique identity, called a TokenID set, that can be used to distinguish multiple Tokens that may exist because of concurrent Process instances or the dividing of the Token for parallel processing within a single Process instance. The parallel dividing of a Token creates a lower level of the TokenID set. The set of all levels of TokenID will identify a Token. The TokenID set for a Token will be in the following format: “TokenID.TokenID. … TokenID,” each level being separated by a dot.

A Start Event generates a Token that must eventually be consumed at an End Event (which may be implicit if not graphically displayed). Tokens can also be consumed through exception handling Intermediate Events, which act like a forced end to a Process level. Note: A Token does not traverse the Message Flows since it is a Message that is passed down those Flows (as the name implies).

Semantics of the Start Event include:

- *This shape is optional*—a Process level (a top-level Process or an expanded Sub-Process) is not required to have this shape:
  - If there is a Start Event, then there has to be at least one End Event.
  - If the Start Event is used, then there can be no other flow elements that do not have incoming Sequence Flow (of those elements that can accept Sequence Flow)—all other flow objects must be a target of at least one Sequence Flow.
    - An exception to this is the Intermediate Event, which can be without an incoming Sequence Flow.
  - If the Start Event is *not* used, then all flow objects that do not have an incoming Sequence Flow (i.e., are not a target of a Sequence Flow) will be instantiated when the Process is instantiated. There is an assumption that there is only one implicit Start Event, meaning that all the starting flow objects will start at the same time.
- There can be multiple Start Events for a given Process level. Each Start Event is an independent event.

Note: A BPD may have more than one Process level (i.e., it can include Expanded Sub-Processes). The use of Start and End Events is independent for each level of the diagram.

That is, when the trigger for a Start Event occurs, Tokens will be generated for each outgoing Sequence Flow from that event. The TokenID set for each of the Tokens will be established such that it can be identified that the Tokens are all from the same parallel Fork (AND-Split) and the number of Tokens in the group. These Tokens will begin their flow and not wait for any other Start Event to be triggered.

If there is a dependency for more than one Event to happen before a Process can start (e.g., two messages are required to start), then the Start Events must flow to the same activity within that Process. The attributes of the activity would specify when the activity could begin. If the attributes specify that the activity must wait for all inputs, then all Start Events will have to be triggered before the Process begins (refer to the section entitled “Attributes” on page 47 (for sub-processes) and “Attributes” on page 53 (for Tasks) for more information about activity attributes). In addition, a correlation mechanism will be required so that different triggered Start Events will apply to the same process instance. Correlation will likely be handled through Event attributes, but this an open issue will be addressed in a later version of the specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

❖ A sub-process, which is a process within a process, if expanded to show its detail, can also have Start Events.

Start Event Triggers

There are many ways that can business process can be started (instantiated). The Trigger for a Start Event is designed to show the general mechanism that will instantiate that particular Process. There are six types of Start Events in BPMN: None, Message, Timer, Rule, Link, and Multiple.

Table 5 displays the types of Triggers and the graphical marker that will be used for each:

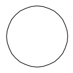
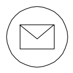
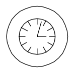
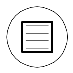
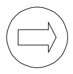
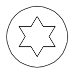
Trigger	Description	Marker
None	The modeler does not display the type of Event. It is also used for a Sub-Process that starts when the flow is triggered by its Parent Process.	
Message	A message arrives from a participant and triggers the start of the Process.	
Timer	A specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the start of the Process.	
Rule	This type of event is triggered when the conditions for a rule such as “S&P 500 changes by more than 10% since opening,” or “Temperature above 300C” become true.	
Link	A Link is a mechanism for connecting the end (Result) of one Process to the start (Trigger) of another. Typically, these are two Sub-Processes within the same parent Process.	
Multiple	This means that there are multiple ways of triggering the Process. Only one of them will be required to start the Process. The attributes of the Start Event will define which of the other types of Triggers apply.	

Table 5 Start Event Types

Attributes

The following are identified attributes of a Start Event:

Attribute	Description
Name ?	Name is an optional property that is text description of the Event.
Trigger (None | Message | Timer | Rule | Link | Multiple) : None	Trigger is a property (default None) that defines the type of trigger expected for that Start.

Attribute	Description
Message: MessageName	If the Trigger is a Message, then the name of the Message must be supplied.
Timer: (Timedate | TimeCycle): Timedate	If the Trigger is a Timer, then a timedate or a timedatecycle must be entered.
Rule: RuleExpression	If the Trigger is a Rule, then an expression must be entered.
Link: LinkName	If the Trigger is a Link, then the name of the Link must be supplied.
Multiple: Trigger + (except Multiple)	If the Trigger is a Multiple, then a list of the Trigger must have the appropriate data.
Assign *: Expression	Zero or more assignments can be made. Each assignment is an expression.
OutgoingSequenceFlow +: SequenceFlowName	One or more outgoing Sequence Flows can be idenitified for the Start Event.
IncomingMessageFlow *: MessageFlowName	Zero or more incoming Message Flows can be idenitified for the Start Event, but the Trigger type must be Message or Multiple (with Message as one of Trigger types). For a Message Flow there will be an associated BPEL4WS *receive* or a BPML *one-way action* to receive the message. For multiple Message Flows, there will be BPEL4WS *pick* or BPML *choice* that will receive only one of the messages.
Pool ?: PoolName	If Pools are used, then the PoolName must be added to the Start Event to identify its location.
Lane ?: LaneName	If that Pool has more than one Lane, then the LaneName must be added.
Association *	Zero or more Associations can be associated with the Start Event.
Documentation ?	The modeler can add optional text documentation about the Start Event.

Table 6 Start Event Attributes

Sequence Flow Connections

Refer to the section entitled “Sequence Flow Rules” on page 24 for the entire set of objects and how they may be source or targets of Sequence Flows.

- ❖ A Start Event cannot be a target for a Sequence Flow; there can be no incoming Sequence Flows.
- ❖ A Start Event can be a source for a Sequence Flow; multiple Sequence Flows can originate from a Start Event. For each Sequence Flow that has the Start Event as a source, there will be a new parallel path generated.
  - ❖ When a Start Event is not used, then all flow objects that do not have an incoming Sequence Flow will be the start of a separate parallel path.

Each path will have a separate unique Token that will traverse the Sequence Flow.

Message Flow Connections

Refer to the section entitled “Message Flow Rules” on page 25 for the entire set of objects and how they may be source or targets of Sequence Flows.

Note: All Message Flows described here must connect two separate Pools. They can connect to the Pool boundary or to flow objects within the Pool boundary. They cannot connect two objects within the same Pool.

- ❖ A Start Event can be the target for Message Flows; it can have 0 (zero) or more incoming Message Flows. Each Message Flow arriving at a Start Event represents an instantiation mechanism (a Trigger) for the process. to see how the incoming Message Flows are mapped to BPEL4WS and BPML elements.
  - ❖ The trigger property of the Start Event must be set to Message or Multiple if there are any incoming Message Flows.
- ❖ A Start Event cannot be a source for a Message Flow; it can have no outgoing Message Flows.

Mapping to Execution Languages

The following two sections describe how the use of Start Events will map to BPEL4WS and BPML, respectively.

BPEL4WS

- ❖ If the Start Event has an expression for the assign property, then this will map to a BPEL4WS *assign*.

Each type of Start Event Trigger will have a different mapping to BPEL4WS:

- ❖ None: this does not map to any BPEL4WS element.
- ❖ Message: A *receive* will be associated with the message defined with the Message Flow that arrives at the Start Event (see Figure 2).

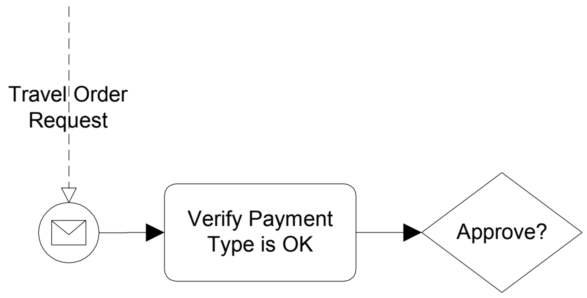


Figure 2 Message Flow connected to a Start Event

  - ❖ If there is more than one connected to the Start Event, then a BPEL4WS *pick* will be required to process the messages with a separate *receive* for each message. This means that a single instance of the process will be instantiated when the first message received through the *pick receives* arrives.

Note: The modeler does not need to connect the Message Flows to the Start Event to model this behavior, however. The receipt of the messages could be spelled out through the modeling of the receiving Tasks as graphical objects (see Figure 3) and using a None Start Event.

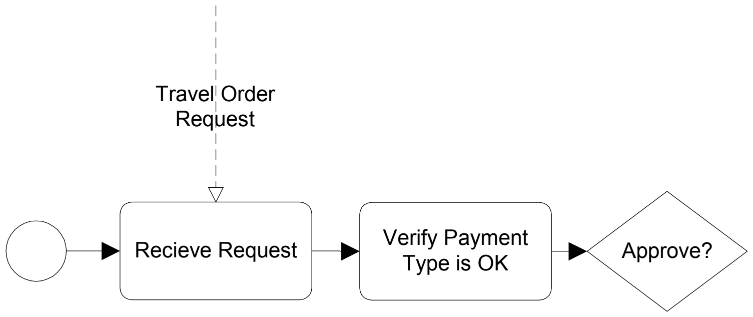


Figure 3 Process Instantiation through Message Receiving Task

- ❖ Timer: TBD.
- ❖ Rule: TBD.
- ❖ Link: this will map to the *receive* element.
- ❖ Multiple: this will map to a combination of *receive* elements.

BPML

- ❖ If the Start Event has an expression for the assign property, then this will map to a BPML *assign*.

Each type of Start Event Trigger will have a different mapping to BPML:

- ❖ None: this does not map to any BPML element.
- ❖ Message: A *one-way action* will be associated with the message defined with the Message Flow that arrives at the Start Event (see Figure 2).
  - ❖ If there is more than one connected to the Start Event, then a BPML *choice* will be required to process the messages with a separate *one-way action* for each message. This means that a single instance of the process will be instantiated when the first message received through the *choice actions* arrives.

Note: The modeler does not need to connect the Message Flows to the Start Event to model this behavior, however. The receipt of the messages could be spelled out through the modeling of the receiving Tasks as graphical objects (see Figure 3).

- Timer: this will map to a *faults case* that is triggered by a *schedule* element within a *context*.
- Rule: TBD.
- Link: this will map to the *signal* within an *event* element.
- Multiple: this will map to a combination of *action*, *schedule*, *signal*, and TBD elements.

4.1.2 End

As the name implies, the End Event indicates where a process will end. In terms of sequence flow, the End Event ends the flow of the Process, and thus, will not have any outgoing Sequence Flows—no Sequence Flows can connect from an End Event.

The End Event shares the same basic shape of the Start Event and Intermediate Event, a circle, but is drawn with a thick single line (see Figure 4). Text associated with the End Event (e.g., its name) can be placed above or below the shape, in any direction or location, or on the incoming Sequence Flow, depending on the preference of the modeler or modeling tool vendor.

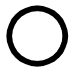


Figure 4 End Event

To continue the discussing how flow proceeds throughout the process, an End Event consumes a Token that had been generated from a Start Event within the same level of Process. If parallel Sequence Flows target the End Event, then the Tokens will be consumed as they arrive. All the Tokens that were generated from the Start Events or through forking during the Process must be consumed before the Process has been completed.

- There can be multiple End Events within a single level of a process.
- *This shape is optional*—a given Process level (a top-level Process or an expanded Sub-Process) is not required to have this shape:
  - If there is an End Event, then there has to be at least one Start Event.
  - If an End Event is used, then there can be no other flow elements that do not have any outgoing Sequence Flows (of those elements that can generate Sequence Flow)—all other flow objects must be a source of at least one Sequence Flow.
  - If the End Event is not used, then all flow objects that do not have any outgoing Sequence Flows (i.e., are not a source of a Sequence Flow) mark the end of the process. However, the process will not end until all parallel paths have completed.

Note: A BPD may have more than one Process level (i.e., it can include Expanded Sub-Processes). The use of Start and End Events is independent for each level of the diagram.

A Token entering the path-ending flow objects will be consumed when the processing performed by those objects are completed (when the path has completed). When all

Tokens for a given instance of the Process are consumed, then the Process will reach a state of being completed.

End Event Results

A BPMN modeler can define the consequence of reaching an End Event. This will be referred to as the End Event Result.

Table 7 displays the types of Results and the graphical marker that will be used for each:

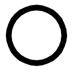
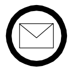
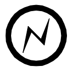
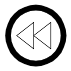
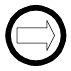
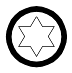
Result	Description	Marker
None	The modeler does not display the type of Event. It is also used for a Sub-Process that end and the flow goes back to its Parent Process.	
Message	A message is sent to a participant at the conclusion of the Process.	
Process Error	This particular End will inform the Process Engine that named Error should be generated. This Error will be caught by an Intermediate Event within the Event Context.	
Compensate	This particular End will inform the Process Engine that a Compensation is necessary. This Compensate identifier will be used by an Intermediate Event when the Process is rolling back.	
Link	A Link is a mechanism for connecting the end (Result) of one Process to the start (Trigger) of another. Typically, these are two Sub-Processes within the same parent Process.	
Multiple	This means that there are multiple consequences of ending the Process. All of them will occur. The attributes of the End Event will define which of the other types of Results apply.	

Table 7 End Event Types

Attributes

The following are identified attributes of an End Event:

Attribute	Description
Name ?	Name is an optional property that is text description of the Event.
Result: (None | Message | ProcessError | Compensate | Rule | Link | Multiple) : None	Result is a property (default None) that defines the type of result expected for that End.
Message: MessageName	If the Result is a Message, then the name of the Message must be supplied.
ProcessError: ErrorCode	If the Result is a Process Error, then the error code must be supplied.
Compensate: CompensateCode	If the Result is a Compensate, then the compensation code must be supplied.
Link: LinkName	If the Result is a Link, then the name of the Link must be supplied.
Multiple: Trigger + (except Multiple)	If the Result is a Multiple, then a list of the Results must have the appropriate data.
Assign *: Expresssion	Zero or more assignments can be made. Each assignment is an expression.
IncomingSequenceFlow +: SequenceFlowName	One or more incoming Sequence Flows can be idenitified for the End.
FlowCondition: (One | All | Complex) : All	If there is more than one, or if there are more than one End Events, then a Flow Condition must be set. The Flow Condition will apply to all End Events for that level of the Process. A Flow Condition of One means that flow will continue up to the Parent Process when one Token arrives. The process will continue and all other Tokens arriving at will be consumed, but no other Token will proceed up to the Parent Process. A Flow Condition of All means that all Tokens generated at that level of the Process must be consumed before a Token is passed back up to the Parent Process.
Complex: Expression	A complex Flow Condition can be set by the modeler. This will consist of an expression that can reference Sequence Flow names and or Process data.
PassThrough: (True | False): False	The definition of the PassThrough property is an open issue that will be handled in a later version of the specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.
OutgoingMessageFlow *: MessageFlowName	Zero or more outgoing Message Flows can be idenitified for the End, but the Result type must be Message or Multiple (with Message as one of Result types). For each Message Flow there will be an associated BPEL4WS *reply* or asynchronous *invoke* or a BPML *notification action* to send the message.

Attribute	Description
Pool ?: PoolName	If Pools are used, then the PoolName must be added to the End Event to identify its location.
Lane ?: LaneName	If that Pool has more than one Lane, then the LaneName must be added.
Association *	Zero or more Associations can be associated with the End Event.
Documentation ?	The modeler can add optional text documentation about the End Event.

Table 8 End Event Attributes

Sequence flow Connections

Refer to the section entitled "Sequence Flow Rules" on page 24 for the entire set of objects and how they may be source or targets of Sequence Flows.

- ❖ An End Event can be a target for a Sequence Flow; there can be multiple incoming Flows. The Flows can come from either alternative or parallel paths. For modeling convenience, each path can connect to a separate End Event object or one End Event can be used.

Thus, the End Event is used as a Sink for all Tokens that arrive at the Event. All Tokens that are generated at the Start Event for that Process must eventually arrive at an End Event or consumed through an exception handling Intermediate Event. The Process will be in a *running* state until all Tokens are consumed.

- ❖ An End Event cannot be a source for a Sequence Flow; there can be no outgoing Flows.

Message Flow Connections

Refer to the section entitled "Message Flow Rules" on page 25 for the entire set of objects and how they may be source or targets of Sequence Flows.

Note: All Message Flows described here must connect two separate Pools. They can connect to the Pool boundary or to flow objects within the Pool boundary. They cannot connect two objects within the same Pool.

- ❖ An End Event cannot be the target for Message Flows; it can have no incoming Message Flows.
- ❖ An End Event can be a source for a Message Flow; it can have one outgoing Message Flow.
    - ❖ However, if there is more than one End Event in the Process, then each of the End Events can have a different outgoing Message Flow.

Mapping to Execution Languages

The following two sections describe how the use of End Events will map to BPEL4WS and BPML, respectively.

BPEL4WS

- ❖ If the End Event has an expression for the assign property, then this will map to a BPEL4WS *assign*.

Each type of End Event Result will have a different mapping to BPML:

- ❖ None: this does not map to any BPEL4WS element. However, it marks the end of a path within the Process and will be used to define the boundaries of complex BPEL4WS elements.
- ❖ Message: A graphically hidden BPEL4WS *reply* will be associated with the message defined with the Message Flow that leaves the End Event (see Figure 5).

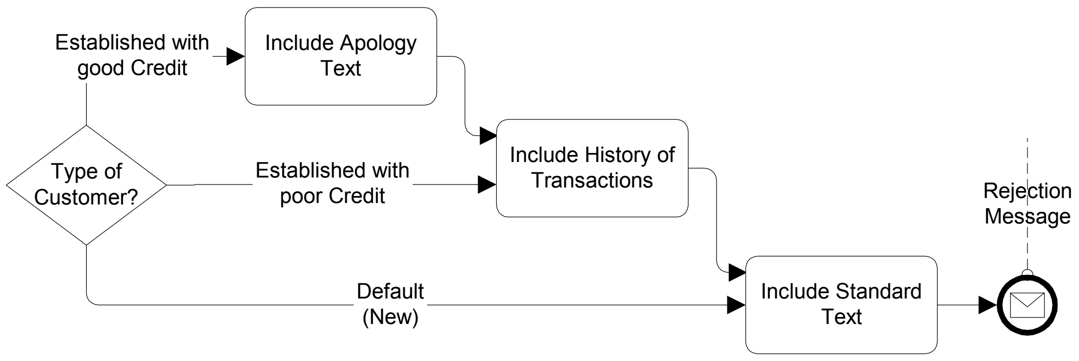


Figure 5 Message Flow leaving an End Event

Note: The modeler does not need to connect the Message Flows from the End Event to model this behavior, however. The sending of the messages could be modeled through the modeling of the sending Tasks as graphical objects (see Figure 6)

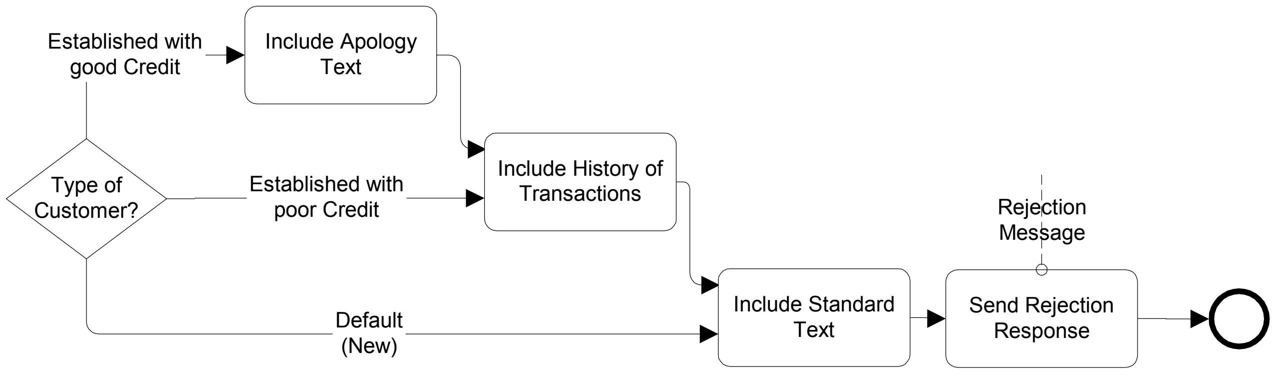


Figure 6 Message Flow from Task that precedes the End Event

- Process Error: this will map to a *throw* element.
- Compensate: this will map to a *compensate* element.
- Link: this will map to the *invoke* element.
- Multiple: this will map to a combination of *invoke*, *throw*, *fault*, and *compensate* elements.

BPML

- If the End Event has an expression for the assign property, then this will map to a BPML *assign*.

Each type of End Event Result will have a different mapping to BPML:

- None: this does not map to any BPML element. However, it marks the end of a path within the Process and will be used to define the boundaries of complex BPML elements.
- Message: A *notification action* will be associated with the message defined with the Message Flow that leaves the End Event (see Figure 5).

Note: The modeler does not need to connect the Message Flows from the End Event to model this behavior, however. The sending of the messages could be modeled through the modeling of the sending Tasks as graphical objects (see Figure 6).

- Process Error: this will map to a *fault* element.
- Compensate: this will map to a *compensate* element.
- Link: this will map to the *raise* element.
- Multiple: this will map to a combination of *action*, *raise*, *fault*, and *compensate* elements.

4.1.3 Intermediate

Intermediate Events occur between a Start Event and an End Event. This is an event that occurs after a Process has been started. It will affect the flow of the process, but will not start or (directly) terminate the process. Intermediate Events can be used to:

- Show where messages or delays are expected within the Process,
- Disrupt the normal flow through exception handling, or
- Show the extra work required for compensating a transaction.

The Intermediate Event shares the same basic shape of the Start Event and End Event, a circle, but is drawn with a thin double line (see Figure 7). Text associated with the Intermediate Event (e.g., its name) can be placed above or below the shape, in any direction or location, or on the outgoing Sequence Flow, depending on the preference of the modeler or modeling tool vendor.

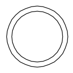


Figure 7 Intermediate Event

One use of Intermediate Events is to represent exception or transaction compensation handling. This will be shown by placing the Intermediate Event on the boundary of a Task or Sub-Process (either collapsed or expanded). Figure 8 displays an example of an Intermediate Event attached to a Task. The Intermediate Event can be attached to any location of the activity boundary and the outgoing Sequence Flow can flow in any direction. However, in the interest of clarity of the diagram, we recommend that the modeler choose a consistent location on the boundary. For example, if the diagram orientation is horizontal, then the Intermediate Events can be attached to the bottom of the activity and the Sequence Flow directed down and then to the right. If the diagram orientation is vertical, then the Intermediate Events can be attached to the left or right side of the activity and the Sequence Flow directed to the left or right and then down.

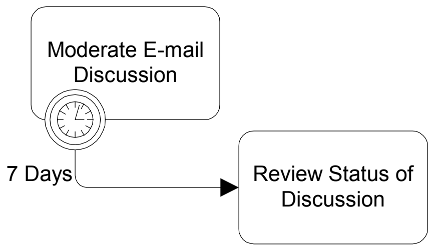


Figure 8 Task with an Intermediate Event attached to its boundary

Intermediate Event Triggers

There are seven types of Intermediate Events in BPMN: Message, Timer, Process Error (exception), Compensate, Rule, Link, and Multiple. These Event types indicate the different ways that a Process may be interrupted or delayed after it has started. Each type of Intermediate Event will have a different icon placed in the center of the Intermediate Event shape to distinguish one from another.

Table 9 displays the types of Triggers and the graphical marker that will be used for each:

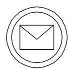
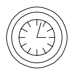
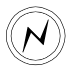
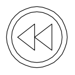
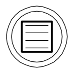
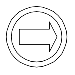
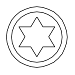
Trigger	Description	Marker
Message	A message arrives from a participant and triggers the Event. This causes the Process to continue if it was waiting for the message, or changes the flow for exception handling.	
Timer	A specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the Event. If used within the main flow it acts as a delay mechanism. If used for exception handling it will change the normal flow into an exception flow.	
Process Error	This is only used for exception handling. It reacts to a named Error (e.g., thrown from an End Event) or to any error if a name is not specified.	
Compensate	This is only used for transaction handling. It reacts to a named Compensate (e.g., thrown from an End Event).	
Rule	This is only used for exception handling. This type of event is triggered when a named Rule becomes true. A Rule is an expression that evaluates some Process data.	
Link	A Link is a mechanism for connecting the end (Result) of one Process to the start (Trigger) of Event-Based Exclusive Decision.	
Multiple	This means that there are multiple ways of triggering the Event. Only one of them will be required. The attributes of the Intermediate Event will define which of the other types of Triggers apply.	

Table 9 Intermediate Event Types

Attributes

The following are identified attributes of an Intermediate Event:

Attribute	Description
Name ?	Name is an optional property that is text description of the Event.
Trigger: (Message | Timer | ProcessError | Compensate | Rule | Multiple) : Message	Trigger is a property (default Message) that defines the type of trigger expected for that Intermediate Event.
Message: MessageName	If the Trigger is a Message, then the name of the Message must be supplied.
Timer: (Timedate | TimeCycle): Timedate	If the Trigger is a Timer, then a timedate or a timedatecycle must be entered.

Attribute	Description
ProcessError: (ErrorCode | None): ErrorCode	If the Trigger is a Process Error, then the error code can be supplied. If there is no error code, then any Error will trigger the Event.
Compensate: CompensateCode	If the Trigger is a Compensate, then the compensation code must be supplied.
Rule: RuleName	If the Trigger is a Rule, then an expression must be entered.
Link: LinkName	If the Trigger is a Link, then the name of the Link must be supplied.
Multiple: Trigger + (except Multiple)	If the Trigger is a Multiple, then a list of the Trigger must have the appropriate data.
Interrupt: (True | False): True	Interrupt is a property that has a default of True. The property defines how the Intermediate Event will affect the Event Context if the Intermediate Event is attached to the boundary of an activity. If the property is True, then the Intermediate Event will interrupt the processing of all the activities within the Event Context. The flow would then be diverted through the outgoing Sequence Flow from the Intermediate Event. If the property is False, all processing of all the activities within the Event Context will continue *and* the flow will also be sent through the outgoing Sequence Flow from the Intermediate Event.
Assign *: Expression	Zero or more assignments can be made. Each assignment is an expression.
IncomingSequenceFlow *: SequenceFlowName	Zero or more incoming Sequence Flows can be idenitified for the Intermediate Event.
OutgoingSequenceFlow +: SequenceFlowName	One or more outgoing Sequence Flows can be idenitified for the Intermediate Event.
IncomingMessageFlow *: MessageFlowName	Zero or more incoming Message Flows can be idenitified for the Intermediate Event, but the Trigger type must be Message or Multiple (with Message as one of Trigger types). For a Message Flow there will be an associated BPEL4WS *receive* BPML *action* to receive the message. For multiple Message Flows, there will be BPEL4WS *pick* or BPML *choice* that will receive only one of the messages.
Pool ?: PoolName	If Pools are used, then the PoolName must be added to the Intermediate Event to identify its location.
Lane ?: LaneName	If that Pool has more than one Lane, then the LaneName must be added.
Association *	Zero or more Associations can be associated with the Intermediate Event.
Documentation ?	The modeler can add optional text documentation about the Intermediate Event.

Table 10 Intermediate Event Attributes

Note: BPMN may include a Process Break element in a future version of the Specification. The Process Break would be used in conjunction with a Timer Intermediate Event and highlights the location of a delay in the Process. This is an open issue. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

Note: the graphical depiction of a false Interrupt property has not been defined. This is an open issue. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

Sequence flow Connections

Refer to the section entitled “Sequence Flow Rules” on page 24 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ An Intermediate Event can be a target for a Sequence Flow; it can have one incoming Flow.
- ❖ An Intermediate Event can be a source for a Sequence Flow; it can have one outgoing Flow.

Message Flow Connections

Refer to the section entitled “Message Flow Rules” on page 25 for the entire set of objects and how the may be source or targets of Sequence Flows.

Note: All Message Flows described here must connect two separate Pools. They can connect to the Pool boundary or to flow objects within the Pool boundary. They cannot connect two objects within the same Pool.

- ❖ An Intermediate Event of type Message can be the target for Message Flows; it can have one incoming Message Flows.
- ❖ An Intermediate Event cannot be a source for a Message Flow; it can have no outgoing Message Flows.

Mapping to Execution Languages

The Mapping to Execution Languages will depend on the type of Intermediate Event. Each of the seven types will be mapped differently.

BPEL4WS

- ❖ If the Intermediate Event has an expression for the assign property, then this will map to a BPEL4WS *assign*.

Each type of Intermediate Event will have a different mapping to BPML:

- Message:
  - If the Intermediate Event follows a Decision (e.g., is part of a *pick*): this will map to an *onMessage* element within a *pick*.
  - If the Intermediate Event is within the normal flow of the Process (but does not follow a Decision): this will map to a *receive*.
  - If the Intermediate Event is attached to the boundary of an activity: this will map to an *onMessage* element within a *scope*.
- Timer:
  - If the Intermediate Event follows a Decision (e.g., is part of a *pick*): this will map to an *onAlarm* element within a *pick*.
  - If the Intermediate Event is within the normal flow of the Process (but does not follow a Decision): this will map to a *wait*.
  - If the Intermediate Event is attached to the boundary of an activity: this will map to a *wait* element, followed by a *throw*. A *scope* is also created that has a *catch* to correspond with the *throw*.
- Process Error: this will map to a *catch* element within a *scope*
- Compensate (must be attached to the boundary of an activity): this will map to an *compensationHandler* element within a *scope*.
- Rule (must be attached to the boundary of an activity): TBD.
- Link (must follow a Decision): this will map to the *onMessage* element of a *pick*.
- Multiple (must be attached to the boundary of an activity): this will map to a combination of *onMessage*, *onAlarm*, *compensationHandler*, *onSignal*, *throw*, *catch,* and *wait* elements within a *context*.

BPML

- If the Intermediate Event has an expression for the assign property, then this will map to a BPML *assign*.

Each type of Intermediate Event will have a different mapping to BPML:

- Message:
  - If the Intermediate Event follows a Decision (e.g., is part of a *choice*): this will map to an *action* within an *event* element within a *choice*.
  - If the Intermediate Event is within the normal flow of the Process (but does not follow a Decision): this will map to an *action* (that expects a message).
  - If the Intermediate Event is attached to the boundary of an activity: this will map to an *action* within an *event* element within an *exception* element within a *context*.
- Timer:
  - If the Intermediate Event follows a Decision (e.g., is part of a *choice*): this will map to a *delay* within an *event* element within a *choice*.

    - ❖ If the Intermediate Event is within the normal flow of the Process (but does not follow a Decision): this will map to a *delay*.
    - ❖ If the Intermediate Event is attached to the boundary of an activity: this will map to an *schedule* element within a *context* that will create a *fault code* that will be captured by a *faults case* within the *context*.
- ❖ Process Error:
    - ❖ If the Intermediate Event is attached to the boundary of an activity: this will map to an *faults case* within a *context*.
- ❖ Compensate (must be attached to the boundary of an activity): this will map to an *compensation* process within a *context*.
- ❖ Rule (must be attached to the boundary of an activity): TBD.
- ❖ Link (must follow a Decision): this will map to a *signal* within an *event* element of a *choice*.
- ❖ Multiple (must be attached to the boundary of an activity): this will map to a combination of *actions*, *events*, *faults*, *schedules*, *exceptions*, and *contexts* elements within a *context*.

4.2 Activities

4.2.1 Processes

A **Process** is an activity performed within a company or organization. In BPMN a Process is depicted as a network of flow objects, which are a set of other activities and the controls that sequence them. The concept of process is intrinsically hierarchical. Processes may be defined at any level from enterprise-wide processes to processes performed by a single person. Low-level processes may be grouped together to achieve a common business goal.

Attributes

The following are identified attributes of a Process:

Attribute	**Description**
Name	Name is a text description of the Process.
Nested:(True | False): False	The Nested property defines whether or not the Process is nested within another Process (thus sharing the Parent Process properties).
ParentProcess: ProcessName	If Nested, then the Parent Process must be identified.

Attribute	Description
Property *:	Modeler-defined Properties can be added to a Process. These Properties are "local" to the Process. All Tasks, Sub-Process objects, and Sub-Processes that are nested have access to these Properties. The fully delineated name of these properties are "<process name>.<property name>" (e.g., "Add Customer.Customer Name"). If a process is nested within another Process, then the fully delineated name would also be preceded by the Parent Process name for as many Parents there are until the top level Process.
Name:	Each Property has a Name (e.g., name="Customer Name").
Type:	Each Property has a Type (e.g., type="String").
AdHoc: (True | False): False	AdHoc is a Boolean property, which has a default of False. This specifies whether the Process is Ad Hoc or not. The activities within an Ad Hoc Process are not controlled by a process engine, they are completely controlled by the performers of the activities. The Process Engine may be able to track the actual instances on the activities within.
CompletionCondition: Expression	If the Process is Ad Hoc, then a Completion Condition must be included, which defines the conditions when the Process will end. The Ad Hoc marker will be placed at the bottom center of the Process or the Sub-Process shape for Ad Hoc Processes.
Assign *: Expression	Zero or more assignments can be made. Each assignment is an expression.
AssignTime: (Start | End): Start	For each assignment the modeler can specify whether the assignment will take place at the start or end of the Process.
IncomingMessageFlow *: MessageFlowName	Zero or more incoming Message Flows can be idenitified for the Process.
Target: (Boundary | Internal): Boundary	Each Message Flow must be associated with the Process itself (Pool Boundary) or with flow objects that the Process contains (Internal). To create a mapping to BPEL4WS or BPML, all the Message Flows should be connected to objects within the Process.
OutgoingMessageFlow *: MessageFlowName	Zero or more outgoing Message Flows can be idenitified for the Process.
Source: (Boundary | Internal): Boundary	Each Message Flow must be associated with the Process itself (Pool Boundary) or with flow objects that the Sub-Process contains (Internal). To create a mapping to BPEL4WS or BPML, all the Message Flows should be connected to objects within the Process.

Attribute	Description
Pool ?: PoolName	If Pools are used, then the PoolName must be added to the Process to identify its location. There is a one-to-one relationship between a Process and a Pool.
Partner *: PartnerName	Zero or more Partner's can be identified for the Process. A Partner is also equivalent to the other Pools of the diagram. Thus, all the other Pool Names in the diagram will be listed in as Partners for the current Process. Additional Partners that are not shown as Pools can also be identified. This will map to the *partner* element of BPEL4WS.
Association *	Zero or more Associations can be associated with the Process.
Documentation ?	The modeler can add optional text documentation about the Process.

Table 11 Process Attributes

4.2.2 Sub-Process

A **Sub-Process** is a compound activity in that it has detail that is defined as a flow of other activities. A Sub-Process is a graphical object within a Process Flow, but it also references another Process (either nested or independent). A Sub-Process shares the same shape as the Task, which is a rectangle that has rounded corners. The Sub-Process can be in a collapsed view that hides its details (see Figure 9) or a Sub-Process can be in an expanded view that shows its details within the view of the Process in which it is contained (see Figure 10). In the collapsed form, the Sub-Process object uses a marker to distinguish it as a Sub-Process, rather than a Task. The marker is a small square with a plus sign (+) inside. The square is positioned at the bottom center of the shape. Text associated with the Sub-Process (e.g., its name) can be placed inside the shape, or above or below the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor.

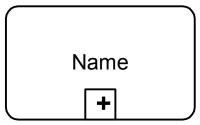


Figure 9 Collapsed Sub-Process

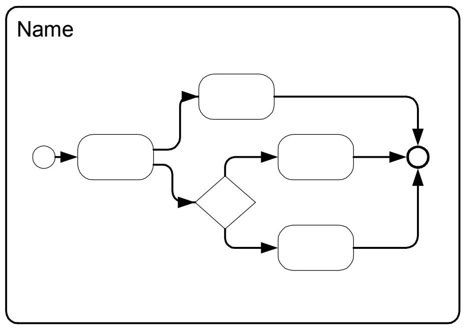


Figure 10 Expanded Sub-Process

BPMN specifies three types of markers for Sub-Processes. The (Collapsed) Sub-Process Marker, seen in Figure 9, can be combined with two other markers: a Loop Marker and an Ad Hoc Marker. A Sub-Process may have one or both of these markers. All the markers that are present will be grouped and the whole group will be centered at the bottom of the shape (see Figure 11).

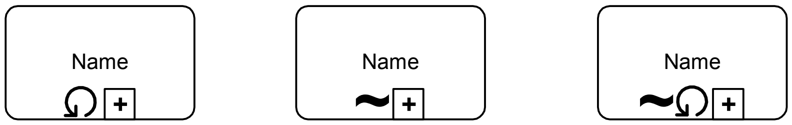


Figure 11 Collapse Sub-Process Marker Combinations

Note: The positioning of the Loop and Ad Hoc Markers is subject to change in a later release of the BPMN specification. This is an open issue. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN.

Note: An additional graphical marker may be included in BPMN for parallel ForEach types of loops. This is an open issue. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN.

Attributes

The following are identified attributes of a Sub-Process:

Attributes	Description
Name	Name is a text description of the Sub-Process.
SubProcessType: (Nested | Independent): Nested	SubProcessType is a property that defines whether the Sub-Process details are embedded within the higher level Process (nested) or refers to another, re-usable Process. The default is Nested.
Process: ProcessName	If the type is Independent, then the name of the referenced Process must be included.
InputMap +: Expression	For Independent, multiple input mappings can be made between Parent Process properties and the properties of the referenced Process. These mappings are in the form of an expression (although a modeling tool can present this to a modeler in any number of ways).
OutputMap +: Expression	For Independent, multiple output mappings can be made between Parent Process properties and the properties of the referenced Process. These mappings are in the form of an expression (although a modeling tool can present this to a modeler in any number of ways).

Attributes	Description
Property *	Modeler-defined Properties can be added to a Sub.Process. These Properties are "local" to the Sub-Process object—not the Process that the Sub-Process object represents. These Properties are only for use within the processing of the Sub-Process object. The fully delineated name of these properties are "<process name>.<sub-process name>.<property name>" (e.g., "Add Customer.Review Credit.Status").
Name	Each Property has a Name (e.g., name="Customer Name").
Type	Each Property has a Type (e.g., type="Text").
Transaction: (True | False): False	Transaction is a Boolean property, which has a default of False. This value automatically becomes True when a Compensate Intermediate Event is attached to the boundary of the Sub-Process.
Compensate: Intermediate Event	The Compensate property lists the name of the Intermediate Event.
EventContext: (True | False): False	EventContext is a Boolean property, which has a default of False. This value automatically becomes True when one or more a Timer, Message, or Process Error Intermediate Events is attached to the boundary of the Sub-Process.
Exception: Intermediate Event	The Exception property lists the names of the Intermediate Events.
LoopType: (None | Standard | ForEach) : None	LoopType is a property and is by default None, but can be set to Standard or ForEach, which means that the Loop marker will be placed at the bottom center of the Sub-Process shape. ForEach Loops require an expression, which specifies the number of instances.
LoopCondition: Expression	Standard Loops required an expression to be evaluated, plus the timing when the expression will be evaluated.
Counter: Number	The Counter property is used at runtime to count the number of loops.
Maximum: Number	The Maximum property is a simple way to add a cap to the number of loops. This gets added to the expression when mapped to BPEL4WS or BPML.
EvaluateCondition: (Before | After) : After	Standard Loops expressions evaluated Before the Sub-Process begins are *while* loops and expressions evaluated After the Sub-Process finishes are *while* loops for BPEL4WS and *until* loops for BPML.
Timing: (Serial | Parallel) : Serial	The Timing property defines whether the ForEach instances will be performed serially or in parallel. A parallel ForEach is equivalent to multi-instance specifications that other notations, such as UML Activity Diagrams use.

Attributes	Description
LoopFlowCondition: (One | All | Complex): All	The LoopFlowCondition, applied only to Parallel ForEach loops, acts the similarly to the FlowCondition for the Sub-Process. A Loop Flow Condition of One means that the Token will continue past the Sub-Process after only on of the Sub-Process instances has completed. The Sub-Process will continue its other instances, but no other Tokens will be passed from the Sub-Process. A Loop Flow Condition of All means that all Sub-Process instances must be completed before the Token can move from the Sub-Process.
Complex: Expression	A complex Loop Flow Condition can be set by the modeler. This will consist of an expression that can reference Process data. The expression will determine the Token will continue past the Sub-Process.
Assign *: Expression	Zero or more assignments can be made. Each assignment is an expression.
AssignTime: (Start | End): Start	For each assignment the modeler can specify whether the assignment will take place at the start or end of the Sub-Process.
IncomingSequenceFlow +: SequenceFlowName	One or more incoming Sequence Flows can be idenitified for the Sub-Process.
FlowCondition: (One | All | Complex): All	If there is more than one, then a Flow Condition must be set. A Flow Condition of One means that the Sub-Process will be started when one Token arrives on any of the Flows. The process will continue and all other Tokens arriving at the Sub-Process will be consumed. A Flow Condition of All means that a Token must arrive from all incoming Flows before the Sub-Process can start.
Complex: Expression	A complex Flow Condition can be set by the modeler. This will consist of an expression that can reference Sequence Flow names and or Process data. The expression will determine when the Sub-Process will start.
OutgoingSequenceFlow +: SequenceFlowName	One or more outgoing Sequence Flows can be idenitified for the Sub-Process.
IncomingMessageFlow *: MessageFlowName	Zero or more incoming Message Flows can be idenitified for the Sub-Process.
Target: (Boundary | Internal): Boundary	Each Message Flow must be associated with the Sub-Process itself (Boundary) or with flow objects that the Sub-Process contains (Internal). For a Boundary Message Flow, it is equivalent to connecting the Message Flows to the Start Event of the Sub-Process.
OutgoingMessageFlow *: MessageFlowName	Zero or more outgoing Message Flows can be idenitified for the Sub-Process.

Attributes	Description
Source: (Boundary | Internal): Boundary	Each Message Flow must be associated with the Sub-Process itself (Boundary) or with flow objects that the Sub-Process contains (Internal). For a Boundary Message Flow, it is equivalent to connecting the Message Flows to the End Event of the Sub-Process.
Pool ?: PoolName	If Pools are used, then the PoolName must be added to the Sub-Process to identify its location.
Lane ?: LaneName	If that Pool has more than one Lane, then the LaneName must be added.
Association *	Zero or more Associations can be associated with the Sub-Process.
Documentation ?	The modeler can add optional text documentation about the Sub-Process.

Table 12 Sub-Process Attributes

Sequence flow Connections

Refer to the section entitled “Sequence Flow Rules” on page 24 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ A Sub-Process can be a target for a Sequence Flow; it can have multiple incoming Flows. An incoming Flow can be from an alternative path or a parallel path. The Flow Condition will determine when the Sub-Process will start.
  - ❖ If the Sub-Process does not have an incoming Sequence Flow, and there is no Start Event for the Process, then the Sub-Process will be instantiated when the process is instantiated.
- ❖ A Sub-Process can be a source for a Sequence Flow; it can have multiple outgoing Flows. If there are multiple outgoing Sequence Flows, then this means that a separate parallel path is being created for each Flow.

Tokens will be generated for each outgoing Sequence Flow from Sub-Process. The TokenIDs for each of the Tokens will be set such that it can be identified that the Tokens are all from the same parallel Fork (AND-Split) and the number of Tokens in the group

  - ❖ If the Sub-Process does not have an outgoing Sequence Flow, and there is no End Event for the Process, then the Sub-Process marks the end of one or more paths in the Process. When the Sub-Process ends and there are no other parallel paths active, then the Process will be completed.

Message Flow Connections

Refer to the section entitled “Message Flow Rules” on page 25 for the entire set of objects and how the may be source or targets of Sequence Flows.

Note: All Message Flows described here must connect two separate Pools. They can connect to the Pool boundary or to flow objects within the Pool boundary. They cannot connect two objects within the same Pool.

- ❖ A Sub-Process can be the target for Message Flows; it can have zero or more incoming Message Flows.
- ❖ A Sub-Process can be a source for a Message Flow; it can have zero or more outgoing Message Flows.

Mapping to Execution Languages

The following two sections describe how the use of Sub-Processes will map to BPEL4WS and BPML, respectively.

BPEL4WS

There are four possibilities, depending on the Message Flows that attach to the Sub-Process boundary:

- ❖ A Sub-Process that has no Message Flows attached to its boundary will map to the BPEL4WS *invoke*. This will invoke another web service, which is another *process*.
- ❖ A Sub-Process that has an incoming Message Flow attached to its boundary will map to a BPEL4WS *receive* followed by a BPEL4WS *invoke*.
- ❖ A Sub-Process that has an outgoing Message Flow attached to its boundary will map to the BPEL4WS *invoke* followed by a BPEL4WS *reply*.
- ❖ A Sub-Process that has both an incoming and an outgoing Message Flow attached to its boundary will map to a BPEL4WS *receive* followed by a BPEL4WS *invoke* followed by a BPEL4WS *reply*.

Sub-Process properties will map as follows:

- ❖ For a Reference Sub-Process type the modeler will have to create the referenced Process independently (with a different name) and then assign the Process to the Sub-Process object. The referenced *process* will be called with the BPEL4WS *invoke*.
  - ❖ InputMap will be mapped to the parameter passing elements of the *call*.
  - ❖ OutputMap will be mapped to the parameter passing elements of the *call*.
- ❖ The mapping for the Transaction property is TBD.
- ❖ If the LoopType is Standard then the Sub-Process will be wrapped by a BPEL4WS *while* or *until*.
  - ❖ A Before EvaluateCondition will map to the BPEL4WS *while*.
  - ❖ An After EvaluateCondition will map to the BPEL4WS *while*. However, to ensure that the Sub-Process is performed at least once, the *activity(s)* appropriate for the Sub-Process Type will be performed first in a *sequence*, which includes the *while*
  - ❖ Any value in Maximum will be appended to the LoopCondition. For example with a LoopCondition of "x < 0" and Maximum of 5 (loops), the final expression would be

"(x < 0) and (<Sub-ProcessName>.Counter <= 5)." An BPEL4WS *assign* will be used to update the Counter property.

- If the LoopType is ForEach then the TBD.

Editor's Note: We have not determined how the Ad Hoc Sub-Process will be mapped to BPEL4WS.

BPML

There are four possibilities, depending on the Message Flows that attach to the Sub-Process boundary:

- A Sub-Process that has no Message Flows attached to its boundary will map to the BPML *call*.
- A Sub-Process that has an incoming Message Flow attached to its boundary will map to a BPML *one-way action* followed by a BPML *call*.
- A Sub-Process that has an outgoing Message Flow attached to its boundary will map to the BPML *call* followed by a BPML *one-way action*.
- A Sub-Process that has both an incoming and an outgoing Message Flow attached to its boundary will map to the BPML *request-response action*.
  - If the Sub-Process type is Independent, then a BPML *call* will be used within the *action*.
  - If the Sub-Process type is Nested, then the activities within the Sub-Process will be mapped as appropriate and then inserted within the *action*.

Sub-Process properties will map as follows:

- For a Reference Sub-Process type the modeler will have to create the referenced Process independently (with a different name) and then assign the Process to the Sub-Process object. The referenced *process* will be called with the BPML *call*.
  - InputMap will be mapped to the parameter passing elements of the *call*.
  - OutputMap will be mapped to the parameter passing elements of the *call*.
- The mapping for the Transaction property is TBD.
- If the LoopType is Standard then the Sub-Process will be wrapped by a BPML *while* or *until*.
  - A Before EvaluateCondition will map to the BPML *while*.
  - An After EvaluateCondition will map to the BPML *until*.
  - Any value in Maximum will be appended to the LoopCondition. For example with a LoopCondition of "x < 0" and Maximum of 5 (loops), the final expression would be "(x < 0) and (<Sub-ProcessName>.Counter <= 5)." An BPML *assign* will be used to update the Counter property.

- If the LoopType is ForEach then the Sub-Process will be wrapped by a BPML *foreach*.
    - If the Time is Parallel, then the Sub-Process will be accessed through BPML *spawn* for each instance and then a BPML *synch* will synchronize them.
    - Mapping the LoopFlowCondition TBD.

Editor's Note: We have not determined how the Ad Hoc Sub-Process will be mapped to BPML.

4.2.3 Task

A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail. Generally, an end-user and/or an application are used to perform the Task when it is executed.

A Task object shares the same shape as the Sub-Process, which is a rectangle that has rounded corners (see Figure 12). Text associated with the Task (e.g., its name) can be placed above or below the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor.

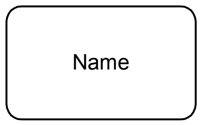


Figure 12 A Task Object

Attributes

The following are identified attributes of a Task:

Attributes	Description
Name	Name is a text description of the Task.
Type (Send | Receive | Service | User): Service	Type is a property that has a default of Service, but can be set to Send, Receive, or User. The type of Task will depend on the Message Flows to and/or from the Task, if Message Flows are used. If there is only an incoming Sequence Flow, then the type must be Receive. If there is only an outgoing Sequence Flow, then the type must be Send. If there is both an incoming and outgoing Sequence Flow, then the type must be Service. User Tasks are necessary to create an asynchronous mechanism of notification and response to handle the long-lived nature of User Tasks. Since a BPML *solicit-response action* is synchronous, this means that a set of BPML elements will be required to create an asynchronous situation that handles complexities of the interactions between a User and a BPM Engine (this is detailed below in the section on "Mapping to Execution Languages").

Attributes	Description
(Receive) **Instantiate** (True | False): False	Receive Tasks can be defined as the instantiation mechanism for the Process with the Instantiate property. This property can only be set to true if the Task is the first activity after the Start Event or a starting activity if there is no Start Event.
Property *	Modeler-defined Properties can be added to a Task. These Properties are "local" to the Task object. These Properties are only for use within the processing of the Task object. The fully delineated name of these properties are "<process name>.<task name>.<property name>" (e.g., "Add Customer.Review Credit Report.Score").
Name	Each Property has a Name (e.g., name="Customer Name").
Type	Each Property has a Type (e.g., type="Text").
Input *: Attribute	Input is an optional property that defines which of the Parent Process attributes are used as either an input for or an output from the Task.
Output *: Attribute	Output is an optional property that defines which of the Parent Process attributes are used as either an input for or an output from the Task.
Transaction: (True | False): False	Transaction is a Boolean property, which has a default of False. This value automatically becomes True when a Compensate Intermediate Event is attached to the boundary of the Task.
Compensate: Intermediate Event	The Compensate property lists the name of the Intermediate Event.
EventContext: (True | False): False	EventContext is a Boolean property, which has a default of False. This value automatically becomes True when one or more a Timer, Message, or Process Error Intermediate Events is attached to the boundary of the Task.
Exception: Intermediate Event	The Exception property lists the names of the Intermediate Events.
LoopType: (None | Standard | ForEach) : None	LoopType is a property and is by default None, but can be set to Standard or ForEach, which means that the Loop marker will be placed at the bottom center of the Task shape. ForEach Loops require an expression, which specifies the number of instances.
LoopCondition: Expression	Standard Loops required an expression to be evaluated, plus the timing when the expression will be evaluated.
Counter: Number	The Counter property is used at runtime to count the number of loops.
Maximum: Number	The Maximum property is a simple way to add a cap to the number of loops. This gets added to the expression when mapped to BPEL4WS or BPML.

Attributes	Description
EvaluateCondition: (Before | After) : After	Standard Loops expressions evaluated Before the Sub-Process begins are *while* loops and expressions evaluated After the Task finishes are *while* loops for BPEL4WS and *until* loops for BPML.
Timing: (Serial | Parallel) : Serial	The Timing property defines whether the ForEach instances will be performed serially or in parallel. A parallel ForEach is equivalent to multi-instance specifications that other notations, such as UML Activity Diagrams use.
LoopFlowCondition: (One | All | Complex): All	The LoopFlowCondition, applied only to Parallel ForEach loops, acts the similarly to the FlowCondition for the Sub-Process. A Loop Flow Condition of One means that the Token will continue past the Task after only on of the Sub-Process instances has completed. The Task will continue its other instances, but no other Tokens will be passed from the Task. A Loop Flow Condition of All means that all Task instances must be completed before the Token can move from the Task.
Complex: Expression	A complex Loop Flow Condition can be set by the modeler. This will consist of an expression that can reference Process data. The expression will determine the Token will continue past the Task.
Assign *: Expression	Zero or more assignments can be made. Each assignment is an expression.
AssignTime: (Start | End): Start	For each assignment the modeler can specify whether the assignment will take place at the start or end of the Task.
IncomingSequenceFlow +: SequenceFlowName	One or more incoming Sequence Flows can be idenitified for the Task.
FlowCondition: (One | All | Complex): All	If there is more than one, then a Flow Condition must be set. A Flow Condition of One means that the Task will be started when one Token arrives on any of the Flows. The process will continue and all other Tokens arriving at the Task will be consumed. A Flow Condition of All means that a Token must arrive from all incoming Flows before the Task can start.
Complex: Expression	A complex Flow Condition can be set by the modeler. This will consist of an expression that can reference Sequence Flow names and or Process data. The expression will determine when the Task will start.
OutgoingSequenceFlow +: SequenceFlowName	One or more outgoing Sequence Flows can be idenitified for the Task.
IncomingMessageFlow *: MessageFlowName	Zero or more incoming Message Flows can be idenitified for the Task.
OutgoingMessageFlow *: MessageFlowName	Zero or more outgoing Message Flows can be idenitified for the Task.

Attributes	Description
Pool ?: PoolName	If Pools are used, then the PoolName must be added to the Task to identify its location.
Lane ?: LaneName	If that Pool has more than one Lane, then the LaneName must be added.
Association *	Zero or more Associations can be associated with the Task.
Documentation ?	The modeler can add optional text documentation about the Task.

Table 13 Task Attributes

Sequence flow Connections

Refer to the section entitled "Sequence Flow Rules" on page 24 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ A Task can be a target for a Sequence Flow; it can have multiple incoming Flows. An incoming Flow can be from an alternative path or a parallel path. The Flow Condition will determine when the Task will start.
  - ❖ If the Task does not have an incoming Sequence Flow, and there is no Start Event for the Process, then the Task will be instantiated when the process is instantiated.
- ❖ A Task can be a source for a Sequence Flow; it can have multiple outgoing Flows. If there are multiple outgoing Sequence Flows, then this means that a separate parallel path is being created for each Flow.

Tokens will be generated for each outgoing Sequence Flow from the Task. The TokenIDs for each of the Tokens will be set such that it can be identified that the Tokens are all from the same parallel Fork (AND-Split) and the number of Tokens in the group

  - ❖ If the Task does not have an outgoing Sequence Flow, and there is no End Event for the Process, then the Task marks the end of one or more paths in the Process. When the Task ends and there are no other parallel paths active, then the Process will be completed.

Message Flow Connections

Refer to the section entitled "Message Flow Rules" on page 25 for the entire set of objects and how the may be source or targets of Sequence Flows.

Note: All Message Flows described here must connect two separate Pools. They can connect to the Pool boundary or to flow objects within the Pool boundary. They cannot connect two objects within the same Pool.

- ❖ A Task can be the target for Message Flows; it can have zero or one incoming Message Flows.
- ❖ A Task can be a source for a Message Flow; it can have zero or more outgoing Message Flows.

Mapping to Execution Languages

The following two sections describe how the use of Tasks will map to BPEL4WS and BPML, respectively.

BPEL4WS

- A Receive Type Task will be mapped to a BPEL4WS *receive*.
    - The Instantiate property will be mapped to the *createInstance* element of the *receive* element. True will be mapped to *yes* and False will be mapped to *no*.
- A Send Type Task will be mapped to a BPEL4WS *reply* or a BPEL4WS *invoke* (with only the *inputContainer* specified)
- A Service Type Task will be mapped to a BPEL4WS *invoke* (with both the *inputContainer* and *outputContainer* specified)
- A User Type Task will not have any Message Flows and will map TBD:
- The mapping for the Transaction property is TBD.
- If the LoopType is Standard then the Task will be wrapped by a BPEL4WS *while* or *until*.
    - A Before EvaluateCondition will map to the BPEL4WS *while*.
    - An After EvaluateCondition will map to the BPEL4WS *while*. However, to insure that the Task is performed at least once, the *activity* appropriate for the Task Type will be performed first in a *sequence*, which includes the *while*.
    - Any value in Maximum will be appended to the LoopCondition. For example with a LoopCondition of "x < 0" and Maximum of 5 (loops), the final expression would be "(x < 0) and (<TaskName>.Counter <= 5)." A BPEL4WS *assign* will be used to update the Counter property.
- If the LoopType is ForEach then the mapping is TBD.

BPML

- A Receive Type Task will be mapped to a BPML *one-way action*.
- A Send Type Task will be mapped to a BPML *notification action*.
- A Service Type Task with both an incoming and outgoing Message Flow will be mapped to a BPML *solicit-response action*.
- A User Type Task will not have any Message Flows and will map TBD:
- The mapping for the Transaction property is TBD.
- If the LoopType is Standard then the Task will be wrapped by a BPML *while* or *until*.
    - A Before EvaluateCondition will map to the BPML *while*.
    - An After EvaluateCondition will map to the BPML *until*.
    - Any value in Maximum will be appended to the LoopCondition. For example with a LoopCondition of "x < 0" and Maximum of 5 (loops), the final expression would be "(x < 0) and (<TaskName>.Counter <= 5)." A BPML *assign* will be used to update the Counter property.

- If the LoopType is ForEach then the Task will be wrapped by a BPML *foreach*.
  - If the Time is Parallel, then the Task will be wrapped in a Nested Process and accessed through BPML *spawn* for each instance and then a BPML *synch* will synchronize them.
  - Mapping the LoopFlowCondition TBD.

Note: BPML *request-response action* is not mapped to a BPMN Task. Although this type of *action* is an atomic activity in BPML, additional activities can be performed before the *action* is complete. Because addtional work can be done for the response to the request, this will be represented as a Sub-Process in BPMN.

4.3 Decisions

Decisions are locations within a business process where the Sequence Flow can take two or more alternative paths. This is basically the "fork in the road" for a process. For a given performance (or instance) of the process, only one of the paths can be taken (this should not be confused with forking of paths—refer to the section entitled "Forking (AND-Split)" on page 86). A Decision is not an activity from the business process perspective, but is an object that controls the flow between activities. It can be thought of as a question that is asked at that point in the Process. The question has a defined set of Alternative answers. Each Decision Alternative is paired with a single outgoing Sequence Flow. When an Alternative is chosen during the performance of the Process, the corresponding Sequence Flow is then chosen. A Token arriving at the Decision would be directed down the appropriate path, based on the chosen Alternative. The Sequence Flows themselves act only as the path through which the Token travels; they do not have their own conditions that can determine whether they are traveled or not.

A Decision is a diamond (see Figure 13), which has been used in many flow chart notations and is familiar to most modelers. Text associated with the Decision (e.g., its name) can be placed inside the shape, or above or below the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor.

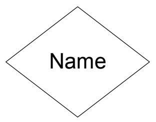


Figure 13 A Decision

Note: Although the shape of the Decision is a diamond, it is not a requirement that incoming and outgoing Sequence Flow must connect to the corners of the diamond. Sequence Flow can connect to any position on the boundary of the Decision shape.

Decisions come in two basic types: Exclusive and Inclusive. The following two sections define these two types.

4.3.1 Exclusive

The Exclusive Decision has two or more outgoing Message Flows, but only one of them may be taken during the performance of the Process. Thus, the Exclusive Decision defines a set of Alternative paths for the Token to take as it traverses the Flows. There are two types of Exclusive Decisions: Data-Based and Event-Based.

Data-Based

The Data-Based Exclusive Decisions are the most commonly used type. We will also refer to them as just Decisions. The set of Alternatives for Data-Based Exclusive Decisions are based on condition expressions. These expressions evaluate the current values of process data to determine which path should be taken (hence the name Data-Based). The conditions should be evaluated in a specific order. The first one that evaluates as true will determine the Sequence Flow that will be taken. One of the condition expressions may be "default," and is the last condition evaluated. This means that if none of the other condition expressions is true at runtime, then the default expression will be chosen—along with its associated Sequence Flow.

Note: the Default Alternative for a Data Decision Type may be defined as being mandatory in a future version of the specification. This is an open issue. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN.

Although the Decision actually contains the condition expressions, they can be displayed on the outgoing Sequence Flows (see Figure 14). While shown on the Sequence Flow, they are not actually a property of the Sequence Flows.

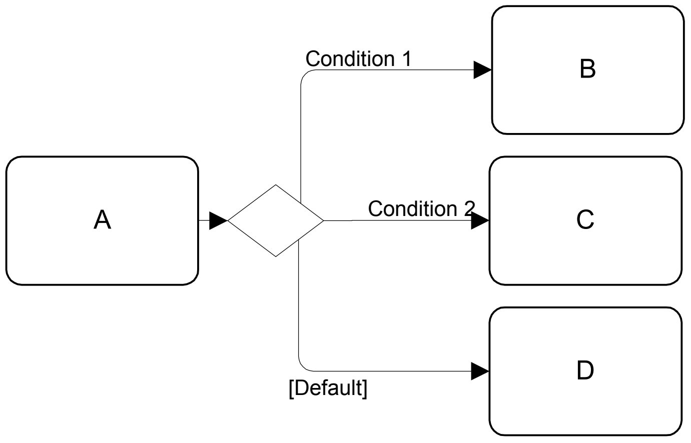


Figure 14 A Data-Based Decision Example

Event-Based

Event-Based Exclusive Decisions are a fairly new development in Business Process Management and will map to the BPEL4WS *pick* or BPML *choice* elements. The basic idea is that this Decision represents a branching point in the process where the Alternatives are based on an Intermediate Event that occurs at that point in the Process. The specific Intermediate Event, usually a message type, determines which of the paths will be taken.

For example, if a company is waiting for a response from a customer, they will perform one set of activities if the customer responds “Yes” and another set of activities if the customer responds “No.” The customer’s response determines which path is taken. The identity of the Message determines which path is taken. That is, the “Yes” Message and the “No” message are completely different messages—they are not the same message with different values within a property of the Message. In addition to Messages, other types of Intermediate Events can be used, such as Timers and Process Errors.

The Event-Based Exclusive Decisions are configured by using the Decision shape and having outgoing Sequence Flows target an Intermediate Event (see Figure 15). All of the outgoing Sequence Flows must target an Intermediate Event; there cannot be a mixing of condition expressions and Intermediate Events for a given Decision.

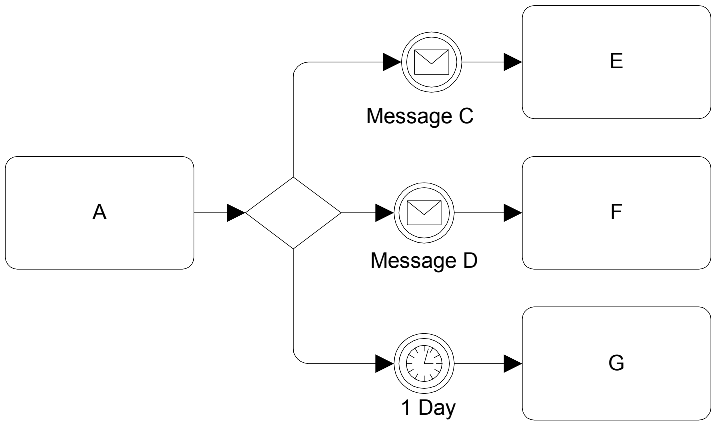


Figure 15 An Event-Based Decision Example

To relate the Event-Based Exclusive Decision to BPEL4WS or BPML, the Decision diamond marks the location of a BPEL4WS *pick* or a BPML *choice* and the Intermediate Events that follow the Decision become the event handlers of the *pick* or *choice*. The activities that follow the Intermediate Events become the contents of the *activity sets* for the event handlers. The boundaries of the activity sets is actually determined by the configuration of the process; that is, the boundaries extend to where all the alternative paths are finally joined together (which could be the end of the Process).

Attributes

The following table displays the identified attributes of an Exclusive Decision:

Attributes	Description
Name	Name is a property that is text description of the Decision.
DecisionType: (Data | Event): Data	DecisionType is a property and is by default Data.
(Data) **Alternative** +: Expression	If the type is Data, there must be one or more Alternatives with their expressions.
OutgoingSequenceFlow: SequenceFlowName	Each Alternative must have an associated Sequence Flow.
Assign +: Expression	Zero or more assignments can be made for each Alternative.
(Data) **DefaultAlternative**?	If the type is Data, then a Default Alternative may be specified.
OutgoingSequenceFlow: SequenceFlowName	The Default Alternative must have an associated Sequence Flow.
Assign +: Expression	Zero or more assignments can be made for the DefaultAlternative.
(Event) **Alternative** 2+: OutgoingSequenceFlow	If the type is Event, then two or more Alternatives are defined as Sequence Flows and their targets must be an Intermediate Event. The Intermediate Events must be of type Message, Timer, or Fault. Only one of the Events can be of type Timer, however.
Target: EventName	The targets of the Sequence flow must be an Intermediate Event
IncomingSequenceFlow *: SequenceFlowName	One or more incoming Sequence Flows can be idenitified for the Decision.
FlowCondition: (One | All | Complex) : All	If there is more than one, then a Flow Condition must be set. A Flow Condition of One means that the Decision will be evaluated when one Token arrives on any of the Flows. The process will continue and all other Tokens arriving at the Decision will be consumed. A Flow Condition of All means that a Token must arrive from all incoming Flows before the Decision can be evaluated.
Complex: Expression	A complex Flow Condition can be set by the modeler. This will consist of an expression that can reference Sequence Flow names and or Process data. The expression will determine when the Decision will be evaluated.
Pool ?: PoolName	If Pools are used, then the PoolName must be added to the Decision to identify its location.
Lane ?: LaneName	If that Pool has more than one Lane, then the LaneName must be added.
Association *	Zero or more Associations can be associated with the Decision.
Documentation ?	The modeler can add optional text documentation about the Decision.

Table 14 Decision Attributes

Sequence flow Connections

Refer to the section entitled “Sequence Flow Rules” on page 24 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ A Data-Based Exclusive Decision can be a target for a Sequence Flow; it can have multiple incoming Flows. An incoming Flow can be from an alternative path or a parallel path.
  - ❖ If the Decision does not have an incoming Sequence Flow, and there is no Start Event for the Process, then the Decision will be evaluated when the process is instantiated.
- ❖ An Exclusive Decision can be a source for a Sequence Flow; it must have two or more outgoing Flows. One of these outgoing Sequence Flows must be a “default” Sequence Flow. The non-default outgoing Flows are evaluated independently to determine if that path will be taken. If none of the non-default Flows taken, then the default Flow will be taken.

Message Flow Connections

Refer to the section entitled “Message Flow Rules” on page 25 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ An Exclusive Decision cannot be a target for a Message Flow.
- ❖ An Exclusive Decision cannot be a source for a Message Flow.

Mapping to Execution Languages

The following two sections describe how the use of Decisions will map to BPEL4WS and BPML, respectively.

BPEL4WS

- ❖ A Data-Based Exclusive Decision will map to a BPEL4WS *switch*. Each ConditionExpression will map to the condition for a *switch case*. The Default condition will map to the Switch *otherwise case*.
  - ❖ The activities that follow the conditions will be included within the *activity* (usually a *sequence*) for that condition. The exact content of the *activity* will depend on the configuration of the Process. Details of how the configuration will be mapped to the activity set can be found in the section entitled “Mapping to Execution Languages” on page 133.
- ❖ An Event-Based Exclusive Decision will map to a BPEL4WS *pick*. Each of the target Intermediate Events will map to the message handlers within the *pick*.
  - ❖ The activities that follow the Intermediate Events will be included within the *activity* (usually a *sequence*) for that message handler. The exact content of the *activity* will depend on the configuration of the Process. Details of how the configuration will be mapped to the activity set can be found in the section entitled “Mapping to Execution Languages” on page 133. If the Intermediate Event is of type Message, then the first activity of the *activity* will be a *receive*.

BPML

- ❖ A Data-Based Exclusive Decision will map to a BPML *switch*. Each ConditionExpression will map to the condition for a *switch case*. The Default condition will map to the Switch *default case*.
  - ❖ The activities that follow the conditions will be included within the *activity set* for that condition. The exact content of the *activity set* will depend on the configuration of the Process. Details of how the configuration will be mapped to the activity set can be found in the section entitled "Mapping to Execution Languages" on page 133.
- ❖ An Event-Based Exclusive Decision will map to a BPML *choice*. Each of the target Intermediate Events will map to the event handlers within the *choice*.
  - ❖ The activities that follow the Intermediate Events will be included within the *activity set* for that event handler. The exact content of the *activity set* will depend on the configuration of the Process. Details of how the configuration will be mapped to the activity set can be found in the section entitled "Mapping to Execution Languages" on page 133. If the Intermediate Event is of type Message, then the first activity of the *activity set* will be a *one-way action*.

4.3.2 Inclusive

Zero to all of the outgoing Sequence Flows from an Exclusive Decision may be taken during the performance of the Process. Thus, an Inclusive Decision is a hybrid between a Fork (AND-Split) and a Decision (OR-Split). In some sense it is a grouping of related independent Binary (Yes/No) Decisions. Since each path is independent, all combinations of the paths may be taken, from zero to all. BPMN extends this concept by insisting that there be a default Flow that is followed if none of the other outgoing Flows are taken. This insures that the process behavior is fully covered through the Sequence Flows. Thus, a BPMN Inclusive Decision really will have one or more of the outgoing Flows taken at runtime.

Editor's Note: the details of the how Inclusive Decisions look and behave is an open issue and will be included in a later version of the specification. Since Inclusive Decisions are a hybrid of forking and splitting, the definition of their behavior may be moved to another section in this specification, depending on how the notation of these Decisions are finalized. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN.

4.4 Pools and Lanes

BPMN has a larger scope than BPEL4WS or BPML, and this scope is expressed in different dimensions. The dimension discussed here has to with defining business processes in a collaborative B2B environment. BPMN uses the concept known as "swimlanes" to help partition and organize activities.

BPEL4WS and BPML are focused on a specific private process that is internal to a given Participant (i.e., a company or organization). BPEL4WS also can define an abstract process. It is possible that a BPMN diagram may depict more than one private process, as well as the processes that show the collaboration between private processes or

Participants. If so, then each private business process will be considered as being performed by different Participants. Graphically, each Participant will be partitioned; that is, will be contained within a rectangular box call a “Pool.” Pools can have sub-swimlanes that are called, simply, “Lanes.”

The section entitled “Uses of BPMN” on page 15 describes the uses of BPMN for modeling private processes and the interactions of processes in B2B scenarios. Pools and Lanes are designed to support these uses of BPMN.

4.4.1 Pool

A Pool is a “swimlane” and a graphical container for partitioning a set of activities from other Pools, usually in the context of B2B situations. It is a square-cornered rectangle that is drawn with a solid single line (as seen in Figure 16). To help with the clarity of the diagram, A Pool will extend the entire length of the diagram, either horizontally or vertically. However, there is no specific restriction to the size and or positioning of a Pool. Modelers and modeling tools can use Pools (and Lanes) in a flexible manner in the interest of conserving the “real estate” of a diagram on a screen or a printed page. Text associated with the Pool (e.g., its name) can be placed inside the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor.

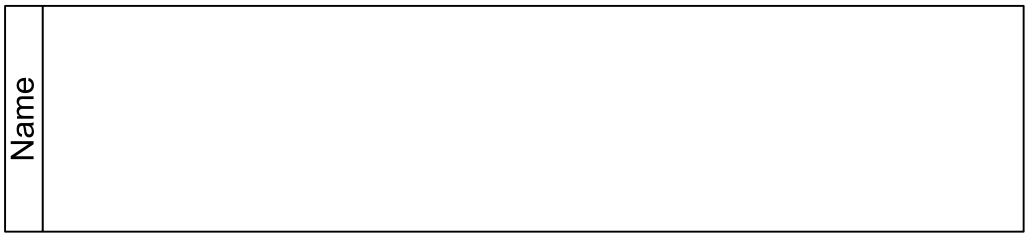


Figure 16 A Pool

A Pool acts as the container for the Sequence Flow between activities. The Sequence Flow can cross the boundaries between Lanes of a Pool, but cannot cross the boundaries of a Pool. The interaction between Pools, e.g., in a B2B context, is shown through Message Flows.

Another aspect of Pools is whether or not there is any activity detailed within the Pool. Thus, a given Pool may be shown as a “White Box,” with all details exposed, or as a “Black Box,” with all details hidden. No Sequence Flow is associated with a “Black Box” Pool, but Message Flows can attach to its boundaries (see Figure 17).

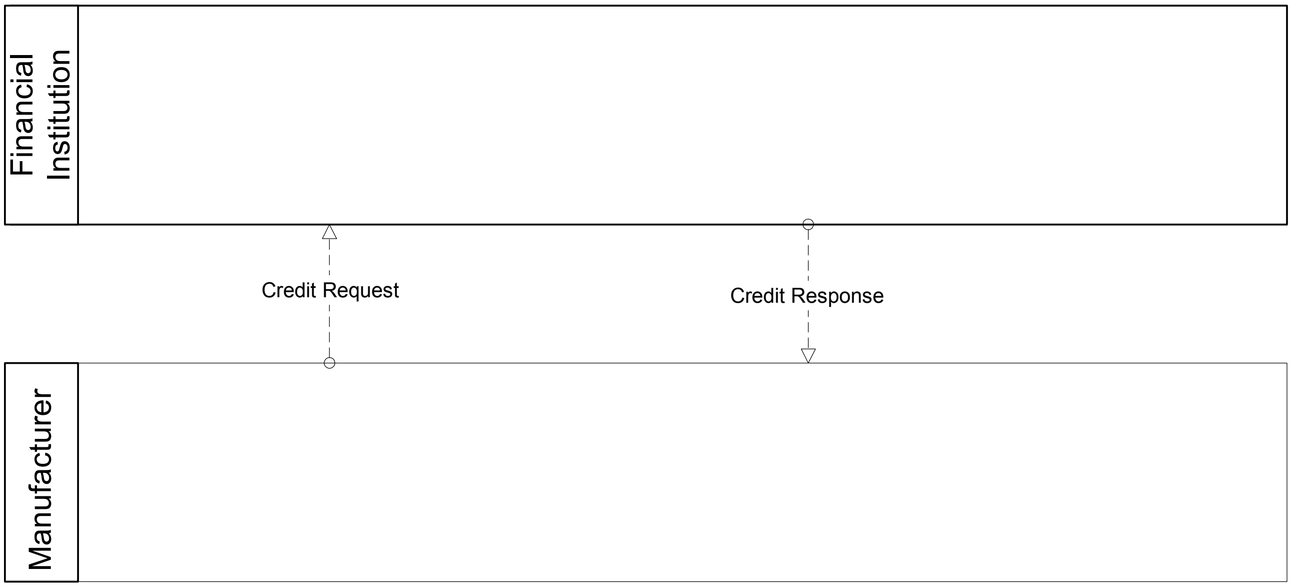


Figure 17 Message Flow connecting to the boundaries of two Pools

For a “White Box” Pool, the activities within are organized within by Sequence Flows. Message Flows can cross the Pool boundary to attach to the appropriate activity (see Figure 18).

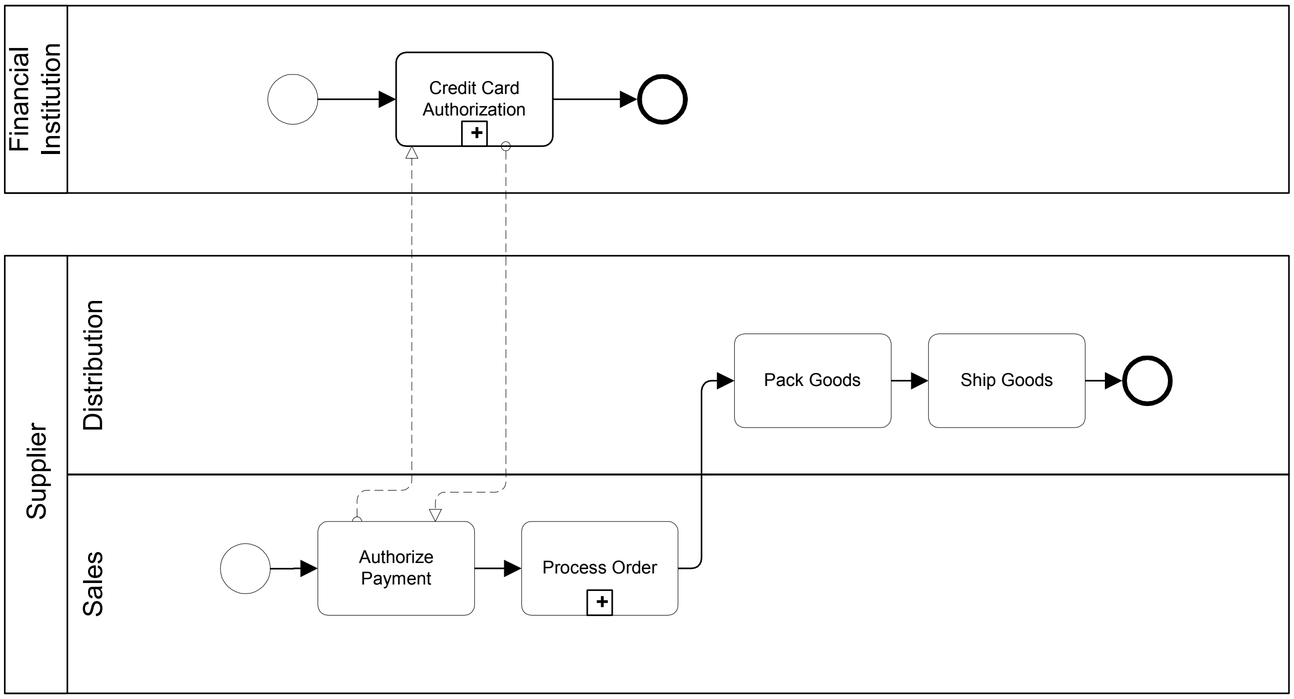


Figure 18 Message Flow connecting to flow objects within two Pools

Attributes

The following table displays the identified attributes of a Pool:

Attribute	Description
Name	Name is a property that is text description of the Pool.
PoolType (Private | Interface | Collaboration): Private	Pool Type is a property that provides information about to which lower-level language the Pool will be mapped. The default type is Private which will be mapped to BPEL4WS or BPML. An Interface Pool is also called the public interface of a process (or other web services) and will be mapped to languages such as WSCI. A Collaboration Pool will have two Lanes that represent business roles (e.g., buyer or seller) and will show the interactions between these roles. These pools will be mapped to languages such as ebXML.
Owner ?	Owner is an optional property that will help identify the point-of-view of the diagram. If the Pool Type is Collaboration, then there is no specific Owner.
IncomingMessageFlow *: MessageFlowName	There can be zero or more incoming Message Flows.
Target (Boundary | FlowObjectName): Boundary	Each Outgoing Message Flow has to have a target specified, which can be either the boundary of the Pool or a flow object within the Pool.
OutgoingMessageFlow *: MessageFlowName	There can be zero or more outgoing Message Flows.
Source (Boundary | FlowObjectName): Boundary	Each incoming Message Flow has to have a source specified, which can be either the boundary of the Pool or a flow object within the Pool.
Lane +: LaneName	There can be one or more Lanes within a Pool. If there is only one Lane, then that Lane shares the name of the Pool and only the Pool name is displayed. If there is more than one, then each Lane has to have its own name and all names are displayed.
Association *	Zero or more Associations can be associated with the Pool.
Documentation ?	The modeler can add optional text documentation about the Pool.

Table 15 Pool Attributes

Mapping to Execution Languages

Pools do not have any specific Mapping to Execution Languages. However, a Pool is associated with a mapping to a specific lower level language. For example, one Pool may encompass a BPEL4WS or BPML document while another Pool might encompass an ebXML BPSS document.

Note: the current specification does not contain mappings to any languages except to the execution definitions in BPEL4WS and to BPML. The mapping to the abstract definitions in BPEL4WS will be included in a later version of the specification.

Note: Interface processes may be allowed to be a Lane within a Pool in a later version of the specification. This is an open issue. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN.

4.4.2 Lane

A Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally (see Figure 19). Text associated with the Lane (e.g., its name) can be placed inside the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor. Our examples place the name as a banner on the left side (for horizontal Pools) or at the top (for vertical Pools) on the other side of the line that separates the Pool name.

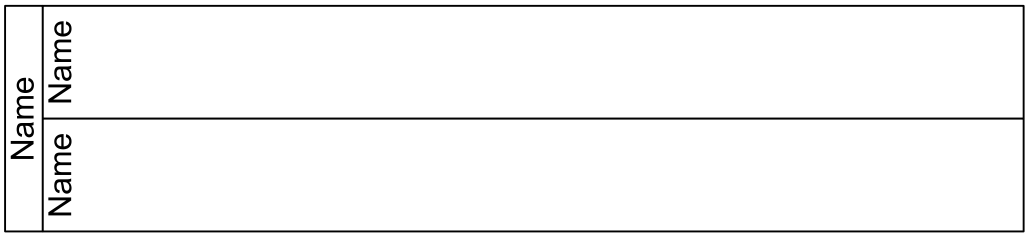


Figure 19 Two Lanes in a Pool

Lanes are used to organize and categorize activities within a Pool. The meaning of the Lanes is up to the modeler. BPMN does not specify the usage of Lanes. Lanes are often used for such things as internal roles (e.g., Manager, Associate), systems (e.g., an enterprise application), an internal department (e.g., shipping, finance), etc. In addition, Lanes can be nested. For example, there could be an outer set of Lanes for company departments and then an inner set of Lanes for roles within each department.

Attributes

The following table displays the identified attributes of a Lane:

Attribute	Description
Name	Name is a property that is text description of the Lane. If the Lane is the only one in the Pool, it will share the name of the Pool.
ParentPool: PoolName	The Parent Pool must be specified. There can be only one Parent.
Association *	Zero or more Associations can be associated with the Lane.
Documentation ?	The modeler can add optional text documentation about the Lane.

Table 16 Lane Attributes

Mapping to Execution Languages

Lanes do not have any specific Mapping to Execution Languages. They are designed to help organize and communicate how activities are grouped in a business process.

4.5 Data Object

In BPMN, a Data Objects are considered artifacts and not a flow object. They are considered an artifact because they do not have any direct affect on the Sequence Flow or Message Flow of the Process, but they do provide information about what the Process does. That is, how documents, data, and other objects are used and updated during the Process. While the name "Data Object" may imply an electronic document, they can be used to represent many different types of objects, both electronic and physical.

In general, BPMN will not standardize many modeling artifacts. These will mainly be up to modelers and modeling tool vendors to create for their own purposes. However, equivalents of the BPMN Data Object are used by Document Management oriented workflow systems and many other process modeling methodologies. Thus, this object is used enough that it is important to standardize its shape and behavior.

The Data Object is a portrait-oriented rectangle that has its upper-right corner folded over (see Figure 20). Text associated with the Data Object (e.g., its name and/or state) can be placed above or below the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor.

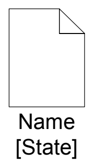


Figure 20 A Data Object

As an artifact, Data Objects generally will be associated with flow objects. An Association will be used to make the connection between the Data Object and the flow object. This means that the behavior of the Process can be modeled without Data Objects for modelers who want to reduce clutter. The same Process can be modeled with Data Objects for modelers who want to include more information without changing the basic behavior of the Process.

In some cases, the Data Object will be shown being sent from one Process to another, via a Sequence Flow (see Figure 21). Data Objects will also be associated with Message Flows. They are not to be confused with the message itself, but could be though of as the "payload" or content of some messages.

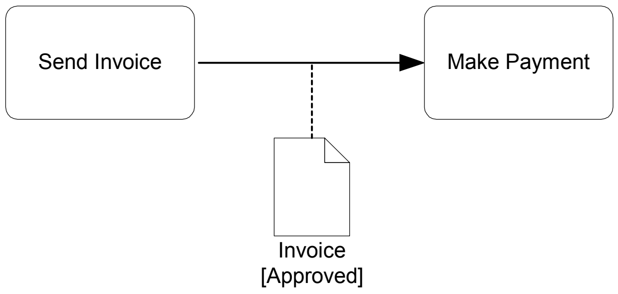


Figure 21 A Data Object associated with a Sequence Flow

In other cases, the same Data Object will be shown as being an input, then an output of a Process (see Figure 22). Directionality added to the Association will show whether the Data Object is an input or an output. Also, the state property of the Data Object can change to show the impact of the Process on the Data Object.

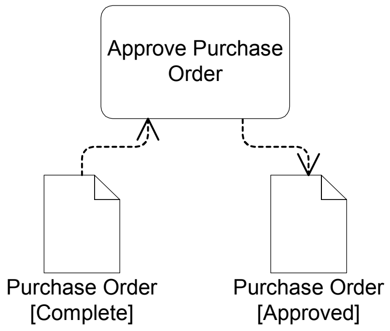


Figure 22 Data Objects shown as inputs and outputs

Attributes

The following table displays the identified attributes of a Text Annotation:

Attribute	Description
Name	Name is a property that is a text description of the Data Object. Multiple Data Objects can share the same name within one Process.
State ?	State is an optional property that indicates the impact the Process has had on the Data Object. Multiple Data Objects with the same name can share the same state within one Process.
Target: ObjectName	Target is an option property that identifies the object that the Data Object is connected to through an Association.
Documentation ?	The modeler can add optional text documentation about the Data Object.

Table 17 Data Object Attributes

Sequence Flow Connections

Refer to the section entitled “Sequence Flow Rules” on page 24 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ A Data Object cannot be a target for a Sequence Flow.
- ❖ A Data Object cannot be a source for a Sequence Flow.

Message Flow Connections

Refer to the section entitled “Message Flow Rules” on page 25 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ A Data Object cannot be a target for a Message Flow.
- ❖ A Data Object cannot be a source for a Message Flow.

Mapping to Execution Languages

Data Objects do not have any Mapping to Execution Languages. They provide detailed information about how data will interact with the flow objects and flows of Processes.

4.6 Text Annotation

Text Annotations are a mechanism for a modeler to provide additional information for the reader of a BPMN diagram. The Text Annotation object is an open rectangle and can be connected to a specific object on the diagram with an Association (see Figure 23). Text associated with the Annotation can be placed within the bounds of the open rectangle.

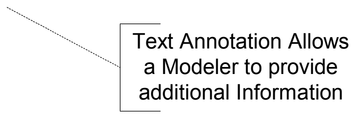

Text Annotation Allows a Modeler to provide additional Information

Figure 23 A Text Annotation

Text Annotations do not affect the flow of the Process and do not map to any BPEL4WS or BPML elements.

Attributes

The following table displays the identified attributes of a Text Annotation:

Attribute	Description
Text: String	Text is a property that is text that the modeler wishes to communicate to the reader of the diagram.
Target ?: ObjectName	Target is an optional property that identifies what object the Annotation is connected to through an Association.

Table 18 Text Annotation Attributes

Sequence Flow Connections

Refer to the section entitled “Sequence Flow Rules” on page 24 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ A Text Annotation cannot be a target for a Sequence Flow.
- ❖ A Text Annotation cannot be a source for a Sequence Flow.

Message Flow Connections

Refer to the section entitled "Message Flow Rules" on page 25 for the entire set of objects and how the may be source or targets of Sequence Flows.

- ❖ A Text Annotation cannot be a target for a Message Flow.
- ❖ A Text Annotation cannot be a source for a Message Flow.

Mapping to Execution Languages

Text Annotations can map to the *documentation* element of BPM execution languages. If the Annotation is associated with a flow object and that object has a straight-forward mapping to a BPM execution language element, then the text of the Annotation will be placed in the *documentation* element of that object. If there is no straight-forward mapping to a BPM execution language element, then the text of the Annotation will be appended to the *documentation* element of the *process*.

5. Connecting Objects

This section defines the graphical objects used to connect two objects together (i.e., the connecting lines of the diagram) and how the flow progresses through a Process (i.e., through a straight sequence or through the creation of parallel or alternative paths).

5.1 Graphical Connecting Objects

There are two ways of connecting objects in BPMN: a Flow, either sequence or message, and an Association. Sequence Flows and Message Flows, to a certain extent, represent orthogonal aspects of the business processes depicted in a model, although they both affect the performance of activities within a Process. In keeping with this, Sequence Flows will generally flow in a single direction (either left to right, or top to bottom) and Message Flows will flow at a 90° from the Sequence Flows. This will help clarify the relationships for a diagram that contains both Sequence Flows and Message Flows. However, BPMN does not restrict this relationship between the two types of Flows. A modeler can connect either type of Flow in any direction at any place in the diagram.

The next three sections will describe how these types of connections function in BPMN.

5.1.1 Sequence Flow

A Sequence Flow is used to show the order that activities will be performed in a Process. Each Flow has only one source and only one target. The source and target must be from the set of the following flow objects: Start Event, Intermediate Event, End Event, Task, Sub-Process, and Decision. During performance or simulation of the process, a Token will leave the source flow object, traverse down the Sequence Flow, and enter the target flow object.

The Sequence Flow is a thin, solid line with a solid arrowhead (see Figure 24). Text can be attached to a Sequence Flow and displayed above and/or below, across, or through the line (for vertical lines).

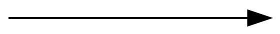


Figure 24 A Sequence Flow

A Sequence Flow, unlike some process modeling languages, does not have a conditional expression property. This means that once the Token leaves the source object, it will *always* traverse the Flow. There is no “decision” to be made. Such “decisions” are made in the Decision object. By adding conditionals to a Flow also adds ambiguity about the exact behavior of the Process.

BPMN is designed to avoid ambiguity to the Sequence Flow to satisfy a general requirement that “BPMN must expose all normal Sequence Flow semantics in the notation—there will be no hidden semantics.” For those notations that employ a condition expression on a flow, there is ambiguity about the behavior of the process. Sometimes a Token will travel down the Flow sometimes not—depending on the evaluation of the condition expression. It is not always clear what happens to the Token if it does not go down the Flow. The Token might “jump” to the end of the Process or some other location in the Process, depending on the process configuration and notation semantics. A viewer of the model has to have a clear understanding of the semantics of the diagram to understand this behavior, which might be difficult in a complex diagram because the behavior is not completely exposed (graphically) to the viewer of the diagram.

It is appropriate for lower-level execution languages to use condition expressions on transitions between activities, since it provides a compact way of defining behavior. However, BPMN is a visual language, intended to communicate the behavior to a human audience. Thus, it is appropriate for BPMN to expose as much of the process behavior as possible and to make this behavior as obvious to the viewer as possible. This may make BPMN a bit more visually verbose, but the behavior of the processes will be obvious. A BPMN diagram can then be mapped to an executable BPM language (BPEL4WS or BPML) that has a well constructed, but less obvious behavior (to humans).

Attributes

The following table displays the identified attributes of a Sequence Flow:

Attribute	Description
Name: String	Name is an property that is text description of the Sequence Flow. If the modeler does not enter a name, the name will default to: “SF”+<SourceObjectName>+”to”+<TargetObjectName>. This name is necessary since Sequence Flow names may be used in Complex Flow Conditions for objects that have multiple incoming Sequence Flows.
Source: FlowObjectName	Source is a property that identifies which flow object the Sequence Flow is connected *from*; i.e., the Sequence Flow is an outgoing flow from that object.
Target: FlowObjectName	Target is a property that identifies which flow object the Sequence Flow is connected *to*; i.e., the Sequence Flow is an incoming flow to that object.
Association *	Zero or more Associations can be associated with the Sequence Flow.
Documentation ?	The modeler can add optional text documentation about the Sequence Flow.

Table 19 Sequence Flow Attributes

Mapping to Execution Languages

The following two sections describe how the use of Sequence Flow will map to BPEL4WS and BPML, respectively.

BPEL4WS

A Sequence Flow may not have a specific mapping to a BPEL4WS in most situations. However, when there is a section of the diagram that contains parallel activities, then Sequence Flow may map to the *link* element. Details of this mapping are TBD. In general, the set of Sequence Flows within a Pool will determine how BPEL4WS elements are derived and the boundaries of those elements. Refer to the section entitled “Mapping to Execution Languages” on page 133 for more details.

BPML

A Sequence Flow does not have a specific mapping to a BPML element. However, the set of Sequence Flows within a Pool will determine how BPML elements are derived and the boundaries of those elements. Refer to the section entitled "Mapping to Execution Languages" on page 133 for more details.

5.1.2 Message Flow

A Message Flow is used to show the flow of messages between two entities that are prepared to send and receive them. In BPMN, two separate Pools in the diagram will represent the two entities. Thus,

❖ Message Flow always connects two Pools, either to the Pools themselves or to flow objects within the Pools. They cannot connect two objects within the same Pool.

The Message Flow is drawn with a dashed line with an open arrowhead (see Figure 25). Text can be attached to a Message Flow and displayed above and/or below, across, or through the line (for vertical lines).

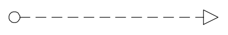


Figure 25 A Message Flow

The Message Flow can connect directly to the boundary of a Pool (See Figure 26), especially if the Pool does not have any process details within (e.g., is a "Black Box").

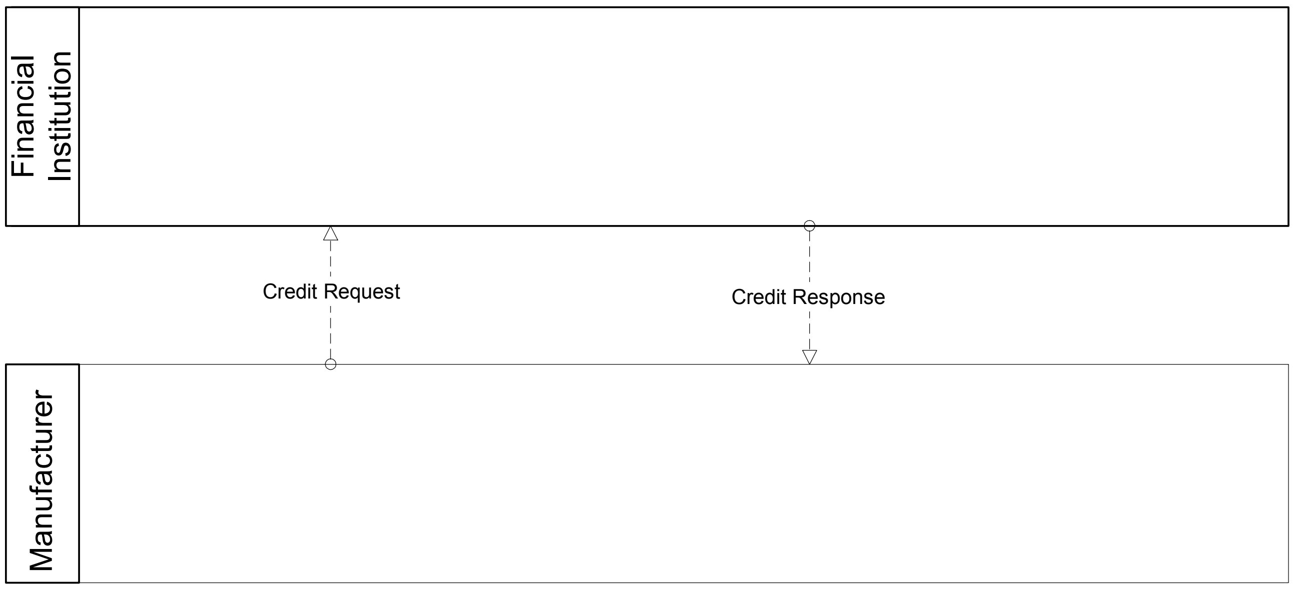


Figure 26 Message Flow connecting to the boundaries of two Pools

A Message Flow can also cross the boundary of a Pool and connect to a flow object within that Pool (see Figure 27).

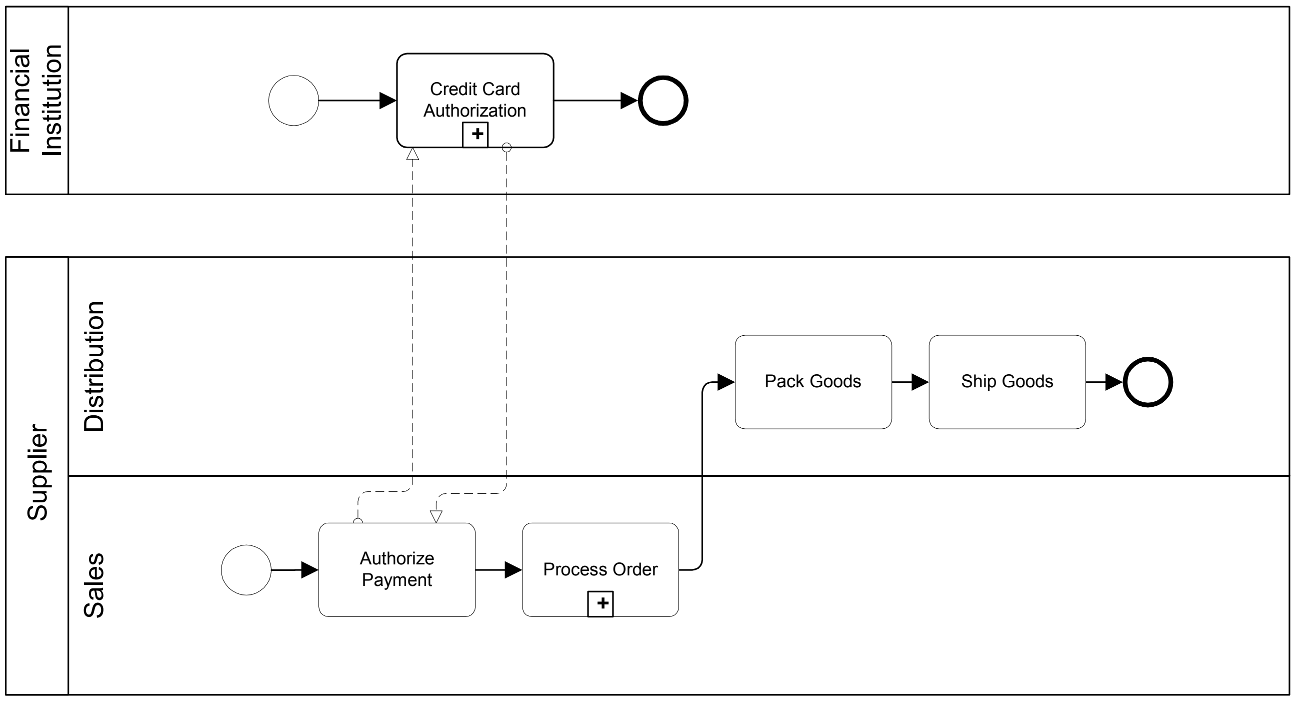


Figure 27 Message Flow connecting to flow objects within two Pools

If there is an Expanded Sub-Process in one of the Pools, then the message flow can be connected to either the boundary of the Sub-Process or to objects within the Sub-Process. If the Message Flow is connected to the boundary to the Expanded Sub-Process, then this is equivalent to connecting to the Start Event for incoming Message Flows or the End Event for outgoing Message Flows (see Figure 28).

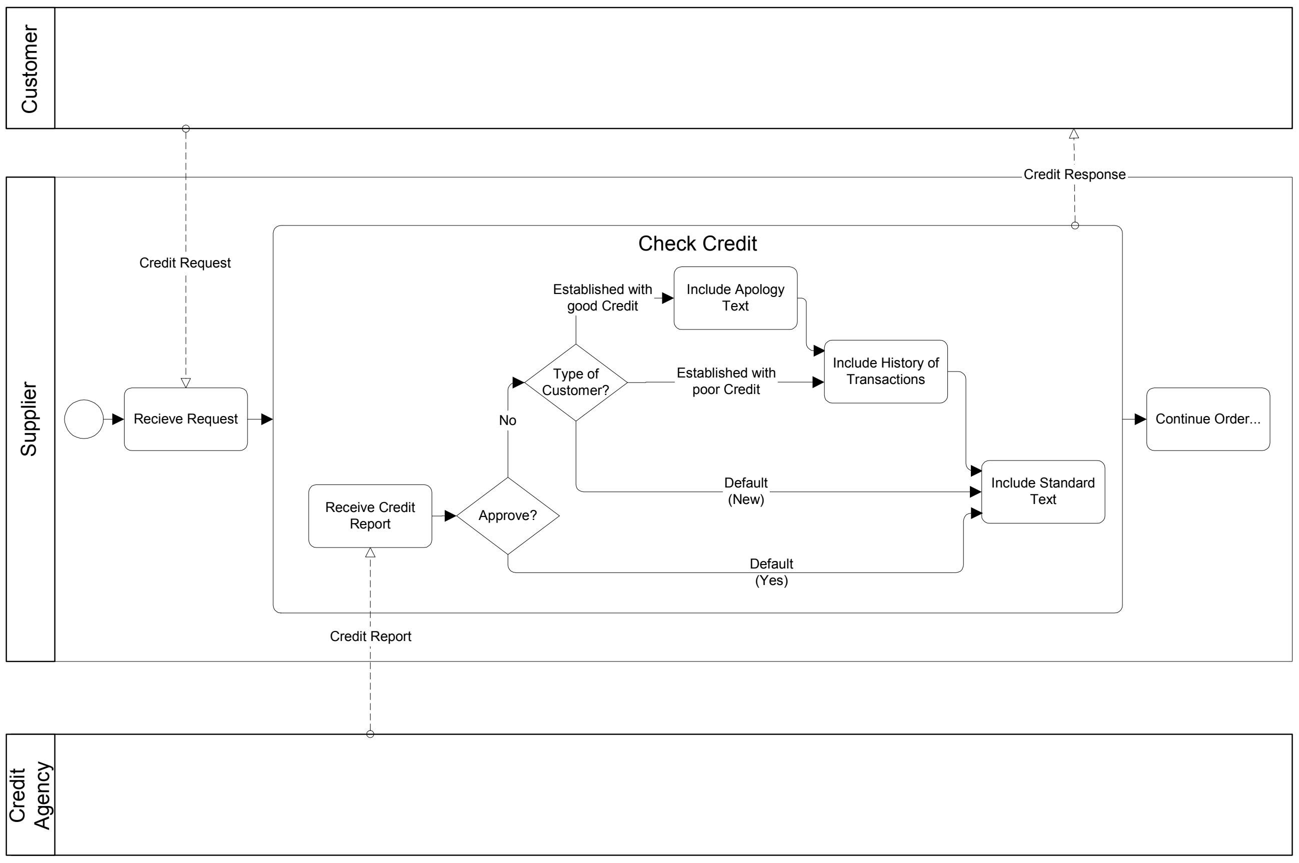


Figure 28 Message Flow connecting to boundary of Sub-Process and Internal objects

Attributes

The following table displays the identified attributes of a Message Flow:

Attribute	Description
Name ?: String	Name is an optional property that is text description of the Message Flow.
Message ?: MessageName	Message is an optional property that identifies the Message that is being sent.
SourcePool: PoolName	SourcePool is a property that identifies which Pool the message Flow is connected *from*; i.e., the Message Flow is an outgoing flow from that Pool. The Message Flow can originate from the boundary of the Pool or an object within the Pool.
SourceObject: FlowObjectName	If the source is an object within the Pool, then the SourceObject property will identify that object. The set of objects that a Message Flow can originate from are: Task, Sub-Process, and End Event.
TargetPool: PoolName	TargetPool is a property that identifies which Pool the Message Flow is connected *to*; i.e., the Message Flow is an incoming flow to that Pool. The Message Flow can target the boundary of the Pool or an object within the Pool.
TargetObject: FlowObjectName	If the target is an object within the Pool, then the TargetObject property will identify that object. The set of objects that a Message Flow can target from are: Task, Sub-Process, Start Event, and Intermediate Event.
Association *	Zero or more Associations can be associated with the Message Flow.
Documentation ?	The modeler can add optional text documentation about the Message Flow.

Table 20 Message Flow Attributes

Mapping to Execution Languages

A Message Flow does not have a specific mapping to a BPEL4WS or BPML element. It represents a message that is send through a WSDL *operation* that is referenced in a BPEL4WS *receive*, *reply*, or *invoke* or a BPML *action*.

5.1.3 Association

An Association is used to associate information and artifacts with flow objects. Text and graphical non-flow objects can be associated with the flow objects and flows. Since the Association does not affect the sequence flow or message flow of the process in any way, it, and the information it is connected to, can be added or subtracted at any time in a diagram (i.e., with a modeling tool) and not affect the behavior of the process.

Note: We have not dealt with Spawning and Joining yet and the Association may be used to support these features. Thus, the definition of the Association may change. It is also possible that an Association may be used to show the bindings between activities in a private process and activities in a Collaboration Process.

The Association is drawn with a dotted line (see Figure 29). Text can be attached to a Association and displayed above and/or below, across, or through the line (for vertical lines).

Figure 29 An Association

If there is a reason to put directionality on the association then a line arrowhead can be added (see Figure 30)

---------------------->

Figure 30 A directional Association

An Association is used to connect user-defined text with a flow object (see Figure 31).

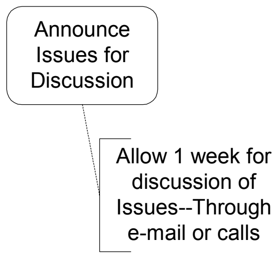

Announce Issues for Discussion
Allow 1 week for discussion of Issues--Through e-mail or calls

Figure 31 An Association of Text Annotation

An Association is also used to associate Data Objects with other objects (see Figure 32). A Data Object is used to show how documents are used throughout a Process. Refer to the section entitled “Data Object” on page 69 for more information on Data Objects.

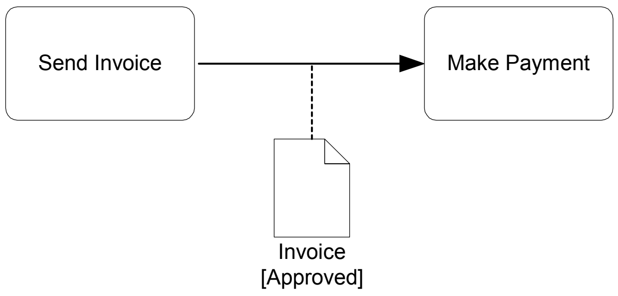


Figure 32 An Association connecting a Data Object with a Flow

Attributes

The following table displays the identified attributes of a Association:

Attribute	Description
Name ?: String	Name is an optional property that is text description of the Association.
Source: ObjectName	Source is a property that identifies which object the Association is connected *from*. The set of objects that an Association can connect to are: Pool, Lane, all Events, Task, Sub-Process, Decision, Sequence Flow, and Message Flow.
Target: ObjectName	Target is a property that identifies which artifact the Association is connected *to*. Associations can only connect to artifacts.
Direction (None | To | From | Both): None	Direction is a property that defines whether or not the Association shows any directionality with an arrowhead. The default is None (no arrowhead). A value of To means that the arrowhead will be at the Source object. A value of From means that the arrowhead will be at the Target artifact. A value of Both means that there will be an arrowhead at both ends of the Association line.
Documentation ?	The modeler can add optional text documentation about the Association.

Table 21 Association Attributes

Mapping to Execution Languages

An Association does not have a specific mapping to an execution language element. These objects and the artifacts they connect to provide additional information for the reader of the BPMN diagram, but do not directly affect the execution of the Process.

5.2 Sequence Flow Mechanisms

The Sequence Flow mechanisms described in the following sections are divided into four types: Normal flow, Exception Flow, Transaction Compensation flow, and Ad Hoc (no flow).

5.2.1 Normal Flow

Normal sequence flow refers to the flow that originates from a Start Event and continues through activities via alternative and parallel paths until it ends at an End Event. As stated above, the normal sequence flow should be completely exposed and no flow behavior hidden. This means that a viewer of a BPMN diagram will be able to trace through a series of flow objects and Sequence Flows, from the beginning to the end of a given level of the Process without any gaps or hidden “jumps” (see Figure 33). In this figure, Sequence Flows connect all the objects in the diagram, from the Start Event to the End Event. The behavior of the Process shown will reflect the connections as shown and not skip any activities or “jump” to the end of the Process.

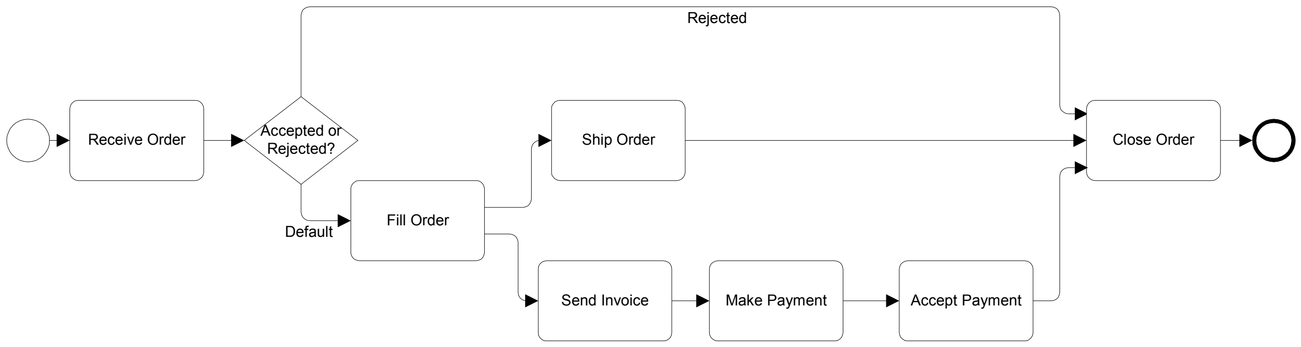


Figure 33 A Process with Normal flow

As the Process continues through the series of Sequence Flows, control mechanisms may divide or combine the Sequence Flows as a means of describing complex behavior. There are two control mechanisms of dividing (forking and splitting) and two control mechanisms of combining (joining and merging) Sequence Flows. A casual look at the definitions of the English terms for these mechanisms would indicate that each pair of terms mean basically the same thing. However, their effect on the behavior of a Process is quite different. We will continue to use these English terms but will provide specific definitions about how they affect the performance of the process in the next few sections of this specification. In addition, we will relate these BPMN terms to the terms OR-Split (for split), Or-Join (for merge), AND-Split (for fork), and AND-Join (for join), as defined by the Workflow Management Coalition.[1]

Editor's Note: The graphical mechanism for forking, joining, and merging may is an open issue and may change in the next version of the specification. Refer to the section entitled "Open Issues" on page 137 for a complete list of the issues open for BPMN. BPMN might employ a special flow objects that control these behaviors. Other notations, such as the UML Activity Diagram, EPCs, and IDEF3, use such control objects. Another possibility is that forking will be restricted to the Start Events, joining will be restricted to End Events, and Expanded Sub-Processes will be employed to show parallel flows within a Process. This approach has many advantages by preventing invalid process designs or processes that produce unexpected behavior. BPMI is investigating old and new concepts for handling forking, joining, splitting, and merging.

The use of an expanded Sub-Process in a Process (see Figure 34), which is the inclusion of one level of the Process within another Level of the Process, can sometimes break the traceability of the flow through the lines of the diagram. The Sub-Process is not required to have a Start Event and an End Event. This means that the series of Sequence Flows will be disrupted from border of the Expanded Sub-Process to the first object within the Expanded Sub-Process. The flow will "jump" to the first object within the Expanded Sub-Process. Expanded Sub-Processes will often be used, as seen in the figure, to include exception handling. A requirement that modelers always include a Start Event and End Event within

1. The *Workflow Management Coalition Terminology & Glossary*. The Workflow Management Coalition. Document Number WFMC-TC-1011. April 1999.

Expanded Sub-Processes would mainly add clutter to the diagram without necessarily adding to the clarity of the diagram. Thus, BPMN does not enforce the use of Start Events and End Events to satisfy the traceability of a diagram that contains multiple levels.

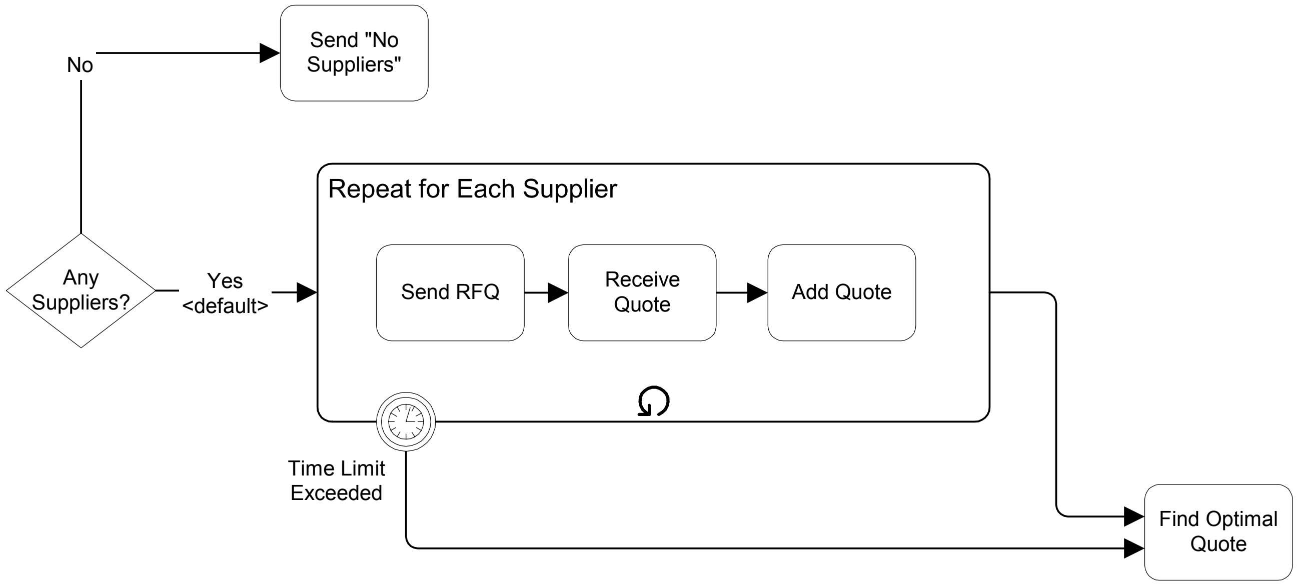


Figure 34 A Process with Expanded Sub-Process without a Start Event and End Event

A modeler may want to ensure the traceability of a diagram and can use a Start Event and End Event in an Expanded Sub-Process. One way to do this would be to attach these events to the boundary of the Expanded Sub-Process (see Figure 35). The incoming Sequence Flow to the Sub-Process can be attached directly to the Start Event instead of the boundary of the Sub-Process. Likewise, the outgoing Sequence Flow from the Sub-Process can connect from the End Event instead of the boundary of the Sub-Process. Doing this, the Normal flow can be traced throughout a multi-level Process.

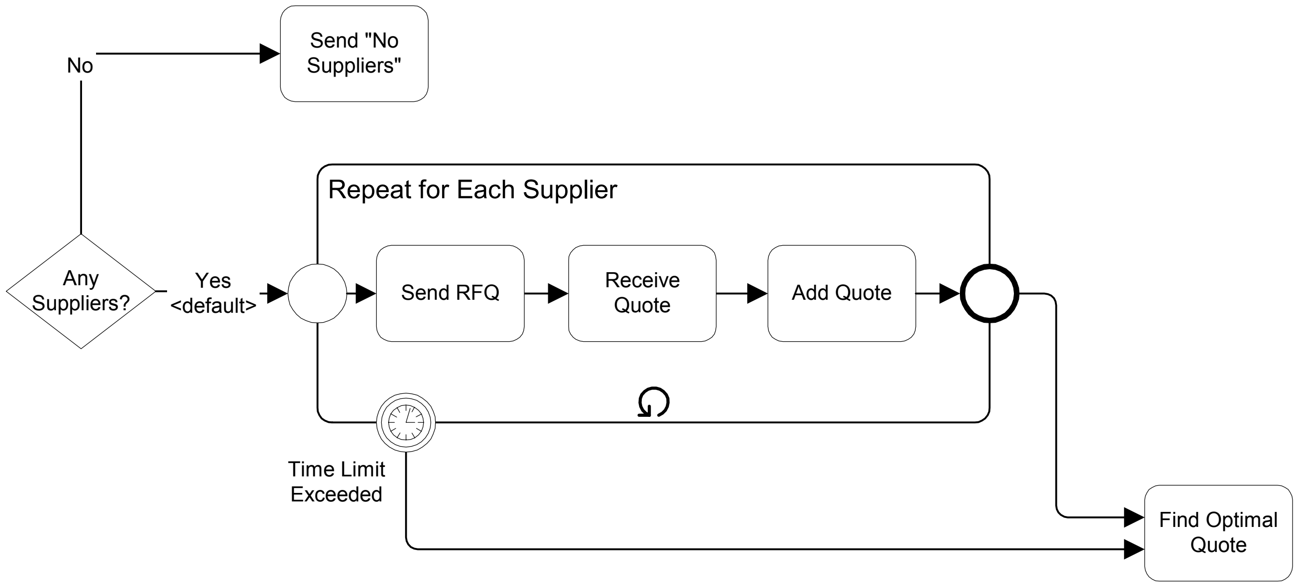


Figure 35 A Process with Expanded Sub-Process with a Start Event and End Event

When dealing with Exception Flow and Transaction Compensation Flow, the traceability requirement is also relaxed (refer to the section entitled “Exception Flow” on page 95 and “Transaction Compensation Flow” on page 99).

Splitting (OR-Split)

BPMN uses the term splitting to refer to the dividing of a path into two or more alternative paths (also known as an OR-Split). It is a place in the Process where a question is asked, and the answer determines which of a set of paths is taken. It is the “fork in the road” where a traveler, in this case a Token, can take only one of the forks (not to be confused with forking—see below).

A Decision is the object that is used in BPMN to show and control the splitting of the flow. This means that when a Token reaches a Decision, the Token will continue down only one of the alternative paths for a given instance of the Process. There are two basic mechanism for making the Decision during the performance of the Process: the first is an evaluation of a condition expression (see Figure 36); the second is receipt of a particular message (see Figure 37). The Decision is the only mechanism in the Normal Flow where alternative paths are designated. “Decisions” on page 58 for details on the Decision Object.

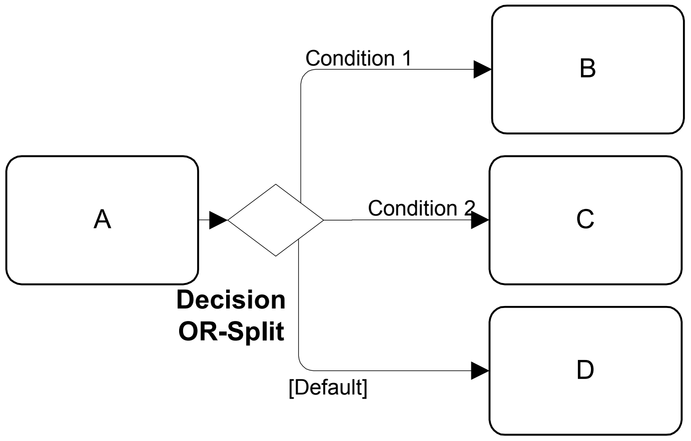


Figure 36 A Data-Based Decision Example

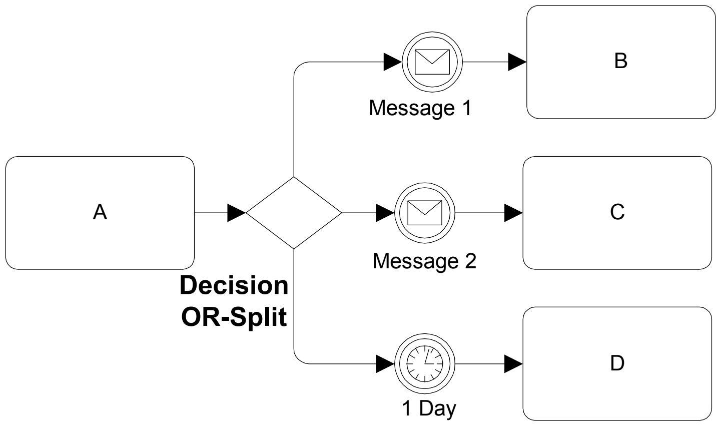


Figure 37 An Event-Based Decision Example

A Decision (an OR-Split) can be thought of as a notational convenience for combining two or more process variations into the same diagram. A Process could be modeled with different variations—each variation appropriate for a given set of conditions and shown a separate diagram. Figure 38 shows the same overall Process as in Figure 36, but modeled with three different variations. The diagram with a Decision object, as shown in Figure 36, provides a more compact and intuitive way to show how the process will be performed.

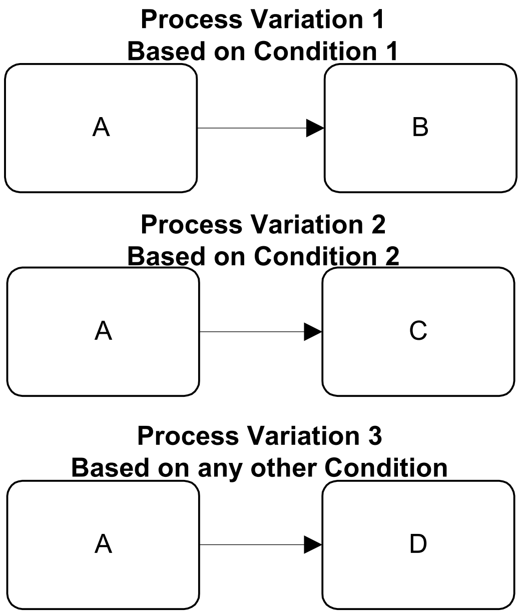


Figure 38 Three variations of a Process

As the number of Decisions increase in a Process, the number of possible variations will increase almost exponentially. However, only one of those variations will actually be performed for a given instance. The concept of a Process as a co-location of a set of variations is important to remember in the next sections that discuss the merging, forking, and joining of the flow. Each variation should be cleanly traceable through the Process and should not be intermingled with parallel groupings of activities.

Merging (Or-Join)

BPMN uses the term merging to refer to the combining of two or more alternative paths into one path (also known as an a OR-Join). It is a place in the process where two or more alternative paths begin to traverse activities that are common to each of the paths. Theoretically, each alternative path can be modeled separately to a completion (an End Event). However, merging allows the paths to overlap and avoids the duplication of activities that are common to the separate paths (variations). For a given instance of the Process, a Token would actually only see the sequence of activities that exist in one of the paths as if it were modeled separately to completion.

The graphical mechanism to merge alternative paths is simple: there are two or more incoming Sequence Flows to a flow object (see Figure 39). In general, this means that a Token will travel down one of the alternative paths (for a given Process instance) and will continue from there. For that instance, Tokens will never arrive down the other alternative paths. However, the continuation of the Token is subject to the Flow Condition Attributes of the target object (refer to the section entitled “Flow Conditions” on page 88 for details on how Flow Conditions will affect the flow of the Process).

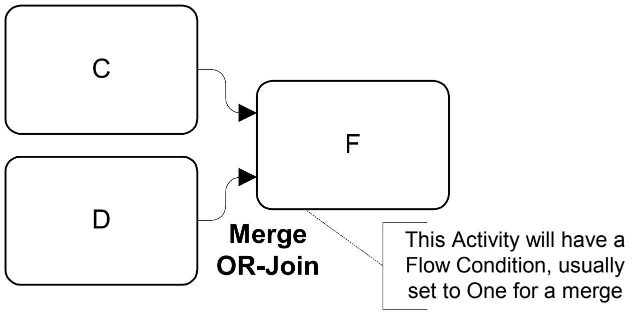

C
D
F
Merge
OR-Join
This Activity will have a Flow Condition, usually set to One for a merge

Figure 39 Merging – the joining of alternative paths

There are no graphical representations for the Flow Conditions of a flow object. These will be hidden attributes. However, the modeler can attach to the object a Text Annotation that displays the Flow Condition.

There is no specific correlation between the merging of a set of paths and the splitting that occurs through a Decision object. For example, a Decision may split a path into three separate paths, but these three paths do not need to be merged at the same object. Figure 40 shows that two of three alternative paths are merged at Task “F.” All of the paths eventually will be merged, but this can happen through any combination of objects, including lone End Events. In fact, each path could end with a separate End Event.

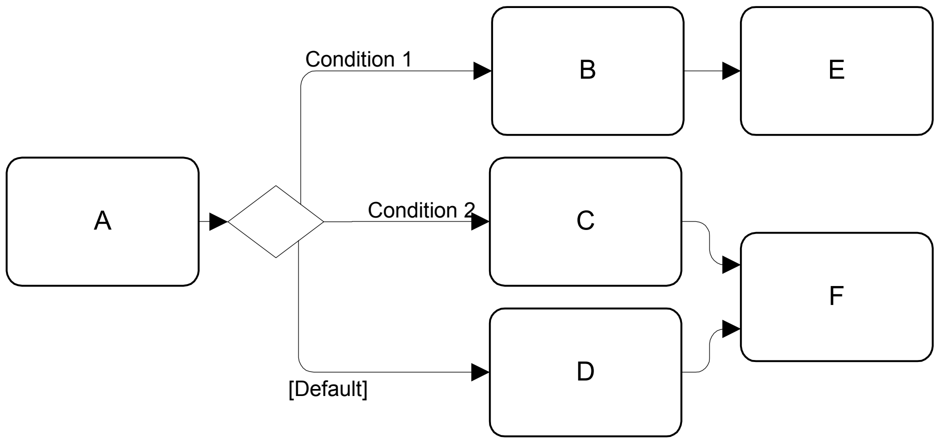


Figure 40 The Split-Merge Relationship is not Fixed

Thus, for alternative flow, BPMN contrasts with BPEL4WS and BPML, which are mainly block structured. A BPEL4WS *switch* and *pick* or a BPML *switch* and *choice*, which map to the BPMN Decision, are specific block structures that have well-defined boundaries. While there are no obvious boundaries to the alternative paths created by a Decision, the appropriate boundaries can be derived by an evaluation of the configuration of Sequence Flows that follow the Decision. The locations in the Process where Tokens of the same identity are merged through multiple incoming Sequence Flows will determine the boundaries for a specific Decision. The boundary may in fact be the end of the Process. More detail on the evaluation of BPEL4WS and BPML element boundaries can be found in the section entitled “Mapping to Execution Languages” on page 133.

The graphical mechanism for merging alternative paths is also the same as the mechanism for joining parallel paths (refer to the section entitled "Joining (AND-Join)" on page 87). Thus, it is possible that a given object may have a combination of parallel and alternative incoming Sequence Flows. This feature allows the modeler flexibility in configuring a process diagram to reflect complex behavior in a compact form. However, the actual behavior and the expected behavior may not coincide in complex situations since there can be ambiguity in the diagram when alternative and parallel flows are combined. This is particularly true when alternative paths cross the implicit boundary of a group of parallel paths. The section entitled "Avoiding Illegal Models and Unexpected Behavior" on page 93 will discuss this issue in more detail.

Forking (AND-Split)

BPMN uses the term forking to refer to the dividing of a path into two or more parallel paths (also known as an AND-Split). It is a mechanism that will allow activities to be performed concurrently, rather than serially. This means a separate Token will be generated to traverse each of the paths. Each Token will have two aspects to its identity. The first aspect, called the TokenID, has to with the single path that is being forked—this identity will be common to all the new Tokens. The second aspect, called the SubTokenID, which is a TokenID nested with a higher-level TokenID, will be unique for each the new paths. The start of a business process will have a single Token with a TokenID. Multiple levels of SubTokenIDs will be created as forks occur through the Process. The number of SubTokenIDs for each fork will be known. The TokenID sets will be used to join the Tokens from a given fork back together.

The graphical mechanism to create parallel paths is simple: there are two or more outgoing Sequence Flows from a flow object (see Figure 41). A special flow control object is not used to fork the path, unlike the Decision object that is used to split the path. All flow objects that can have outgoing Sequence Flows can create a fork in the flow; except for Decisions, which can have multiple, but alternative outgoing Sequence Flows.

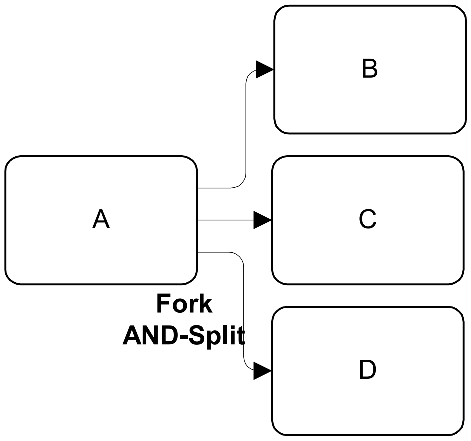


Figure 41 Forking – the creation of parallel paths

Most of the time, the paths that have been divided with a fork are combined back together through a join (refer to the next section) and synchronized before the flow will continue. However, BPMN provides advanced methods for more complex handling of parallel paths. The exact behavior will be determined by Flow Condition attributes that are contained within objects that have two or more incoming Sequence Flows (refer to the section entitled "Flow Conditions" on page 88 for more details).

Joining (AND-Join)

BPMN uses the term joining to refer to the combining of two or more parallel paths into one path (also known as an AND-Join). The graphical mechanism to join parallel paths is simple: there are two or more incoming Sequence Flows to a flow object (see Figure 42). In general, this means that Tokens created at a fork will travel down parallel paths and then meet at the joining object.

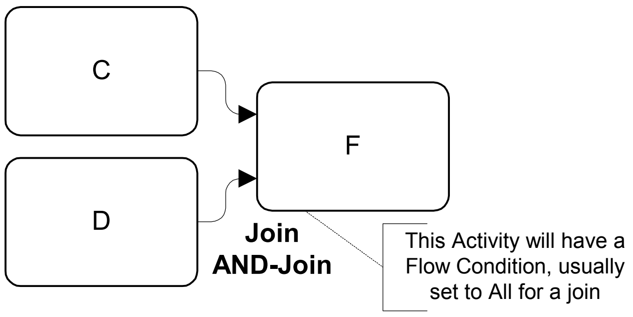

C
D
F
Join
AND-Join
This Activity will have a Flow Condition, usually set to All for a join

Figure 42 Joining – the joining of parallel paths

Most of the time, only one Token will continue past the joining object. Attributes of the object will determine how and when the Token will continue (refer to the section entitled "Flow Conditions" on page 88 for details on how Flow Conditions will affect the flow of the Process).

There are no graphical representations for the Flow Conditions of a flow object. These will be hidden attributes. However, the modeler can attach to the object a Text Annotation that displays the Flow Condition.

There is no specific correlation between the joining of a set of parallel paths and the forking that created the parallel paths. For example, a an activity may have three outgoing Sequence Flows, which creates a fork of three parallel paths, but these three paths do not need to be joined at the same object. Figure 43 shows that two of three parallel paths are joined at Task "F." All of the paths eventually will be joined, but this can happen through any combination of objects, including lone End Events. In fact, each path could end with a separate End Event.

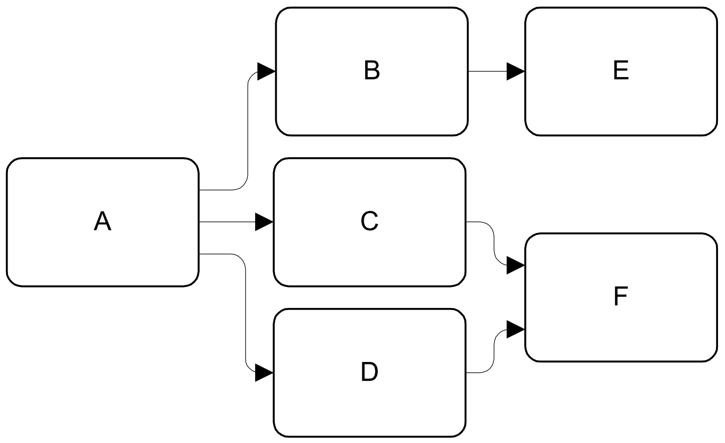


Figure 43 The Fork-Join Relationship is not Fixed

Thus, for parallel flow, BPMN contrasts with BPEL4WS and BPML, which are mainly block structured. A BPEL4WS *flow* or a BPML *all*, which map to a set of BPMN parallel activities, is a specific block structure that has a well-defined boundary. While there are no obvious boundaries to the parallel paths created by a fork, the appropriate boundaries can be derived by an evaluation of the configuration of Sequence Flows that follow the fork. The locations in the Process where Tokens of the same TokenID and all the appropriate SubTokenIDs are joined with through multiple incoming Sequence Flows will determine the boundaries for a specific block of parallel activities. The boundary may in fact be the end of the Process. More detail on the evaluation of BPEL4WS and BPML element boundaries can be found in the section entitled “Mapping to Execution Languages” on page 133.

The graphical mechanism for joining parallel paths is also the same as the mechanism for merging alternative paths (refer to the section entitled “Merging (Or-Join)” on page 84). Thus, it is possible that a given object may have a combination of parallel and alternative incoming Sequence Flows. This feature allows the modeler flexibility in configuring a process diagram to reflect complex behavior in a compact form. However, the actual behavior and the expected behavior may not coincide in complex situations since there can be ambiguity in the diagram when alternative and parallel flows are combined. This is particularly true when alternative paths cross the implicit boundary of a group of parallel paths. The section entitled “Avoiding Illegal Models and Unexpected Behavior” on page 93 will discuss this issue in more detail.

Flow Conditions

If an object has more than one incoming Sequence Flows, then a Flow Condition must be set to specify how Tokens will be handled when they arrive, which will determine when the object is ready to be instantiated. The Flow Condition will not be evaluated until at least one Token has arrived at the object. As a result of the evaluation of the condition, the object will be instantiated or the object will wait until another Token arrives before the Flow Condition is evaluated again. The TokenID and any SubTokenIDs of the Token will be taken into consideration for the evaluation of the Flow Condition. The IDs have to match up appropriately if more than one Token is required for object instantiation. This includes any Token that arrives via an upstream connection of a Sequence Flow (a loop), which will have a Path-SubTokenID set that identifies it as being a part of a loop. More detail on the evaluation of Token Path-SubTokenIDs can be found in the section entitled “Mapping to Execution Languages” on page 133.

There are three types of Flow Conditions, as described in the following sections: One, All, and Complex.

One

The One Flow Condition is mainly used for incoming Sequence Flows that are alternative (see Figure 44). The condition means that the first Token that arrives to the object through any of the incoming Sequence Flows will cause the instantiation of the object. If there are parallel incoming Sequence Flows, then addition Tokens will follow the first one, but they will not cause another instantiation of the object. The additional Tokens will be consumed, but ignored for additional flow. Note: It is an open issue to provide a mechanism that will specify that the additional Tokens will not be ignored, but will also continue throughout the flow (in a sense, they are not joined at the joining location). Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

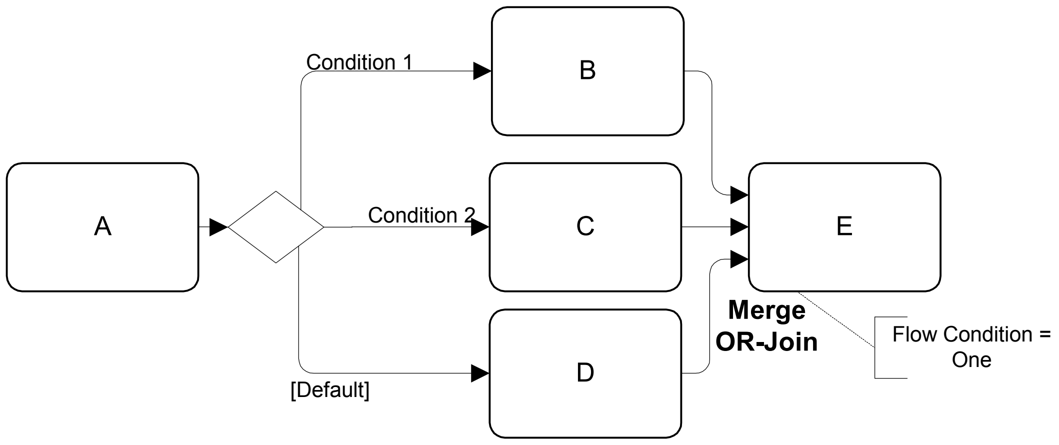


Figure 44 Flow Condition of One

All

The All Flow Condition is mainly used for incoming Sequence Flows that are parallel (see Figure 45). The condition means that one Token for each incoming Sequence Flow must arrive at the object before that object can be instantiated.

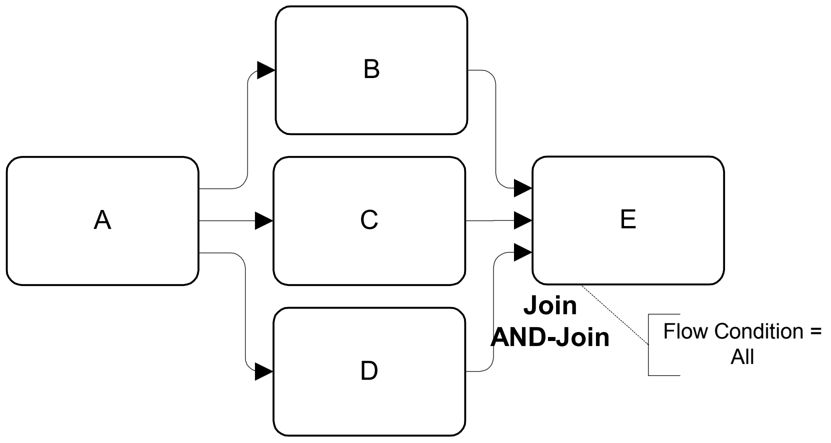


Figure 45 Flow Condition of All

A modeler must be careful in using the All Flow Condition. If there are alternative incoming Sequence Flows, then it is not possible for all of the Tokens to arrive for a given instance of the Process. Thus, the activity will not be able to start since it will be waiting for a Token that will never arrive. The section entitled “Avoiding Illegal Models and Unexpected Behavior” on page 93 will discuss this issue in more detail.

Complex

A Complex Flow Condition is any condition that is not One or All. The simplest case would be a single number (e.g., two) that would be required before the condition is satisfied. Figure 46 shows a Process where Task “G” has three incoming Sequence Flows. However, because of the Decision that precedes Task “G,” only two of the three incoming Sequence Flows will have Tokens when the Process is performed. If the modeler wants all preceding Tasks (that will be performed) to complete before Task “G” is performed, then a Flow Condition of “two” would be required.

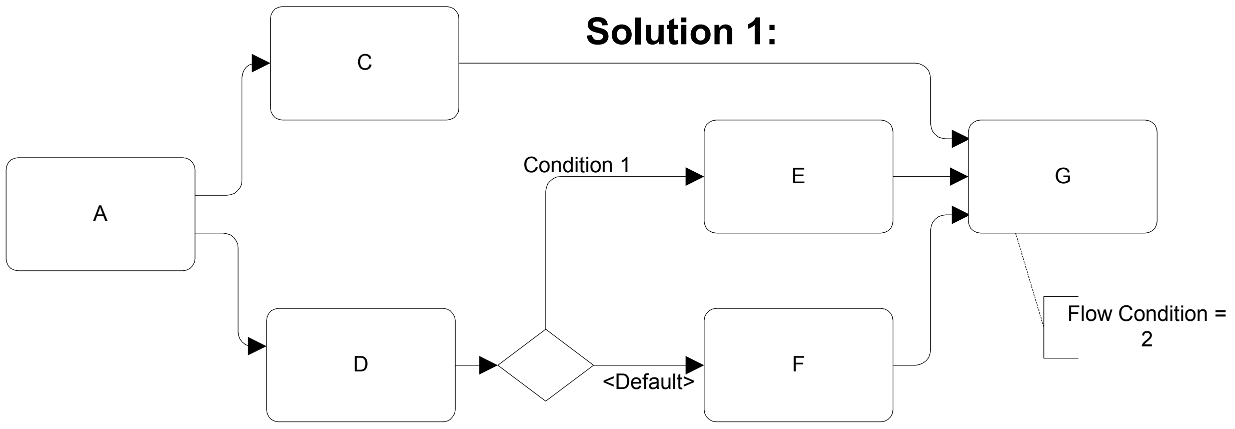


Figure 46 A Complex Flow Condition

If the number of parallel incoming Sequence Flows is greater than the number required, as specified by the Flow Condition, then the flow will continue when the required number of Tokens arrives, but any additional Tokens will be consumed and then ignored in terms of continuing the flow. Note: As with the One Flow Condition, it is an open issue to provide a mechanism that will specify that the additional Tokens will not be ignored, but will also continue throughout the flow (in a sense, they are not joined at the joining location). Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

A Complex Flow Condition may reference Process data attributes, Activities, and/or Sequence Flows for evaluation to determine when the object is to be instantiated. For example, the Flow Condition for Task “G” in Figure 47 could be set to check the names of the Sequence Flows that are incoming to the object for the Tokens to travel through. The Flow Condition would specify that a Token is expected from the Sequence Flows coming from Task “C” and Task “E” or, alternatively Task “C” and Task “F.” The Flow Condition could be set as follows:

(SequenceFlow=”SFCtoG” AND SequenceFlow=”SFEtoG”) OR
(SequenceFlow=”SFCtoG” AND SequenceFlow=”SFFtoG”)

This expression would achieve the same results as a Flow Condition of “two.”

Looping

BPMN provides 2 (two) mechanisms for looping within a Process. The first involves the use of attributes of activities to define the loop. The second involves the connection of Sequence Flows to "upstream" objects.

Activity Looping

The attributes of Tasks and Sub-Processes will determine if they are repeated or performed once. There are two types of loops: Standard and ForEach.

For Standard Loops:

- If the loop condition is evaluated before the activity, this is generally referred to as a "while" loop. This means that the activities will be repeated as long as the condition is true. The activities may not be performed at all (if the condition is false the first time) or performed many times.
- If the loop condition is evaluated after the activity, this is generally referred to as an "until" loop. This means that the activities will be repeated until a condition becomes true. The activities will be performed at least once or performed many times.

For ForEach Loops:

- If the Timing is serial, then this becomes much like a while loop with a set number of iterations the loop will go through. These are often used in processes where a specific type of item will have a set number of sub-items or line items. A ForEach loop will be used to process each of the line items.
- If the Timing is parallel, this is generally referred to as a multiple instance of the activities. An example of this type of feature would be used in a process to write a book, there would be a Sub-Process to write a chapter. There would be as many copies or instances of the Sub-Process as there are chapters in the book. All the instances could begin at the same time.

Note: In a later version of the BPMN specification, the Loop Marker might be modified to indicate that Loop is set to be performed in parallel.

Those activities that are repeated (looped) will have a loop marker placed in the bottom center of the activity shape (see Figure 47 and Figure 48).

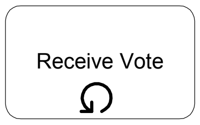


Figure 47 A Task with a Loop Marker

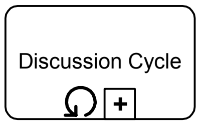


Figure 48 A Collapsed Sub-Process with a Loop Marker

Expanded Sub-Processes also can have a loop marker placed at the bottom center of the Sub-Process rectangle (see Figure 49). The entire contents of the Sub-Process will be repeated as defined in the attributes.

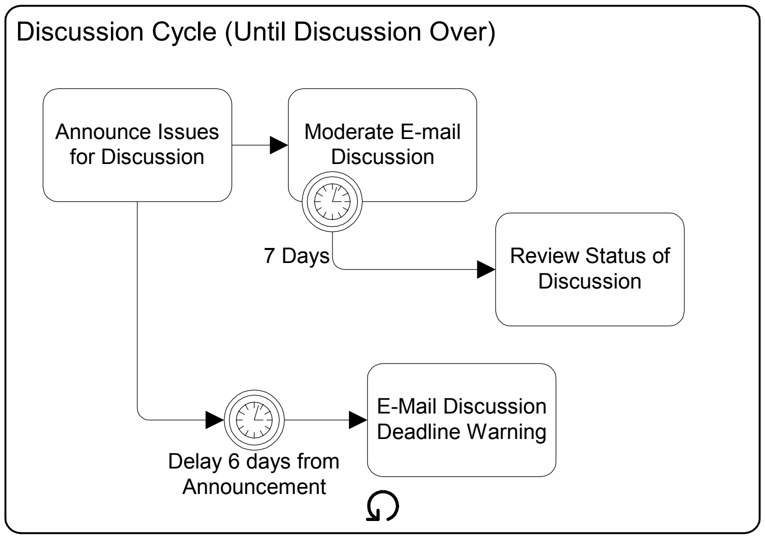

Discussion Cycle (Until Discussion Over)
Announce Issues for Discussion
Moderate E-mail Discussion
7 Days
Review Status of Discussion
E-Mail Discussion Deadline Warning
Delay 6 days from Announcement

Figure 49 An Expanded Sub-Process with a Loop Marker

Sequence Flow Looping

Loops can also be created by connecting a Sequence Flow to an "upstream" object. An object is considered to be upstream if that object has an outgoing Sequence Flow that leads to a series of other Sequence Flows, the last of which turns out to be an incoming Sequence Flow to the original object. That is, that object produces a Token and that Token traverses a set of Sequence Flows until the Token reaches the same object again.

Usually these connections follow a Decision so that the loop is not infinite (see Figure 50). If the Sequence Flow goes directly from a Decision to an upstream object, this is an "until" loop. The set of looped activities will occur until a certain condition is true.

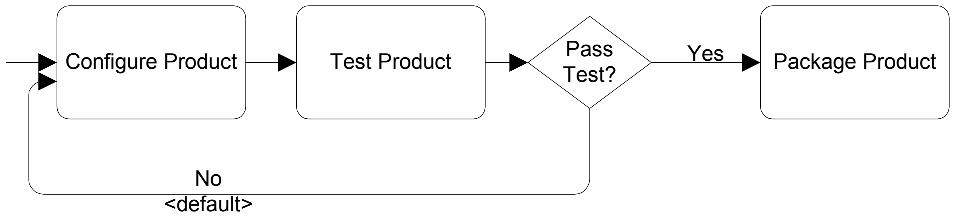


Figure 50 An Until Loop

A while loop is created by making the decision first and then performing the repeating activities or moving on in the Process (see Figure 51). The set of looped activities may not occur or may occur many times.

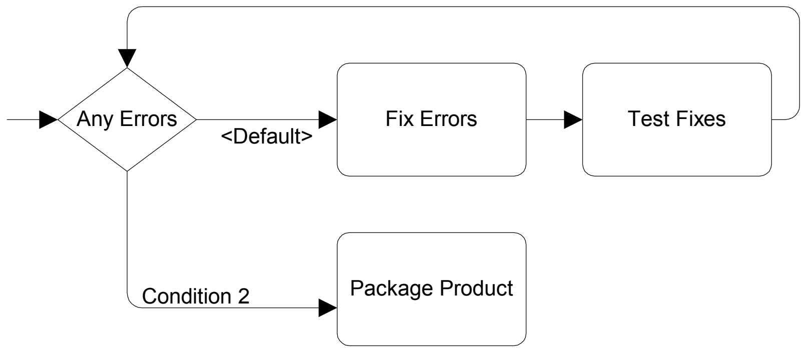


Figure 51 A While Loop

Passing the Sequence Flow to and from Sub-Processes

If a Process is used within another Process, then it is a Sub-Process. The Sequence Flow will start at the parent Process and then pass to the Sub-Process and then will pass back to the parent process. Most of the time the flow (a Token) will reach a Sub-Process, get transferred to the Start Event of the Sub-Process, traverse the Sequence Flows of the Sub-Process, reach the End Event of the Sub-Process, and, finally, get transferred back to the parent Process to continue. If the Sub-Process contains parallel flows, then all the flows must complete before the Token is transferred back to the parent Process. This functionality treats the Sub-Process as a self-contained “box” of activities.

Refer to the section entitled “References” on page 134 for a complete list of the issues open for BPMN.

In many process methodologies, the flow will pass back to the parent only when all the activity within the Sub-Process has completed, even if there are multiple Sequence Flows from the Sub-Process object.

End Events have a Boolean (True/False) property named PassThrough, which is False by default. This property specifies if an End Event can pass the flow back to a parent before the Sub-Process has completed.

Avoiding Illegal Models and Unexpected Behavior

BPMN, being a graph-structured diagram, rather than having a block-structure like BPEL4WS or BPML, provides a great flexibility for depicting complex process behavior in a fairly compact form. However, the free-form nature of BPMN can create modeling situations that cannot be executed or will behave in a manner that is not expected by the modeler. These types of modeling problems can occur because there is no tight relationship between forks and joins or splits and merges. A block structure provides these tight relationships, but a graph-structure allows these flow control mechanisms to be mixed and matched at the discretion of the modeler. Some combinations of these control elements will create Processes that cannot be executed or will create behavior that was not intended by the modeler. The situation where alternative paths cross the implicit boundary of a group of parallel paths can cause an invalid model.

Figure 52 shows such a model. Task “D” is an activity that has two incoming Sequence Flows; one from a forked path (after a split path) and one from a split path. This can create a problem at Task “E,” which also has multiple incoming Sequence Flows. The Sequence

Flow from Task “B” is crossing the implicit boundary of the fork created after Task “A.” As a result, if the “Yes” Sequence Flow is taken from the Decision in the diagram (Variation 1), then Task “E” can expect two Tokens to arrive—one from Task “C” and one from Task “D.” However, if the default Sequence Flow is taken from the Decision (Variation 2), Task “E” can expect only one Token will arrive—one from Task “D.”

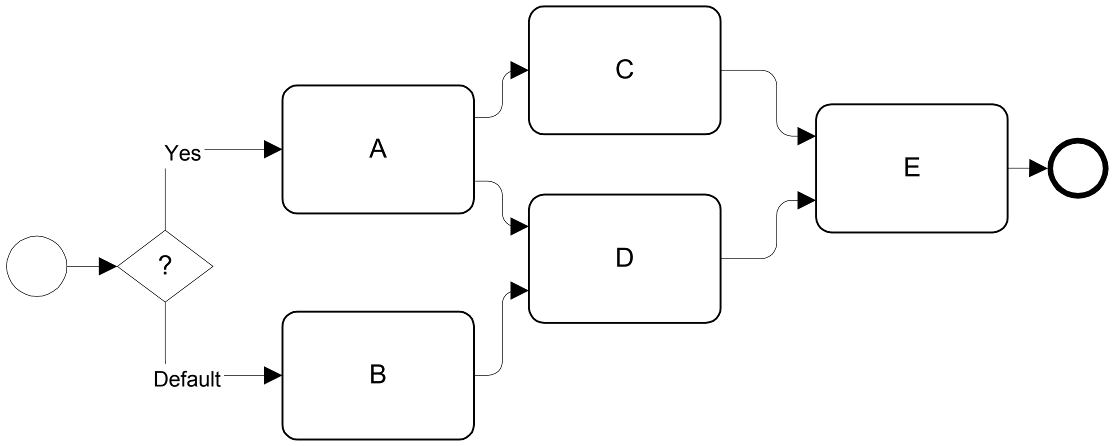


Figure 52 Potentially an invalid model

If the Flow Condition for Task “E” is set to One, then the behavior will be fine for Variation 1, but in Variation 2, Task “E” will not wait for both Task “C” and “D” to complete. If the Flow Condition for Task “E” is set to All, then the behavior will be fine for Variation 1, but in Variation 2, Task “E” will be waiting for a Token that never arrives and the Process will be stuck at that location.

This flow can be executed properly if the Flow Condition for Task “E” is set properly to take into account which path is taken from the Decision and the appropriate number of paths that will be needed for each variation. The need to be careful in defining the Flow Condition may not be obvious to the modeler whose diagram is more complex than the one in the figure.

Another type of problem occurs with looping back to upstream activities. If the loop Decision is made within the implicit boundaries of a set of parallel paths, then the behavior of the loop becomes ambiguous (see Figure 53).

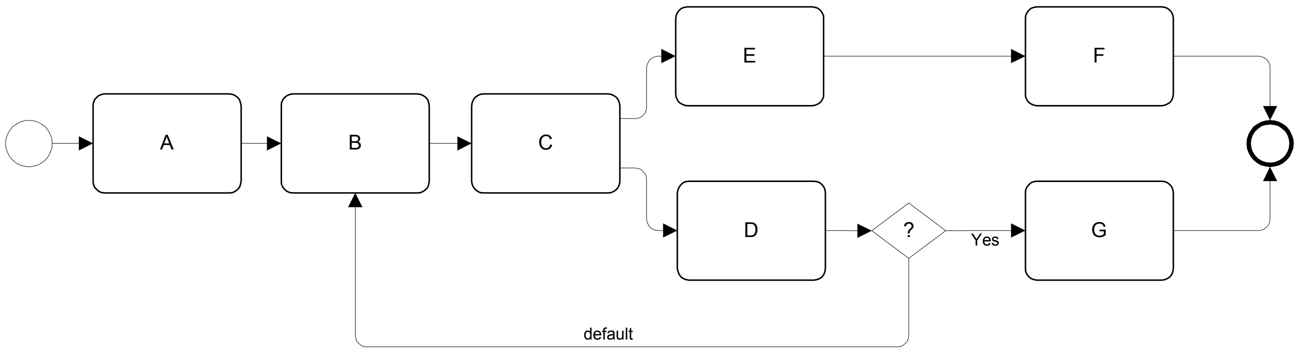


Figure 53 Improper Looping

In general, the analysis of how Tokens will flow through the model will help find models that cannot be executed properly. This Token flow analysis will be used to create some of the mappings to BPEL4WS and BPML. Since BPEL4WS and BPML are properly executable, if the Token flow analysis cannot create a valid BPEL4WS or BPML process, then the model is not structured correctly. This is an open issue that will be resolved in a later version of the specification. The section entitled “Mapping to Execution Languages” on page 133 will detail the Token flow analysis. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

In addition, BPMI is investigating how the fork, join, and merge control mechanisms might be improved so that such modeling situations are easily avoided. This is an open issue that will be resolved in a later version of the specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

5.2.2 Link Events

Start, Intermediate, and End Events can all be defined as being the type Link. Link Events are used to coordinate specific paths of a Process that are separated by a graphical distance or by differing levels of the Process. An example of how Link Events are used can be seen in the section entitled “BPMN by Example” on page 103.

A full description of how Link Events are used within BPMN is an open issue that will be handled in a later version of the specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

5.2.3 Spawning and Synchronizing Activities

Spawning is a mechanism for starting activities and not waiting on them to complete before continuing with the flow to other activities. At some other point in the Process, the completion of the spawned activities may be required to continue the flow. This is called synchronization.

Spawning and synchronization are a special case of process flow and not part of the Normal Flow, although these functions do interact with Normal Flow. One factor that makes spawning a special case is that activities that are spawned are not a part of the event context that spawned them. That is, if an activity were spawned within an interruptible Sub-Process, that activity would be aborted if the Sub-Process were aborted through an Intermediate Event. Thus, the spawned activity should not reside in the confines of the Sub-Process. This means that the flow from within the Sub-Process must extend to a position outside that Sub-Process. Normal Sequence Flow cannot cross the Sub-Process boundary. The same issue applies to the synchronization of the spawned activity. Therefore, the graphical mechanism for spawning and synchronizing must not (entirely) utilize Sequence Flows.

The graphical mechanisms for spawning activities and then synchronizing the spawned activities have not been defined for this version of the specification. It is an open issue that will be handled in a later version of the specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

5.2.4 Exception Flow

Exception flow occurs outside the normal flow of the Process and is based upon an event (an Intermediate Event) that occurs during the performance of the Process. Intermediate Events can be included in the normal flow to set delays or breaks to wait for a message. However, exception flow is created by attaching the Intermediate Event to the boundary of an activity, either a Task or a Sub-Process (see Figure 54). Multiple Intermediate Events can be attached to the boundary of an activity.

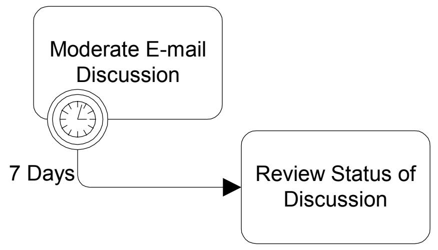


Figure 54 A Task with Exception Flow (Interrupts Event Context)

By doing this, the modeler is creating an Event Context. An Event Context is an activity of a Process that responds to an Intermediate Event. The activity will only respond if it is active (running) at the time of the Event. If the activity has completed, then the event may occur with no response. If there are a group of Tasks that the modeler wants to include in an Event Context, then an Expanded Sub-Process can be added to encompass the Tasks and to handle any events by having them attached to its boundary (see Figure 55).

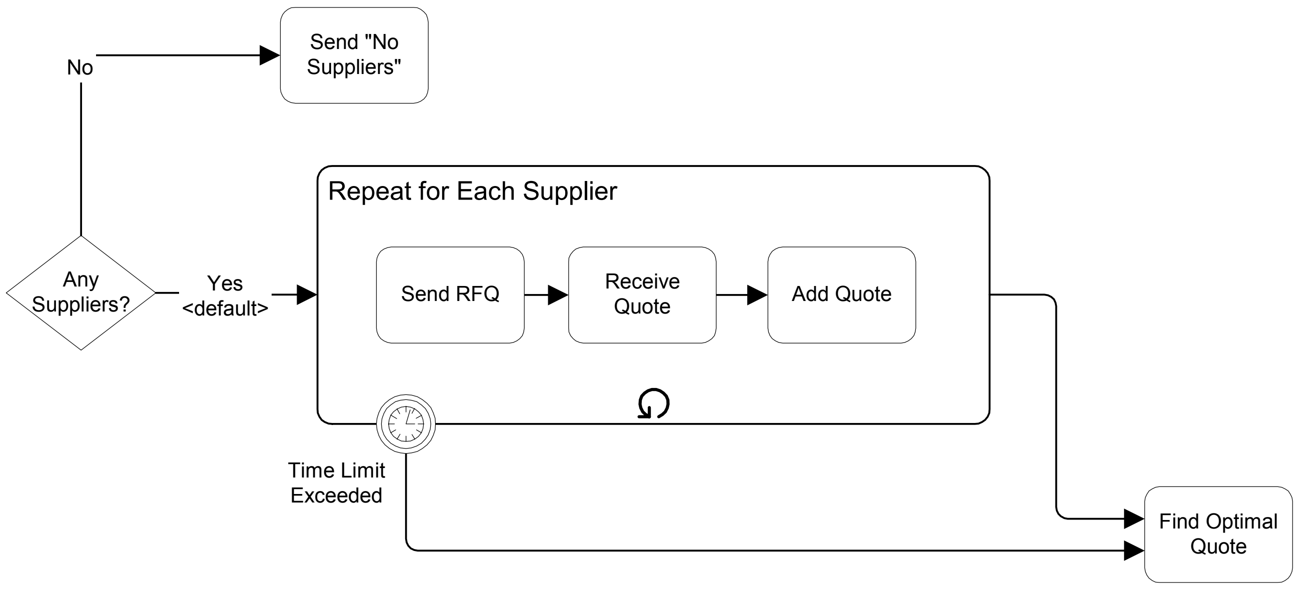


Figure 55 A Sub-Process with Exception Flow (Interrupts Event Context)

Three types of Intermediate Event are used by Event Contexts: Timer, Message, and Process Error. A Timer Event occurs when the Time and Date as specific in the Intermediate Event is exceeded during the performance. Usually the time is relative to the start of the Event Context. A Message Event occurs when a message, with the exact identity as specified in the Intermediate Event, is received by the Process. A Process Error Event occurs when the Process detects a Process Error. If an Error Code is specified in the Intermediate Event, then the code of the detected Error must match for the Event Context to respond. If the Intermediate Event does not specify an Error Code, then any Process Error will trigger a response from the Event Context.

If this event does not occur while the Event Context is ready, then the Process will continue through the normal flow as defined through the Sequence Flows. If the event does occur while the Event Context is ready, then one of two things can happen:

- Interrupting the Event Context: The Event Context will be aborted meaning those activities within the Event Context that are active will be aborted and those that have not yet been performed will be cancelled. A Token will also be generated from the appropriate Intermediate Event on the Event Context boundary and proceed through the Sequence Flows that follow. The Token will have the same TokenID set as the Token that entered the Event Context.
- Continuing the Event Context: The Event Context will continue without interruption until normal completion. In addition, a forked Token will also be generated from the appropriate Intermediate Event on the Event Context boundary and proceed through the Sequence Flows that follow. The Token will have the same TokenID set as the Token that entered the Event Context, except that an additional SubTokenID will be appended onto the set. Likewise, the Token that leaves the Event Context will also have an additional SubTokenID will be appended onto its TokenID set.
    - The graphical mechanism to distinguish an Intermediate Event that interrupts the Event Context versus one that does not interrupt the Event Context is an open issue and will be defined in a later version of this specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN

Mapping to Execution Languages

The following two sections describe how exception flow will map to BPEL4WS and BPML, respectively.

BPEL4WS

For an activity with an Intermediate Event attached to its boundary:

- ❖ The activity will be placed inside a *scope*.
- ❖ A *faultHandler* element will be defined for the *scope*.
    - ❖ If an Intermediate Event is of type Process Error and no name has been specified for the error, then a *catchAll* element will be added to the *faultHandler* element.
    - ❖ If an Intermediate Event is of type Process Error and a name has been specified for the error, then a *catch* element will be added to the *faultHandler* element with the name of the error placed in the *faultName* attribute.
    - ❖ If an Intermediate Event is of type Message, then:
        - ❖ The source activity will be placed inside a *flow* (which is inside the *scope*)
        - ❖ A *sequence* will also be placed within the *flow*
        - ❖ The first element of the *sequence* will be a *receive* that will wait for the message identified in the Intermediate Event.
        - ❖ The next element of the *sequence* will be a *throw* that will have a *faultName* with a name that is constructed from the Message Name with “Fault” appended.
        - ❖ A *catch* element will be added to the *faultHandler* element (within the *scope*) with the “<message name>Fault” error placed in the *faultName* attribute.
    - ❖ If the Intermediate Event is of type Timer, then:
        - ❖ The source activity will be placed inside a *flow* (which is inside the *scope*)

        - A *sequence* will also be placed within the *flow*
        - The first element of the *sequence* will be a *wait* that will use the time information identified in the Intermediate Event.
        - The next element of the *sequence* will be a *throw* that will have a *faultName* with a name that is constructed from the Intermediate Event Name with "Fault" appended.
        - A *catch* element will be added to the *faultHandler* element (within the *scope*) with the "<Intermediate Event name>Fault" error placed in the *faultName* attribute.
    - If there is only one activity that flows from the Intermediate Event, then the appropriate mapping for this activity will be used as the *activity* for the *faultHandler*.
        - A *sequence* will be used as the *activity* for the *faultHandler* if the mapping is complex or there are more than one activity that follows the Intermediate Event.

BPML

For a Task with a Intermediate Event attached to its boundary:

- A hidden *nested process* will be created with the Task mapped to an *action* within the *nested process*.
    - The name of the Task and the name of the BPML *nested process* will be the same except that the *nested process* will have "Sub" appended.

Then for all activities:

- A *context* will be defined for the *nested process*.
    - If an Intermediate Event is of type Message, an *exception* element will be defined for the *context* of the *nested process*.
        - Within the *exception* will be an *action* within an *event*.
        - An *action* to receive the message named in the Intermediate Event will be the first *activity* of the *exception activity set*.
        - All the BPMN activities that follow the flow from the Intermediate Event will be a part of the *exception activity set*.
    - If an Intermediate Event is of type Process Error or Time, then a *faults* element will be added to the *context*.
        - If the Intermediate Event is of type Process Error, then
            - If the Intermediate Event specifies a *code* for the Process Error, then the *code* will be inserted into the *faults* element as a *case*.
            - If the Intermediate Event does not specifies a code for the Process Error, then a *default* element will be added to the *faults* element.
        - If the Intermediate Event is of type Timer, then *schedule* will be added to the context that will generate a *fault code* at the appropriate time.
            - The *code* will be inserted into the *faults* element as a *case*.

- All the BPMN activities that follow the flow from the Intermediate Event will be a part of the *case* or *default activity set*.

5.2.5 Transaction Compensation Flow

Transaction Compensation Flow occurs outside the normal flow of the Process and is based upon an event (an Intermediate Event) that is triggered during the rolling back of a Process that has started, but is later cancelled. This flow originates from an Intermediate Event with a Compensate marker. The graphical line for a Transaction Compensation Flow is drawn the same as a normal Sequence Flow (see Figure 56). The direction of flow can be set to any orientation, but it will make the diagram more understandable if the flow is shown going in the opposite direction of the normal flow (as in the figure below).

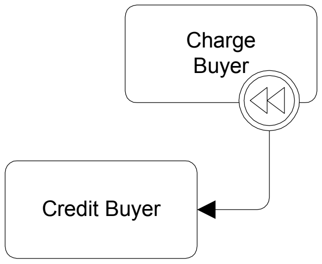


Figure 56 A Task with Transaction Compensation Flow

A transaction is a special activity that creates a “product” that cannot be simply undone or rolled back. The undoing of the product requires compensation through another activity. The compensation activity does not occur unless the Process is aborted and rolled back. This is why the Compensate marker for Events looks like a “rewind” symbol for a tape player. Thus, this Transaction Compensation Flow only occurs when the whole process flow is going backwards—the Tokens will move from the their current positions back to the Start. When the Tokens reach an activity (either Task or Sub-Process) that has a Compensate Intermediate Event attached to its border, the Token may traverse the flow from the Compensate Event to the activity that handles the compensation. The compensation will occur if:

- The End Event that started the rollback is set to compensate all transactions or
- The End Event that started the rollback is set to compensate a specific transaction. The name of the transaction will be the name shown on the Compensate Intermediate Event attached to the boundary of the activity.

After the compensation has been completed, the Process will continue its rollback.

In Figure 56, the buyer was charged for a purchase, but then later cancelled the order before the Process was completed. The charge cannot be simply erased. A separate activity that was not performed during the original flow must occur during rollback to credit the buyer.

Note: In a later version of the specification, activities that are defined as transactions may be drawn with a different line style or have a marker to show that they are transactions and not normal activities. This is an open issue. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

Mapping to Execution Languages

The following two sections describe how transaction compensation flow will map to BPEL4WS and BPML, respectively.

BPEL4WS

For an activity with a Compensate Intermediate Event attached to its boundary:

- ❖ The activity will be placed inside a *scope*.
- ❖ A *compensationHandler* element will be defined for the *scope*.
  - ❖ If there is only one activity that flows from the Intermediate Event, then the appropriate mapping for this activity will be used as the *activity* for the *compensationHandler*.
    - ❖ A *sequence* will be used as the *activity* for the *compensationHandler* if the mapping is complex or there are more than one activity that follows the Intermediate Event.

BPML

For a Task with a Compensate Intermediate Event:

- ❖ A hidden *nested process* will be created with the Task mapped to an *action* within the *nested process*.
  - ❖ The name of the Task and the name of the BPML *nested process* will be the same except that the *nested process* will have “_sub” appended.

Then for all activities:

- ❖ A *context* will be defined for the *nested process*.
- ❖ A *transaction* element will be defined for the *context* of the *nested process*.
  - ❖ The *transaction* type will be *open*.
  - ❖ The name of the *transaction* will be the name of the Compensate Intermediate Event.
  - ❖ All the BPMN activities that follow the flow from the Compensate Intermediate Event will be a part of the *transaction activity set*.

5.2.6 Ad Hoc

An Ad Hoc Process is a group of activities that have no pre-definable sequence relationships. A set of activities can be defined for the Process, but the sequence and number of performances for the activities is completely determined by the performers of the activities and cannot be defined beforehand.

A Sub-Process is marked as being an Ad Hoc with a “tilde” symbol placed at the bottom center of the Sub-Process shape (see Figure 57 and Figure 58). Activities within the Process are disconnected from each other. During execution of the Process, any one or more of the activities may be active and they can be performed in almost any order or frequency.

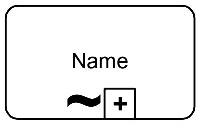


Figure 57 A Collapsed Ad Hoc Sub-Process

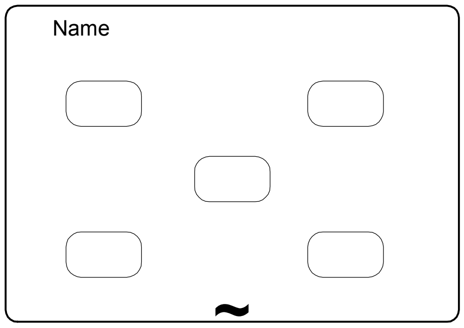


Figure 58 An Expanded Ad Hoc Sub-Process

The performers determine when activities will start, when they will end, what the next activity will be, and so on. Examples of the types of Processes that are Ad Hoc include computer code development (at a low level), sales support, and writing a book chapter. If we look at the details of writing a book chapter, we could see that the activities within this Process include: researching the topic, writing text, editing text, generating graphics, including graphics in the text, organizing references, etc. (see Figure 59). There may be some dependencies between Tasks in this Process, such as writing text before editing text, but there is not necessarily any correlation between an instance of writing text to an instance of editing text. Editing may occur infrequently and based on the text of many instances of the writing text Task.

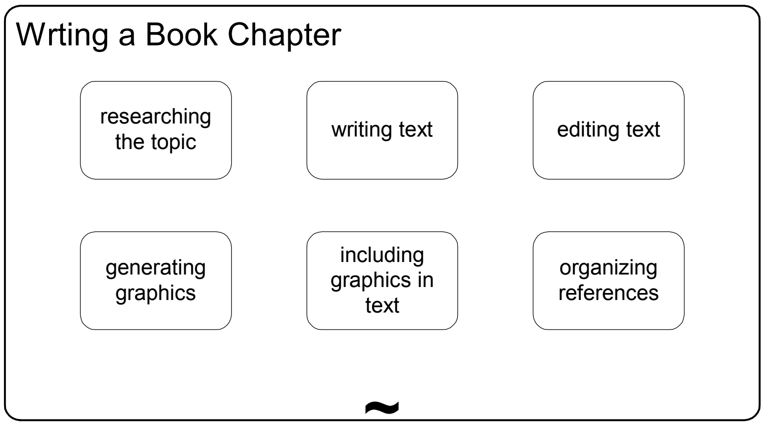


Figure 59 An Ad Hoc Process for Writing a Book Chapter

It is a challenge for a BPM engine to monitor the status of Ad Hoc Processes, usually these kind of processes are handled through groupware applications (such as e-mail), but BPMN allows modeling of Processes that are not necessarily executable and should provide the mechanisms for those BPM engines that can follow an Ad Hoc Process. Given this, at some point, the Process will have completed and this can be determined by evaluating a Completion Condition that evaluates Process attributes that will have been updated by an activity in the Process.

Mapping to Execution Languages

The Mapping to Execution Languages for Ad Hoc Processes is an open issue has not been determined for this version of the specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

6. BPMN by Example

This section will provide an example of a business process modeled with BPMN. The process that will be described is a process that BPMI has been using to develop this notation. It is a process for resolving issues through e-mail votes (see Figure 60). This Process is small, but fairly complex and will provide examples for many of the features of BPMN. There are some unusual features of this business process, such as infinite loops. Although not a typical process, it will help illustrate that BPMN can handle simple and unusual business processes and still be easily understandable for readers of the diagram. The sections below will highlight these features as we describe how the Process works.

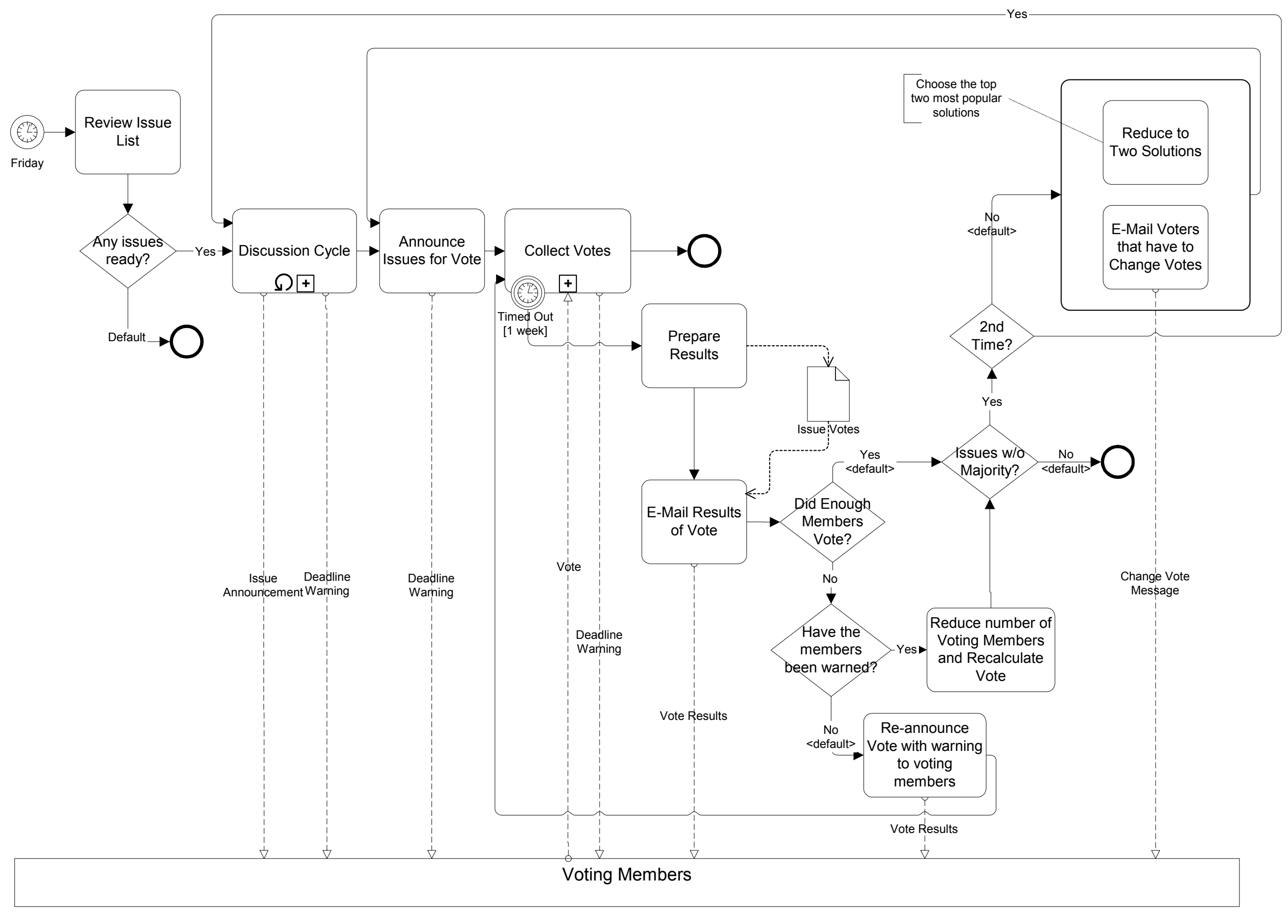


Figure 60 E-Mail Voting Process

The Process has a point of view that is from the perspective of the manager of the Issues List and the discussion around this list. From that point of view, the voting members of the working group are considered as external Participants who will be communicated with by messages (shown as Message Flow).

6.1 The Beginning of the Process

The Process starts with Timer Start Event that is set to trigger the Process every Friday (see Figure 61).

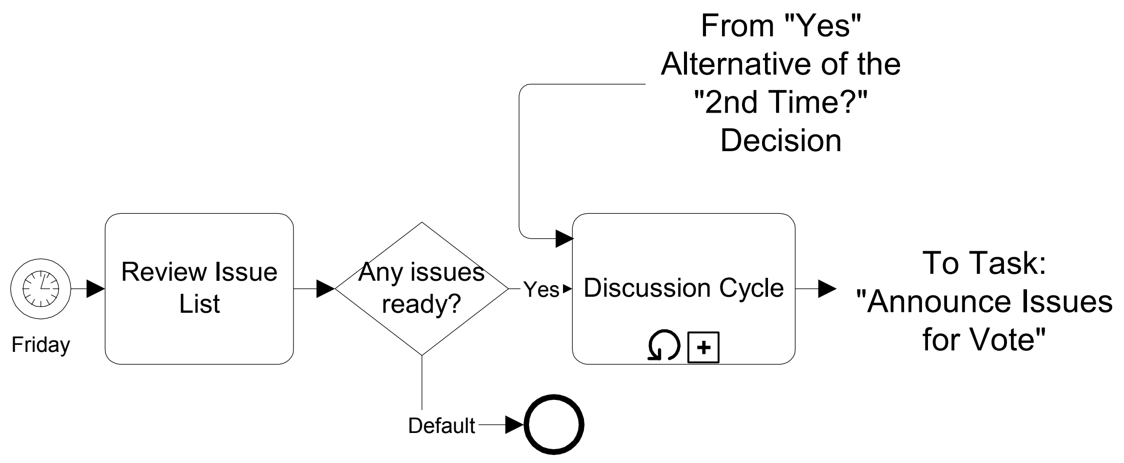

From "Yes" Alternative of the "2nd Time?" Decision
Friday
Review Issue List
Any issues ready?
Yes
Discussion Cycle
To Task: "Announce Issues for Vote"
Default

Figure 61 The Start of the Process

The Issue List Manager will review the list and determine if there are any issues that are ready for going through the discussion and voting cycle. Then a Decision must be made. If there are no issues ready, then the Process is over for that week. If there are issues ready, then the Process will continue with the discussion cycle. The “Discussion Cycle” Sub-Process is the first activity after the “Any issues ready?” Decision and this Sub-Process has two incoming Sequence Flows, one of which originates from a downstream Decision and is thus part of a loop. It is one of a set of five complex loops that exist in the Process. The contents of the “Discussion Cycle” Sub-Process and the activities that follow will be described below.

6.1.1 Mapping to BPEL4WS

Processes must begin with a *recieve* activity that instantiates the process (i.e., it “bootstraps” itself). The “E-Mail Voting Process” is scheduled to start every Friday as shown by the Timer Start Event. Thus, an additional Process will have to be created that will run indefinitely and will send a starting message to the “E-Mail Voting Process” every Friday. This additional Process is not shown in this example.

The modeler-defined properties of the Process will be placed in a BPEL4WS *container* named “processData.” The same container will be used in all derived *processes* in this example.

The “Review Issue List” Task will map to a BPEL4WS *invoke*. This *invoke* will be placed inside a *sequence* since other activities follow the *invoke*. This Task type is Service, which means that the *invoke* will be synchronous and an *outputContainer* included.

Note: the names of BPD objects have all non-alphanumeric characters stripped from them when they are mapped to BPEL4WS (or BPML) *name* elements to match the element restrictions.

The “Any Issues Ready?” Decision will map to a BPEL4WS *switch*. The Alternative labeled “No (default)” will map to the *otherwise case* of the *switch*. This *case* will only contain an *empty activity* since there is nothing to do and the Process is over. Note that *empty* does not have any corresponding activity in the BPMN diagram. The Decision Alternative labeled “Yes” will map to other *case* for the *switch*. This *case* will have a *condition* that checks the number of issues that are ready. This *case* will contain an *activity set* that holds the entire contents of the rest of the Process because the *switch* needs a definitive boundary and *otherwise* is connected to the end of the Process. Thus, the end of the Process is the boundary for the *switch*. However, the rest of the Process will be segmented into a set of *processes* to account for the diagram configuration that includes three upstream Sequence Flow that define some intertwining loops.

If the loop shown in this section of the model were merely a simple loop, then a BPEL4WS *while* would be used to handle the loop. In this situation, though, the looping has to be handled through a set of derived *processes* that are accessed by *invoking* them (as a web service). There would no specific diagram element to represent these *nested processes*; indeed, a modeler would not want to create a set of related Processes to handle complex looping. While an execution engine can easily handle a complex set of language documents and elements, a human developing and monitoring this process will want to see the Process in an easy-to-read format (such as BPMN). In this example, all derived *processes* will be named "DerivedProcess<number>" and the number will be incremented as they are created. Any naming scheme will work as long as all the *processes* have unique names. Thus, to handle the rest of the Process, a derived *nested process* named "DerivedProcess1" is created and then a BPEL4WS *invoke* is used to access this *process* from the "Yes" *case* of the "Any issues ready?" *switch*. We shall see that later in the Process the same *process* is accessed through another *invoke*, marking the source of the loop.

Note: All the derived *processes* in the BPEL4WS samples are accessed through an asynchronous *invoke*. That is, the *invoke* uses a one-way WSDL operation and the *outputContainer* element is not used in the *invoke*. Thus, the "calling" process will not wait until the "called" process completes. This would be equivalent to the BPML *spawn*. The BPML samples use a synchronous *call*, however, since they are calling *nested processes*, instead of independent *processes*.

All the sub-processes and derived processes in the BPEL4WS documents must be started with the receipt of a message. This *receive* will be the first *activity* inside a *sequence* that will be the main *activity* of the *process*. These *receive activities* will have the *createInstance* attribute set to "Yes." A partner named "internal," a portType name "processPort" will be created to support all of these process to process communications. The WSDL operations that will support these communications will all be named "call<process name>" (as noted above, the processes are actually spawned).

The "Discussion Cycle" Sub-Process shown in Figure 61 will start the *sequence* (after the *receive* that instantiates the *process*) for the "DerivedProcess1" *process*. Since "Discussion Cycle" is a Sub-Process it will map to a separate BPEL4WS *process* that is access through an *invoke* (synchronous, since we don't want to continue the Process until the Sub-Process has completed). However, since it has a loop marker, it will first be wrapped in a *while*. In this situation, the looping mechanism is simple. The attributes of the Sub-Process will tell us the details. The "Discussion Cycle" Sub-Process's relevant attributes are:

- LoopType = "standard"
- LoopCondition = "DiscussionOver = True"
- EvaluateCondition = "after"

All variations of LoopType will map to a BPEL4WS *while*. “not(DiscussionOver = True)” will be the condition for the *while*. The default value for the “DiscussionOver” property is False, thus an activity within the Sub-Process will have to change it to True before the *until* loop is over. The logical opposite of the expression that is shown in the Sub-Process attributes is used since the EvaluationCondition property is “after.” We will look into the details of the “Discussion Cycle” Sub-Process in the section entitled “The First Sub-Process” on page 110.

Example 1 displays some sample BPEL4WS code that reflects the portion of the Process that was just discussed and is shown in Figure 61.

```
<process name="EMailVotingProcess">
  <!-- The Process data is defined first-->
  <sequence>
    <!--This starts the beginning of the Process. The process that sends the
         starting message every Friday is not shown here.-->
    <receive partner="Internal" portType="tns:processPort"
            operation="callEMailVotingProcess" container="processData"
            createInstance="Yes"/>
    <invoke name="ReviewIssueList" partner="Internal"
            portType="tns:internalPort" operation="sendIssueList"
            inputContainer="processData" outputContainer="processData"/>
    <switch name="Anyissuesready">
      <!-- name="Yes" -->
      <case condition="bpws:getContainerProperty(ProcessData,NumIssues)>0">
        <!--A chunk of this process is separated into a derived process so
        that it can be called from a complex loop. Thus, it is called from
        here and from ”Collect Votes” as part of a loop-->
        <invoke name="DerivedProcess1" partner="Internal"
              portType="tns:processPort" operation="callDerivedProcess1"
              inputContainer="processData"/>
      </case>
      <!--name="otherwise" -->
      <otherwise>
        <!--This is one of the two ways to the end of the Process-->
        <empty/>
      </otherwise>
    </switch>
  </sequence>
</process>

<process name="DerivedProcess1">
  <!-- The Process data is defined first-->
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
            operation="callDerivedProcess1" container="processData"
            createInstance="Yes"/>
    <while condition="bpws:getContainerProperty(ProcessData,DiscussionOver)
                    =false">
      <!--This calls the first Sub-Process-->
      <invoke process="DiscussionCycle" partner="Internal"
              portType="tns:processPort operation="callDiscussionCycle"
              inputContainer="processData" outputContainer="processData"/>
    </while>
    <invoke name="DerivedProcess2" partner="Internal" portType="tns:processPort"
           operation="callDerivedProcess2" inputContainer="processData"/>
  </sequence>
  </process>
  <!--A lot of other stuff (not shown)-->
```

Example 1 BPEL4WS Sample for Beginning of E-Mail Voting Process

6.1.2 Mapping to BPML

Processes must begin with instantiation events (a *message* or *signal*) or through a *spawn*, *call*, or a *schedule time event*. The "E-Mail Voting Process" is scheduled to start every Friday as shown by the Timer Start Event. Thus, an additional Process will have to be created that will contain a *schedule* that will instantiate the "E-Mail Voting Process" every Friday. This additional Process is not shown in this example.

The modeler-defined properties of the Process will be placed in a BPML *property* elements within the *context* of the main *process*. All the properties will be available to all the *nested processes* in this example.

The "Review Issue List" Task will map to a BPML *action*. This *action* will be placed inside a *sequence* since other activities follow the *action*.

The "Any Issues Ready?" Decision will map to a BPML *switch*. The Alternative labeled "No (default)" will map to the *default case* of the *switch*. This *case* will only contain an *empty activity* since there is nothing to do and the Process is over. Note that *empty* does not have any corresponding activity in the BPMN diagram. The Decision Alternative labeled "Yes" will map to other *case* for the *switch*. This *case* will have a *condition* that checks the number of issues that are ready. This *case* will contain an *activity set* that holds the entire contents of the rest of the Process because the *switch* needs a definitive boundary and *default* is connected to the end of the Process. Thus, the end of the Process is the boundary for the *switch*. However, the rest of the Process will be segmented into a set of *nested processes* to account for the diagram configuration that includes three upstream Sequence Flow that define some intertwining loops.

If the loop shown in this section of the model were merely a simple loop, then a BPML *while* or *until* would be used to handle the loop. In this situation, though, the looping has to be handled through a set of derived *nested processes* that are accessed through a *call* or *spawn*. There would no specific diagram element to represent these *nested processes*; indeed, a modeler would not want to create a set of related Processes to handle complex looping. While an execution engine can easily handle a complex set of language documents and elements, a human developing and monitoring this process will want to see the Process in an easy-to-read format (such as BPMN). In this example, all derived *nested processes* will be named "DerivedProcess<number>" and the number will be incremented as they are added to the BPML document. Any naming scheme will work as long as all the *nested processes* have unique names. Thus, to handle the rest of the Process, a derived *nested process* named "DerivedProcess1" is created and then a BPML *call* is used to access this *nested process* from the “Yes” *case* of the “Any issues ready?” *switch*. We shall see that later in the Process the same *nested process* is accessed through another *call*, marking the source of the loop.

Note: All the derived *nested processes* in the BPML samples are accessed through a synchronous *call*. Thus, the “calling” process will wait until the “called” process completes. The BPEL4WS samples use an asynchronous *invoke*, however, since they are calling independent *processes*, instead of *nested processes*.

The “Discussion Cycle” Sub-Process shown in Figure 61 will start the *activity set* for the “DerivedProcess1” *nested process*. Since “Discussion Cycle” is a Sub-Process it will map to a BPML *call*. However, since it has a loop marker, it will first be wrapped in a looping object. In this situation, the looping mechanism is simple. The attributes of the Sub-Process will tell us the details. The “Discussion Cycle” Sub-Process’s relevant attributes are:

- LoopType = “standard”
- LoopCondition = “DiscussionOver = True”
- EvaluateCondition = “after”

A “standard” LoopType that has an EvaluateCondtion that is “after” will map to a BPML *until*. “DiscussionOver = True” will be the condition for the *until*. The default value for the “DiscussionOver” property is False, thus an activity within the Sub-Process will have to change it to True before the *until* loop is over. We will look into the details of the “Discussion Cycle” Sub-Process in the section entitled “The First Sub-Process” on page 110.

Example 2 displays some sample BPML code that reflects the portion of the Process that was just discussed and is shown in Figure 61.

```
<process name="EMailVotingProcess">
  <action name="ReviewIssueList" portType="tns:internalPort"
          operation="sendIssueList">
    <output property="NumIssues" element="…"/>
  </action>
  <switch name="Anyissuesready">
    <case name="Yes">
      <condition>NumIssues>0<condition/>
      <!--A chunk of this process is separated into a derived nested
        process so that it can be called from a complex loop. Thus,
        it is called from here and from ”Collect Votes” as part of a loop-->
      <call name="DerivedProcess1"/>
    </case>
    <default name="Default">
      <!--…This is one of the two ways to the end of the Process-->
      <empty/>
    </default>
  </switch>
  <context>
  <!-- The Process data is defined first-->
  <process name="DerivedProcess1">
    <until>
      <condition>DiscussionOver<condition/>
      <!--This calls the first Sub-Process-->
      <call process="DiscussionCycle"/>
    </until>
    …
  </process>
  <!--A lot of other stuff (not shown)-->
  </context>
</process>
```

Example 2 BPML Sample for Beginning of E-Mail Voting Process

6.2 The First Sub-Process

Figure 62 shows the details of the “Discussion Cycle” Sub-Process.

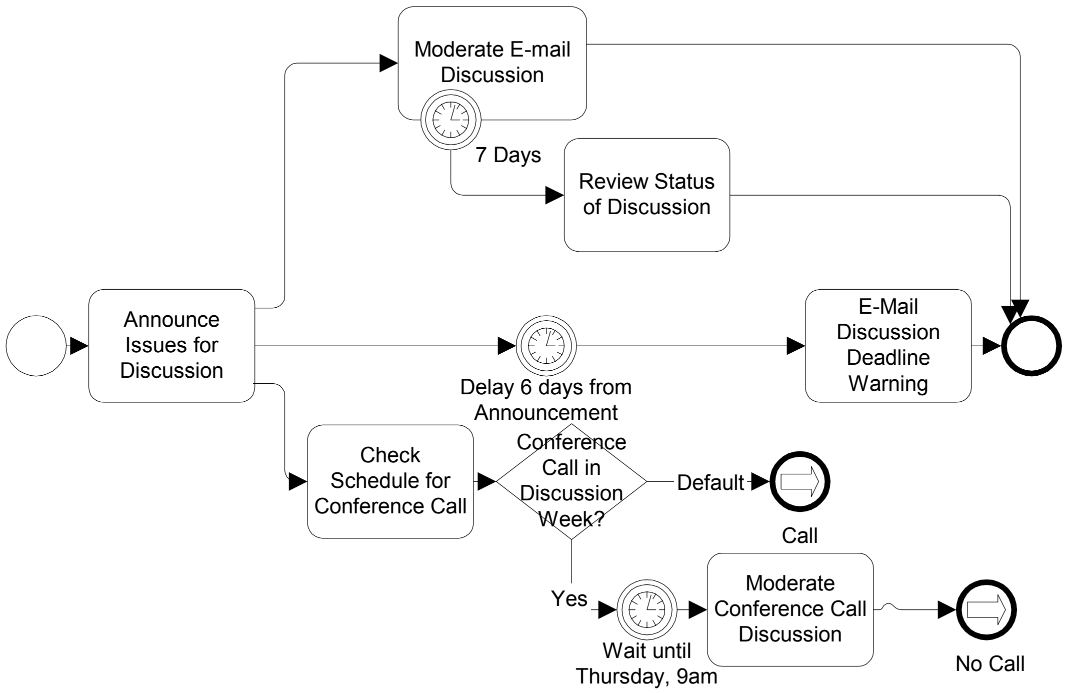

Moderate E-mail Discussion
7 Days
Review Status of Discussion
Announce Issues for Discussion
Delay 6 days from Announcement
E-Mail Discussion Deadline Warning
Check Schedule for Conference Call
Conference Call in Discussion Week?
Default
Call
Yes
Wait until Thursday, 9am
Moderate Conference Call Discussion
No Call

Figure 62 “Discussion Cycle” Sub-Process Details

The Sub-Process starts of with a Task for the Issue List Manager to send an e-mail to the working group that a set of Issues are now open for discussion through the working group’s message board. Since this Task sends a message to an outside Participant (the working group members), an outgoing Message Flow is seen from the “Discussion Cycle” Sub-Process to the “Voting Members” Pool in Figure 60. Basically, the working group will be discussing the issues for one week and proposing additional solutions to the issues. After the first Task, three separate parallel paths are followed.

The top parallel path in the figure starts with a long-running Task, “Moderate E-mail Discussion,” that has a Timer Intermediate Event attached to its boundary. The Task “Review Status of Discussion” is intended to occur only when the timeout occurs.

The middle parallel path of the fork, which contains an Intermediate Event and a Task. A Timer Intermediate Event used in this situation will cause a delay that is set to 6 days. The “E-Mail Discussion Deadline Warning” Task will follow. Again, since this Task sends a message to an outside Participant, an outgoing Message Flow is seen from the “Discussion Cycle” Sub-Process to the “Voting Members” Pool in Figure 60.

The bottom parallel path of the fork contains more than one object, first of which is Task where the issue list manager checks the calendar to see if there is a conference call this week. The output of the Task will be an update to the attribute “ConCall,” which will be true or false. After the Task, a Decision with its two Alternatives follows. The Alternative labeled “default” flows directly to an End Event. But this is a Link End Event, which indicates that there will be a corresponding Start or Intermediate Event at some point later in the Process. The Decision Alternative labeled “Yes” will have a *condition* that checks the value of the

"ConCall" attribute (set in the previous Task) to see if there will be a conference call during the coming week. If so, the Timer Intermediate Event indicates delay, since all conference calls for the working group start at 9am PDT on Thursdays. The Task for moderating the conference call follows the delay, which is followed by another Link End Event. We will see how Link Events are used later.

6.2.1 Mapping to BPEL4WS

The Sub-Process starts of with a Task, which maps to a BPEL4WS *invoke* (which is after the automatically generated receive that starts the *process*). After the first Task, three separate parallel paths are followed. The forking of the flow marks the start of a BPEL4WS *flow*. The *flow* will extend the entire (Sub) Process, since the paths do not join back together until the End Events.

In the upper parallel path of the fork, the Task, "Moderate E-mail Discussion," has a Timer Intermediate Event attached to its boundary. Because of this, the Task is placed in its own Event Context. A BPEL4WS *scope* will be required to set up the Event Context. The Task itself is mapped to a BPEL4WS *invoke* (synchronous). The Timer Intermediate Event must be set up to create a *fault* at the appropriate time. To do this, another *flow* is created that contains the above *invoke* plus a *sequence* that contains a *wait*. The *wait* is set to the duration that is defined in the Timer Intermediate Event. After the *wait*, a *throw* creates a fault name after the Intermediate Event with "_fault" appended. The *scope* will contain a *faultHandler*, and a *catch* element within the *faultHandler*. The *catch* will be triggered by the *fault* generated by the above *throw*. The Task "Review Status of Discussion" is intended to occur only when the timeout occurs, thus, it will map to an *invoke* that is the *activity* of the *catch* fault handler of the *scope*.

The middle parallel path of the fork has a string of two objects. Since this sequence appears in the middle of a BPEL4WS *flow*, a *link* element will be used. The link will be automatically generated and will be named "<source name> to <target name>." A Timer Intermediate Event used in this situation will map to a BPEL4WS *wait* (set to 6 days). The wait will also have a target element that refers to the name of the generated link. The "E-Mail Discussion Deadline Warning" Task will map to an *invoke* that follows the *wait*. Included within the *invoke* will be a *source* element that refers to the same generated *link*. In addition, this invoke can be asynchronous since a response is not required. This means that the *outputContainer* will not be included.

The bottom parallel path of the fork also contains more than one object, so four more *links* will be automatically generated for the string of objects. The path also contains a Decision, which normally will map to a *switch*, as will happen later in the process, but in this situation the Decision is mapped to *links* controlled by *transitionConditions*. The first object is a Task, which will map to an *invoke* (synchronous) that has two *target* elements referring to two of the new *links*. There are two Target *links* because the Task is followed by the Decision with its two Alternatives. One *link* will target the *wait* that follows the "Yes" Alternative and the other *link* will target the *invoke* that is created from the Link End Event that follows the "default" Alternative. The condition for the Decision Alternative labeled "Yes" will map to the *transitionCondition* that checks the value of the "ConCall" property (set in the previous Task) to see if there will be a conference call during the coming week. This alternative leads to a *wait*, as indicated by the Timer Intermediate Event, since all conference calls for the working group start at 9am PDT on Thursdays. This *wait* will have a *source* element that corresponds to the *target* element from the previous *invoke*. The *wait* will also have a *target* element to link to the following *invoke*. The Task for moderating the conference call follows the *wait*, which will map to an *invoke* (synchronous). This *invoke* will have a *source* element that corresponds to the *target* element from the previous *wait*. It will also include a *target* element to link to the next *invoke* (described next). Following the Task is a Link End Event, which represents a specific message that is sent to another *process* (in this case a *pick* will receive the message, as we shall see later). To do this, an *invoke* (asynchronous) is included. This *invoke* will have a *source* element that corresponds to the *target* element from the previous *invoke*. The *operation* for the *invoke* will be named "send<End Event name>." The Alternative labeled "default" will map to the *otherwise* of the *switch*. This Alternative flows directly to an End Event. But this is a Link End Event and thus, instead of an *empty*, the *activity* of the *otherwise* will be an *invoke* (asynchronous). This *invoke* creates a different message than what was sent due to the above End Event and will have a *source* element that corresponds to the *target* element from a previous *invoke*.

Note: Although the Decision described above was mapped to a set of *links* with *transitionConditions*, it is possible that all Decisions be mapped to *switches*, even if the mapping will exist within a BPEL4WS *flow* element. The BPMN specification does not require one method or the other. For Decisions that will not be within a *flow*, a mapping to a *switch* is required.

Example 3 displays some sample BPEL4WS code that reflects the portion of the Process as described above and shown in Figure 62.

```
<process name="DiscussionCycle">
  <!-- The Process data is defined first-->
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
           operation="callDiscussionCycle" container="processData"
           createInstance="Yes"/>
    <invoke name="AnnounceIssuesforDiscussion" partner="WGVoter"
          portType="tns:emailPort" operation="sendDiscussionAnnouncement"
          inputContainer="processData"/>
    <flow>
      <links>
        <link name="Delay6daysfromDiscussionAnnouncementtoEMailDiscussion
                DeadlineWarning"/>
        <link name="CheckCalendarforConferenceCalltoWaituntilThursday,9am"/>
        <link name="CheckCalendarforConferenceCalltoCall"/>
        <link name="WaituntilThursday,9amtoModerateConferenceCallDiscussion"/>
      </links>
      <!-- This is the first of the three paths of the fork. -->
      <scope>
        <flow>
          <invoke name="ModerateEmailDiscussion" partner="internal"
                portType="tns:internalPort" operation="sendDiscussion"
                inputContainer="processData"
                outputContainer="processData"/>
          <sequence>
            <wait name="7days" for="tns:OneWeek"/>
            <throw faultName="7days_fault"/>
          </sequence>
    </flow>
    <faultHander>
      <catch faultName="7days_fault">
        <invoke name="ReviewStatusofDiscussion" partner="internal"
                portType="tns:internalPort"
                operation="receiveDiscussionStatus"
                inputContainer="processData"
                outputContainer="processData"/>
      </catch>
    </faultHander>
  </scope>
  <!-- This is the second of the three paths of the fork. -->
  <wait name="Delay6daysfromDiscussionAnnouncement" for="P6D">
    <target linkName="Delay6daysfromDiscussionAnnouncementtoEMail
                      DiscussionDeadlineWarning"/>
  </wait>
  <invoke name="EMailDiscussionDeadlineWarning" partner="WGVoter"
          portType="tns:emailPort" operation="sendDiscussionWarning"
          inputContainer="processData">
    <source linkName="Delay6daysfromDiscussionAnnouncementtoEMail
                      DiscussionDeadlineWarning"/>
  </invoke>
  <!-- This is the third of the three paths of the fork. -->
  <invoke name="CheckCalendarforConferenceCall" partner="internal"
         portType="tns:internalPort" operation="receiveCallSchedule"
         inputContainer="processData" outputContainer="processData">
    <target linkName="CheckCalendarforConferenceCalltoWaituntilThursday9am"
         transitionCondition="bpws:getContainerProperty(processData,conCall)
                              =true"/>
    <target linkName="CheckCalendarforConferenceCalltoCall"
       transitionCondition="(bpws:getContainerProperty(processData,conCall)
                            =true)=false"/>
  </invoke>
  <!-- name="Yes" -->
  <wait name="WaituntilThursday9am" for="P6DT9H">
    <source linkName="CheckCalendarforConferenceCalltoWaituntilThursday9am">
    <target linkName="WaituntilThursday9amtoModerateConferenceCall
                     Discussion"/>
  </wait>
  <invoke name="ModerateConferenceCallDiscussion" partner="internal"
          portType="tns:internalPort" operation="sendConCall"
          inputContainer="processData" outputContainer="processData">
    <source linkName="WaituntilThursday9amtoModerateConferenceCall
                      Discussion"/>
    <target linkName="ModerateConferenceCallDiscussiontoNoCall"/>
  </invoke>
  <!-- This is used as a message to be used for a pick in the "Collect Votes"
       process -->
  <invoke name="NoCall" partner="internal" portType="tns:processlPort"
          operation="sendNo_Call" inputContainer="processData">
    <source linkName="ModerateConferenceCallDiscussiontoNoCall"/>
  </invoke>
```

```
      <!-- name="otherwise" -->
      <!-- This is used as a message to be used for a pick in the "Collect Votes"
           process -->
      <invoke name="Call" partner="internal" portType="tns:process1Port"
              operation="sendCall" inputContainer="processData">
        <source linkName="CheckCalendarforConferenceCalltoCall"/>
      </invoke>
    </flow>
  </sequence>
</process>
```

Example 3 BPEL4WS Sample of “Discussion Cycle” Sub-Process Details

6.2.2 Mapping to BPML

The Sub-Process starts of with a Task, which maps to a BPML *action*. After the first Task, three separate parallel paths are followed. The forking of the flow marks the start of a BPML *all*. The *all* will extend the entire (Sub) Process, since the paths do not join back together until the End Events.

In the upper parallel path of the fork, the Task, “Moderate E-mail Discussion,” has a Timer Intermediate Event attached to its boundary. Because of this, the Task is placed in its own Event Context. A BPML *nested process* will be required to set up this Event Context. The Task itself is mapped to a BPML *action*, but this *action* will be separated from this level of the process and placed in a separate *nested process*. The *nested process* will be accessed through a BPML *call* that is one of the activities of the *all*. The *nested process* will contain a *context*, an *exception* element within the *context*, and a *scheduled fault handler* (a timeout), set to 1 week, within the *exception* element. The Task “Review Status of Discussion” is intended to occur only when the timeout occurs, thus, it will map to an *action* that is in the *activity set* of the *case* within the *fault handler* of the *nested process*.

The middle parallel path of the fork has a string of two objects, whose mappings will wrapped in a BPML *sequence*. A Timer Intermediate Event used in this situation will map to a BPML *delay* (set to 6 days), since it has an incoming Sequence Flow, rather than an *fault handler* as the one described above, which was attached to the boundary of a Task. The “E-Mail Discussion Deadline Warning” Task will map to an *action* that follows the *delay*.

The bottom parallel path of the fork also contains more than one object, so these objects will be wrapped in a BPML *sequence*. The first object is a Task, which will map to an *action*. The output of the Task will be an update to the property “ConCall,” which will be true or false. After the Task, a Decision with its two Alternatives follows, which will map to a BPML *switch*. The Alternative labeled “default” will map to the *default case* of the *switch*. This Alternative flows directly to an End Event. But this is a Link End Event and thus, instead of an *empty activity*, the *activity set* of the *default case* will be a *raise* activity (which raises a *signal* that will be received with a *synch* later). The *signal* for the *raise* activity will be named “Call” after then name of the End Event. The Decision Alternative labeled “Yes” will map to other *case* for the *switch*. This *case* will have a *condition* that checks the value of the “ConCall” property (set in the previous Task) to see if there will be a conference call during the coming week. The *activity set* for the case will start with a *delay*, as indicated by the Timer Intermediate Event, since all conference calls for the working group start at 9am PDT on Thursdays. The Task for moderating the conference call follows the *delay*, which will map to an *action*. The Task is followed by another Link End Event. This will map to another *raise activity* that creates a *signal* named “No Call.” We shall see how this *signal* and the “Call” *signal* are used later in the process.

Example 4 displays some sample BPML code that reflects the portion of the Process as described above and shown in Figure 62.

```
<process name="DiscussionCycle">
  <sequence>
    <action name="AnnounceIssuesforDiscussion" portType="tns:emailPort"
             operation="sendDiscussionAnnouncement"/>
    <all>
      <call process="ModerateEmailDiscussionProcess"/>
      <sequence>
        <delay name="Delay6daysfromAnnouncement" duration="P6D"/>
        <action name="EMailDiscussionDeadlineWarning" portType="tns:emailPort"
                 operation="sendDiscussionWarning"/>
      </sequence>
      <sequence>
        <action name="CheckCalendarforConferenceCall" … >
          <output property="ConCall" element="…" />
        </action>
        <switch name="ConferenceCallinDiscussionWeek">
          <case name="Yes">
            <condition>ConCall=true<condition/>
            <delay name="WaituntilThursday9am" dateTime="P6DT9H"/>
            <action name="ModerateConferenceCall Discussion" … />
            <raise signal="NoCall"/>
          </case>
          <default name="Default">
            <raise signal="Call"/>
          </default>
      </sequence>
    </all>
  </sequence>
</process>

<process name="ModerateEmailDiscussionProcess">
  <action name="ModerateEmailDiscussion" … />
  <context>
    <schedule code="OneWeek" duration="P7D"/>
    <fault>
      <case code="OneWeek">
        <action name="ReviewStatusofDiscussion" portType="tns:internalPort"
                 operation="receiveDiscussionStatus">
          <output property="DiscussionOver" element="…"/>
        </action>
      </case>
    </fault>
  </context>
</process>
```

Example 4 BPML Sample of “Discussion Cycle” Sub-Process Details

6.3 The Second Sub-Process

Figure 63 shows the next section of the Process, which includes the expanded details of the “Collect Votes” Sub-Process.

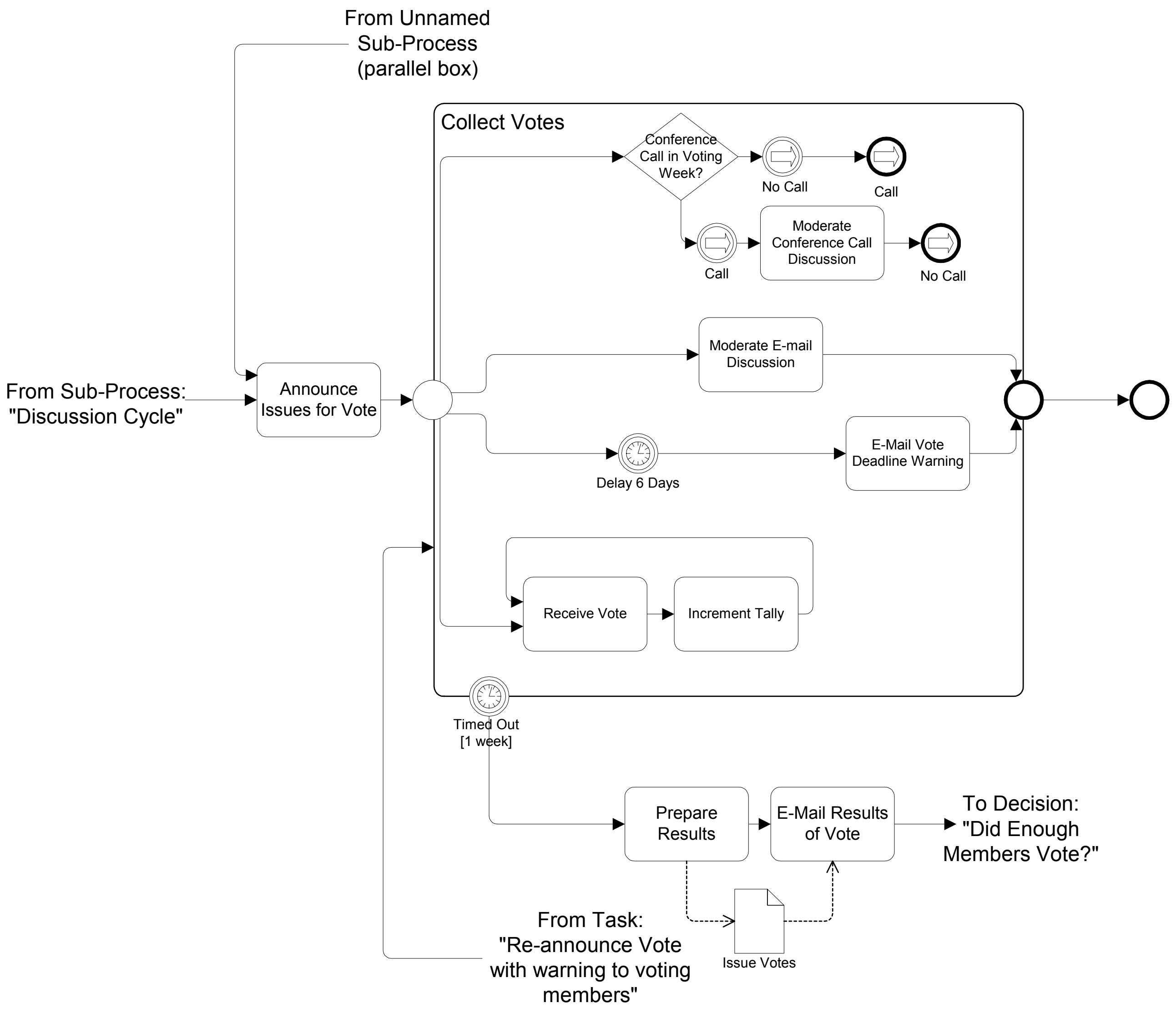

From Unnamed Sub-Process (parallel box)
Collect Votes
Conference Call in Voting Week?
No Call
Call
Call
Moderate Conference Call Discussion
No Call
Moderate E-mail Discussion
From Sub-Process: "Discussion Cycle"
Announce Issues for Vote
Delay 6 Days
E-Mail Vote Deadline Warning
Receive Vote
Increment Tally
Timed Out [1 week]
Prepare Results
E-Mail Results of Vote
To Decision: "Did Enough Members Vote?"
From Task: "Re-announce Vote with warning to voting members"
Issue Votes

Figure 63 “Collect Votes” Sub-Process Details

This part of the process starts out with a Task for the issue list manager to send out an e-mail to announce to the working group, and the voting members in particular, which lets them know that the issues are now ready for voting. Since this Task sends a message to an outside Participant (the working group members), an outgoing Message Flow is seen from the “Announce Issues for Vote” Task to the “Voting Members” Pool in Figure 60. This Task is also a target for one of the complex loops in the Process.

The “Collect Votes” Sub-Process follows the Task, and is also a target of one of the looping Sequence Flows. This Sub-Process is basically a set of four parallel paths that extend from the beginning to the end of the Sub-Process.

The first branch of the fork leads to a Decision that determines whether or not a conference call will occur during the upcoming week. The Decision is an Event-Based Exclusive Decision that uses Link Events that were created earlier in the Process. Basically, if there was a call last week, then there will not be a call this week and vice versa. The appropriate Link Events were created in the "Discussion Cycle" Process for use now. If the "No Call" Intermediate Event gets triggered (because the call occurred last week during the discussion cycle), then there is no call this week, but since this whole week might be repeated, a new "Call" Link End Event is used so that a call will happen next week. If the "Call" Intermediate Event gets triggered (because there was no call last week), then the "Moderate Conference Call" Task will occur. This Task will be followed by a Link End Event that will ensure that a call will not occur next week if the voting cycle is repeated.

The second and third branches forks work the same way as the similar activities in the "Discussion Cycle" Sub-Process, except that the "Moderate E-Mail Discussion" Task does not have a Timer Intermediate Event attached. The "E-Mail Vote Deadline Warning" Task sends a message to an outside Participant (the working group members), thus, an outgoing Message Flow is seen from the "Collect Votes" Sub-Process to the "Voting Members" Pool in Figure 60.

The fourth branch of the fork is rather unique in that the diagram uses a loop that does not utilize a Decision. Thus, it is, as it is intended to be, an infinite loop. The policy of the working group is that voting members can vote more than once on an issue; that is, they can change their mind as many times as they want throughout the entire week. The first Task in the loop receives a message from the outside Participant (the working group members), thus, an incoming Message Flow is seen from the "Voting Members" Pool to the "Collect Votes" Sub-Process in Figure 60. The Timer Intermediate Event attached to the boundary of the Sub-Process is the mechanism that will end the infinite loop, since all work inside the Sub-Process will be ended when the timeout is triggered. All the remaining work of the Process is conducted after the timeout and flows from the Timer Intermediate Event.

Note: The modeler could have organized the ending section differently. The Timer Intermediate Event could have lead directly to an End Event (instead of the rest of the Process) and the "Collect Votes" Sub-Process could have lead directly to the rest of the Process (instead of the End Event). The behavior would be the same in either case. This is mentioned to point out that when a Sub-Process is interrupted by an Intermediate Event, the flow follows the Intermediate Event until the Exception Sequence Flow is ended, and then the flow will go back to the end of the Sub-Process and the Normal Sequence Flow will continue as if the Sub-Process was never interrupted. That is, Intermediate Events do not interrupt the whole Process. They only interrupt the level to which they are attached and then the higher level Process will still continue.

Figure 63 shows that there are Two Tasks that follow the timeout. First, a Task will prepare all the voting results, then a Task will send the results to the voting members. A Document Object, "Issue Votes," is shown in the diagram to illustrate how one might be used, but it will not map to anything in the execution languages. The remaining activities of the Process will be described in the next section.

Although the Sub-Process itself is connected to an End Event, this End Event will not be reached until all work that follows from the Timer Intermediate Event has been completed.

6.3.1 Mapping to BPEL4WS

The first Task of this section of the Process is also a target for one of the complex loops in the Process, thus, it will map to an *invoke* (asynchronous) that is placed inside another derived *process* (“DerivedProcess2”). This derived *process* will be *invoked* from “DerivedProcess1,” after the “Discussion Cycle” process has been completed, as part of the normal flow and then from another part of the Process as part of the looping flow. Thus, “DerivedProcess2” will require a (instantiation) *receive* to accept the message from “DerivedProcess1” and from “DerivedProcess4” (as we shall see later).

The “Collect Votes” Sub-Process follows the Task, but is also a target of one of the looping Sequence Flows. Thus, it will also be set inside a derived *process* (“DerivedProcess3”). Thus, “DerivedProcess3” will require a (instantiation) *receive* to accept the message from “DerivedProcess2” and from the fault handler of “Collect Votes” (as we shall see later). The “Collect Votes” Sub-Process will map to an *invoke* (asynchronous) and the details will be in a *process* referenced through the *invoke*.

Example 5 shows sample BPEL4WS code that defines the two derived *processes*.

```
<process name="DerivedProcess2">
  <!-- This starts the middle section of the Process and is call from
        the first time and then from “Collect Votes” during a loop-->
  <!-- The Process data is defined first-->
    <sequence>
      <receive partner="Internal" portType="tns:processPort"
                operation="callDerivedProcess2" container="processData"
                createInstance="Yes"/>
      <invoke name="AnnounceIssuesforVote" partner="WGVoter"
              portType="tns:emailPort" operation="sendVoteAnnouncement"
              inputContainer="processData"/>
      <invoke name="DerivedProcess3" partner="Internal"
              portType="tns:processPort" operation="callDerivedProcess3"
              inputContainer="processData"/>
    </sequence>
</process>

<process name="DerivedProcess3">
  <!-- this calls the second Sub-Process and then continues. It is also
       called from “Collect Votes” as part of a loop-->
  <!-- The Process data is defined first-->
    <sequence>
      <receive partner="Internal" portType="tns:processPort"
               operation="callDerivedProcess3" container="processData"
               createInstance="Yes"/>
      <invoke name="CollectVotes" partner="Internal" portType="tns:processPort"
              operation="callCollectVotes" inputContainer="processData"/>
    </sequence>
</process>
```

Example 5 BPEL4WS Sample that sets up the Access for the Second Sub-Process

The “Collect Votes Sub-Process is basically a set of four parallel paths that extend from the beginning to the end of the Sub-Process. Thus, the *activity* for the *process* it maps to will be a *flow*.

The first branch of the fork introduces the BPEL4WS *pick*, since an Event-Based Exclusive Decision was used. The *pick* will use one of two possible messages that could be generated during the “Discussion Cycle” *process* or from the “Collect Votes” process itself if it is re-instantiate through a loop. The Link Intermediate Events from the Event-Based Exclusive Decision will map to *onMessage* event handlers within the *pick*. The *activity* for each *event handler* in the *pick* also *raised* another *signal* that will be used by this same *pick* if voting week was repeated by a loop.

The second and third branches of the fork are rather straightforward mappings of two Tasks to *invokes* (one synchronous and one asynchronous), and Timer Intermediate Event to a *delay*. One *link* is created so that one of the *invokes* will wait for the *delay*.

The fourth branch of the fork is the location the infinite loop. This loop will map to a BPEL4WS *while* with a *condition* of “1=0,” which will always be false. Inside the *while* is a *sequence* of two *invokes* (one synchronous and one asynchronous), which are mapped from the two Tasks in the loop.

To exit out of the infinite loop and the whole “Collect Votes” Sub-Process, a *scope* will be wrapped around the main *flow* of the *process*, which will include a *faultHandler*. The Timer Intermediate Event must be set up to create a *fault* at the appropriate time. To do this, another *flow* is created that contains the above *flow* plus a *sequence* that contains a *wait*. The *wait* is set to the duration that is defined in the Timer Intermediate Event. After the *wait*, a *throw* creates a fault name after the Intermediate Event with “Fault” appended. The *scope* will contain a *faultHandler*, and a *catch* element within the *faultHandler*. The *catch* will be triggered by the *fault* generated by the above *throw*. The *activity* for the *catch* will be a *sequence* and will be the source of all the remaining activities of the Process, since all the remaining Sequence Flow begins from the Timer Intermediate Event. The first two Tasks, as shown in the figure, will map to *invokes* (one synchronous and the other asynchronous). The Document Object shown in the figure is not mapped into BPEL4WS. The remainder of the Process will be described in the next section.

Example 6 shows sample BPEL4WS code that defines the “Collect Votes” Sub-Process.

```
<process name="CollectVotes">
  <!--This is a nested process for the E-Mail Voting collection. It consists of
      an all and a faultHandler (for a timeout). The all will never complete
      normally since there is an infinite loop inside. The timeout is intended to
      be the normal way of ending the process-->
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
             operation="callCollectVotes" container="processData"
             createInstance="Yes"/>
    <scope>
      <flow>
        <sequence>
          <wait name="7days" for="P7D"/>
          <throw faultName="7daysFault"/>
        </sequence>
        <flow>
          <links>
            <link name="Delay6daysfromVoteAnnouncementtoEMailVote
                        DeadlineWarning"/>
          </links>
    <!--This is the first of the four paths of the fork. -->
    <pick name="ConferenceCallthisWeek">
      <!-- name="Call"  -->
      <onMessage partner="internal" portType="tns:processlPort"
                 operation="sendCall" Container="processData">
        <invoke name="ModerateConferenceCallDiscussion" partner="internal"
                portType="tns:internalPort" operation="sendConCall"
                inputContainer="processData" outputContainer="processData"/>
        <invoke name="NoCall" partner="internal" portType="tns:processlPort"
                 operation="sendNo_Call" inputContainer="processData"/>
      </onMessage>
      <!-- name="No Call"  -->
      <onMessage partner="internal" portType="tns:processlPort"
                 operation="sendNo_Call" Container="processData">
        <invoke name="Call" partner="internal" portType="tns:processlPort"
                 operation="sendCall" inputContainer="processData"/>
      </onMessage>
    </pick>
    <!-- This is the second of the four paths of the fork. -->
    <invoke name="ModerateEMailDiscussion" partner="internal"
             portType="tns:internalPort" operation="sendDiscussion"
             inputContainer="processData"
             outputContainer="processData"/>
    <!--This is the third of the four paths of the fork.-->
    <wait name="Delay6daysfromVoteAnnouncement" for="P6D">
      <target linkName="Delay6daysfromVoteAnnouncementtoEMailVote
                         DeadlineWarning"/>
    </wait>
    <invoke name="EMailVoteDeadlineWarning" partner="WGVoter"
             portType="tns:emailPort" operation="sendVoteWarning"
             inputContainer="processData">
      <source linkName="Delay6daysfromVoteAnnouncementtoEMailVote
                         DeadlineWarning"/>
    </invoke>
    <!--This is the fourth of the four paths of the fork. This branch of the
          all is intended to be an infinite loop that is eventually
          interrupted by the Time Out.  This is necessary since any voter can
          change their vote until the deadline. -->
    <while condition="1=0">
      <sequence>
        <receive name="ReceiveVote" partner="WGVoter"
                   portType="tns:emailPort" operation="receiveVote"
                   container="processData"/>
        <invoke name="IncrementTally" partner="internal"
                 portType="tns:internalPort" operation="sendReceiveTotal"
                 inputContainer="processData" outputContainer="processData"/>
      </sequence>
    </while>
  </flow>
</flow>
<faultHander>
  <catch faultName="7 days_fault">
```

```
        <!-- The BPMN diagram shows that the Timer Intermediate Event connects
             directly to the rest of the Process. Thus, they will show up in
             this activity set. -->
        <sequence>
          <invoke name="PrepareResults" partner="internal"
                  portType="tns:internalPort" operation="sendReceiveResults"
                  inputContainer="processData" outputContainer="processData"/>
          <invoke name="EMailResultsofVote" partner="WGVoter"
                  portType="tns:emailPort" operation="sendVotingResults"
                  inputContainer="processData"/>
    <!--the rest of the process is not shown-->
      </faultHander>
    </scope>
  </sequence>
</process>
```

Example 6 BPEL4WS Sample of the Second Sub-Process

6.3.2 Mapping to BPML

The first Task of this section of the Process is also a target for one of the complex loops in the Process, thus, it will map to an *action* that is inside another derived *nested process* ("DerivedProcess2"). This derived *nested process* will be called from "DerivedProcess1" as part of the normal flow and then from another part of the Process as part of the looping flow.

The "Collect Votes" Sub-Process follows the Task, but is also a target of one of the looping Sequence Flows. Thus, it will also be set inside a derived Sub-Process ("DerivedProcess3"). This derived *nested process* will be called from "DerivedProcess2" as part of the normal flow and then from another part of the Process as part of the looping flow. The "Collect Votes" Sub-Process will map to a *call* and the details will be in a *nested process* referenced by the *call*. Example 7 shows sample BPML code that defines the two derived *nested processes*.

```
<process name="DerivedProcess2">
  <!-- This starts the middle section of the Process and is call from
       "DerivedProcess1" the first time and then from "Collect Votes"
       during a loop-->
  <action name="Announce Issues for Vote" portType="tns:emailPort"
          operation="sendVoteAnnouncement"/>
  <call process="DerivedProcess3"/>
</process>

<process name="DerivedProcess3">
  <!-- this calls the second Sub-Process and then continues. It is also
      called from "Collect Votes" as part of a loop-->
  <call process="Collect Votes"/>
</process>
```

Example 7 BPML Sample that sets up the Access for the Second Sub-Process

The "Collect Votes Sub-Process is basically a set of four parallel paths that extend from the beginning to the end of the Sub-Process. Thus, the *activity set* for the *nested process* will be an *all*.

The first activity introduces the BPML *choice* and uses *signals* that were generated in the "Discussion Cycle" Sub-Process. A working group conference call may occur this week, but only if there was not a call during the last discussion week, since calls occur every other week. After the "Discussion Cycle" Sub-Process determined whether or not there would be a call at that time, the Sub-Process created a *signal* (either "Call" or "No Call") that can be used by the "Collect Votes" Sub-Process to know the call status for this week. An Event-Based Exclusive Decision maps to a *choice*. The Link Intermediate Events will map to *event* handlers within the *choice*. The *activity set* for each *event* handler in the *choice* also *raised* another *signal* that will be used by this same *pick* if voting week was repeated by a loop.

The second and third activities for the *all* are rather straightforward mappings of a Task to an *action*, and then a *sequence* that has a *delay* and an *action*.

The fourth activity of the *all* is the location the infinite loop. This loop will map to a BPML *until* with a *condition* of "1=0," which will always be false. Inside the *until* are two *actions* which are mapped from the two Tasks in the loop.

To exit out of the infinite loop and the whole Sub-Process, the process will be given a *context*, which will include a *fault handler* that will be triggered by a *schedule*. The Timer Intermediate Event attached to the Sub-Process boundary sets the *schedule* to one week. The *activity set* for the *exception* will be the source of all the remaining activities of the Process, since all the remaining Sequence Flow begins from the Timer Intermediate Event. The first two Tasks, as shown in the figure, will map to *actions*. The Document Object shown in the figure is not mapped into BPML. The remainder of the Process will be described in the next section.

Example 8 shows sample BPML code that defines the "Collect Votes" Sub-Process.

```
<process name="CollectVotes">
  <!--This is a nested process for the E-Mail Voting collection. It consists of
  an all and a scheduled fault handler (a timeout). The all will never complete
  normally since there is an infinite loop inside. The timeout is intended to be
  the normal way of ending the process-->
  <all>
    <!--This is the first of the four paths of the fork-->
    <choice name="ConferenceCallthisWeek">
      <event>
        <synch name="Call" signal="Call"/>
        <action name="ModerateConferenceCallDiscussion"
                portType="tns:internalPort" operation="sendConCall"/>
        <raise name="NoCall" signal="NoCall"/>
      </event>
      <event>
        <synch name="NoCall" signal="NoCall"/>
        <raise name="Call" signal="Call"/>
      </event>
    </choice>
    <!--the second and third paths of the fork are not shown-->
    <!--This is the fourth of the four paths of the fork and is intended to be an
    infinite loop that is eventually interrupted by the Time Out. This is
    necessary since any voter can change their vote until the deadline-->
    <until>
      <condition>1=0<condition/>
      <action name="ReceiveVote" portType="tns:emailPort"
              operation="receiveVote"/>
      <action name="IncrementTally" portType="tns:internalPort"
              operation="sendReceiveTotal">
        <output property="AllItemsCompleted" element="..."/>
        <output property="NoMajority" element="..."/>
      </action>
    </until>
  </all>
  <context>
    <schedule code="OneWeek" duration="P7D"/>
    <fault>
      <case code="OneWeek">
        <action name="PrepareResults" portType="tns:internalPort"
                operation="sendReceiveResults">
          <output property="NumVoted" element="..."/>
          <output property="NoMajority" element="..."/>
        </action>
        <action name="EMailResultsofVote" portType="tns:emailPort"
                operation="sendVotingResults"/>
        <!--the rest of the process is not shown-->
      </case>
    </fault>
  </context>
</process>
```

Example 8 BPML Sample of the Second Sub-Process

6.4 The End of the Process

Figure 64 shows the last section of the Process, which includes a complex set of Decisions and loops.

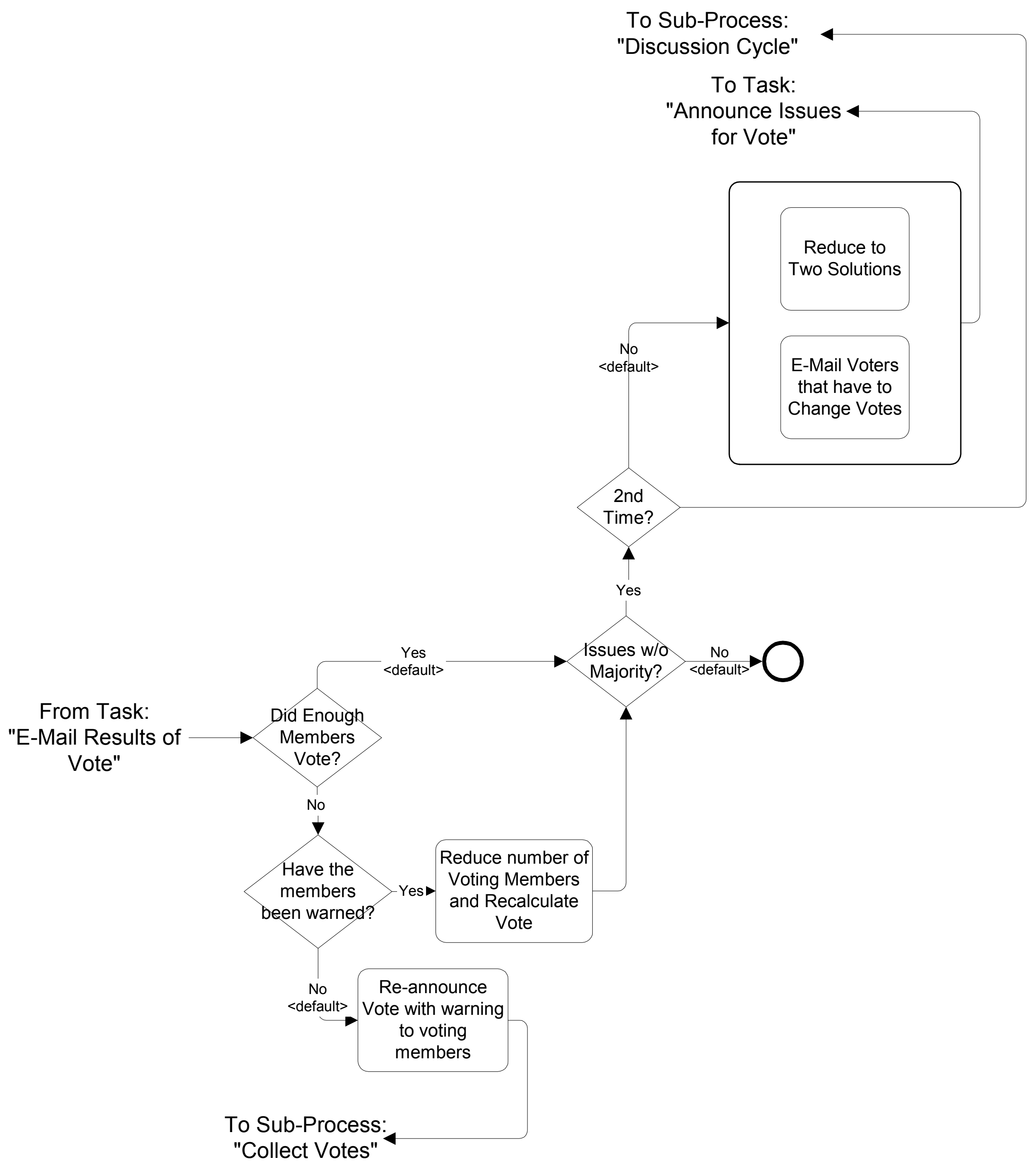

To Sub-Process: "Discussion Cycle"
To Task: "Announce Issues for Vote"
Reduce to Two Solutions
E-Mail Voters that have to Change Votes
No <default>
2nd Time?
Yes
Yes <default>
Issues w/o Majority?
No <default>
From Task: "E-Mail Results of Vote"
Did Enough Members Vote?
No
Have the members been warned?
Yes
Reduce number of Voting Members and Recalculate Vote
No <default>
Re-announce Vote with warning to voting members
To Sub-Process: "Collect Votes"

Figure 64 The last segment of the E-Mail Voting Process

This segment of the Process continues from where the last segment left off (as described in the section above). It contains four Decisions that interact with each other and create loops to upstream activities.

The first Decision, “Did Enough Members Vote?,” is necessary since two-thirds of the voting members are required to approve any solution to an issue. If less than two-thirds of the voting members cast votes, which sometimes happens, the issues can't be resolved. This Decision flows to another Decision for both of its Alternatives. The "No" Alternative is followed by the "Have the Members been Warned?" Decision. If a voting member misses a vote, they are warned. If they miss a second vote, they lose their status as a voting member and the voting percentages are recalculate through a Task ("Reduce number of Voting Members and Recalculate Vote"). If they haven't been warned, then a warning is sent and the voting week is repeated.

If all issues are resolved, then the Process is done. If not, then another Decision is required. The voting is given two chances before it goes back to another cycle of discussion. The first time will see a reduction of the number of solutions to the two most popular based on the vote (more if there are ties). Some voting members will have to change their votes just because their solution is no longer valid. These two activities are placed in a Sub-Process to show how a Sub-Process without Start and End Events can be used to create a simple set of parallel activities. Informally, this is called a "parallel box." It is not a special object, but another use of Sub-Processes. For simple situations, it can be used to show a set of parallel activities without the extra clutter of a lot of Sequence Flows. In actuality, these two Tasks cannot actually be done in parallel, but they are modeled this way to highlight the optional use of Start and End Events.

After the parallel box, the flow loops back to the "Collect Votes" Sub-Process. If there already has been two cycles of voting, then the process flows back to the "Decision Cycle" Sub-Process.

6.4.1 Mapping to BPEL4WS

As mentioned above, the entire contents of this segment follow a Timer Intermediate Event, which means they are contained in the *faultHandler* of the *scope* within the "Collect Votes" *process*. Each of the Decisions in this section will map to a BPEL4WS *switch*.

The first Decision, "Did Enough Members Vote?," flows to another Decision for both of its Alternatives. Thus, each of the *switch cases* will contain another *switch*. The "No" Alternative is followed by the "Have the Members been Warned?" Decision. Each Alternative from this Decision is followed by a Task, which maps to *Invokes* (one synchronous and the other asynchronous). The "No (default)" Alternative leads to a loop. This looping is handled by using an *invoke* (asynchronous) to the "DerivedProcess3" *process*, which was created just for the purpose of this loop (since it is in the context of a more complex looping situation).

Notice that the "Issues w/o Majority?" Decision can be reached through the alternative paths from two different Decisions. This creates a situation that has two impacts on the Mapping to Execution Languages. First, it creates a section of the Process in which the Alternatives from two Decisions overlap. This is possible in a graph-structured diagram like BPMN, but in a block-structured (and acyclical) language like BPEL4WS, these two sections cannot overlap because they have different block boundaries. This means that this section must be repeated in some way in both of the appropriate *switch case activities*. All these elements could be actually duplicated or they can be separated into a derived *process* and then *invoked* from the appropriate place. The later method was used in this example (see Example 9 and Example 10).

Note: At this point, BPMN does not specify whether a reused section of a BPMN diagram should map to a derived *process* that is *invoked* from each location of duplication, or whether the section should remain intact and be duplicated in each appropriate location. This is left up to the specific implementation of BPMN since both solutions will behave equivalently.

The second impact of the multiple incoming Sequence Flows into the "Issues w/o Majority?" Decision has to do with how the three visible loops are created (actually there are five loops). Normally, Sequence Flow loops will map to a BPEL4WS *while*. If there are multiple loops in the Process they have to be physically separated or completely nested because of the block-structured nature of the BPML looping elements. The alternative paths of the Decisions cannot be mixed and still maintain the BPEL4WS blocks they way that the end of the "E-mail Voting" Process mixes the paths.

A different type of looping mechanism is required. This method requires the creation of a set of derived *processes* that can reference each other and also themselves. In this way, a block-structured language can simulate a set of interleaving loops (as seen in a graph-structured diagram). Thus, in this BPMN example, derived *processes* were created to mark places where loops can be targeted within the BPML code from the "downstream" elements. A BPML *invoke* (asynchronous) is used to re-perform activities that had already been executed in the process.

Note: A synchronous *invoke* could also be used to access the *processes* to perform the loop. With a synchronous *invoke*, the source *process* would remain active until the target *process* (and any other loops that follow) have been completed before it also completes. With an asynchronous *invoke*, the source *process* would complete immediately, reducing the number of active BPM system resources. At this point, BPMN does not specify whether an asynchronous *invoke* or a synchronous *invoke* should be used in this situation.

Example 9 displays the BPEL4WS code for first part of the end of the “E-Mail Voting Process.”

```
    <!--This segment of the code is within the context of the “Collect
        Votes” nested process-->
<catch property="tns:OneWeek" type="duration">
    <!--The BPMN diagram shows that the Timer Intermediate Event connects
        directly to the rest of the Process. Thus, they will show up in this
        activity set-->
    <!--The first two actions are not shown-->
  <sequence>
    <invoke name="PrepareResults" partner="internal" portType="tns:internalPort"

             operation="sendReceiveResults" inputContainer="processData"
             outputContainer="processData"/>
    <invoke name="EMailResultsofVote" partner="WGVoter"
             portType="tns:emailPort" operation="sendVotingResults"
             inputContainer="processData"/>
    <switch name="DidEnoughMembersVote">
      <!-- name="No" -->
      <case condition="bpws:getContainerProperty(ProcessData,NumVoted)>
                      (.7)*(bpws:getContainerProperty(ProcessData,NumVWGM))">
        <switch name="Havethemembersbeenwarned">
          <!-- name="Yes" -->
          <case condition="bpws:getContainerProperty(ProcessData,VotersWarned)
                          =true">
            <sequence>
              <invoke name="ReducenumberofVotingMembersandRecalculateVote"
                       partner="internal" portType="tns:internalPort"
                       operation="sendReceiveNumVoters"
                       inputContainer="processData"
                       outputContainer="processData"/>
              <!--Some elements of the process were separated into a derived
                   process since they would have been repeated. They would have
                   been repeated because they are arrived by alternative paths that
                   do not close a set of alternative paths. -->
              <invoke name="DerivedProcess4" partner="Internal"
                      portType="tns:processPort" operation="callDerivedProcess4"
                      inputContainer="processData"/>
            </sequence>
          </case>
          <!-- name="No (otherwise)" -->
          <otherwise>
            <sequence>
              <invoke name="ReannounceVotewithwarningtovotingmembers"
                      partner="WGVoter" portType="tns:emailPort"
                      operation="sendReannounceVote" inputContainer="processData"
                      outputContainer="processData"/>
              <invoke name="DerivedProcess3" partner="Internal"
                      portType="tns:processPort" operation="callDerivedProcess3"
                      inputContainer="processData"/>
            </sequence>
          </otherwise>
        </switch>
      </case>
      <!-- name="Yes (otherwise)" -->
      <otherwise>
        <!-- Some elements of the process were separated into a derived process
             since they would have been repeated. They would have been repeated
             because they are arrived by alternative paths that do not close a
             set of alternative paths. -->
        <invoke process="DerivedProcess4" partner="Internal"
                portType="tns:processPort" operation="callDerivedProcess4"
                inputContainer="processData"/>
      </otherwise>
    </switch>
  </sequence>
</catch>
```

Example 9 Sample BPEL4WS code for the last section of the Process

Example 10 shows the BPEL4WS code for the Process from the “Issues w/o Majority?” Decision until the end of the Process or loops. The mappings are a fairly straightforward set of *switches*. If all issues are resolved, then the Process is done. If not, then another Decision is required. The “parallel box,” as is any forking situation, will map to a BPEL4WS *flow*. After the parallel box, the flow loops back to the “Collect Votes” Sub-Process. This looping is handled by using an *invoke* (asynchronous) to the “DerivedProcess2” *process*, which was created just for the purpose of this loop.

If there has already been two cycles of voting, then the process flows back to the “Decision Cycle” Sub-Process. This looping is handled by using an *invoke* (asynchronous) to the “DerivedProcess1” *process*, which was created just for the purpose of this loop

```
<process name="DerivedProcess4">
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
             operation="callDerivedProcess4" container="processData"
             createInstance="Yes"/>
    <switch name="IssueswoMajority">
      <case name="Yes" condition="NoMajority=true">
        <switch name="2ndTime">
          <!-- name="Yes"  -->
          <case condition="bpws:getContainerProperty(ProcessData,VotedOnce)
                           =true">
            <!--This is done to do the complex looping situation. -->
            <invoke name="DerivedProcess1" partner="Internal"
                    portType="tns:processPort" operation="callDerivedProcess1"
                    inputContainer="processData"/>
          </case>
          <!-- name="No (otherwise)"-->
          <otherwise>
            <sequence>
```

```
              <flow>
                <invoke name="ReducetoTwoSolutions" partner="internal"
                        portType="tns:internalPort"
                        operation="sendReceiveSolutions"
                        inputContainer="processData"
                        outputContainer="processData"/>
                <invoke name="EMailVotersthathavetoChangeVotes"
                        partner="WGVoter" portType="tns:emailPort"
                        operation="sendVoteWarning" inputContainer="processData"/>
              </flow>
              <invoke process="DerivedProcess2" partner="Internal"
                      portType="tns:processPort" operation="callDerivedProcess2"
                      inputContainer="processData"/>
            </sequence>
          </otherwise>
        </switch>
      </case>
      <otherwise name="Nootherwise">
        <!-- This is one of the two ways to the end of the Process. -->
        <empty/>
      </otherwise>
    </switch>
  </sequence>
</process>
```

Example 10 Sample BPEL4WS code for derived *process* for repeated elements

6.4.2 Mapping to BPML

As mentioned above, the entire contents of this segment follow a Timer Intermediate Event, which means they are contained in the *case* of a *fault hander* in the “Collect Votes” *nested process*. Each of the Decisions in this section will map to a BPML *switch*.

The first Decision, “Did Enough Members Vote?,” flows to another Decision for both of its Alternatives. Thus, each of the *switch cases* will contain another *switch*. The “No” Alternative is followed by the “Have the Members been Warned?” Decision. Each Alternative from this Decision is followed by a Task, which maps to *actions*. The “No (default)” Alternative leads to a loop. This looping is handled by using a *call* to the “DerivedProcess3” *nested process*, which was created just for the purpose of this loop (since it is in the context of a more complex looping situation).

Notice that the “Issues w/o Majority?” Decision can be reached through the alternative paths from two different Decisions. This creates a situation that has two impacts on the Mapping to Execution Languages. First, it creates a section of the Process in which the Alternatives from two Decisions overlap. This is possible in a graph-structured diagram like BPMN, but in a block-structured language like BPML, these two sections cannot overlap because they have different block boundaries. This means that this section must be repeated in some way in both of the appropriate *switch case activity sets*. All these elements could be actually duplicated or they can be separated into a derived *nested process* and then *call*ed from the appropriate place. The later method was used in this example (see Example 11 and Example 12).

Note: At this point, BPMN does not specify whether a reused section of a BPMN diagram should map to a derived *nested process* that is *call*ed from each location of duplication, or whether the section should remain intact and be duplicated in each appropriate location. This is left up to the specific implementation of BPMN since both solutions will behave equivalently.

The second impact of the multiple incoming Sequence Flows into the "Issues w/o Majority?" Decision has to do with how the three visible loops are created (actually there are five loops). Normally, Sequence Flow loops will map to a BPML *while, until*, or *foreach*. If there are multiple loops in the Process they have to be physically separated or completely nested because of the block-structured nature of the BPML looping elements. The alternative paths of the Decisions cannot be mixed and still maintain the BPML blocks they way that the end of the "E-mail Voting" Process mixes the paths.

A different type of looping mechanism is required. This method requires the creation of a set of derived *nested processes* that can reference each other and also themselves. In this way, a block-structured language can simulate a set of interleaving loops (as seen in a graph-structured diagram). Thus, in this BPMN example, derived *nested processes* were created to mark places where loops can be targeted within the BPML code from the "downstream" elements. A BPML *call* is used to re-perform activities that had already been executed in the process.

Note: A *spawn* could also be used to access the *nested processes* to perform the loop. With a *call*, the source *nested process* would remain active until the target *nested process* (and any other loops that follow) have been completed before it also completes. With a *spawn*, the source *nested process* would complete immediately, reducing the number of active BPM system resources. At this point, BPMN does not specify whether a *spawn* or a *call* should be used in this situation.

Example 11 displays BPML code for the first part of the end of the “E-Mail Voting Process.”

```
    <!--This segment of the code is within the context of the “Collect
    Votes” nested process-->
<schedule code="OneWeek" duration="P7D"/>
<fault>
  <case code="OneWeek">
    <!--The BPMN diagram shows that the Timer Intermediate Event connects
        directly to the rest of the Process. Thus, they will show up in this
        activity set-->
    <!--The first two actions are not shown-->
    <switch name="DidEnoughMembersVote">
      <case name="No">
        <condition>NumVoted>(.7)*(NumVWGM)<condition/>
        <switch name="Havethemembersbeenwarned">
          <case name="Yes">
            <condition>VotersWarned=true<condition/>
            <action name="ReducenumberofVotingMembersandRecalculateVote"
                    portType="tns:internalPort" operation="sendReceiveNumVoters"/>
          <!--Some elements of the process were separated into a derived nested
          process since they would have been repeated. They would have been
          repeated because they are arrived by more than one alternative paths-->
            <call process="DerivedProcess4"/>
          </case>
          <default name="Nodefault">
            <action name="ReannounceVote with warning to voting members"
                    portType="tns:emailPort" operation="sendReannounceVote">
              <output property="VotersWarned" element="..."/>
            </action>
            <call process="DerivedProcess3"/>
          </default>
        </switch>
      </case>
      <default name="Yesdefault">
      <!--Some elements of the process were separated into a derived nested
      process since they would have been repeated. They would have been
      repeated because they are arrived by more than one alternative paths-->
        <call process="DerivedProcess4"/>
      </default>
    </switch>
  </case>
</fault>
```

Example 11 Sample BPML code for the last section of the Process

Example 12 shows the BPML code for the Process from the “Issues w/o Majority?” Decision until the end of the Process or loops. The mappings are a fairly straightforward set of *switches*. If all issues are resolved, then the Process is done. If not, then another Decision is required. The “parallel box,” as is any forking situation, will map to a BPML *all*. After the parallel box, the flow loops back to the “Collect Votes” Sub-Process. This looping is handled by using a *call* to the “DerivedProcess2” *nested process*, which was created just for the purpose of this loop.

If there has already been two cycles of voting, then the process flows back to the “Decision Cycle” Sub-Process. This looping is handled by using a *call* to the “DerivedProcess1” *nested process*, which was created just for the purpose of this loop.

```
<process name="DerivedProcess4">
  <switch name="IssueswoMajority">
    <case name="Yes">
      <condition>NoMajority=true<condition/>
      <switch name="2ndTime">
        <case name="Yes">
          <condition>VotedOnce=true<condition/>
          <!--This is done to do the complex looping situation-->
          <call process="DerivedProcess1"/>
        </case>
        <default name="Nodefault">
          <all>
            <action name="ReducetoTwoSolutions" portType="tns:internalPort"
                    operation="sendReceiveSolutions">
              <output property="NoMajority" element="..."/>
            </action>
            <action name="EMailVotersthathavetoChangeVotes"
                    portType="tns:emailPort" operation="sendVoteWarning"/>
            <call process="DerivedProcess2"/>
          </all>
        </default>
      </switch>
    </case>
    <default name="Nodefault">
      <!--This is one of the two ways to the end of the Process-->
      <empty/>
    </default>
  </switch>
</process>
```

Example 12 Sample BPML code for derived *nested process* for repeated elements

7. Mapping to Execution Languages

Many of the above sections specified the Mapping to Execution Languages for the topics of those sections. Those mappings will not be duplicated in this section. This section will cover the mappings to BPEL4WS and BPML that are derived by analyzing the relationships between the elements described in the above sections. For example, a Decision object marks the beginning of a *switch*, but the end of the *switch* will have to be determined by tracing the alternative paths from the Decision and finding the point in the Process where all the alternative paths have merged, which may be the end of the Process. The strings of activities that lie between the Decision and the merging point will comprise the *activity sets* for each of the *switch cases*. The location of the final merging point could be complicated by the inclusion of intermediary Decisions and/or parallel sections of the Process. BPMN does not include a specific merging object that will be tied one-to-one with a specific Decision object that will allow the quick identification of the merging point. Likewise, BPMN does not have paired objects to mark the beginning and end of parallel activities that would fit into the BPML *all* element or BPEL4WS *flow* element. Furthermore, BPMN is cyclical in that it allows Sequence Flows to connect to upstream objects so that a modeler can easily visualize looping situations. The exact configuration of these loops will determine how they are mapped to BPM execution constructions, some of which are acyclical.

To determine the appropriate merging and joining points that are needed to construct execution language elements such as *switch*, the configuration of the Process needs to be analyzed. The mechanism we are proposing is called Token Analysis. This involves the creation of a conceptual Token that will “traverse” all the Sequence Flows of the Process. The Token will have a hierarchical TokenID set that will expand or contract based on the forking and joining and/or splitting and merging that occurs throughout the Process. By matching the TokenID set of Tokens that arrive at objects that have multiple incoming Sequence Flows, it will be possible to determine the boundaries of execution language structured activities.

Editor’s Note: the finalization of the Mapping to Execution Languages through Token Analysis is an open issue and will be developed further in a later version of the specification. Refer to the section entitled “Open Issues” on page 137 for a complete list of the issues open for BPMN.

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

BPML

(BPML) 1.0, BPMI, November 2002

http://www.BPMI.org

BPEL4WS

(BPEL4WS) 1.0, IBM/Microsoft/BEA, August 2002

http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

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

Business Process modeling

Jean-Jacques Dubray, “A Novel Approach for Modeling Business Process Definitions,” 2002

http://www.ebpml.org/ebpml2.2.doc

Dublin Core Meta Data

Dublin Core Metadata Element Set, Dublin Core Metadata Initiative

http://dublincore.org/documents/dces/

ebXML BPSS

Jean-Jacques Dubray, “A new model for ebXML BPSS Multi-party Collaborations and Web Services Choreography,” 2002

http://www.ebpml.org/ebpml.doc

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

WfMC Glossary

Workflow Management Coalition Terminology and Glossary.

http://www.wfmc.org/standards/docs.htm

9. Open Issues

The following elements or features of BPMN are not fully defined in this version of the specification:

- Use of Link Start and End Events (to dynamically pass Tokens between levels of a Process).
- Event properties for defining how they can be correlated with a specific process instance.
- Use of the PassThrough property for End Events. This is related to the Link Start and End Events.
- Process Breaks.
- The position of Loop and Ad Hoc markers.
- Enhancing the Loop marker to show parallelism.
- Inclusive Decisions.
- Will Decisions have a mandatory default alternative?
- Enhanced graphical mechanisms for forking, joining, and merging.
- Expanding Flow Conditions so that multiple Tokens can proceed after a join. This is related to the use of a PassThrough property.
- Should interface processes be a Lane within a Pool? Or should they be separated in their own Pool?
- Using an Intermediate Event without interrupting an Event Context.
- Visual marker or queue for Transactions.
- Spawning and Synchronizing Activities.
- Linking activities in private business processes to their corresponding activities in an abstract or collaboration process.
- Completed Mapping to Execution Languages (including the abstract definitions in BPEL4WS and WSCI).
- Specification of BPMN as an XML language.

Appendix A: E-Mail Voting Process BPEL4WS

This appendix provides the complete BPEL4WS code for the example BPMN business process that is described in the section entitled “BPMN by Example” on page 103.

```
<!-- The Main Process -->
<process name="EMailVotingProcess">
  <containers>
    <container name="processData" messageType="processDataMessage"/>
  </containers>
  <sequence>
    <!--This starts the beginning of the Process. The process that sends the
        starting message every Friday is not shown here.-->
    <receive partner="Internal" portType="tns:processPort"
             operation="callEMailVotingProcess" container="processData"
             createInstance="Yes"/>
    <invoke name="ReviewIssueList" partner="Internal"
            portType="tns:internalPort" operation="sendIssueList"
            inputContainer="processData" outputContainer="processData"/>
    <switch name="AnyIssuesReady">
      <!--name="Yes" -->
      <case condition="bpws:getContainerProperty(ProcessData,NumIssues)>0">
        <!-- A chunk of this process is separated into a derived process so that
            it can be called from a complex loop. -->
        <invoke name="DerivedProcess1" partner="Internal"
                portType="tns:processPort" operation="callDerivedProcess1"
                inputContainer="processData"/>
      </case>
      <!--name="otherwise" -->
      <otherwise>
        <!--This is one of the two ways to the end of the Process.-->
        <empty/>
      </otherwise>
    </switch>
  </sequence>
<!-- A Derived Process -->
<process name="DerivedProcess1">
  <containers>
    <container name="processData" messageType="processDataMessage"/>
  </containers>
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
             operation="callDerivedProcess1" container="processData"
             createInstance="Yes"/>
    <while condition="bpws:getContainerProperty(ProcessData,DiscussionOver)
                  =false">
      <!--This calls the first Sub-Process-->
      <invoke process="DiscussionCycle" partner="Internal"
              portType="tns:processPort operation="callDiscussionCycle"
              inputContainer="processData"/>
    </while>
    <invoke name="DerivedProcess2" partner="Internal" portType="tns:processPort"
            operation="callDerivedProcess2" inputContainer="processData"/>
  </sequence>
</process>
</process>
<!-- A Derived Process -->
<process name="DerivedProcess2">
  <!-- This starts the middle section of the process. -->
  <containers>
    <container name="processData" messageType="processDataMessage"/>
  </containers>
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
             operation="callDerivedProcess2" container="processData"
             createInstance="Yes"/>
    <invoke name="AnnounceIssuesforVote" partner="WGVoter"
            portType="tns:emailPort" operation="sendVoteAnnouncement"
            inputContainer="processData"/>
    <invoke name="DerivedProcess3" partner="Internal" portType="tns:processPort"

            operation="callDerivedProcess3" inputContainer="processData"/>
  </sequence>
</process>
<!-- A Derived Process -->
<process name="DerivedProcess3">
  <!--this calls the second Sub-Process. After the Collect Votes Sub-Process
      times out, the rest of the process will be in the fault handler
      of that process. Calls from there will loop back into other processes.-->
  <containers>
    <container name="processData" messageType="processDataMessage"/>
  </containers>
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
            operation="callDerivedProcess3" container="processData"
            createInstance="Yes"/>
    <invoke name="CollectVotes" partner="Internal" portType="tns:processPort"
           operation="callCollectVotes" inputContainer="processData"/>
  </sequence>
</process>
<!-- A Derived Process -->
<process name="DerivedProcess4">
  <containers>
    <container name="processData" messageType="processDataMessage"/>
  </containers>
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
            operation="callDerivedProcess4" container="processData"
            createInstance="Yes"/>
    <switch name="IssueswoMajority">
      <case name="Yes" condition="NoMajority=true">
```

```
        <switch name="2ndTime">
          <!-- name="Yes" -->
          <case condition="bpws:getContainerProperty(ProcessData,VotedOnce)
                          =true">
            <!--This is done to do the complex looping situation. -->
            <invoke name="DerivedProcess1" partner="Internal"
                    portType="tns:processPort" operation="callDerivedProcess1"
                    inputContainer="processData"/>
          </case>
          <!-- name="No (otherwise)" -->
          <otherwise>
            <sequence>
              <flow>
                <invoke name="ReducetoTwoSolutions" partner="internal"
                        portType="tns:internalPort"
                        operation="sendReceiveSolutions"
                        inputContainer="processData"
                        outputContainer="processData"/>
                <invoke name="EMail Voters that have to Change Votes"
                         partner="WGVoter" portType="tns:emailPort"
                         operation="sendVoteWarning" inputContainer="processData"/>
              </flow>
              <invoke process="DerivedProcess2" partner="Internal"
                      portType="tns:processPort" operation="callDerivedProcess2"
                      inputContainer="processData"/>
            </sequence>
          </otherwise>
        </switch>
      </case>
      <otherwise name="Nootherwise">
        <!-- This is one of the two ways to the end of the Process. -->
        <empty/>
      </otherwise>
    </switch>
  </sequence>
</process>
<!-- A User Built Process -->
<process name="DiscussionCycle">
  <!--This defines the first Sub-Process. -->
  <containers>
    <container name="processData" messageType="processDataMessage"/>
  </containers>
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
             operation="callDiscussionCycle" container="processData"
             createInstance="Yes"/>
    <invoke name="AnnounceIssuesforDiscussion" partner="WGVoter"
            portType="tns:emailPort" operation="sendDiscussionAnnouncement"
            inputContainer="processData"/>
    <flow>
      <links>
        <link name="Delay6daysfromDiscussionAnnouncementtoEMailDiscussion
                    DeadlineWarning"/>
    <link name="CheckCalendarforConferenceCalltoWaituntilThursday9am"/>
    <link name="CheckCalendarforConferenceCalltoCall"/>
    <link name="WaituntilThursday9amtoModerateConferenceCallDiscussion"/>
  </links>
  <!-- This is the first of the three paths of the fork. -->
  <scope>
    <flow>
      <invoke name="ModerateEmailDiscussion" partner="internal"
              portType="tns:internalPort" operation="sendDiscussion"
              inputContainer="processData"
              outputContainer="processData"/>
      <sequence>
        <wait name="7days" for="P7D"/>
        <throw faultName="7days_fault"/>
      </sequence>
    </flow>
    <faultHander>
      <catch faultName="7days_fault">
        <invoke name="ReviewStatusofDiscussion" partner="internal"
                portType="tns:internalPort"
                operation="receiveDiscussionStatus"
                inputContainer="processData" outputContainer="processData"/>
      </catch>
    </faultHander>
  </scope>
  <!-- This is the second of the three paths of the fork. -->
  <wait name="Delay6daysfromDiscussionAnnouncement" for="P6D">
    <target linkName="Delay6daysfromDiscussionAnnouncementtoEMail
                     DiscussionDeadlineWarning"/>
  </wait>
  <invoke name="EMailDiscussionDeadlineWarning" partner="WGVoter"
          portType="tns:emailPort" operation="sendDiscussionWarning"
          inputContainer="processData">
    <source linkName="Delay6daysfromDiscussionAnnouncementtoEMail
                     DiscussionDeadlineWarning"/>
  </invoke>
  <!-- This is the third of the three paths of the fork. -->
  <invoke name="CheckCalendarforConferenceCall" partner="internal"
          portType="tns:internalPort" operation="receiveCallSchedule"
          inputContainer="processData" outputContainer="processData">
    <target linkName="CheckCalendarforConferenceCalltoWaituntilThursday9am"
        transitionCondition="bpws:getContainerProperty(processData,conCall)
                          =true"/>
    <target linkName="Check Calendar for Conference Call to Call"
          transitionCondition="(bpws:getContainerProperty(processData,
                               conCall)=true)=false"/>
  </invoke>
  <!-- name="Yes" -->
  <wait name="WaituntilThursday9am" for="P6DT9H">
    <source linkName="CheckCalendarforConferenceCalltoWaituntil
                     Thursday9am"/>
    <target linkName="WaituntilThursday9amtoModerateConferenceCall
                    Discussion"/>
      </wait>
      <invoke name="ModerateConferenceCallDiscussion" partner="internal"
              portType="tns:internalPort" operation="sendConCall"
              inputContainer="processData" outputContainer="processData">
        <source linkName="WaituntilThursday9amtoModerateConferenceCall
                         Discussion"/>
        <target linkName="ModerateConferenceCallDiscussiontoNoCall"/>
      </invoke>
      <!-- This is used as a message to be used for a pick in the "Collect Votes"
           process -->
      <invoke name="NoCall" partner="internal" portType="tns:processlPort"
             operation="sendNo_Call" inputContainer="processData">
        <source linkName="ModerateConferenceCallDiscussiontoNoCall"/>
      </invoke>
      <!-- name="otherwise" -->
      <!-- This is used as a message to be used for a pick in the "Collect Votes"
           process -->
      <invoke name="Call" partner="internal" portType="tns:processlPort"
              operation="sendCall" inputContainer="processData">
        <source linkName="CheckCalendarforConferenceCalltoCall"/>
      </invoke>
    </flow>
  </sequence>
</process>
<!-- A User Built Process -->
<process name="CollectVotes">
  <!--This is a process for the E-Mail Voting collection. It consists of an all
      and a timeout event handler. The all will never complete normally since
      there is an infinite loop inside. The timeout is intended to be the normal
      way of ending the process. -->
  <containers>
    <container name="processData" messageType="processDataMessage"/>
  </containers>
  <sequence>
    <receive partner="Internal" portType="tns:processPort"
             operation="callCollectVotes" container="processData"
             createInstance="Yes"/>
    <scope>
      <flow>
        <sequence>
          <wait name="7days" for="P7D"/>
          <throw faultName="7days_fault"/>
        </sequence>
        <flow>
          <links>
            <link name="Delay6daysfromVoteAnnouncementtoEMailVoteDeadline
                        Warning"/>
          </links>
          <!--This is the first of the four paths of the fork. -->
          <pick name="ConferenceCallthisWeek">
            <!-- name="Call"  -->
            <onMessage partner="internal" portType="tns:processlPort"
                       operation="sendCall" Container="processData">
            <invoke name="ModerateConferenceCallDiscussion" partner="internal"
                    portType="tns:internalPort" operation="sendConCall"
                    inputContainer="processData" outputContainer="processData"/>
            <invoke name="NoCall" partner="internal" portType="tns:processlPort"
                    operation="sendNo_Call" inputContainer="processData"/>
          </onMessage>
          <!-- name="No Call"  -->
          <onMessage partner="internal" portType="tns:processlPort"
                     operation="sendNo_Call" Container="processData">
            <invoke name="Call" partner="internal" portType="tns:processlPort"
                     operation="sendCall" inputContainer="processData"/>
          </onMessage>
        </pick>
        <!-- This is the second of the four paths of the fork. -->
        <invoke name="ModerateEMailDiscussion" partner="internal"
                portType="tns:internalPort" operation="sendDiscussion"
                inputContainer="processData"
                outputContainer="processData"/>
        <!--This is the third of the four paths of the fork.-->
        <wait name="Delay6daysfromVoteAnnouncement" for="P6D">
          <target linkName="Delay6daysfromVoteAnnouncementtoEMailVote
                            DeadlineWarning"/>
        </wait>
        <invoke name="EMailVoteDeadlineWarning" partner="WGVoter"
                portType="tns:emailPort" operation="sendVoteWarning"
                inputContainer="processData">
          <source linkName="Delay6daysfromVoteAnnouncementtoEMailVote
                            DeadlineWarning"/>
        </invoke>
        <!--This is the fourth of the four paths of the fork. This branch of the
             all is intended to be an infinite loop that is eventually
             interrupted by the Time Out.  This is necessary since any voter can
             change their vote until the deadline. -->
        <while condition="1=0">
          <sequence>
            <receive name="ReceiveVote" partner="WGVoter"
                     portType="tns:emailPort" operation="receiveVote"
                     container="processData"/>
            <invoke name="IncrementTally" partner="internal"
                    portType="tns:internalPort" operation="sendReceiveTotal"
                    inputContainer="processData" outputContainer="processData"/>
          </sequence>
        </while>
      </flow>
    </flow>
    <faultHander>
      <catch faultName="7days_fault">
        <!-- The BPMN diagram shows that the Timer Intermediate Event connects
             directly to the rest of the Process. Thus, they will show up in
             this activity set. -->
        <sequence>
```

```
<invoke name="PrepareResults" partner="internal"
       portType="tns:internalPort" operation="sendReceiveResults"
       inputContainer="processData" outputContainer="processData"/>
<invoke name="EMailResultsofVote" partner="WGVoter"
       portType="tns:emailPort" operation="sendVotingResults"
       inputContainer="processData"/>
<switch name="DidEnoughMembersVote">
  <!-- name="No" -->
  <case condition="bpws:getContainerProperty(ProcessData,NumVoted)>
               (.7)*(bpws:getContainerProperty(ProcessData,NumVWGM))">
    <switch name="Havethemembersbeenwarned">
      <!-- name="Yes" -->
      <case condition="bpws:getContainerProperty(ProcessData,
                     VotersWarned)=true">
        <sequence>
          <invoke name="ReducenumberofVotingMembersandRecalculateVote"
                 partner="internal" portType="tns:internalPort"
                 operation="sendReceiveNumVoters"
                 inputContainer="processData"
                 outputContainer="processData"/>
          <!--Some elements of the process were separated into a
              derived process since they would have been repeated. They
               would have been repeated because they are arrived by
               alternativepaths that do not close a set of alternative
               paths. -->
          <invoke name="DerivedProcess4" partner="Internal"
                 PortType="tns:processPort"
                 operation="callDerivedProcess4"
                 inputContainer="processData"/>
        </sequence>
      </case>
      <!-- name="No (otherwise)" -->
      <otherwise>
        <sequence>
          <invoke name="ReannounceVotewithwarningtovotingmembers"
                 partner="WGVoter" portType="tns:emailPort"
                 operation="sendReannounceVote"
                 inputContainer="processData"
                 outputContainer="processData"/>
          <invoke name="DerivedProcess3" partner="Internal"
                 portType="tns:processPort"
                 operation="callDerivedProcess3"
                 inputContainer="processData"/>
        </sequence>
      </otherwise>
    </switch>
  </case>
  <!-- name="Yes (otherwise)" -->
  <otherwise>
    <!-- Some elements of the process were separated into a derived
         process since they would have been repeated. They would
         have been repeated because they are arrived by alternative
         that do not close a set of alternative paths. -->
```

```
                 <invoke process="DerivedProcess4" partner="Internal"
                          portType="tns:processPort" operation="callDerivedProcess4"
                          inputContainer="processData"/>
               </otherwise>
             </switch>
           </sequence>
         </catch>
       </faultHander>
     </scope>
   </sequence>
 </process>
```

Appendix B: E-Mail Voting Process BPML

This appendix provides the complete BPML code for the example BPMN business process that is described in the section entitled “BPMN by Example” on page 103.

```
<process name="EMailVotingProcess">
  <!--The process is started by a schedule time event that is defined in a
      separate process and is not shown here.-->
  <action name="ReviewIssueList" portType="tns:internalPort"
          operation="sendIssueList">
    <output property="NumIssues" element="..."/>
  </action>
  <switch name="AnyIssuesReady">
    <case name="Yes">
      <condition>NumIssues>0<condition/>
      <!--A chunk of this process is separated into a derived nested
      process so that it can be called from a complex loop.-->
      <call name="DerivedProcess1"/>
    </case>
    <default name="Default">
      <!--This is one of the two ways to the end of the Process.-->
      <empty/>
    </default>
  </switch>
  <context>
    <property name="tns:NumVWGM" type="xsd:integer">
      <value>10</value>
    </property>
    <property name="tns:VotedOnce" type="xsd:boolean">
      <value>false</value>
    </property>
    <property name="tns:NoMajority" type="xsd:boolean">
      <value>true</value>
    </property>
    <property name="tns:DiscussionOver" type="xsd:boolean">
      <value>false</value>
    </property>
    <property name="tns:NumIssues" type="xsd:integer">
      <value>0</value>
    </property>
    <property name="tns:NumVoted" type="xsd:integer">
      <value>0</value>
    </property>
    <property name="tns:VotersWarned" type="xsd:boolean">
      <value>false</value>
    </property>
    <property name="tns:ConCall" type="xsd:boolean">
      <value>false</value>
    </property>
```

```
<process name="DerivedProcess1">
  <until>
    <condition>DiscussionOver<condition/>
    <!--This calls the first Sub-Process. -->
    <call process="Discussion Cycle"/>
  </until>
  <call process="DerivedProcess2"/>
</process>
<process name="DerivedProcess2">
  <!--This starts the middle section of the process.-->
  <action name="AnnounceIssuesforVote" portType="tns:emailPort"
          operation="sendVoteAnnouncement"/>
  <call process="DerivedProcess3">
</process>
<process name="DerivedProcess3">
  <!--This calls the second Sub-Process. After the Collect Votes Sub-Process
  times out, the rest of the process will be in the fault handler of
  that process. Calls from there will loop back into other processes. -->
  <call name="CollectVotes" process="CollectVotes"/>
</process>
<process name="DerivedProcess4">
  <switch name="Issuesw/oMajority">
    <case name="Yes">
      <condition>NoMajority=true<condition/>
      <switch name="2ndTime">
        <condition>VotedOnce=true<condition/>
        <case name="Yes">
          <!--This is done to do the complex looping situation.-->
          <call process="DerivedProcess1"/>
        </case>
        <default name="Nodefault">
          <all>
            <action name="ReducetoTwoSolutions" portType="tns:internalPort"
                    operation="sendReceiveSolutions">
              <output property="NoMajority" element="..."/>
            </action>
            <action name="EMailVotersthathavetoChangeVotes"
                    portType="tns:emailPort" operation="sendVoteWarning"/>
            <call process="DerivedProcess2"/>
          </all>
        </default>
      </switch>
    </case>
    <default name="Nodefault">
      <!--This is one of the two ways to the end of the Process.-->
      <empty/>
    </default>
  </switch>
</process>
<process name="DiscussionCycle">
  <!--This defines the first Sub-Process.-->
  <action name="AnnounceIssuesforDiscussion" portType="tns:emailPort"
          operation="sendDiscussionAnnouncement"/>
    <all>
      <call process="ModerateEmailDiscussionProcess"/>
      <sequence>
        <delay name="Delay6daysfromDiscussionAnnouncement" duration="P6D"/>
        <action name="EMailDiscussionDeadlineWarning" portType="tns:emailPort"
                operation="sendDiscussionWarning"/>
      </sequence>
      <sequence>
        <action name="CheckCalendarforConferenceCall"
            portType="tns:internalPort" operation="receiveCallSchedule">
          <output property="ConCall" element="…"/>
        </action>
        <switch name="ConferenceCallinDiscussionWeek">
          <case name="Yes">
            <condition>ConCall=true<condition/>
            <delay name="WaituntilThursday9am" instant="P6DT9H"/>
            <action name="ModerateConferenceCallDiscussion"
                    portType="tns:internalPort" operation="sendConCall"/>
            <raise name="NoCall" signal="NoCall"/>
          </case>
          <default name="Default">
            <raise signal="Call"/>
          </default>
        </switch>
      </sequence>
    </all>
  </process>
  <process name="ModerateEmailDiscussionProcess">
    <action name="ModerateEmailDiscussion" portType="tns:internalPort"
            operation="sendDiscussion"/>
    <context>
      <schedule code="OneWeek" duration="P7D"/>
      <fault>
        <case code="OneWeek">
          <action name="ReviewStatusofDiscussion" portType="tns:internalPort"
                  operation="receiveDiscussionStatus">
            <output property="DiscussionOver" element="…"/>
          </action>
        </case>
      </fault>
    </context>
  </process>
  <process name="CollectVotes">
    <!--This is a nested process for the E-Mail Voting collection. It
    consists of an all and a timeout event handler. The all will never
    complete normally since there is an infinite loop inside. The
    timeout is intended to be the normal way of ending the process.-->
    <all>
      <!--This is the first of the four paths of the fork. -->
      <choice name="ConferenceCallthisWeek">
        <event>
          <synch name="Call" signal="Call"/>
        <action name="ModerateConferenceCallDiscussion"
                portType="tns:internalPort" operation="sendConCall"/>
        <raise name="NoCall" signal="NoCall"/>
      </event>
      <event>
        <synch name="NoCall" signal="NoCall"/>
        <raise name="Call" signal="Call"/>
      </event>
    </choice>
    <!--This is the second of the four paths of the fork. -->
    <action name="ModerateEMailDiscussion" portType="tns:internalPort"
            operation="sendDiscussion"/>
    <!--This is the third of the four paths of the fork. -->
    <sequence>
      <delay name="Delay6daysfromVoteAnnouncement" duration="P6D"/>
      <action name="EMailVoteDeadlineWarning" portType="tns:emailPort"
              operation="sendVoteWarning"/>
    </sequence>
    <!--This is the fourth of the four paths of the fork.
    This branch of the all is intended to be an infinite loop that
    is eventually interrupted by the Time Out. This is necessary
    since any voter can change their vote until the deadline. -->
    <until>
      <condition>1=0<condition/>
      <action name="ReceiveVote" portType="tns:emailPort"
              operation="receiveVote"/>
      <action name="IncrementTally" portType="tns:internalPort"
              operation="sendReceiveTotal">
        <output property="AllItemsCompleted" element="..."/>
        <output property="NoMajority" element="..."/>
      </action>
    </until>
  </all>
  <context>
    <schedule code="OneWeek" duration="P7D"/>
    <fault>
      <case code="OneWeek">
        <!--The BPMN diagram shows that the Timer Intermediate Event
        connects directly to the rest of the Process. Thus, they will
        show up in this activity set. -->
        <action name="PrepareResults" portType="tns:internalPort"
                operation="sendReceiveResults">
          <output property="NumVoted" element="..."/>
          <output property="NoMajority" element="..."/>
        </action>
        <action name="EMailResultsofVote"
                portType="tns:emailPort"
                operation="sendVotingResults"/>
        <switch name="DidEnoughMembersVote">
          <case name="No">
            <condition>NumVoted>(.7)*(NumVWGM)<condition/>
```

```
              <switch name="Havethemembersbeenwarned">
                <case name="Yes" condition="VotersWarned=true">
                  <condition>VotersWarned=true<condition/>
                  <action name="ReducenumberofVotingMembersandRecalculateVote"
                          portType="tns:internalPort"
                          operation="sendReceiveNumVoters"/>
                <!--Some elements of the process were separated into a derived
                nested process since they would have been repeated. They would have
                been repeated because they are arrived by alternative paths
                that do not close a set of alternative paths. -->
                  <call process="DerivedProcess4"/>
                </case>
                <default name="Nodefault">
                  <action name="ReannounceVotewithwarningtovotingmembers"
                           portType="tns:emailPort" operation="sendReannounceVote">
                    <output property="VotersWarned" element="..."/>
                  </action>
                  <call process="DerivedProcess3"/>
                </default>
              </switch>
            </case>
            <default name="Yesdefault">
            <!--Some elements of the process were separated into a derived nested
            process since they would have been repeated. They would have been
            repeated because they are arrived by alternative paths that do not
            close a set of alternative paths.-->
              <call process="DerivedProcess4"/>
            </default>
          </switch>
        </case>
      </fault>
    </context>
  </process>
  </context>
</process>
```

Appendix C: Glossary

Some of the terminology definitions listed here will differ from the definitions listed in the Terminology section of the BPML specification (see BPML). Some of the terms that are used in both specifications, but have different definitions are: activity, atomic activity, flow, and process.

A

Activity (BPML):	(from the BPML 1.0 specification[1]) 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*.
Activity (BPMN):	An activity is a generic term for work that company performs via business processes. An activity can be atomic or non-atomic (compound). The types of activities that are a part of a Process Model are: Process, Sub-Process, and Task.
AND-Join:	(from the WfMC Glossary[2]) An AND-Join is a point in the Process where two or more parallel executing activities converge into a single common thread of Sequence Flow. See "Join."
AND-Split:	(from the WfMC Glossary[3]) An AND-Split is a point in the Process where a single thread of Sequence Flow splits into two or more threads which are executed in parallel within the Process, allowing multiple activities to be executed simultaneously. See "Fork."
Artifact:	An artifact is a graphical object that provides supporting information about the Process or elements within the Process. However, it does not directly affect the flow of the Process. BPMN has standardized the shape of a Data Object. Other examples of artifacts include critical success factors and milestones.
Association:	An Association is a dotted graphical line that is used to associate information and artifacts with flow objects. Text and graphical non-flow objects can be associated with the flow objects and flows.

1. Some terms were italicized based on the current specification's typographical conventions
2. The underlined terms in this definition were changed from the original definition. "Process" is used in place of "workflow." "Sequence Flow" is used in place of "control."
3. See previous footnote.

Atomic Activity (BPML):	(from the BPML 1.0 specification[1]) 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.
Atomic Activity (BPMN):	An atomic activity is an activity not broken down to a finer level of Process Model detail. It is a leaf in the tree-structure hierarchy of Process activities. Graphically it will appear as a Task in BPMN.

C

Choreography:	Choreography is an ordered sequence of B2B message exchanges.
Collaboration:	Collaboration is the act of sending messages between any two Participants in a BPMN model. The two Participants represent two separate BPML processes.
Collaboration Process:	A Collaboration Process depicts the interactions between two or more business entities.
Collapsed Sub-Process:	A Collapsed Sub-Process is a Sub-Process that hides its flow details. The Collapsed Sub-Process object uses a marker to distinguish it as a Sub-Process, rather than a Task. The marker is a small square with a plus sign (+) inside.
Complex Activity (BPML):	(from the BPML 1.0 specification[2]) 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.
Compound Activity (BPMN):	A compound activity is an activity that has detail that is defined as a flow of other activities. It is a branch (or trunk) in the tree-structure hierarchy of Process activities. Graphically, it will appear as a Process or Sub-Process in BPMN.

1. Some terms were italicized based on the current specification’s typographical conventions
2. Some terms were italicized based on the current specification’s typographical conventions

D

Decision:	Decisions are locations within a business process where the Sequence Flow can take two or more alternative paths. This is basically the “fork in the road” for a process. For a given performance (or instance) of the process, only one of the forks can be taken. A Decision is a diamond. See “Or-Split.”

E

End Event:	As the name implies, the End Event indicates where a process will end. In terms of sequence flow, the End Event ends the flow of the Process, and thus, will not have any outgoing Sequence Flows. An End Event can have a specific Result that will appear as a marker within the center of the End Event shape. End Event Results are Message, Process Error, Compensate, Link, and Multiple. The End Event shares the same basic shape of the Start Event and Intermediate Event, a circle, but is drawn with a thick single line
Event Context:	An Event Context is the set of activities that can be interrupted by an exception (Intermediate Event). This can be one activity or a group of activities in an expanded Sub-Process.
Exception:	An Exception is an event that occurs during the performance of the process that causes normal flow of the process to be diverted or stopped. Exceptions can be generated by a time out, fault, message, etc.
Exception Flow:	Exception Flow is a set of Sequence Flow that originates from an Intermediate Event that is attached to the boundary of an activity. The Process will not traverse this flow unless an Exception occurs during the performance of that activity (through an Intermediate Event).
Expanded Sub-Process:	An Expanded Sub-Process is a Sub-Process that exposes its flow detail within the context of its Parent Process. It will maintain its rounded rectangle shape, but will be enlarged to a size sufficient to display the flow objects within.

F

Fault (BPML):	(from the BPML 1.0 specification[1]) A Fault is an error that occurs while executing an *activity*. Specifically, it is an error that occurs while executing an operation. See "Process Error."
Flow (BPML):	(from the BPML 1.0 specification[2]) A *flow* is the series of executing *activities* resulting from the execution of an *activity set*.
Flow (BPMN):	A Flow is a graphical line connecting two objects in a BPD. There are two types of Flow: Sequence Flow and Message Flow, each with their own line style.
Flow Object:	A flow object is one of the set of following graphical objects: Start Event, Task, Sub-Process, Intermediate Event, Decision, and End Event.
Fork:	A fork is a point in the Process where a single flow is divided into two or more flows. It is a mechanism that will allow activities to be performed concurrently, rather than serially. BPMN does not have a specific graphical object to represent a fork. See "AND-Split."
Interface Process:	An interface process represents the interactions between a private business process and another process or participant.
Intermediate Event:	An Intermediate Event is an event that occurs after a Process has been started. It will affect the flow of the process, but will not start or (directly) terminate the process. An Intermediate Event will show where messages or delays are expected within the Process, disrupt the normal flow through exception handling, or show the extra flow required for compensating a transaction. The Intermediate Event shares the same basic shape of the Start Event and End Event, a circle, but is drawn with a thin double line.

J

Join:	A Join is a point in the Process where two or more parallel Sequence Flows are combined into one Sequence Flow. BPMN does not have a specific graphical object to represent a fork. See "AND-Join."

1. Some terms were italicized based on the current specification's typographical conventions
2. Some terms were italicized based on the current specification's typographical conventions

L

Lane:	An Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally. Lanes are used to organize and categorize activities within a Pool. The meaning of the Lanes is up to the modeler.

M

Merge:	A Merge is a point in the process where two or more alternative Sequence Flows are combined into one Sequence Flow. BPMN does not have a specific graphical object to represent a fork. See “OR-Join.”
Message:	A Message is the object that is transmitted through a Message Flow. The Message will have an identity that can be used for alternative branching of a Process through the Event-Based Exclusive Decision.
Message Flow:	A Message Flow is a dashed line that is used to show the flow of messages between two entities that are prepared to send and receive them. In BPMN, two separate Pools in the diagram will represent the two entities.

N

Normal Flow:	Normal Flow is the flow that originates from a Start Event and continues through activities via alternative and parallel paths until it ends at an End Event.

O

OR-Join:	(from the WfMC Glossary[1]) An Or-Join is a point in the Process where two or more alternative activity(s) Process branches re-converge to a single common activity as the next step within the Process. (As no parallel activity execution has occurred at the join point, no synchronization is required.) See “Merge.”
OR-Split:	(from the WfMC Glossary[2]) An OR-Split is a point in the Process where a single thread of Sequence Flow makes a decision upon which branch to take when encountered with multiple alternative Process branches. See “Decision.”

1. The underlined terms in this definition were changed from the original definition. “Process” is used in place of “workflow.” “Sequence Flow” is used in place of “control.”
2. See previous footnote

P

Parent Process:	A Parent Process is the Process that holds a Sub-Process within its boundaries.
Participant:	A Participant is a business entity, usually a company, company division, or a customer, which controls or is responsible for a business process. If Pools are used, then a Participant would be associated with one Pool.
Pool:	A Pool is a “swimlane” and a graphical container for partitioning a set of activities from other Pools, usually in the context of B2B situations. It is a square-cornered rectangle that is drawn with a solid single line. A Pool acts as the container for the Sequence Flow between activities. The Sequence Flow can cross the boundaries between Lanes of a Pool, but cannot cross the boundaries of a Pool. The interaction between Pools, e.g., in a B2B context, is shown through Message Flows.
Private Business Process:	A private business process is internal to a specific organization and is the type of process that has been generally called a workflow or BPM process. A single private business process will map to a single BPML document.
Process (BPML):	(from the BPML 1.0 specification[1]) A *process* is a progressively continuing procedure consists of a series of controlled *activities* that are systematically directed toward a particular result or end.
Process (BPMN):	A Process is any activity performed within a company or organization. In BPMN a Process is depicted as a network of flow objects, which are a set of other activities and the controls that sequence them.

R

Result:	A Result is consequence of reaching an End Event. Results can be of different types, including: Message, Process Error, Compensate, Link, and Multiple.

S

Sequence Flow:	A Sequence Flow is a solid graphical line that is used to show the order that activities will be performed in a Process. Each Flow has only one source and only one target.

1. Some terms were italicized based on the current specification’s typographical conventions

Start Event:	A Start Event indicates where a particular Process will start. In terms of sequence flow, the Start Event starts the flow of the Process, and thus, will not have any incoming Sequence Flows. A Start Event can have a Trigger that indicates how the Process starts: Message, Timer, Rule, Link, or Multiple. The Start Event shares the same basic shape of the Intermediate Event and End Event, a circle, but is drawn with a single thin line
Sub-Process:	A Sub-Process is Process that is included within another Process. The Sub-Process can be in a collapsed view that hides its details. A Sub-Process can be in an expanded view that shows its details within the view of the Process in which it is contained. A Sub-Process shares the same shape as the Task, which is a rectangle that has rounded corners.
Swimlane:	A swimlane is a graphical container for partitioning a set of activities from other activities. BPMN has two different types of swimlanes. See “Pool” and “Lane.”

T

Task:	A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail. Generally, an end-user and/or an application are used to perform the Task when it is executed. A Task object shares the same shape as the Sub-Process, which is a rectangle that has rounded corners.
Token:	A Token is a descriptive construct used to describe how the flow of a process will proceed at runtime. By tracking how the Token traverses the flow objects, gets diverted through alternative paths, and gets split into parallel paths, the normal sequence flow should be completely definable.A Token will have a unique identity that can be used to separate multiple Tokens that may exist because of concurrent process instances or the splitting of the Token for parallel processing within a single process instance.
Transaction (BPML):	(from the BPML 1.0 specification[1]) 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*.

1. Some terms were italicized based on the current specification’s typographical conventions

Transaction (BPMN):	A Transaction is one or more activities that perform work that cannot be undone with a simple rollback. A separate set of activities will be required to compensate for the work done within the Transaction. An Intermediate Event and Transaction Flow define the compensation activities.
Transaction Flow:	Transaction Flow is defines the set of activities that are performed during the roll-back of a transaction to compensate for activities that were performed during the normal flow of the Process.
Trigger:	A Trigger is a mechanism that signals the start of a business process. Triggers are associated with a Start Events and Intermediate Events and can be of the type: Message, Timer, Rule, Link, and Multiple.