6. Mapping to BPEL4WS

This section will cover the mappings to BPEL4WS that are derived by analyzing the elements and the relationships between the elements described in the above sections.

6.1 Business Process Diagram Mappings

A Business Process Diagram can be made up of a set of (semi-) independent components, which are shown as separate Pools. Thus, there is not a specific mapping to the diagram itself. Rather, there are separate mappings to each of the Pools that are in the diagram. That is, each Pool in the diagram, if it is a “white box” that contains process elements, will map to an individual BPEL4WS *process*. However, in the course of mapping the contents of the Process, there may be one or more derived *processes* necessary to handle complex behavior, such as looping. The attributes of “black box” Pools will also be used in determining specific BPEL4WS elements, such as *partnerLink*.

The following table displays a set of mappings for the attributes of a Business Process Diagram that can be mapped to BPEL4WS:

Business Process Diagram	Mapping to BPEL4WS
Id, Name, **Version, Author, Language, CreationDate, ModificationDate,** Pool, and Documentation	These Elements do not map to any BPEL4WS elements or attributes.
ExpressionLanguage attribute	This attribute will be used for all the Processes that are within the Business Process Diagram. The attribute will map to the *expressionLanguage* attribute of each BPEL4WS *process*.
QueryLanguage attribute	This attribute will be used for all the Processes that are within the Business Process Diagram. The attribute will map to the *queryLanguage* attribute of each BPEL4WS *process*.

Table 44 Business Process Diagram Mappings to BPEL4WS

6.2 Business Process Mappings

There can be one or more Business Processes within a Business Process Diagram, each within a separate Pool. The following table displays a set of mappings from attributes of a Process to BPEL4WS elements (the mappings for the objects contained within a Process, its contents, are mapped separately and these mappings can be found in the sections that follow):

Process	Mapping to BPEL4WS
ProcessType	If the Process is to be used to create a BPEL4WS document, then the attribute MUST be set to Private or Abstract. If the attribute is set to Private, then the *abstractProcess* attribute of the BPEL4WS *process* MUST be set to "no." If the attribute is set to Abstract, then the *abstractProcess* attribute of the BPEL4WS *process* MUST be set to "yes."
Id, Category, and Documentation	These Elements do not map to any BPEL4WS elements or attributes.
Name	The Name attribute of the Process SHALL map to *name* attribute of the appropriate *process*. The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *name* attribute. Note that there may be two or more elements with the same name after the BPMN name has been stripped.
GraphicalElements	This is a list of all the graphical elements contained within the Process. Each of these elements will have their mapping, as defined in the sections below.

Properties	The set of Properties of a Process, as a whole, will map to a BPEL4WS *variable*. The *variable* element will be structured as follows:
	`<variable name="[Process.Name]_Data"`
	`messageType="[Process.Name]_ProcessDataMessage" />`
	The individual Properties will map to the *parts* of a WSDL *message*. The *message* element will be structured as follows:
	`<message name="[Process.Name]_ProcessDataMessage" >`
	`<part name="[Property.Name]"`
	`type="xsd:[Property.Type]" />`
	`</message>`
	There will be as many *parts* to the *message* as there are Properties in the input group.

Correlation = True	This only applies to Properties of Type = "Set." The Name of the Property will map to the name of a *correlationSet*. The Name of each child Property for the Set wil be added to the list of *properties* of the *correlationSet*.
Adhoc	Ad Hoc Processes are not executable. Thus, this attribute MUST be set to False if the Process is to be mapped to BPEL4WS.
AdHocCompletionCondition	This attribute only applies to Ad Hoc Processes. Thus, it will not be mapped to BPEL4WS.
With Assign Expression	This will map to a BPEL4WS *assign*. Refer to the section entitled "Property Assignments" on page 203 for more details about the mappings associated with the *assign* element.
AssignTime = Start	A BPEL4WS *sequence* will be created and the *assign* will follow the instantiation of the process (through a *receive* or a *pick*).

Process	Mapping to BPEL4WS
AssignTime = End	A BPEL4WS sequence will be created and the assign will follow
SuppressJoinFailure	This maps to the BPEL4WS *process* attribute *suppressJoinFailure*.
EnableInstanceCompensation	This maps to the BPEL4WS *process* attribute *enableInstanceCompensation*.

Table 45 Business Process Mappings to BPEL4WS

- The BPEL4WS *process* attributes *targetNamespace* and *xmlns* MUST be provided by the modeling tool that generates the mapping to BPEL4WS.

6.3 Common Object Mappings

The following table displays a set of mappings for the attributes common to Events, Activities, and Gateways:

Objects	Mapping to BPEL4WS
Id, Pool, Lane, Category, and Documentation	These Elements do not map to any BPEL4WS elements or attributes.
Name	The Name attribute of the object SHALL map to *name* attribute of the appropriate derived BPEL4WS element (as per mappings described in the sections below). The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *name* attribute. Note that there may be two or more elements with the same name after the BPMN name has been stripped.
Assign	Each Assign Expression will map to a BPEL4WS *assign* activity. Refer to the section entitled “Property Assignments” on page 203 for more details about the mappings associated with the *assign* element.

Table 46 Common Object Attribute Mappings to BPEL4WS

6.4 Events

6.4.1 Start Event Mappings

The following table displays a set of mappings from the variations of a Start Event to BPEL4WS elements (these mappings extend the mappings common to objects--refer to the section entitled “Common Object Mappings” on page 153):

Start Event	Mapping to BPEL4WS
EventType = Start and Trigger	The mapping to BPEL4WS is specific to the Trigger setting. These mappings are defined in the rows below.
None	There is no BPEL4WS element that a Start Event will map to with a Trigger that is None. The object(s) that are the Target(s) of Sequence Flow that originate from the Start Event will determine the first BPEL4WS element of the Process. Note that a valid BPEL4WS *process* must begin with a *receive* or a *pick* activity that has a *createInstance* set to “yes.” The *receive* or *pick* will likely be placed within a *sequence* or a *flow*.
Message	This will map to the *receive* element. The *createInstance* attribute of the *receive* element will be set to “yes.”
Message	The Message attribute maps to the *variable* attribute of the *receive* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
Implementation = Web Service	The Implementation attribute MUST be a Web service or MUST be converted to a Web Service for mapping to BPEL4WS. The Web Service Attributes are mapped as follows: The Entity attribute is mapped to the *partnerLink* attribute of the BPEL4WS activity. The Interface attribute is mapped to the *portType* attribute of the BPEL4WS activity. The Operation attribute is mapped to the *operation* attribute of the BPEL4WS activity.
Timer	This will map to the *receive* element. The *createInstance* attribute of the *receive* element will be set to “yes.” The remaining attributes of the *receive* will be mapped as shown for the Message Start Event (see above). The functionality of the timing as defined in the Start Event must be implemented in a separate process that will start itself, then use a *wait* element for the defined time, and then use an *invoke* to send a message that will be received by the above *receive* element. A specific Message and Web service implementation must be provided so that the mappings to *receive* element can be completed.
Rule	This will map to the *receive* element. The *createInstance* attribute of the *receive* element will be set to “yes.” The remaining attributes of the *receive* will be mapped as shown for the Message Start Event (see above). The functionality of the timing as defined in the Start Event must be implemented in a separate process that will start itself, then use a *wait* element for the defined time, and then use an *invoke* to send a message that will be received by the above *receive* element. A specific Message and Web service implementation must be provided so that the mappings to *receive* element can be completed.
Link	This will map to the *receive* element. The *createInstance* attribute of the *receive* element will be set to “yes.” The remaining attributes of the *receive* will be mapped as shown for the Message Start Event (see above). A specific Message and Web service implementation must be provided so that the mappings to *receive* element can be completed.

Start Event	Mapping to BPEL4WS
Multiple	This will map to a BPEL4WS *pick* will be required to process the messages with a separate *onMessage* for each defined Trigger. The *createInstance* attribute of the *pick* element will be set to “yes.” This means that a single instance of the process will be instantiated when the first message received through the *pick onMessage* is triggered. The onMessage mappings are the same as that of a receive and as defined for the Message Start Event (see above).
With Assign Expression	Each Assign Expression will map to a BPEL4WS *assign* that will follow the *receive*. Refer to the section entitled “Property Assignments” on page 203 for more details about the mappings associated with the *assign* element.

Table 47 Start Event Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The Timer and Rule Trigger mappings have been defined.
- The definition of the Link Trigger mapping was expanded.
- The mapping for the Multiple Trigger was changed to be a BPEL4WS *pick* element.
- The part of the definition of the Message Trigger that described the mapping if there were multiple incoming Message Flows was removed. This was due that only Multiple Triggers can have multiple incoming Message Flows.

6.4.2 End Event Mappings

The following table displays a set of mappings from the variations of a End Event to BPEL4WS elements (these mappings extend the mappings common to objects--refer to the section entitled “Common Object Mappings” on page 153):

End Event	Mapping to BPEL4WS
EventType = End and Result	The mapping to BPEL4WS is specific to the Result setting. These mappings are defined in the rows below.
None	There is no BPEL4WS element that a End Event will map to with a Result that is None. However, it marks the end of a path within the Process and will be used to define the boundaries of complex BPEL4WS elements. The object(s) that are the Source(s) of Sequence Flow that Target the End Event will determine the final BPEL4WS elements of the Process.
Message	This will map to a BPEL4WS *reply* or an *invoke*. The appropriate BPEL4WS activity will be determined by the implementation defined for the Event. That is, the *portType* and *operation* of the Message will be used to check to see if an upstream Message Event have the same *portType* and *operation*. If these two attributes are matched, then the Event will map to a *reply*, if not, the Event will map to an *invoke*.
Message	The Message attribute maps to the *variable* attribute of the *reply* or the *outputVariable* of the *invoke*. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
Implementation = Web Service	The Implementation attribute MUST be a Web service or MUST be converted to a Web Service for mapping to BPEL4WS. The Web Service Attributes are mapped as follows: The Entity attribute is mapped to the *partnerLink* attribute of the BPEL4WS activity. The Interface attribute is mapped to the *portType* attribute of the BPEL4WS activity. The Operation attribute is mapped to the *operation* attribute of the BPEL4WS activity.
Exception	This will map to a *throw* element. The ExceptionCode attribute of the Event will map to the *faultName* attribute of the *throw*.
Cancel	The mapping of the Cancel Intermediate Event to BPEL4WS is an open issue. Refer to the section entitled “Open Issues” on page 245 for other Open Issues.
Compensation	This will map to a *compensate* element. The Name of the activity referenced by the Compensation Event will map to the *scope* attribute of the *compensate* element.
Link	This will map to a (one-way) *invoke* element.
LinkId	The LinkId attribute maps to the *outputVariable* of the *invoke*. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
ProcessRef	The The Implementation attribute MUST be a Web service or MUST be converted to a Web Service for mapping to BPEL4WS. The Web Service Attributes are mapped as follows: The Entity attribute of the Pool where the Process is contained is mapped to the *partnerLink* attribute of the BPEL4WS activity. The Name attribute of the Process is mapped to the *portType* attribute of the BPEL4WS activity. The LinkId attribute is mapped to the *operation* attribute of the BPEL4WS activity.
Terminate	This will map to the *terminate* element.

End Event	Mapping to BPEL4WS
Multiple	This will map to a this will map to a combination of *invoke*, *throw*, *fault*, and *compensation* elements as they are defined above.
With Assign Expression	This will map to a BPEL4WS *assign* that will precede any other mappings required by teh Event. Refer to the section entitled “Property Assignments” on page 203 for more details about the mappings associated with the *assign* element.

Table 48 End Event Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The mapping for the Message, Exception, Compensation, and Link End Events was updated and expanded.
- The mapping to the Return End Event was removed, since that type of Event has been removed.

6.4.3 Intermediate Event Mappings

The following table displays a set of mappings from the variations of a Intermediate Event to BPEL4WS elements (these mappings extend the mappings common to objects--refer to the section entitled “Common Object Mappings” on page 153):

Intermediate Event	Mapping to BPEL4WS
EventType = Intermediate and Trigger	The mapping to BPEL4WS is specific to the Trigger setting. These mappings are defined in the sections below.
With Assign Expression	this will map to a BPEL4WS *assign*. Refer to the section entitled “Property Assignments” on page 203 for more details about the mappings associated with the *assign* element.

Table 49 Intermediate Event Mappings to BPEL4WS

None Intermediate Events

The mappings for None Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled “Intermediate Event Mappings” on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = None	There is no BPEL4WS element that a End Event will map to with a Result that is None. However, it marks the end of a path within the Process and will be used to define the boundaries of complex BPEL4WS elements. The object(s) that are the Source(s) of Sequence Flow that Target the End Event will determine the final BPEL4WS elements of the Process.

Table 50 None Intermediate Mappings to BPEL4WS

Message Intermediate Events

The mappings for Message Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled “Intermediate Event Mappings” on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Message	This mapping is defined in the next five (5) rows.
Within the normal flow	If the Entity defined in the To attribute of the Message is the same Entity as that of the Process that contains the Event, then this will map to a *receive*. The *createInstance* attribute of the *receive* element will be set to “no.” If the Entity defined in the From attribute of the Message is the same Entity as that of the Process that contains the Event, then this will map to an (one-way) *invoke*.
Message	The Message attribute maps to the *variable* attribute of the *reply* or the *outputVariable* of the *invoke*. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
Implementation = Web Service	The Implementation attribute MUST be a Web service or MUST be converted to a Web Service for mapping to BPEL4WS. The Web Service Attributes are mapped as follows: The Entity attribute is mapped to the *partnerLink* attribute of the BPEL4WS activity. The Interface attribute is mapped to the *portType* attribute of the BPEL4WS activity. The Operation attribute is mapped to the *operation* attribute of the BPEL4WS activity.
Attached to an Activity Boundary	The mappings of the activity (to which the Event is attached) will be placed within a *scope*. A *faultHandler*s element will be defined for the *scope*. A *catch* element will be added to the *faultHandlers* element with “<message name>_Exit” as the *faultName* attribute. An *eventHandler*s element will be defined for the *scope*. The Event will map to an *onMessage* element within the *eventHandler*s. The mapping to the *onMessage* attributes is the same as described for the *receive* above. The activity for the *onMessage* will be a *throw* with “<message name>_Exit” as the *faultName* attribute.
Used in an Event-Based Decision	This will map to an *onMessage* within a *pick*.The mapping to the *onMessage* attributes is the same as described for the *receive* above.

Table 51 Message Intermediate Mappings to BPEL4WS

Timer Intermediate Events

The mappings for Timer Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled “Intermediate Event Mappings” on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Timer	This mapping is defined in the next three (3) rows.
Within the normal flow	This will map to a *wait*. The TimeDate attribute maps to the *until* attribute of the *wait*. The TimeCycle attribute maps to the *for* attribute of the *wait*.

Intermediate Event	Mapping to BPEL4WS
Attached to an Activity Boundary	The mappings of the activity (to which the Event is attached) will be placed within a *scope*. A *faultHandler*s element will be defined for the *scope*. A *catch* element will be added to the *faultHandlers* element with "<Event name>_Exit" as the *faultName* attribute. An *eventHandler*s element will be defined for the *scope*. The Event will map to an *onAlarm* element within the *eventHandler*s. The TimeDate attribute maps to the *until* attribute of the *onAlarm*. The TimeCycle attribute maps to the *for* attribute of the *onAlarm*. The activity for the *onAlarm* will be a *throw* with "<message name>_Exit" as the *faultName* attribute.
Used in an Event-Based Decision	This will map to an *onAlarm* within a *pick*. The TimeDate attribute maps to the *until* attribute of the *onAlarm*. The TimeCycle attribute maps to the *for* attribute of the *onAlarm*.

Table 52 Timer Intermediate Mappings to BPEL4WS

Exception Intermediate Events

The mappings for Exception Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled "Intermediate Event Mappings" on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Exception	This mapping is defined in the next two (2) rows.
Within the normal flow	This will map to a *throw* element.
Attached to an Activity Boundary	The mappings of the activity (to which the Event is attached) will be placed within a *scope*. This Event will map to a *catch* element within a *scope*. If the Exception Event does not have an ExceptionCode, then a *catchAll* element will be added to the *faultHandlers* element. If the Exception Event does has an ExceptionCode, then a *catch* element will be added to the *faultHandlers* element with the ExceptionCode mapping to the *faultName* attribute.

Table 53 Exception Intermediate Mappings to BPEL4WS

Cancel Intermediate Events

The mappings for Cancel Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled "Intermediate Event Mappings" on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Cancel	The mapping of the Cancel Intermediate Event to BPEL4WS is an open issue. Refer to the section entitled "Open Issues" on page 245 for other Open Issues.

Table 54 Cancel Intermediate Mappings to BPEL4WS

Rule Intermediate Events

The mappings for Rule Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled “Intermediate Event Mappings” on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Rule	This mapping is defined in the next two (2) rows.
Within the normal flow	This will map to the *receive* element. The *createInstance* attribute of the *receive* element will be set to “no.” The remaining attributes of the *receive* will be mapped as shown for the Message Start Event (see above).
Attached to an Activity Boundary	The activity (to which the Event is attached) will be placed within a *scope*. This will map to an *onMessage* element within a *scope*. The mapping to the *onMessage* attributes is the same as described for the *receive* above.
Attached to an Activity Boundary	The mappings of the activity (to which the Event is attached) will be placed within a *scope*. A *faultHandler*s element will be defined for the *scope*. A *catch* element will be added to the *faultHandlers* element with “<message name>_Exit” as the *faultName* attribute. An *eventHandler*s element will be defined for the *scope*. The Event will map to an *onMessage* element within the *eventHandler*s. The mapping to the *onMessage* attributes is the same as described for the *receive* for the Message Event above. The activity for the *onMessage* will be a *throw* with “<message name>_Exit” as the *faultName* attribute.
Used in an Event-Based Decision	This will map to an *onMessage* element within a *pick*. The mapping to the *onMessage* attributes is the same as described for the *receive* for the Message Event above.

Table 55 Rule Intermediate Mappings to BPEL4WS

Compensation Intermediate Events

The mappings for Compensation Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled “Intermediate Event Mappings” on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Compensation	This mapping is defined in the next two (2) rows.
Within the normal flow	This will map to a *compensate* element.The Name of the activity referenced by the Compensation Event will map to the *scope* attribute of the *compensate* element.
Attached to an Activity Boundary	The activity (to which the Event is attached) will be placed within a *scope*. This Event map to an *compensationHandler* element within a *scope*.

Table 56 Compensation Intermediate Mappings to BPEL4WS

Link Intermediate Events

The mappings for Link Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled “Intermediate Event Mappings” on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Link	This mapping is defined in the next four (4) rows.
With an outgoing Sequence Flow	This will map to a *receive*. The *createInstance* attribute of the *receive* element will be set to “no.” The mapping to the *receive* attributes is the same as described for the *receive* for the Message Event above.
With an incoming Sequence Flow	This will map to a (one-way) *invoke* element. The mapping to the *onMessage* attributes is the same as described for the *invoke* for the Message Event above.
Attached to an Activity Boundary	The mappings of the activity (to which the Event is attached) will be placed within a *scope*. A *faultHandler*s element will be defined for the *scope*. A *catch* element will be added to the *faultHandlers* element with “<message name>_Exit” as the *faultName* attribute. An *eventHandler*s element will be defined for the *scope*. The Event will map to an *onMessage* element within the *eventHandler*s. The mapping to the *onMessage* attributes is the same as described for the *receive* for the Message Event above. The activity for the *onMessage* will be a *throw* with “<message name>_Exit” as the *faultName* attribute.
Used in an Event-Based Decision	This will map to an *onMessage* element within a *pick*. The mapping to the *onMessage* attributes is the same as described for the *receive* for the Message Event above.

Table 57 Link Intermediate Mappings to BPEL4WS

Multiple Intermediate Events

The mappings for Multiple Intermediate Events are described in the following table (these mappings extend the mappings common to Intermediate Events--refer to the section entitled “Intermediate Event Mappings” on page 157):

Intermediate Event	Mapping to BPEL4WS
Trigger = Mutliple	This will map to a this will map to a combination of the mappings as they are defined in the Intermediate Event sections above.

Table 58 Multiple Intermediate Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The mapping for the Message, Exception, Compensation, Rule, and Link End Events was updated and expanded.

6.5 Activities

6.5.1 Common Activity Mappings

The following table displays a set of mappings from the variations of activities to BPEL4WS elements (these mappings extend the mappings common to objects -- refer to the section entitled “Common Object Mappings” on page 153 -- Note that Table 60 contain additional mappings that must be included within this set if extended by any other mapping table):

Activity	Mapping to BPEL4WS

Properties	The set of Properties of an activity, as a whole, will map to a BPEL4WS *variable*. The *variable* element will be structured as follows:
	`<variable name="[activity.Name]_ActivityData"`
	`messageType="[activity.Name]_ActivityDataMessage" />`
	The individual Properties will map to the *parts* of a WSDL *message*. The *message* element will be structured as follows:
	`<message name="[activity.Name]_ActivityDataMessage" >`
	`<part name="[Property.Name]"`
	`type="xsd:[Property.Type]" />`
	`</message>`
	There will be as many *parts* to the *message* as there are Properties in the input group.

With Assign Expression	This will map to a BPEL4WS *assign*. Refer to the section entitled “Property Assignments” on page 203 for more details about the mappings associated with the *assign* element.
AssignTime = Start	A BPEL4WS sequence will be created and the assign will precede
AssignTime = End	A BPEL4WS sequence will be created and the assign will follow

Table 59 Common Activity Mappings to BPEL4WS

Activity Loop Mapping

The mapping to BPEL4WS for looping activities is complex and is made up of a number of activities that will surround the original mapping of the activity itself (which may be complex). The description of this mapping is divided into three sections to describe the basic setup of the loop (common to all loops), then the details of Standard looping, then the details of Multi-Instance looping.

Basic Loop Setup

The basic set up mappings, which are common to both Standard and Multi-Instance looping activities, are described in the following table (these mappings extend the mappings common to objects--refer to the section entitled “Common Activity Mappings” on page 162):

Looping	Mapping to BPEL4WS
Activities with internal looping	Activities that have either a Standard or MultiInstance loop setting will result in a pattern of BPEL4WS elements, depending on the exact settings. This pattern will be placed within a BPEL4WS *sequence* activity. The details of the other mappings are described in the rows that follow.

Looping	Mapping to BPEL4WS
LoopCounter	This attribute will map to a BPEL4WS *variable*, which will be part of the *process* definition. The variable will be structured as follows: `<variable name="[activity.Name]_loopCounter" messageType="loopCounterMessage" />` *Note: The LoopCounter mappings described in the this and the next three rows are only required for Multi-Instance loops and Standards loops that use the LoopMaximum attribute. For all looping activities, the LoopCounter can be used for reporting purposes.*

Supporting WSDL Message	A WSDL *message* element will have to be created to support this *variable*. This *message* can be used for multiple *variables*. The *message* will be structured as follows:
	`<message name="loopCounterMessage" >`
	`  <part name="loopCounter" type="xsd:integer" />`
	`</message>`

Initialization of the LoopCounter	An *assign* activity will be created to initialize the *variable* before the start of the loop. This activity precede the *while* activity. This will be the first activity within the *sequence* activity. The *assign* will be structured as follows:
	`<assign name="[activity.Name]_initialize_loopCounter">`
	`  <copy>`
	`  <from expression="0"/>`
	`  <to variable="[activity.Name]_loopCounter" part="loopCounter" />`
	`  </copy>`
	`</assign>`

Incrementing the LoopCounter	An *assign* activity will be created to update the loopCounter *variable* at the end of the *while* activity (see below). This activity will the last activity of the *sequence* activity that is within the *while* activity. The *assign* will be structured as follows:
	`<assign name="[activity.Name]_increment_loopCounter">`
	`  <copy>`
	`  <from expression="bpws:getVariableData([activity.Name]_loopCounter, loopCount) + 1"/>`
	`  <to variable="[activity.Name]_loopCounter" part="loopCounter" />`
	`  </copy>`
	`</assign>`

Table 60 Basic Activity Loop Mappings to BPEL4WS

Standard Loops

The loop mappings for Standard loops are described in the following table (these mappings extend the mappings of the Basic Loop Setup--refer to the previous section):

Looping	Mapping to BPEL4WS
LoopType = Standard	For a Standard Looping activity, the mapping of the base BPMN activity will be placed within a BPEL4WS *sequence* that is within a *while*, and this will follow the *assign* described in the Basic Loop Setup (see Figure 101 and Example 1). Refer to the section entitled “Sub-Process Mappings” on page 177 or the section entitled “Task Mappings” on page 179 for details about how the base activity will be mapped to BPEL4WS.
LoopCondition	The LoopCondition, which MUST be a boolean expression, will be used as the *condition* attribute of the *while* element.The *while condition* be structured as follows: `<while condition="[loopCondition]">`
TestTime = After	An After TestTime will map to the BPEL4WS *while* activity. However, to insure that the Task is performed at least once (i.e., the functionality of an until loop), a copy of the mapping for BPMN activity will be performed first in a *sequence*, followed by the *while* (which will contain the original copy of the mapping for the BPMN activity).
TestTime = Before	A Before TestTime does not require any additional mappings.
LoopMaximum	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 ([ActivityName].LoopCounter <= 5).”

Table 61 Standard Activity Loop Mappings to BPEL4WS

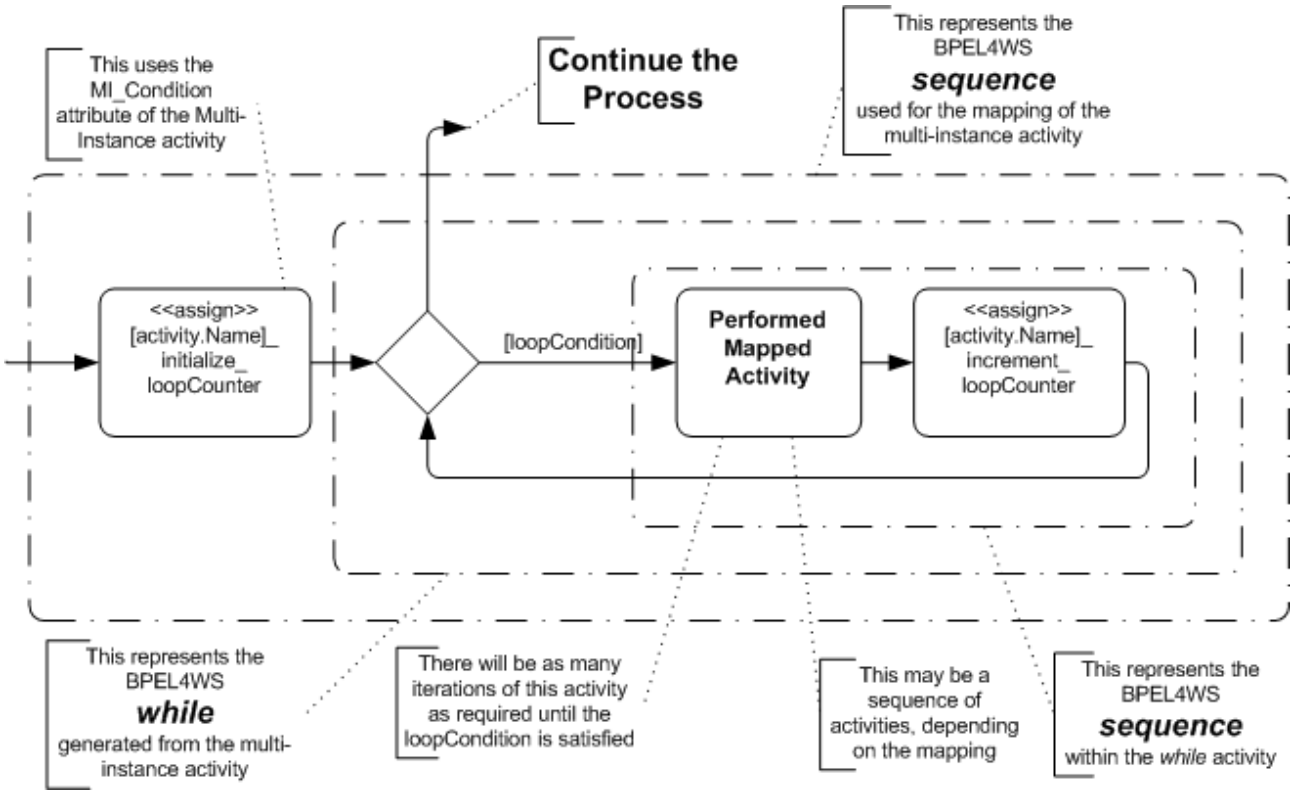

Figure 101 BPMN Depiction of BPEL4WS Pattern for a Standard loop, TestTime = Before

Example 1 displays sample BPEL4WS code that reflects the mapping of a Standard loop.

```
<!-- The Process data is defined first-->
<variable name="[activity.Name]_loopCounter" messageType="loopCounterMessage" />
<!-- The contents of the process prior to the looping activity are here-->
<sequence>
  <assign name="[activity.Name]_initialize_loopCounter">
    <copy>
      <from expression="0"/>
      <to variable="[activity.Name]_loopCounter" part="loopCounter" />
    </copy>
  </assign>
  <!-- If the TestTime is set to After, the mappings of the original activity
       are placed here, as well as within the while.-->
  <while condition="[loopCondition]">
    <sequence>

      <!--The mappings of the original activity are placed here.-->

      <assign name="[activity.Name]_increment_counter">
        <copy>
          <from expression="bpws:getVariableData([activity.Name]_loopCounter,loopCount)+1"/>
          <to variable="[activity.Name]_loopCounter" part="loopCounter" />
        </copy>
      </assign>
    </sequence>
  </while>
</sequence>
<!-- The contents of the process after the looping activity are here-->
```

Example 1 BPEL4WS Sample for a Standard Loop

Multi-Instance Loop Setup

The loop mappings for Multi-Instance loops are described in the following table (these mappings extend the mappings of the Basic Loop Settings--refer to the section entitled “Basic Loop Setup” on page 163):

Multi-Instance	Mapping to BPEL4WS
LoopType = MultiInstance	For a Multi-Instance Looping activity, the mapping of the BPMN activity will be placed within a BPEL4WS *sequence* that is within a *while*, and this will follow the *assign* described in the Basic Loop Setup (see Figure 101 and Example 1). Refer to the section entitled “Sub-Process Mappings” on page 177 or the section entitled “Task Mappings” on page 179 for details about how the base activity will be mapped to BPEL4WS.

MI_Condition	This applies to both Sequential and Parallel MI_Ordering (see below).
	The MI_Condition, which MUST be a numeric expression, will map to an *assign* activity. This will be the first activity of the generated *sequence* activity (as described in the row above).
	First, a BPEL4WS *variable* must be created with a derived name and will have a structure as follows:
	`<variable name="[activity.Name]_forEachCount"`
	`messageType="forEachCounterMessage" />`
	Second, an *assign* activity will be used to generate the number of instances that will be required. The *assign* will be structured as follows:
	`<assign name="[activity.Name]_determine_instances">`
	`<copy>`
	`<from expression="[MI_Condition Exprssion]"/>`
	`<to variable="[activity.Name]_forEachCount"`
	`part="forEachCount" />`
	`</copy>`
	`</assign>`

Supporting WSDL Message	A WSDL *message* element will have to be created to support the *variable*. This *message* can be used for multiple *variables*. The *message* will be structured as follows:
	`<message name="forEachCounterMessage" >`
	`<part name="forEachCount" part="xsd:integer" />`
	`</message>`

The condition for the *while*	The *condition* attribute of the *while* will be a derived expression that utilizes the loopCounter variable and compares it to the derived forEachCount (described in the row above). The *while condition* be structured as follows:
	`<while condition="`
	`bpws:getVariableData([activity.Name]_loopCounter,`
	`loopCounter) >=`
	`bpws:getVariableData([activity.Name]_forEachCount,`
	`forEachCount)">`

Table 62 Multi-Instance Activity Loop Setup Mappings to BPEL4WS

Sequential Multi-Instance Loops

The loop mappings for Sequential Multi-Instance loops are described in the following table (these mappings extend the mappings of the Multi-Instance Setup--refer to the section above):

Multi-Instance	Mapping to BPEL4WS
MI_Ordering = Sequential	This type of looping utilizes both the Basic Loop Setup mappings and the above Multi-Instance mappings. No further mappings are necessary. See Figure 102 and Example 2 for the complete mappings.

Table 63 Sequential Multi-Instance Activity Loop Mappings to BPEL4WS

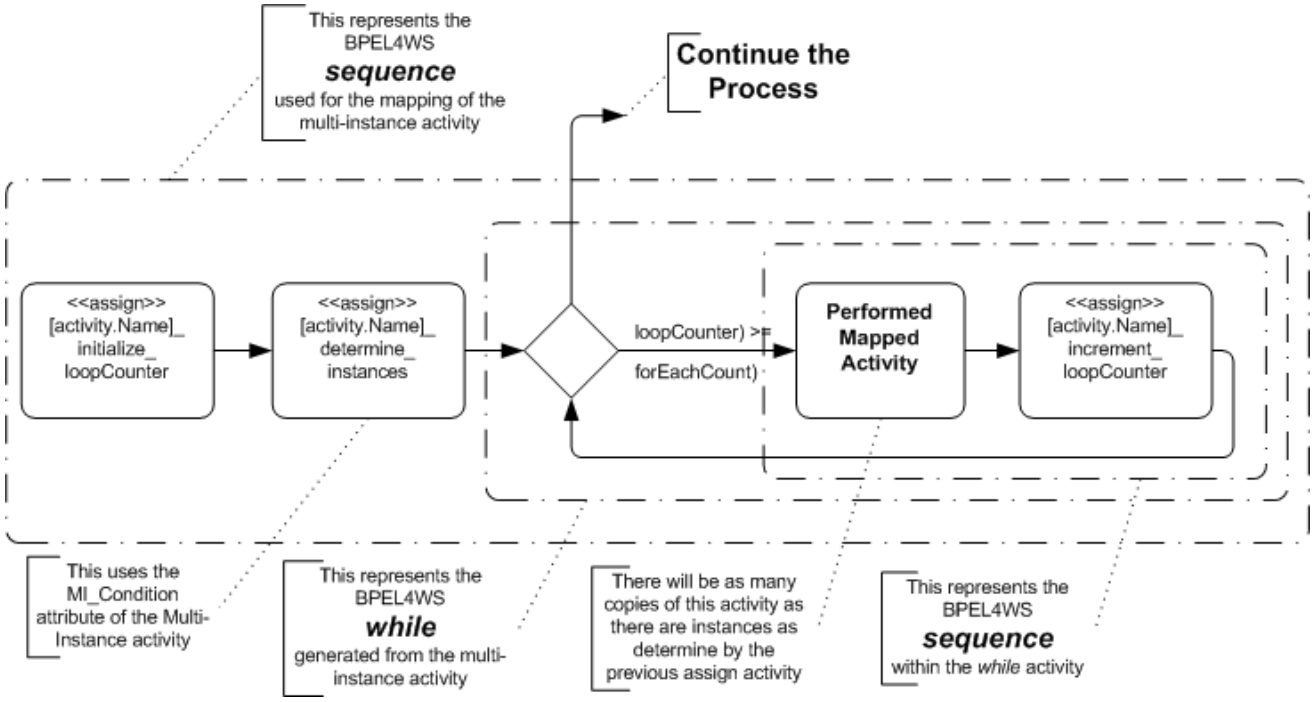

Figure 102 BPMN Depiction of BPEL4WS Pattern for a Sequential Multi-Instance loop

Example 2 displays some sample BPEL4WS code that reflects the mapping of a Standard loop.

```
<!-- The Process data is defined first-->
<variable name="[activity.Name]_loopCounter" messageType="loopCounterMessage" />
<variable name="[activity.Name]_forEachCount" messageType="forEachCounterMessage" />
<!-- The contents of the process prior to the looping activity are here-->
<sequence>
  <assign name="[activity.Name]_initialize_loopCounter">
    <copy>
      <from expression="0"/>
      <to variable="[activity.Name]_loopCounter" part="loopCounter" />
    </copy>
  </assign>
  <assign name="[activity.Name]_determine_instances">
    <copy>
      <from expression="[MI_Condition Exprssion]"/>
      <to variable="[activity.Name]_forEachCount" part="forEachCount" />
    </copy>
  </assign>
  <while condition="bpws:getVariableData([activity.Name]_loopCounter,loopCounter) >=
                    bpws:getVariableData([activity.Name]_forEachCount,forEachCount)">
    <sequence>

      <!--The mappings of the original activity are placed here.-->

      <assign name="[activity.Name]_increment_counter">
        <copy>
          <from expression="bpws:getVariableData([activity.Name]_loopCounter,loopCount)+1"/>
          <to variable="[activity.Name]_loopCounter" part="loopCounter" />
        </copy>
      </assign>
    </sequence>
  </while>
</sequence>
<!-- The contents of the process after the looping activity are here-->
```

Example 2 BPEL4WS Sample for a Multi-Instance Loop with Sequential Ordering

Parallel Multi-Instance Loop Setup

The loop mappings for Sequential Multi-Instance loops are described in the following table (these mappings extend the mappings of the Multi-Instance Setup--refer to the section above):

Multi-Instance	Mapping to BPEL4WS
MI_Ordering = Parallel	A BPEL4WS *while* activity will also be used for Parallel ordering. However, since the Task is to be performed in parallel, the mapping to the Tasks cannot be contained within the *while*. To get the parallel behavior, each copy of the multi-instance Task will be placed into a separate, derived BPEL4WS *process*[1]. A one-way *invoke* will be used to "spawn" each *process* and, thus, each instance of the Task. Since the *invoke* is only one-way, and doesn't wait for a response from the *process*, the *invoke* will complete quickly and the *while* will cycle through all of its iterations quick enough that the instantiations of the Task mappings will be effectively, if not literally, in parallel. The setting for the MI_FlowCondition attribute will determine what BPEL4WS elements will follow the *while* activity. These mappings will be described in the next four sections.
The *while* condition	The *while* condition will be the same as that of the Sequential ordering (see previous section).

Multi-Instance	Mapping to BPEL4WS
Spawning the process	In the *while* activity, a one-way *invoke* activity will be created and used to "spawn" each of the derived *processes*. The *name* attribute for each derived *invoke* will be in the following format: `<invoke name="Spawn_Process_For_[activity.Name]" ... >` This invoke will replace the mappings of the original activity, which was in the while for Standard loops and Sequential Multi-Instance Loops.
The spawned process	The derived *process* will start with a *receive* that accepts the message that is sent by the one-way *invoke* that is within the *while* loop of the original *process*. The name of the process will be `"Spawned_Process_For_[activity.Name]."` The original Task will be mapped and those BPEL4WS elements will follow the initial *receive*. After all the mapped elements have been completed, then a one-way *invoke* will be used to send a message back to the original *process* has a notification that the spawned *process* is completed. This will be the last element of the spawned *process* (see Figure 103 and Example 3). The *name* attribute for the derived *invoke* will be in the following format: `<invoke name="[activity.Name]_Completed" ... >`
Copying variables to/ from the spawned processes	Since the Parallel Multi-Instance Task mappings are going to be performed within the a different process instance, the variables of the original *process* will need to be passed to the spawned *process* through the *inputVariable* of the one-way *invoke* that spawns the *process*. Likewise, any variables that are updated in the spawned *process* will need to be passed back to the original *process* through the *inputVariable* of the one-way *invoke* that indicates that the spawned *process* has completed. *Note: Once the individual derived processes are instantiated, they will be blind to any changes in process variables. From the BPMN point of view, all the multi-instance activities are within the same context as the original Process and, thus, should be able to utilize any dynamic changes to Process Properties (BPEL4WS variables) as they occur (this is especially true for multi-instance Sub-Processes). It is up to the BPEL4WS execution environment to provide a "virtual context" for all the derived processes to "share" the process variables.*
Receiving completion messages	As mentioned above, the spawned *processes* will send a message back to the original *process* after it has completed performing the behavior of the original activity. A BPEL4WS *receive* activity will be used to receive the messages back from all the spawned *processes*. The settings of the MI_FlowCondition will determine The *name* attribute for each derived *receive* will be structured as follows: `<receive name="[activity.Name]_Completed" ... >` The setting of the MI_FlowCondition attribute will determine how many *receive* activities will be required. Once the appropriate number of messages have been received back from the spawned *processes*, the original *process* will continue.

1.Note: BPEL4WS does not have a sub-process capability. It is likely that sub-processes, both embedded and independent, will be added to BPEL4WS in the future. When this capability has been added, the mapping for derived processes will be updated.

Table 64 Parallel Multi-Instance Activity Loop Mappings to BPEL4WS

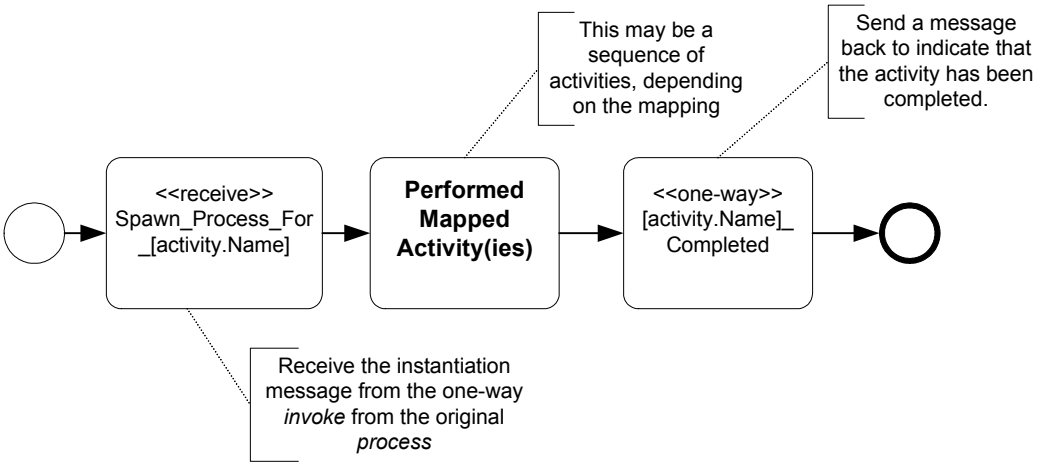

Figure 103 Structure of Process to be Spawned for Parallel Multi-instance

Example 5 displays some sample BPEL4WS code that reflects the mapping of a Multi-Instance loop that has Parallel ordering and must synchronize all the looped activities.

```
<process name="Spawned_Process_For_[activity.Name]" ... >
  <sequence>
    <receive name="Spawn_Process_For_[activity.Name]" ... >

    <!--The mappings of the original activity are placed here.-->

    <invoke name="[activity.Name]_Completed" ... >
  </sequence>
</process>
```

Example 3 BPEL4WS Sample for the derived *process* spawned for Parallel Multi-Instance loops

Parallel Multi-Instance Loops -- Flow Condition All

The loop mappings for Parallel Multi-Instance loops that have a MI_FlowCondition of All are described in the following table (these mappings extend the mappings of the Parallel Multi-Instance Setup--refer to the section above):

Multi-Instance	Mapping to BPEL4WS
MI_FlowCondition = All	This setting utilizes the mechanisms described above for the Parallel ordering. The “All” setting requires that all of the spawned *processes* must be completed before the original *process* can continue (see Figure 104 and Example 4).
Synchronizing the completion of the spawned processes	The synchronization from the spawned *processes* is managed through the messages sent by those *processes* when they have completed the behavior defined by the original activity. These messages will be received by the original *process* and when the messages from all the spawned *processes* are received, then the original *process* can continue. To ensure that all the messages are received, a second *while* activity will used. This while will contain a *receive* activity (for the completion messages) and an *assign* activity to increment the loop counter. The *while condition* attribute will be the same as the condition for the *while* that generated all the spawned *processes*, so that the same number of messages will be received as there were spawned *processes*.

Multi-Instance	Mapping to BPEL4WS
Resetting the loop Counter	Prior to the second *while* activity, another *assign* will be required to reset the loop counter. The contents of the *assign* activity will be the same as the *assign* that originally initialized the loopCounter. The *name* attribute for the derived *assign* will be in the following format: `<assign name="[activity.Name]_reset_loopCounter" ... >`

Table 65 Parallel Multi-Instance Activity, MI_FlowCondition = All, Loop Mappings to BPEL4WS

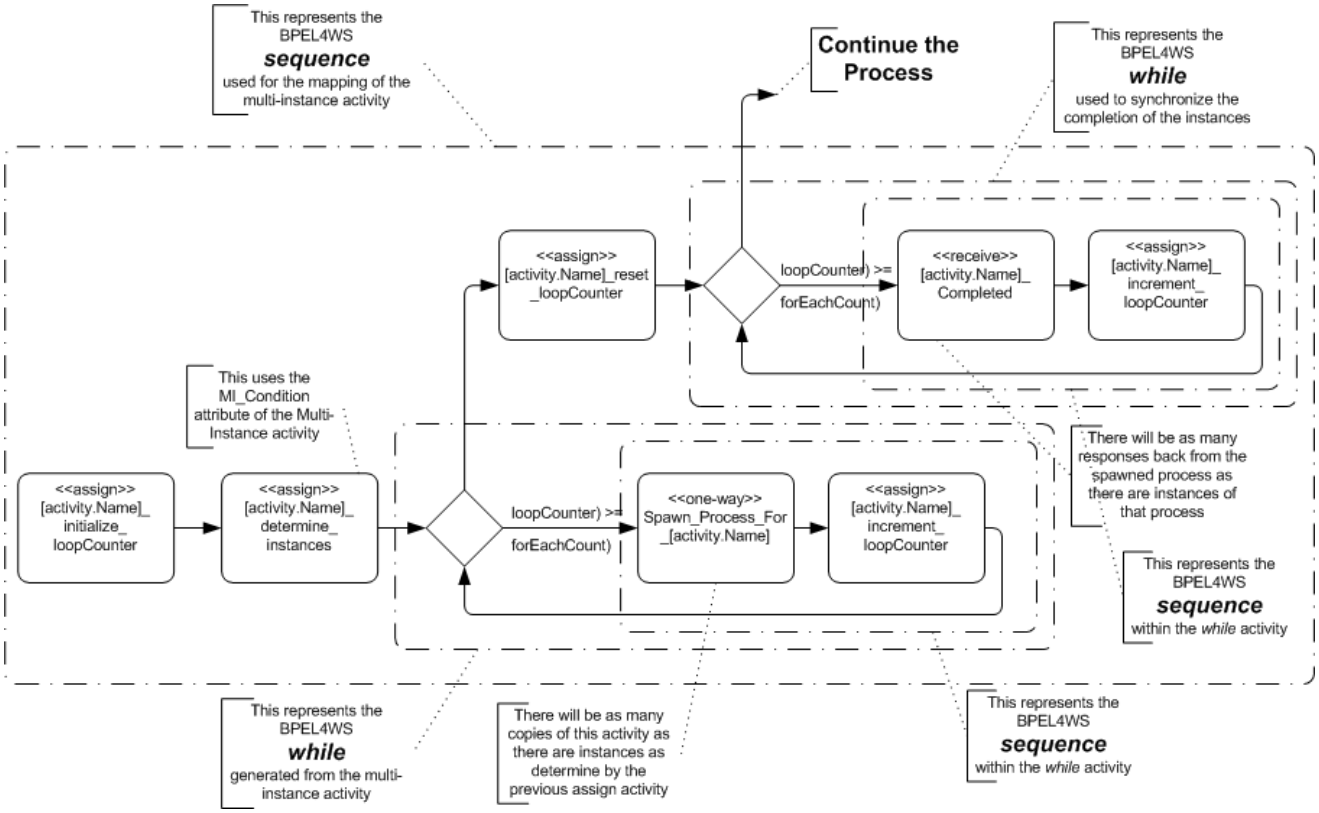

Figure 104 BPMN Depiction of BPEL4WS Pattern for a Parallel Multi-instance MI_FlowCondition = All

```
<!-- The Process data is defined first-->
<variable name="[activity.Name]_loopCounter" messageType="loopCounterMessage" />
<variable name="[activity.Name]_forEachCount" messageType="forEachCounterMessage" />
<!-- The contents of the process prior to the looping activity are here-->
<sequence>
  <assign name="[activity.Name]_initialize_loopCounter">
    <copy>
      <from expression="0"/>
      <to variable="[activity.Name]_loopCounter" part="loopCounter" />
    </copy>
  </assign>
  <assign name="[activity.Name]_determine_instances">
    <copy>
      <from expression="[MI_Condition Exprssion]"/>
      <to variable="[activity.Name]_forEachCount" part="forEachCount" />
    </copy>
  </assign>
  <while condition=" bpws:getVariableData([activity.Name]_loopCounter,loopCounter) >=
                     bpws:getVariableData([activity.Name]_forEachCount,forEachCount)">
    <sequence>
      <invoke name=" Spawn_Process_For_[activity.Name]" ... >
      <assign name="[activity.Name]_increment_counter">
        <copy>
          <from expression="bpws:getVariableData([activity.Name]_loopCounter,loopCount)+1"/>
          <to variable="[activity.Name]_loopCounter" part="loopCounter" />
        </copy>
      </assign>
    </sequence>
  </while>
  <assign name="[activity.Name]_reset_loopCounter">
    <copy>
      <from expression="0"/>
      <to variable="[activity.Name]_loopCounter" part="loopCounter" />
    </copy>
  </assign>
  <!-- Set a while to receive all the return messages. The condition will be the same.-->
  <while condition=" bpws:getVariableData([activity.Name]_loopCounter,loopCounter) >=
                     bpws:getVariableData([activity.Name]_forEachCount,forEachCount)">
    <sequence>
      <receive name="[activity.Name]_Completed" ... >
      <assign name="[activity.Name]_increment_counter">
        <copy>
          <from expression="bpws:getVariableData([activity.Name]_loopCounter,loopCount)+1"/>
          <to variable="[activity.Name]_loopCounter" part="loopCounter" />
        </copy>
      </assign>
    </sequence>
  </while>
</sequence>
<!-- The contents of the process after the looping activity are here-->
```

Example 4 BPEL4WS Sample for a Multi-Instance Loop with Parallel Ordering MI_FlowCondition = All

Parallel Multi-Instance Loops -- Flow Condition One

The loop mappings for Parallel Multi-Instance loops that have a MI_FlowCondition of One are described in the following table (these mappings extend the mappings of the Parallel Multi-Instance Setup--refer to the section above):

Multi-Instance	Mapping to BPEL4WS
MI_FlowCondition = One	This setting utilizes the mechanisms described above for the Parallel ordering. The "One" setting requires that only one of the spawned *processes* must be completed before the original *process* can continue (see Figure 105 and Example 5).
Receiving the completion message	Only one message is required from any one of the spawned *processes* before the original *process* can continue. Thus, there will be a single *receive* activity following the *while* activity. The *receive* will be the last element of the *sequence* that was started for the mapping of the Multi-Instance activity. The other spawned *processes* will continue there activities in parallel, but their completion will have no direct impact on the flow of the main process (their messages won't be received). *Note: As mentioned above, it is up to the BPEL4WS execution environment to provide a "virtual context" for all the derived processes to "share" the process variables that may be updated by the spawned processes with the original process, even if there are no specific BPEL4WS activities to manage this information.*

Table 66 Parallel Multi-Instance Activity Loop, MI_FlowCondition = One, Mappings to BPEL4WS

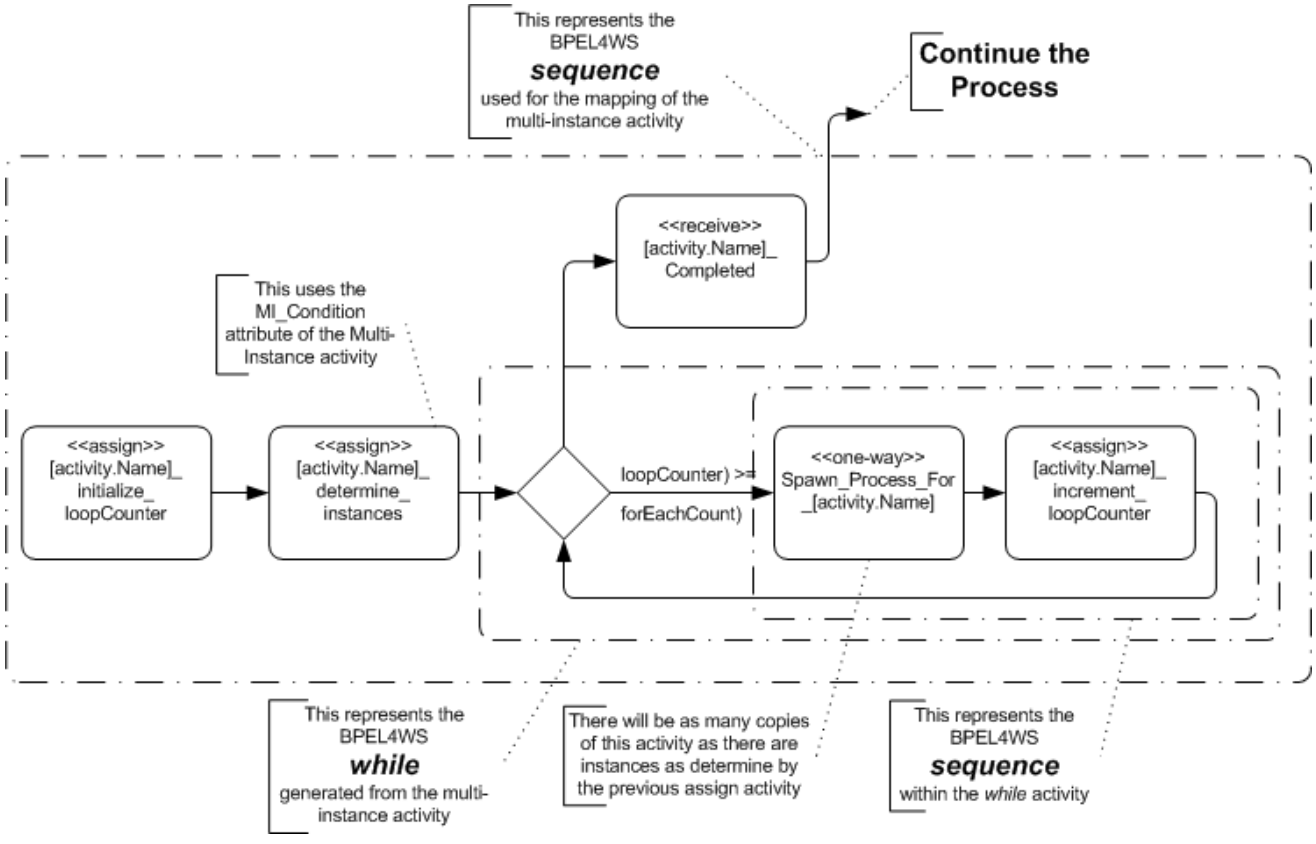

Figure 105 BPMN Depiction of BPEL4WS Pattern for a Parallel Multi-instance MI_FlowCondition = One

Example 5 displays some sample BPEL4WS code that reflects the mapping of a Multi-Instance loop that has Parallel ordering and must wait for only one of the looped activities.

```
<!-- The Process data is defined first-->
<variable name="[activity.Name]_loopCounter" messageType="loopCounterMessage" />
<variable name="[activity.Name]_forEachCount" messageType="forEachCounterMessage" />
<!-- The contents of the process prior to the looping activity are here-->
<sequence>
  <assign name="[activity.Name]_initialize_loopCounter">
    <copy>
      <from expression="0"/>
      <to variable="[activity.Name]_loopCounter" part="loopCounter" />
    </copy>
  </assign>
  <assign name="[activity.Name]_determine_instances">
    <copy>
      <from expression="[MI_Condition Exprssion]"/>
      <to variable="[activity.Name]_forEachCount" part="forEachCount" />
    </copy>
  </assign>
  <while condition="bpws:getVariableData([activity.Name]_loopCounter,loopCounter) >=
                    bpws:getVariableData([activity.Name]_forEachCount,forEachCount)">
    <sequence>

      <!--The mappings of the original activity are placed here.-->

      <assign name="[activity.Name]_increment_counter">
        <copy>
          <from expression="bpws:getVariableData([activity.Name]_loopCounter,loopCount)+1"/>
          <to variable="[activity.Name]_loopCounter" part="loopCounter" />
        </copy>
      </assign>
    </sequence>
  </while>
  <receive name="[activity.Name]_Completed" ... >
</sequence>
<!-- The contents of the process after the looping activity are here-->
```

Example 5 BPEL4WS Sample for a Multi-Instance Loop with Parallel Ordering MI_FlowCondition = One

Parallel Multi-Instance Loops -- Flow Condition Complex

The loop mappings for Parallel Multi-Instance loops that have a MI_FlowCondition of Complex are described in the following table (these mappings extend the mappings of the Parallel Multi-Instance Setup--refer to the section above):

Multi-Instance	Mapping to BPEL4WS
MI_FlowCondition = Complex	The mapping for this setting is almost the same as the MI_FlowCondition of All mapping (as described above) and seen in Figure 104 and Example 4). The difference is that the number of return messages required before the process flow will continue must be determined and the messages have been received.
The while condition for receiving completion messages	The second while in the sequence will be used to receive the appropriate number of completion messages. The ComplexMI_FlowCondition, which MUST be a boolean expression, will determine this number. The *while condition* be structured as follows: `<while condition="[ComplexMI_FlowCondition]">`

Table 67 Parallel Multi-Instance Activity Loop, MI_FlowCondition = Complex, Mappings to BPEL4WS

Parallel Multi-Instance Loops -- Flow Condition None

The loop mappings for Parallel Multi-Instance loops that have a MI_FlowCondition of None are described in the following table (these mappings extend the mappings of the Parallel Multi-Instance Setup--refer to the section above):

Multi-Instance	Mapping to BPEL4WS
MI_FlowCondition = None	This means that there is not synchronization or control of the Tokens that are generated through the multi-instance activity. This means that each Token will continue on independently and each Token will create a separate instantiation of each activity they encounter. Basically, it means there is a separate copy of the whole process, for each copy of the Multi-Instance activity, after that point. Each copy of the remainder of the process will continue independently. To create this behavior, the remainder of the process will moved into a new, derived *process*.

Spawning the rest of the process	This *process* will be spawned through a one-way *invoke* that will be placed within the *while* activity, after the mappings of the original BPMN activity (see Figure 106 and Example 6). The *name* attribute for the derived *invoke* will be in the following format:
	`<invoke name=`
	`"Spawn_Remainder_of_Process_from_[activity.Name]"...>`

Table 68 Parallel Multi-Instance Activity Loop, MI_FlowCondition = None, Mappings to BPEL4WS

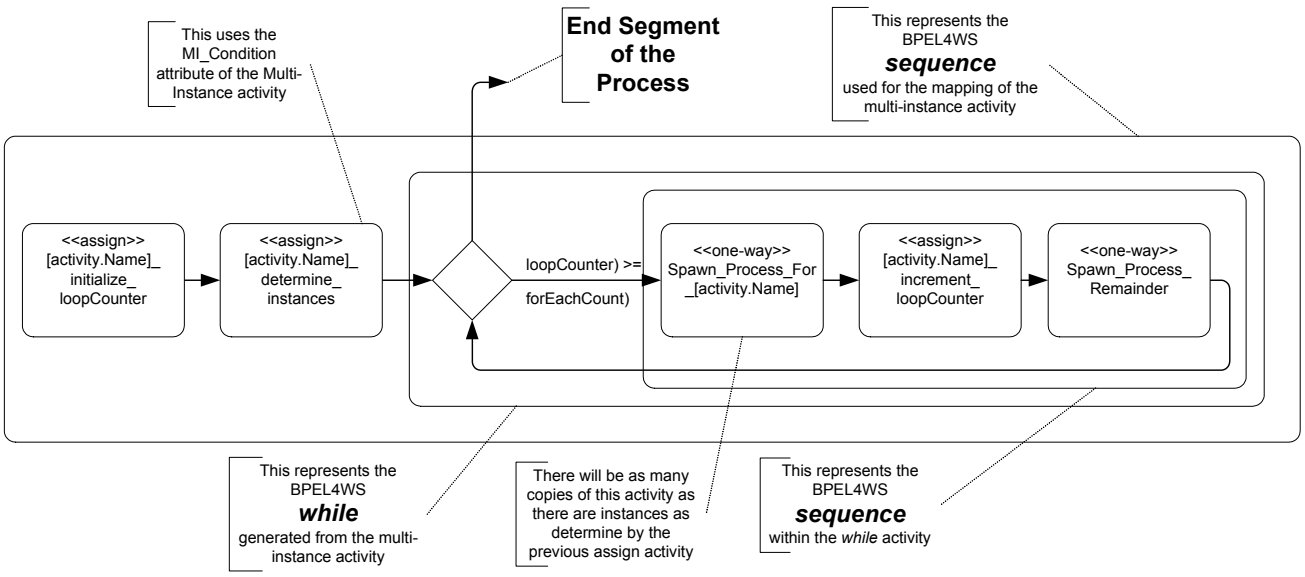

Figure 106 BPMN Depiction of BPEL4WS Pattern for a Parallel Multi-instance MI_FlowCondition = None

Example 6 displays some sample BPEL4WS code that reflects the mapping of a Multi-Instance loop that has Parallel ordering and must wait for none of the looped activities.

```
<!-- The Process data is defined first-->
<variable name="[activity.Name]_loopCounter" messageType="loopCounterMessage" />
<variable name="[activity.Name]_forEachCount" messageType="forEachCounterMessage" />
<!-- The contents of the process prior to the looping activity are here-->
<sequence>
  <assign name="[activity.Name]_initialize_loopCounter">
    <copy>
      <from expression="0"/>
      <to variable="[activity.Name]_loopCounter" part="loopCounter" />
    </copy>
  </assign>
  <assign name="[activity.Name]_determine_instances">
    <copy>
      <from expression="[MI_Condition Exprssion]"/>
      <to variable="[activity.Name]_forEachCount" part="forEachCount" />
    </copy>
  </assign>
  <while condition=" bpws:getVariableData([activity.Name]_loopCounter,loopCounter) >=
                     bpws:getVariableData([activity.Name]_forEachCount,forEachCount)">
    <sequence>

      <!--The mappings of the original activity are placed here.-->

      <assign name="[activity.Name]_increment_counter">
        <copy>
          <from expression="bpws:getVariableData([activity.Name]_loopCounter,loopCount)+1"/>
          <to variable="[activity.Name]_loopCounter" part="loopCounter" />
        </copy>
      </assign>
    </sequence>
  </while>
  <invoke name="Spawn_Remainder_of_Process_from_[activity.Name]" ... >
</sequence>
<!-- The contents of the process after the looping activity are here-->
```

Example 6 BPEL4WS Sample for a Multi-Instance Loop with Parallel Ordering MI_FlowCondition = None

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The activity looping mappings were completely reorganized and revised.

6.5.2 Sub-Process Mappings

The following table displays a set of mappings from the variations of a Sub-Process to BPEL4WS elements (This extends the mappings that are defined for all activities--refer to the section entitled “Common Activity Mappings” on page 162):

Sub-Process	Mapping to BPEL4WS
ActivityType = SubProcess	The SubProcessType attribute will determine the exact mapping of a Sub-Process. See the next two sub-sections for these mappings.
IsATransaction	The mapping of a Sub-Process set to a Transaction is an Open Issue (Refer to the section entitled “Open Issues” on page 245 for other Open Issues). Thus, there is no mapping defined when the IsATransaction is set to True, or the sub-attributes TransactionId, TransactionProtocol, and TransactionMethod.

Table 69 Sub-Process Mappings to BPEL4WS

Embedded Sub-Process

The following table displays a set of mappings from the variations of an Embedded Sub-Process to BPEL4WS elements (This extends the mappings that are defined for all activities--refer to the section entitled “Sub-Process Mappings” on page 177):

Sub-Process	Mapping to BPEL4WS
SubProcessType = Embedded	This will map to a BPEL4WS *scope* element. The scope is not an independent *process* and will share the *process variables* of the higher-level *process*.
GraphicalElements	This is a list of all the graphical elements contained within the Process. Each of these elements will have their mapping, as defined in the sections below.
Adhoc	Ad Hoc Processes are not executable. Thus, this attribute MUST be set to False if the Process is to be mapped to BPEL4WS. The AdHocCompletionCondition and the AdHocOrdering attributes are only valid if the AdHoc attribute is True. Thus, these attributes will not be mapped to BPEL4WS.

Table 70 Embedded Sub-Process Mappings to BPEL4WS

Independent Sub-Process

The following table displays a set of mappings from the variations of an Independent Sub-Process to BPEL4WS elements (This extends the mappings that are defined for all activities--refer to the section entitled “Sub-Process Mappings” on page 177):

Task	Mapping to BPEL4WS
SubProcessType = independent	BPEL4WS does not have a sub-process element. Thus, independent Sub-Processes MUST map to a BPEL4WS *process*. That is, the contents of the Sub-Process, whether it is expanded or collapsed, will be contained within a separate *process*. The ProcessRef attribute will identify the *process* that will be used for the mapping to the BPEL4WS *process*. The attributes of the other BPEL4WS *process* element will be filled from the mapping of the referenced Process. Refer to the section entitled “Business Process Mappings” on page 152 for the details of this mapping. The Sub-Process object itself maps to an *invoke* activity that “calls” the *process*.
InputPropertyMap	This attribute is actually a mapping of Process Properties to the Process Properties of the Process being referenced by the Sub-Process Object. The OutputPropertyMap attribute maps to the *inputVariable* attribute of the *invoke* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
OutputPropertyMap	This attribute is actually a mapping of Process Properties to the Process Properties of the Process being referenced by the Sub-Process Object. The InputPropertyMap attribute maps to the *outputVariable* attribute of the *invoke* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.

Table 71 Independent Sub-Process Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- Separate sub-sections were create for the mappings for embedded and independent Sub-Processes.
- The Embedded Sub-Process mapping was changed from an *invoke* of another *process* to a *scope*.

6.5.3 Task Mappings

The following table displays a set of mappings from the variations of a Task to BPEL4WS elements (This extends the mappings that are defined for all activities--refer to the section entitled “Common Activity Mappings” on page 162):

Task	Mapping to BPEL4WS
ActivityType = Task	The TaskType attribute will determine the exact mapping of a Task. See the next eight (8) sub-sections for these mappings.
Web service Mappings	The Implementation attribute MUST be a Web service or MUST be converted to a Web Service for mapping to BPEL4WS. The Web Service Attributes are mapped as follows: The Entity attribute is mapped to the *partnerLink* attribute of the BPEL4WS activity. The Interface attribute is mapped to the *portType* attribute of the BPEL4WS activity. The Operation attribute is mapped to the *operation* attribute of the BPEL4WS activity.

Table 72 Task Mappings to BPEL4WS

Service Task

The following table displays a set of mappings from the variations of a Service Task to BPEL4WS elements:

Task	Mapping to BPEL4WS
TaskType = Service	This type of Task maps to an *invoke* activity.
InMessage	The InMessage attribute maps to the *inputVariable* attribute of the *invoke* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
OutMessage	The OutMessage attribute maps to the *outputVariable* attribute of the *invoke* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
Implementation = Web Service	This will map as defined in Table 72.

Table 73 ServiceTask Mappings to BPEL4WS

Receive Task

The following table displays a set of mappings from the variations of a Receive Task to BPEL4WS elements (This extends the mappings that are defined for all Tasks--refer to the section entitled “Task Mappings” on page 179):

Task	Mapping to BPEL4WS
TaskType = Receive	This type of Task maps to a *receive* activity.
Message: Message	The Message attribute maps to the *variable* attribute of the *receive* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
Instantiate: Boolean: False	If the Instantiate attribute of the Task is set to False, then the *createInstance* attribute of the *receive* will not be included or it will be set to “no.” If the Instantiate attribute of the Task is set to True, then the *createInstance* attribute of the *receive* will be set to “yes.”
Implementation = Web Service	This will map as defined in Table 72.

Table 74 Receive Task Mappings to BPEL4WS

Send Task

The following table displays a set of mappings from the variations of a Send Task to BPEL4WS elements:

Task	Mapping to BPEL4WS
TaskType = Send	This type of Task maps to a *reply* or an *invoke* activity. The appropriate BPEL4WS activity will be determined by the implementation defined for the Task. That is, the *portType* and *operation* of the Task will be used to check to see if an upstream Receive Task have the same *portType* and *operation*. If these two attributes are matched, then the Send Task will map to a *reply*, if not, the Send Task will map to an *invoke*.
Message: Message	The Message attribute maps to the *variable* attribute of the *reply* activity or it maps to the *inputVariable* attribute of the *invoke* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
Implementation = Web Service	This will map as defined in Table 72.

Table 75 Send Task Mappings to BPEL4WS

User Task

The following table displays a set of mappings from the variations of a User Task to BPEL4WS elements:

Task	Mapping to BPEL4WS
TaskType = User	This type of Task maps to an *invoke* activity.
Performer: String	The Performer is information needed by the implementation. Thus, it will be included in the InMessage being sent to the Web service, through the *inputVariable* attribute of the *invoke* activity.
InMessage	The InMessage attribute maps to the *inputVariable* attribute of the *invoke* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
OutMessage	The OutMessage attribute maps to the *outputVariable* attribute of the *invoke* activity. Refer to the section entitled “Messages” on page 202 for more information about how a BPMN Message maps to BPEL4WS and WSDL.
Implementation = Web Service	This will map as defined in Table 72.

Table 76 User Task Mappings to BPEL4WS

Script Task

The following table displays a set of mappings from the variations of a Script Task to BPEL4WS elements:

Task	Mapping to BPEL4WS
TaskType = Script	This type of Task maps to an *invoke* activity. Since this activity is performed by a process engine, the default settings of the engine must be used to determine the settings of the *invoke* activity. That is, *partnerLink*, *portType*, *operation*, *inputVariable*, and maybe *outputVariable* are derived by these default settings. The script itself is performed when the appropriate Web service of the process engine in invoked.

Table 77 Script Task Mappings to BPEL4WS

Manual Task

The Manual Task does not map to BPEL4WS. Thus, this type of Task should not be used in a Process that is intended to generate BPEL4WS code.

Reference Task

The following table displays a set of mappings from the variations of a Reference Task to BPEL4WS elements:

Task	Mapping to BPEL4WS
TaskType = Reference	This type of Task is not directly mapped to BPEL4WS, since BPEL4WS does not support this type of referencing. However, the Task will be used as a placeholder for the Task that will be mapped (see next row).
TaskRef: Task	This attribute references another Task in the Process. It is the reference Task that will be mapped and the mappings will be placed in the location of the Reference Task. That is, another copy of the entire mapping of the referenced Task will be created in this location (the mappings will also exist in the referenced Task's original location).

Table 78 Reference Task Mappings to BPEL4WS

None Task

The following table displays a set of mappings from the variations of a None Task to BPEL4WS elements:

Task	Mapping to BPEL4WS
TaskType = None	This type of Task maps to an *empty* activity.

Table 79 None Task Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The mapping was separated into separate sub-sections for each type of Task.
- The reference to a BPEL4WS *until* activity was removed.
- The mapping for the Reference Task was included.

6.6 Gateways

6.6.1 Common Gateway Mappings

The following table displays a set of mappings are common for Gateways to BPEL4WS elements (these mappings extend the mappings common to objects -- refer to the section entitled “Common Object Mappings” on page 153):

Data-Based Exclusive Gateways	Mapping to BPEL4WS
Gateway	A Gateway will map to a variety of BPEL4WS elements (e.g., *switch*, *pick*, *flow*) and patterns of elements.
Incoming Flow	A Gateway, as with activities, is a location where Sequence Flow can converge. The convergence of Sequence Flow potentially marks the end of a BPEL4WS structured element, if the correct number of flow converge. Refer to the section entitled “Determining the Extent of a BPEL4WS Structured Element” on page 202 for more details on converging of Sequence Flow and their mapping to BPEL4WS.
Outgoing Flow	The mapping will begin at the location of the Gateway. The BPMN elements that follow the Gateway, until all the outgoing paths have converged, will be contained within the extent of the mapping (e.g., they will be placed with in a *sequence* within a *switch case*). The end of the mapping will be determined by the convergence of the paths, through a variety of mechanisms (refer to the section entitled “Determining the Extent of a BPEL4WS Structured Element” on page 202).
Assignments associated with Gates	This will map to a BPEL4WS *assign*. Refer to the section entitled “Property Assignments” on page 203 for more details about the mappings associated with the *assign* element.

Table 80 Common Gateway Mappings to BPEL4WS

6.6.2 Exclusive

Data-Based

The following table displays a set of mappings from the variations of a Data-Based Exclusive Gateway to BPEL4WS elements (these mappings extend the mappings common to objects -- refer to the section entitled “Common Gateway Mappings” on page 183):

Data-Based Exclusive Gateways	Mapping to BPEL4WS
Gateway (GatewayType = XOR; XORType = Data)	The Gateway will map to a BPEL4WS *switch*.
MarkerVisible	This does not have a mapping to BPEL4WS. Its purpose is to determine whether or not a graphical marker will be displayed.
Incoming Flow	
Outgoing Flow	
Gate	Each Gate will map to a *case* of the *switch*. The *cases* will be listed in the *switch* in the same order as they are listed for the Gateway.
Condition for the Sequence Flow associated with the Gate	This will map to the *condition* for a *switch* case.
BPMN Elements that follow the Gate.	If there is more than one element that follows the Gate, and this includes Assignments for the Gate, then these elements will be placed inside a *sequence* activity after these elements have been mapped.
DefaultGate	This will map to the *otherwise* element of the *switch*.
BPMN Elements that follow the DefaultGate.	If there is more than one element that follows the DefaultGate, and this includes Assignments for the Gate, then these elements will be placed inside a *sequence* activity after these elements have been mapped.

Table 81 Data-Based Exclusive Gateway Mappings to BPEL4WS

Event-Based

The following table displays a set of mappings from the variations of a Event-Based Exclusive Gateway to BPEL4WS elements (these mappings extend the mappings common to objects -- refer to the section entitled “Common Gateway Mappings” on page 183):

Event-Based Exclusive Gateways	Mapping to BPEL4WS
Gateway (GatewayType = XOR; XORType = Event)	This Gateway will map to a BPEL4WS *pick*. The elements of the *pick* will be determined by the targets of the outgoing Sequence Flow. The specific mappings are described in the rows below.
Instantiate	If the Instantiate attribute of the Gateway is set to False, then the *createInstance* attribute of the *pick* SHALL NOT be included OR it MUST be set to “no.” If the Instantiate attribute of the Gateway is set to True, then the *createInstance* attribute of the *pick* MUST be set to “yes.”
Gate with Receive Task as Target	The Receive Task will map to an *onMessage* element within the *pick*. The attributes of the Receive Task will map to the appropriate elements of the *onMessage*, such as *operation* and *portType*. Refer to the section entitled “Receive Task” on page 180 for the mapping of the Receive Task. Note that the name of the Task does not have a corresponding attribute within the *onMessage* element.
Gate with Message Intermediate Event as Target	A Message Intermediate Event will map to an *onMessage* element within the *pick*. The attributes of the Message Intermediate Event will map to the appropriate elements of the *onMessage*, such as *operation* and *portType*. Refer to the section entitled “Intermediate Event Mappings” on page 157 for the mapping of the Message Intermediate Event.
Gate with Timer Intermediate Event as Target	A Timer Intermediate Event, which is the Target of a Sequence Flow associated with the Gate, will map to an onAlarm element within the pick. The Timedate attribute of the Event will map to the until element of the *onAlarm* element. The Timecycle attribute of the Event will map to the for element of the *onAlarm* element.
Gate with Link Intermediate Event as Target	A Rule Intermediate Event, in this situation, will be considered as the same as receiving a message from a process. Thus, this will map to an *onMessage* element within the *pick*. The attributes of the Message Intermediate Event will map to the appropriate elements of the *onMessage*, such as *operation* and *portType*. Refer to the section entitled “Intermediate Event Mappings” on page 157 for the mapping of the Message Intermediate Event.
Gate with Rule Intermediate Event as Target	A Rule Intermediate Event, in this situation, will be considered as the same as receiving a message from a system that tracks and generates Rule events. Thus, this will map to an *onMessage* element within the *pick*. The attributes of the Message Intermediate Event will map to the appropriate elements of the *onMessage*, such as *operation* and *portType*. Refer to the section entitled “Intermediate Event Mappings” on page 157 for the mapping of the Message Intermediate Event.
BPMN Elements that follow the first target of a Gate.	If there is more than one element that follows the first target of a Gate, and this includes Assignments for the Gate, then these elements will be placed inside a *sequence* activity after these elements have been mapped.

Table 82 Data-Based Exclusive Gateway Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- A mapping for the Instantiate attribute of the Gateway was added.
- A mapping for the Exception Intermediate Event, as a Target, was removed.
- Those mappings that were incomplete in the previous version, were completed.

6.6.3 Inclusive

The following table displays a set of mappings from the variations of a Inclusive Gateway to BPEL4WS elements (these mappings extend the mappings common to objects -- refer to the section entitled “Common Gateway Mappings” on page 183):

Inclusive Gateways	Mapping to BPEL4WS
Gateway (GatewayType = OR)	The Gateway will map to a set of BPEL4WS *switches* within a BPEL4WS *flow*. An additional *switch* will be required if the DefaultGate is used (see below)
Gate	Each Gate will map to a *switch*. Each *switch* will binary in nature. That is, each switch will have exactly one *case* and one *otherwise*.
Condition for the Sequence Flow associated with the Gate	This will map to the *condition* for the *switch* case.
BPMN Elements that follow the Gate.	If there is more than one element that follows the Gate, and this includes Assignments for the Gate, then these elements will be placed inside a *sequence* activity after these elements have been mapped. If a DefaultGate is used, then an *assign* activity will follow all the other mappings (see below for details).
The *otherwise* element for the *switch*	The *otherwise* element for each *switch* will contain an *empty* activity. However, if the DefaultGate is used, then
DefaultGate	The DefaultGate will map to a *switch*. However, by using the DefaultGate, the mapping to BPEL4WS is more complicated (see Figure 107 and Example 7). This is the path that is taken if none of the other paths are taken. Thus, the decision about whether the Default Gate should be taken will occur after all the other Gate decisions have been determined. This means that the *switch* for the DefaultGate will follow the *flow* activity generated for all the Gates of the Gateway. Also, a *sequence* activity must encompass the *flow* and the *switch*.
Create the tracking variable	A *variable* must be used so that the switch for the DefaultGate will know whether or not the Default BPMN path should be taken. To do this, a BPEL4WS *variable* must be created with a derived name and will have a structure as follows: `<variable name="[Gateway.Name]_noDefaultRequired" messageType="noDefaultRequired" />`
Supporting WSDL Message	A WSDL *message* element will have to be created to support this *variable*. This *message* can be used for multiple *variables*. The *message* will be structured as follows: `<message name="noDefaultRequired" ><part name="noDefault" type="xsd:boolean" /></message>`

Inclusive Gateways	Mapping to BPEL4WS

Initialization of the tracking variable	An *assign* activity will be created to initialize the *variable* before the start of the loop. This *assign* precede the *flow* activity that contains all the *switches* derived from the Gates. This will be the first activity within the *sequence* activity. The *assign* will be structured as follows:
	`<assign name="[Gateway.Name]_initialize_noDefault">`
	`  <copy>`
	`  <from expression="false"/>`
	`  <to variable="[Gateway.Name]_noDefaultRequired"`
	`  part="noDefault" />`
	`  </copy>`
	`</assign>`

The *switch cases*	The condition for the *switch case* will used the noDefaultRequired *variable* and will structured as follows:
	`<switch>`
	`  <case condition="bpws:getVariableProperty(`
	`  [Gateway.Name]_noDefaultRequired,noDefault)=true">`
	`  <sequence>`
	
	`<!--The mappings of the original activity are placed here.-->`
	`<!--An assign activity (see below) is placed here.-->`
	
	`  </sequence>`
	`  </case>`
	`  <otherwise>`
	`  <empty/>`
	`  </otherwise>`
	`</switch>`

BPMN Elements that follow the DefaultGate	If there is more than one element that follows the DefaultGate, and this includes Assignments for the Gate, then these elements will be placed inside a *sequence* activity after these elements have been mapped. An *assign* activity will be placed in the sequence after all the other mappings (see next row).

Setting of the tracking variable	If any of the *switches* within the flow passes the condition of the switch case, then the noDefaultRequired must be set to True. This will ensure that the DefaultGate switch will bypass the mapped activities for the BPMN Default Gate.
	An *assign* activity will be created to set the *variable* to True. This will be the last activity within the *sequence* activity within the *switch*. The *assign* will be structured as follows:
	`<assign name="[Gateway.Name]_set_noDefault">`
	`  <copy>`
	`  <from expression="true"/>`
	`  <to variable="[Gateway.Name]_noDefaultRequired"`
	`  part="noDefault" />`
	`  </copy>`
	`</assign>`

Table 83 Inclusive Gateway Mappings to BPEL4WS

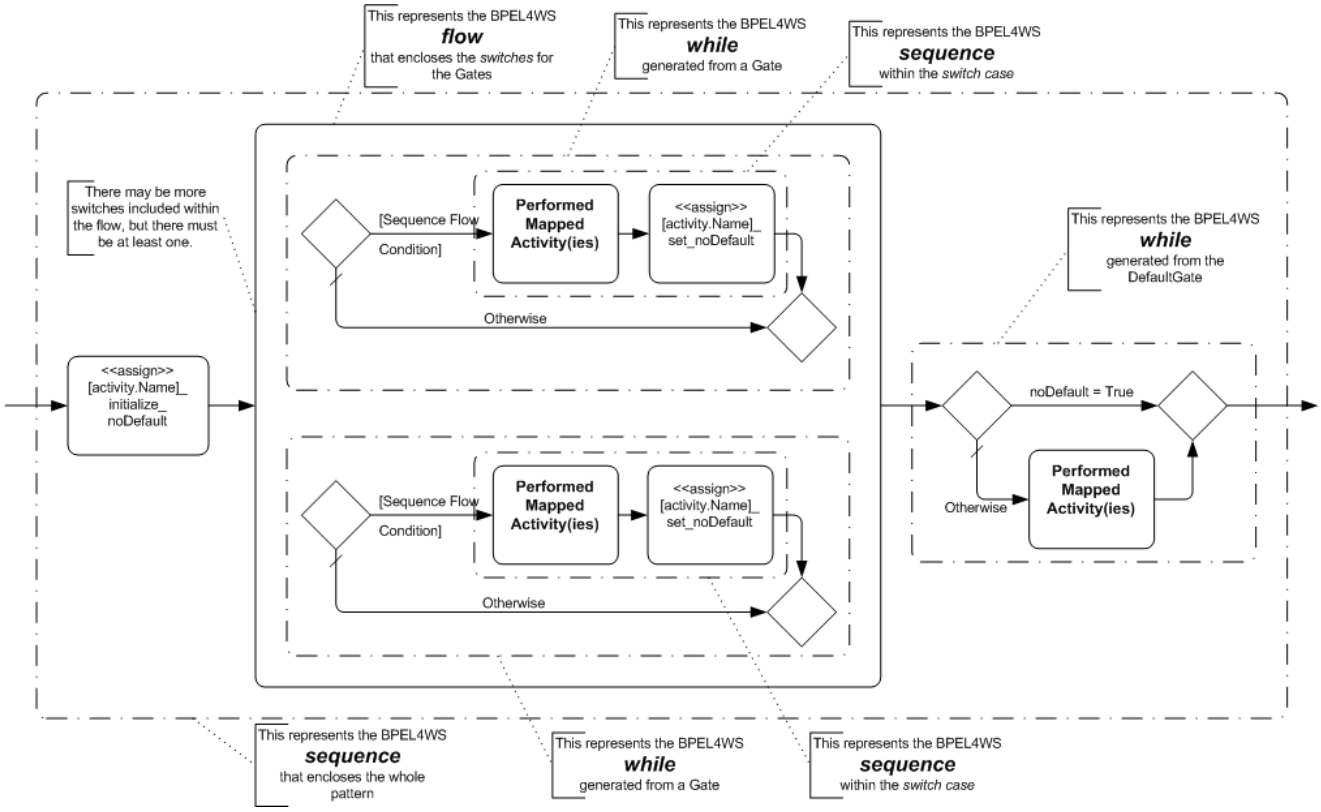

Figure 107 BPMN Depiction of BPEL4WS Pattern for an Inclusive Decision with two (2) Gates and a DefaultGate

Example 7 displays some sample BPEL4WS code that reflects the mapping of a Multi-Instance loop that has Parallel ordering and must synchronize all the looped activities.

```
<!-- The Process data is defined first-->
<variable name="[activity.Name]_loopCounter" messageType="loopCounterMessage" />
<!-- The contents of the process prior to the looping activity are here-->
<sequence>
  <assign name="[Gateway.Name]_initialize_noDefault">
    <copy>
      <from expression="false"/>
      <to variable="[Gateway.Name]_noDefaultRequired" part="noDefault" />
    </copy>
  </assign>
  <flow>
    <!--There will be as many copies of the switch below as there are Gates.-->
    <switch>
      <case condition="[Sequence Flow Condition">
        <sequence>

      <!--The mappings of the activities are placed here.-->

          <assign name="[Gateway.Name]_initialize_noDefault">
            <copy>
              <from expression="true"/>
              <to variable="[Gateway.Name]_noDefaultRequired" part="noDefault" />
            </copy>
          </assign>
        </sequence>
      </case>
      <otherwise>
        <empty/>
      </otherwise>
    </switch>
  </flow>
  <switch>
    <case condition=
          "bpws:getVariableProperty([Gateway.Name]_noDefaultRequired,noDefault)=true">
    <sequence>

      <!--The mappings of the activities are placed here.-->

    </sequence>
  </case>
  <otherwise>
    <empty/>
  </otherwise>
  </switch>
</sequence>
```

Example 7 BPEL4WS Sample for the Pattern for an Inclusive Decision with a DefaultGate

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- This mapping, which was not defined in the last version, was defined.

6.6.4 Complex

The behavior and usage of Complex Gateways have not been well enough established for a mapping to BPEL4WS to be defined.

6.6.5 Parallel

The following table displays a set of mappings from the variations of a Parallel Gateway to BPEL4WS elements (these mappings extend the mappings common to objects -- refer to the section entitled “Common Gateway Mappings” on page 183):

Parallel Gateways	Mapping to BPEL4WS
Gateway (GatewayType = AND)	The Gateway will map to a BPEL4WS *flow*.

Table 84 Parallel Gateway Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- This mapping, which was not defined in the last version, was defined.

6.7 Pool

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 document while another Pool might encompass B2B Collaboration process.

6.8 Lane

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.

6.9 Artifacts

As a general rule, Artifacts do not map to BPEL4WS elements. They provide detailed information about how data will interact with the flow objects and flows of Processes.

However, 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*.

For any new Artifact that is added to a BDP through a modeling tool, it will have to be determined whether or not that Artifact maps to any BPEL4WS element.

6.10 Sequence Flow

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.

The following table displays a set of mappings from Sequence Flow to BPEL4WS elements:

Sequence Flow	Mapping to BPEL4WS
Sequence Flow	This MAY map to a BPEL4WS *link* element. The location of the Sequence Flow within the Process will determine how or if it is mapped to a *link*. Even if the Sequence Flow is not mapped to a link, attributes, such as Condition, will be mapped to BPEL4WS elements, as described below.
Id, Category, and Documentation	These Elements do not map to any BPEL4WS elements or attributes.
Name: String	If the Sequence is not being mapped to a *link*: This attribute does not map to any BPEL4WS elements or attributes. If the Sequence is being mapped to a *link*: The Name attribute of the Process SHALL map to *name* attribute of the *link*. The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *name* attribute. Note that there may be two or more elements with the same name after the BPMN name has been stripped.
Source	If the Sequence is not being mapped to a *link*: This attribute does not map to any BPEL4WS elements or attributes. If the Sequence is being mapped to a *link*: This mapping is described in the next four (4) Rows.
Source Object is an Activity (for a *link*)	The mapping of the source activity will now include a *source* element. The Name of the Sequence Flow will map to *linkName* attribute of the *source* element.The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *name* attribute. Note that there may be two or more elements with the same name after the BPMN name has been stripped. For an exception to the location of the *source* element, see the description of the mapping for a ConditionExpression when the Source object is an Activity below.
Source Object is a Gateway (for a *link*)	This mapping is described in the next two (2) Rows.
The Gateway maps to an activity (e.g., *switch*)	This mapping is the same as if the source object is an activity (see above).
The Gateway does not map to an activity	This Sequence Flow will be essentially combined with one of the Gateway's incoming Sequence Flow. (There will be a separate *link* for each of the incoming Sequence Flow). The Source of the second Sequence will be used at the Source for the original Sequence Flow. Then, this mapping is the same as if the Source object is an activity (see above).

Sequence Flow	Mapping to BPEL4WS
Target	If the Sequence is not being mapped to a *link*: This attribute does not map to any BPEL4WS elements or attributes. If the Sequence is being mapped to a *link*: This mapping is described in the next four (4) Rows.
Target Object is an Activity	The mapping of the target activity will now include a *target* element. The Name of the Sequence Flow will map to *linkName* attribute of the *target* element.The extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *name* attribute. Note that there may be two or more elements with the same name after the BPMN name has been stripped.
Target Object is a Gateway	This mapping is described in the next two (2) Rows.
The Gateway maps to an activity (e.g., *switch*)	This mapping is the same as if the target object is an activity (see above).
The Gateway does not map to an activity	This Sequence Flow will be essentially combined with one of the Gateway's outgoing Sequence Flow. (There will be a separate *link* for each of the outgoing Sequence Flow). The Target of the second Sequence will be used at the Target for the original Sequence Flow. Then, this mapping is the same as if the target object is an activity (see above).
ConditionType = None	If the Sequence is not being mapped to a *link*: This attribute does not map to any BPEL4WS elements or attributes. If the Sequence is being mapped to a *link*: This means that there is no condition placed on the transition between elements (through the link). Thus, there is nothing to be mapped to BPEL4WS.
ConditionType = Expression	This mapping is described in the next two (2) Rows.
Source Object is a Gateway	The mapping of the Sequence Flow in this situation is described in the section entitled "Exclusive" on page 184, in the section entitled "Inclusive" on page 186, in the section entitled "Complex" on page 190.
Source Object is an Activity	Since a Sequence Flow MAY NOT have a Condition if the Source is an activity, unless there are multiple outgoing Sequence Flow, a BPEL4WS *flow* will be required and the Sequence Flow will map to a *link* element. An *empty* activity will be placed in the flow and will contain all the *source* elements. The ConditionExpression will then map to the *transitionCondition* attribute of the *source* element that is contained in the appropriate BPEL4WS activity (see a description of locating the source above).
ConditionType = Default	The mapping of the Sequence Flow in this situation is described in the section entitled "Exclusive" on page 184, in the section entitled "Inclusive" on page 186, in the section entitled "Complex" on page 190.
Quantity: Integer: 1	The mapping of the Quantity attribute, if its value is greater than one (1), BPEL4WS is an open issue. Refer to the section entitled "Open Issues" on page 245 for other Open Issues.

Table 85 Exception Flow Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The details of the mapping, which had been TDB, were included.

6.10.1 When to Map a Sequence Flow to a BPEL4WS Link

In many situations, a Sequence Flow will not map to a BPEL4WS *link* element.

- ❖ To connect activities that are listed in a BPEL4WS structured activity (e.g., a *sequence*), the *link* elements are not required.

The ordering of the list in the sequence provides the direction of flow (see Figure 108).

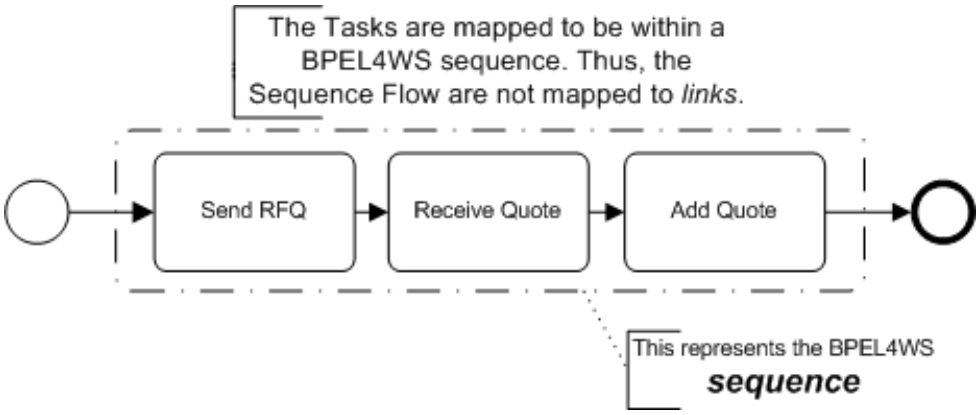

Figure 108 Example: Sequence Flow that are not used for BPEL4WS *links*

- ❖ *Link* elements are only appropriate when the Sequence Flow are connecting objects that are within a BPEL4WS *flow*.

However, it is only the Sequence Flow that are completely contained within the boundaries of the *flow* will be mapped to a *link* (see Figure 110). It should be noted that if another structured activity (e.g., a *switch*) is contained within the flow, then the Sequence Flow that would be appropriate for the contents of the structured activity, would not be mapped to a *link*.

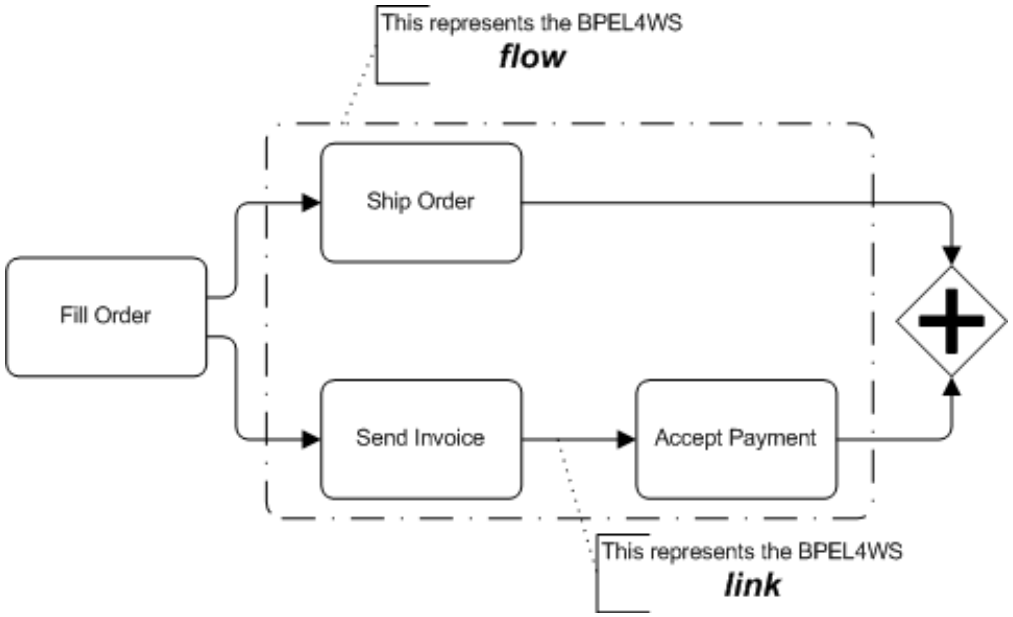

Figure 109 Example: A Sequence Flow that is used for a BPEL4WS *link*

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- This section was added.

6.11 Message Flow

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

6.12 Association

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.

6.13 Exception Flow

BPMN Exception Flow is all the activities, connected by Sequence Flow, which flow from an Intermediate Event attached to the boundary of an activity, until the flow merges back into the normal flow (sometimes at the point of an End Event).

BPEL4WS handles exceptions in a much more structured and programmatic manner. If triggered through a *fault*, the activities in an *exceptionHandlers* will be performed and completed, and then the *process* will continue from the point where the interrupted activity

would have completed normally. Thus, the *exceptionHandlers* element is a completely contained structured element.

Since BPMN handles Exception Flow with much more flexiblity, so that the modeler can have the Exception Flow go wherever it is appropriate, there are different challenges to the BPEL4WS mapping, depending on the configuration of the BPMN model.

The following table displays the mapping Exception FIlow to BPEL4WS:

Exception Flow	Mapping to BPEL4WS
Activities within the Exception Flow	All the activities that follow the attached Intermediate Event, until the Exception Flow merges back into the Normal Flow, will be mapped to BPEL4WS and then placed within the *exceptionHandlers* element for the *scope* of the activity (and usually within a *sequence*).

Table 86 Common Exception Flow Mappings to BPEL4WS

Additional BPEL4WS mapping patterns for Exception Flow will be described in the next three (3) sections.

The Exception Flow Merges back into the Normal Flow After the Activity

In this situation, the Exception Flow may contain one or more activities, but will merge back into the Normal Flow at the same object that follows the activity that is the source of the Exception Flow (see Figure 110). This situation maps closely to the BPEL4WS mechanism for exception handling. Thus, no special mapping mechanisms are required.

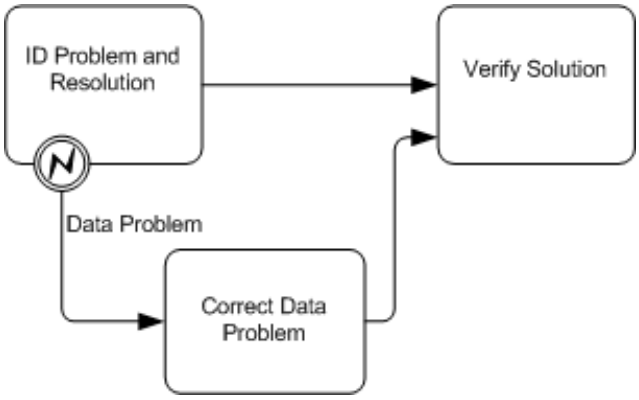

Figure 110 Exception Flow Merging back into the Normal Flow Immediately after the Interrupted Activity

The Exception Flow Merges back into the Normal Flow Further Downstream

In this situation, the activities in the Exception Flow substitute for some of the Normal Flow activities and, thus, the Exception Flow will skip these activities and merge into the Normal Flow further downstream (see Figure 111). Alternatively, the exception may create a situation where the Process must end prematurely, which means that the Exception Flow will merge with the Normal Flow at an End Event (see Figure 112). In either situation, special BPEL4WS patterns will have to appended to the basic Exception Flow mappings.

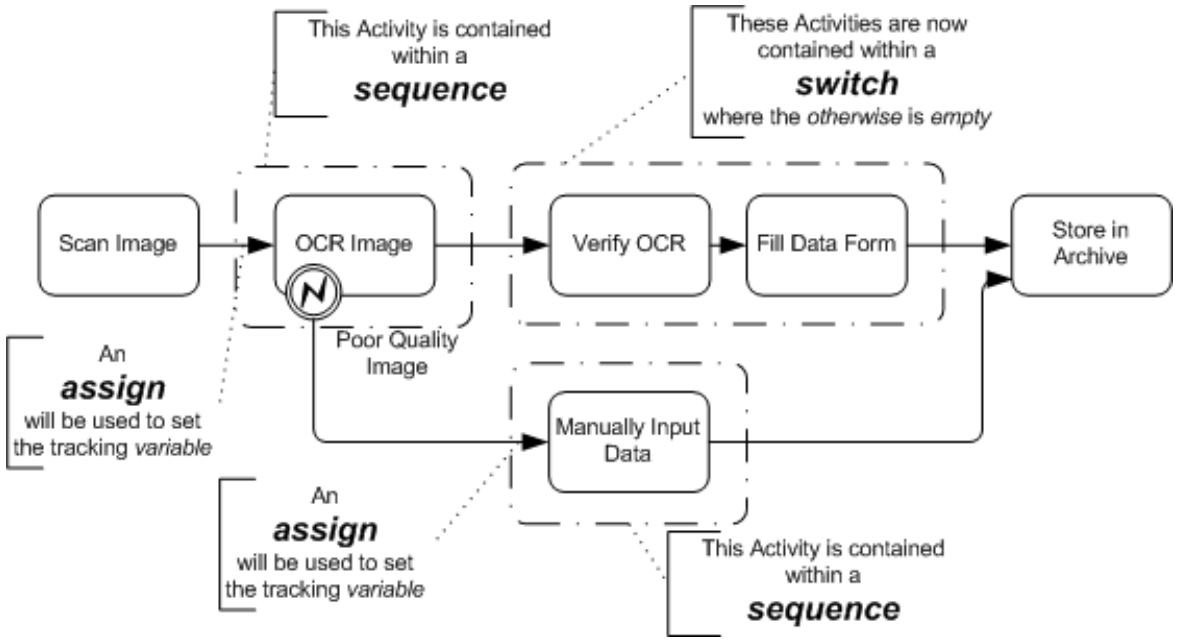

Figure 111 Exception Flow Merging back into the Normal Flow Further Downstream

The following table displays the mapping Exception Fllow to BPEL4WS (these mappings extend the mappings common to Exception Flow -- see above):

Exception Flow	Mapping to BPEL4WS
Activities within the Exception Flow	If there is only one activity in the *exceptionHandlers* element for the *scope* of the activity, then this activity will be placed within a *sequence* and preceded by an *assign* (as described below).
Original Activity	The mapping of the original activity will be placed within a *sequence* (if it had not bee already).
After the Original Activity	The orginal activity will now be followed by a *switch*, instead of what would have been normally mapped there.
Switch Characteristics	The *switch* will be binary in nature. There will be one *case* and an *otherwise* element.

Create the tracking variable	A *variable* must be used so that the switch will know whether or not the Exception Flow or Normal Flow had reached that point in the Process. To do this, a BPEL4WS *variable* must be created with a derived name and will have a structure as follows:
	`<variable name="[activty.Name]_normalCompletion"`
	`messageType="noDefaultRequired" />`

Supporting WSDL Message	A WSDL *message* element will have to be created to support this *variable*. This *message* can be used for multiple *variables*. The *message* will be structured as follows:
	`<message name="noDefaultRequired" >`
	`<part name="normalCompletion" type="xsd:boolean" />`
	`</message>`

Exception Flow	Mapping to BPEL4WS

Initialization of the Tracking Variable	An *assign* activity will be created to initialize the *variable* before the start of the original activity. It will be the first activity in the *sequence* described above. The *assign* will be structured as follows:
	`<assign name="[activity.Name]_initialize_normalCompletion">`
	`  <copy>`
	`  <from expression="true"/>`
	`  <to variable="[activity.Name]_normalCompletion"`
	`  part="normalCompletion" />`
	`  </copy>`
	`</assign>`

Setting of the tracking variable	If a fault is thrown and *exceptionHandlers* is activated, then an *assign* activity will be used to set the *variable* to False. This will be the first activity within the *sequence* activity of the *exceptionHandlers*. The *assign* will be structured as follows:
	`<assign name="[activity.Name]_set_normalCompletion">`
	`  <copy>`
	`  <from expression="false"/>`
	`  <to variable="[activity.Name]_normalCompletion"`
	`  part="normalCompletion" />`
	`  </copy>`
	`</assign>`

Switch cases	The case for the switch will contain all the mappings for all activities that occur in the Process until the Exception Flow has merged back (which could be the end of the Process), usually within a *sequence*. The otherwise for the switch will contain an *empty* activity.
	The condition for the *switch case* will used the `normalCompletion` *variable* and will structured as follows:
	`<switch>`
	`  <case condition="bpws:getVariableProperty(`
	`  [activity.Name]_normalCompletion,`
	`  normalCompletion)=true">`
	`  <sequence>`
	
	`<!--The mappings of the Process activities until the merging of the Exception Flow are placed here.-->`
	
	`  </sequence>`
	`  </case>`
	`  <otherwise>`
	`  <empty/>`
	`  </otherwise>`
	`</switch>`

Potential Invalid Model	If the Exception Flow occurs in the larger context of a set of parallel activities, then the Exception Flow must merge back into the Normal Flow prior to the end of the parallel activities (a BPEL4WS *flow*), or this will create an invalid model.

Table 87 Exception Flow Merging back into the Normal Flow Further Downstream

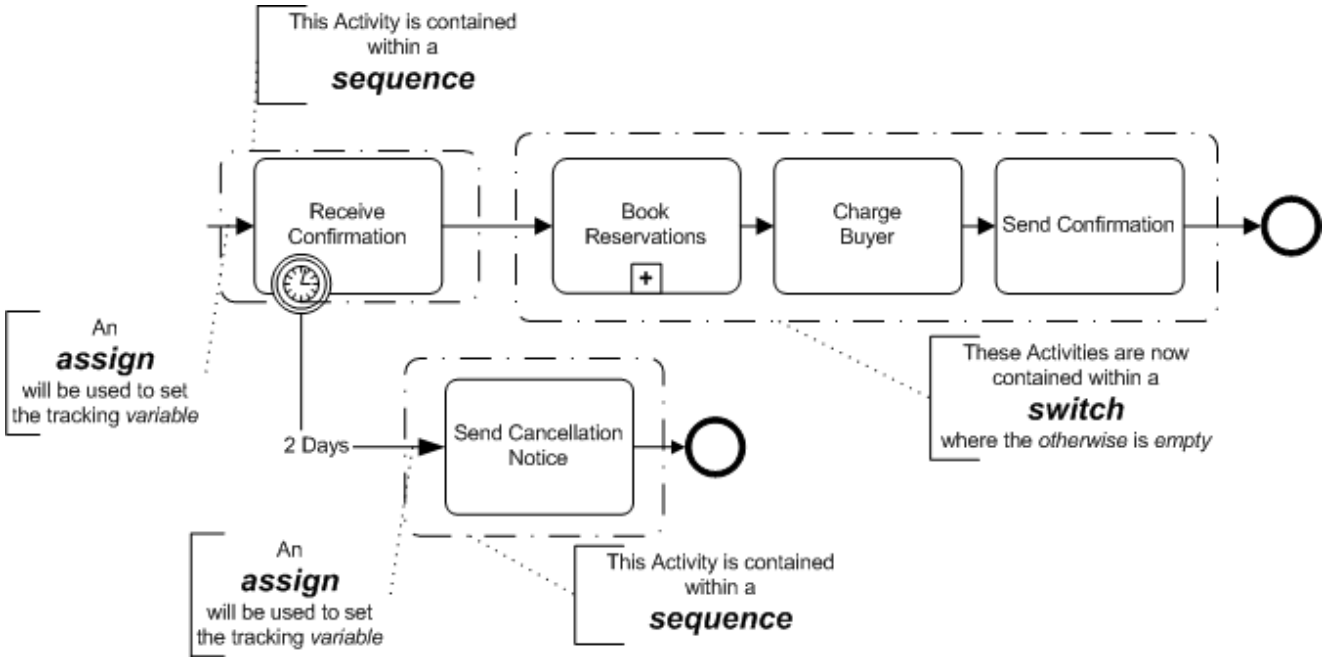

Figure 112 Exception Flow Merging back into the Normal Flow at the End Event

The Exception Flow Loops back into the Normal Flow Upstream

In this situation, the Exception Flow will loop back into the Normal Flow prior to the completion of the activity that is the source of the Exception Flow (see Figure 113). This is a particularly challenging mapping and cannot be done entirely within the confines of the original BPEL4WS *process*. Another process will need to be derived and then “spawned” until the original activity can be completed normally.

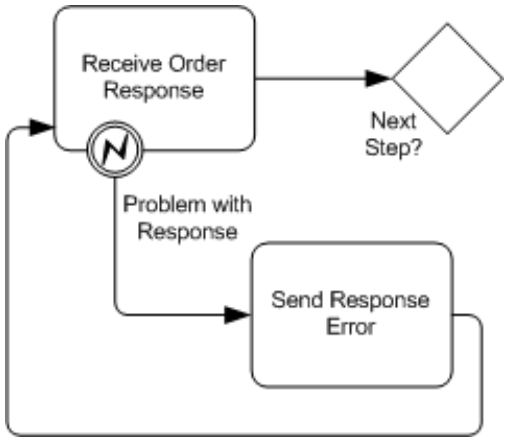

Figure 113 Example of Exception Flow Looping Back into the Normal Flow Upstream

This part of the Process will be modified at the BPEL4WS level so that the loop can be performed (through calling another *process*). If the flow moves to the *exceptionHandlers* activity, this means that the original activity will need to be performed again. Thus, the original activity will be duplicated in another *process* and the *exceptionHandlers* will contain a one-way *invoke* to “spawn” this other process (see Figure 114). In addition, the original process will wait

with a *receive* activity for a message from the derived process that the original activity has completed normally.

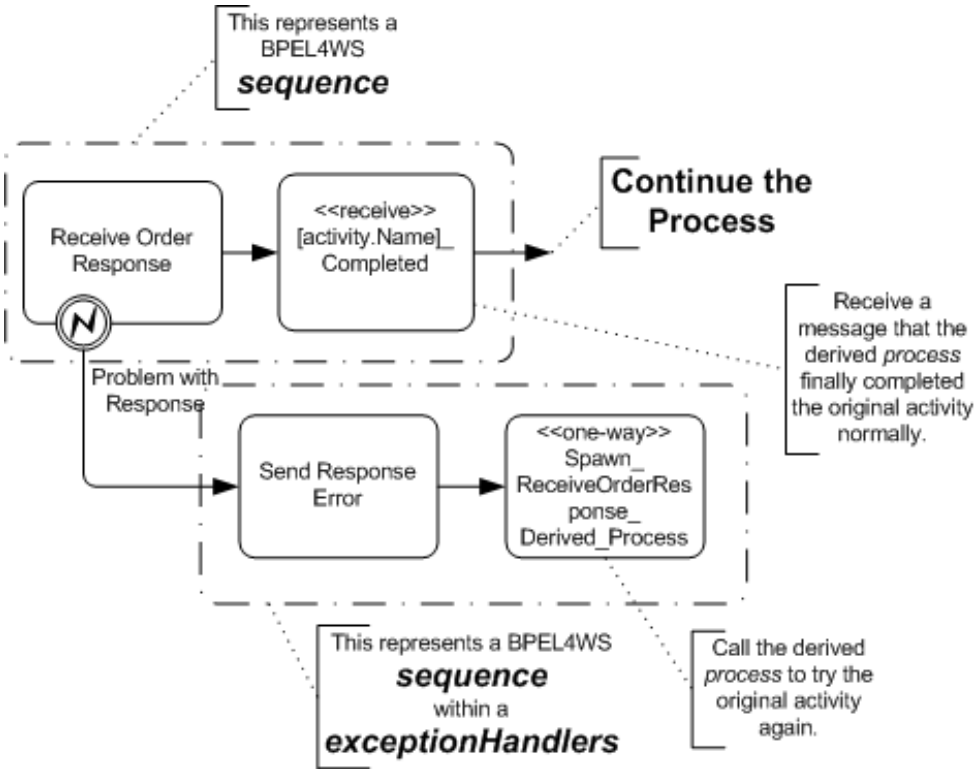

Figure 114 Example of Modification at BPEL4WS level to Handle the Loop

The derived process will be structure much like the corresponding section of the original process (see Figure 115). The mappings of the original activities, from the point of the BPMN Process where the Exception Flow loops into the Normal Flow to the point of the source of the Exception Flow, will be in the derived *process*. The same exceptionHandlers will be attached to the scope around the original activity. The *exceptionHandlers* will also contain a one-way *invoke* to “spawn” itself if the fault occurs again.

When the orginal activity finally completes normally, one-way *invoke* will be used to send a message back to the original *process* so that normall activities can continue.

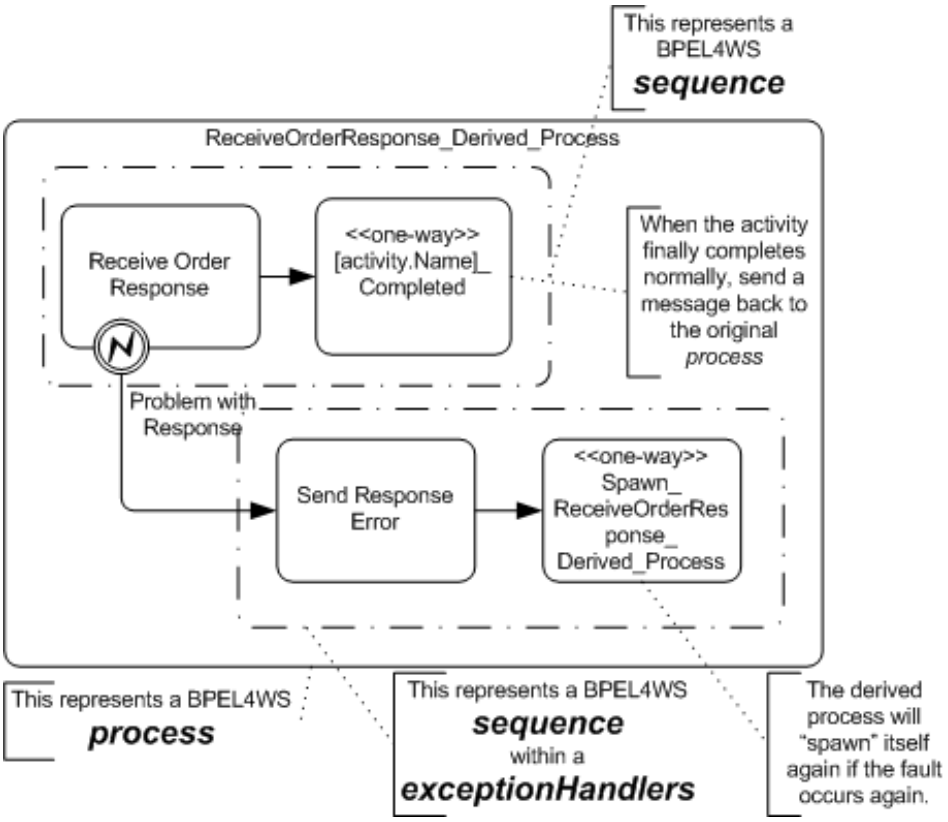

Figure 115 Example of a Derived Process to Handle the Looping

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The details of the Intermediate Event mappings was moved to the Intermediate Event mapping sections.
- The BPEL4WS mapping related to how the Exception Flow merges back into the Normal Flow was added.

6.14Compensation Association

The following table displays a set of mappings from a Compensation Association to BPEL4WS elements:

Compensation Association	Mapping to BPEL4WS
A Compensation Intermediate Event attached to an activity boundary	The mapping of the Compensation Event is described in the section entitled "Compensation Intermediate Events" on page 160. The mapping of the activity Associated with the Intermediate Event will follow the mapping rules defined in the section entitled "Task Mappings" on page 179 or in the section entitled "Sub-Process Mappings" on page 177 will be placed within the *compensationHandler* element.

Table 88 Exception Flow Mappings to BPEL4WS

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The details of the Compensation Intermediate Event mapping was moved to the Intermediate Event mapping section.
- The description of the mapping was updated to reflect that only a single activity can be associated with a Compensation Intermediate Event that is attached to the boundary of an activity.

6.15 Assignment Mapping

The following table displays a set of mappings from the variations of an Assignment expression to BPEL4WS elements:

Assignment	Mapping to BPEL4WS

To	The To attribute will map to the *to* element of the BPEL4WS *assign* activity. A variable and supporting WSDL message should have already be created for the Property used in for the Assignment To attribute. Thus, the structure of the to element will be as follows:
	If the Property is an attribute of a Process:
	`<to variable="[Process.Name]_ProcessData" part="[Property.Name]" />`
	If the Property is an attribute of an activity:
	`<to variable="[activity.Name]_ActivityData" part="[Property.Name]" />`

From	The From expression will map to the *from* element of the BPEL4WS *assign* activity. `<from expression="[From Expression]" />`

Table 89 Assignment Mappings to BPEL4WS

6.16 BPMN Supporting Type Elements

This section describes the mapping to BPEL4WS of a non-graphical elements that are part of BPMN. Messages, which are linked with Message Flow, do have impact on how many other BPMN elements are mapped to BPEL4WS.

Messages

The following are the mappings of a Message. These mappings are used to create a BPEL4WSE4WS XML file, plus a supporting WSDL supporting file. These mappings are used for a Start Event, End Event, Intermediate Event, and Task:

Attributes	Description
Name	The Name attribute maps to the *name* attribute of a BPEL4WS *variable* element. Note that the extra spaces and non-alphanumeric characters MUST be stripped from the Name to fit with the XML specification of the *name* attribute. Note that there may be two or more elements with the same name after the BPMN name has been stripped. The *messageType* attribute of the *variable* element refers to a WSDL *message* type definition. Thus, the *messageType* will share the same Name and a corresponding WSDL *message* must be created.
Properties	Each Properties of the BPMN Message will map to a *part* element of the WSDL *message*. The Name attribute of the Property will map to the *name* attribute of the *part*. The Type attribute of the Property will map to the *type* attribute of the *part*.

Table 90 Message Attributes

6.17 Determining the Extent of a BPEL4WS Structured Element

The structure and vocabulary of BPMN differs from BPEL4WS. BPMN allows flexible, and free-form methods of connectiing activities through Sequence Flow. Furthermore, BPMN is cyclical in that it allows Sequence Flows to connect to upstream objects so that a modeler can easily visualize looping situations. BPEL4WS has a much more structured form of creating a process flow. The *flow* activity in BPEL4WS does allow some flexibility with its *link* elements, but is acyclical. Thus, there is not going to be a one-to-one mapping of the BPMN elements to the BPEL4WS elements, without restricting the connection capability of BPMN.

This is particularly true of the BPEL4WS . In BPEL4WS, structure elements, such as *switch*, *pick*, and *while,* have a clear beginning and end. BPMN does not provide specific markers for the start and end of these elements. The exact configuration of the Sequence Flow connections will determine how the Process will be mapped to the BPEL4WS elements.

To determine the appropriate merging and joining points that are needed to construct the structured elements, 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.

A BPMN Gateway will usually indicate the start of a BPEL4WS structured element, but even this may not be one-to-one if there are loops involved. The end of the BPEL4WS structured element is even less obvious, since it could be marked by the convergence of Sequence Flow into most types of BPMN elements.

The following sections will describe how different BPMN configurations will map to the BPEL4WS structure elements and show how conceptual Tokens can be used to determine the extent of the BPEL4WS elements.

Identifying the Start of a BPEL4WS Element

The most basic structured element of BPEL4WS is the sequence.

- If the *process*, or the activity of a structured element (e.g., a *switch case*), contains more than one activity, then it is likely a *sequence* will be needed. Nearly any set of of activities connected by Sequence Flow, which is not going to be mapped to the contents of a *flow*, will be contained within a *sequence*. The *sequence* will envelope all the remaining elements to the extent of the context in which the *sequence* exists. E.g., the *sequence* will extend the length of the *process*, or the length of a *switch case*, etc.

For the other types of BPEL4WS elements, their extend is determined by tracing through the Process with conceptual Tokens:

- First the start of the BPEL4WSE4WS structured element (e.g., *flow*, *switch*, *pick*, etc.) must be identified. This is done by performing the mapping of the BPMN elements, starting with the Start Event or first element(s) if there is no Start Event, and proceeding down the Sequence Flow . The start of the structured element is usually a Gateway or if an activity has multiple outgoing Sequence Flow (see Figure 116 and Figure 118).
  - Note that some structured elements (mainly a *sequence*, but including others such as a *switch*) are needed for mapping a particular BPMN activity (as described in the sections above). In these cases, the extent of these structured elements are known.

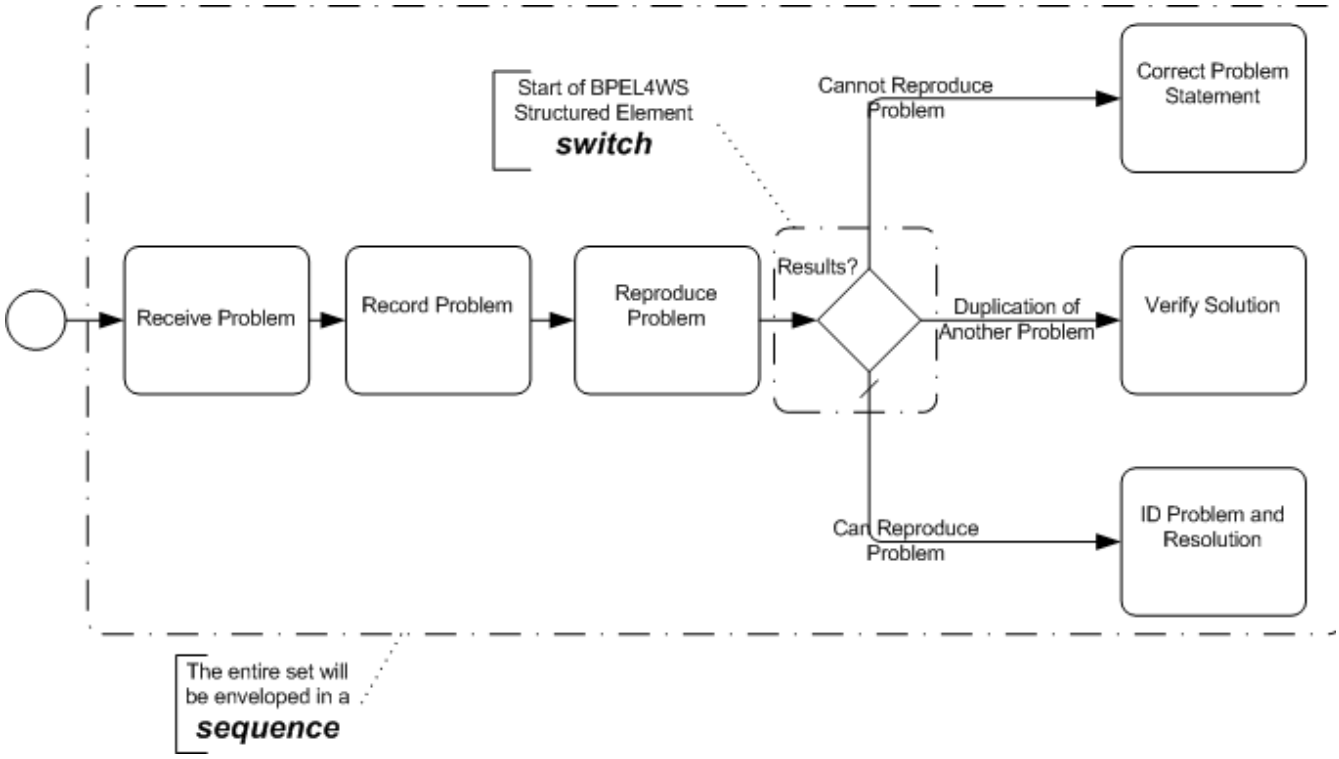

Figure 116 Identification of BPEL4WS structured element

- The number paths that make up the structured element MUST be determined. To do this, the all outgoing paths from the location of the structured element will be identified. A

conceptual Token can be used to trace the paths. The Tokens are given an ID that uniquely identifies the precedent of the structure element being determined and the number of paths being traced for that element (see Figure 117).

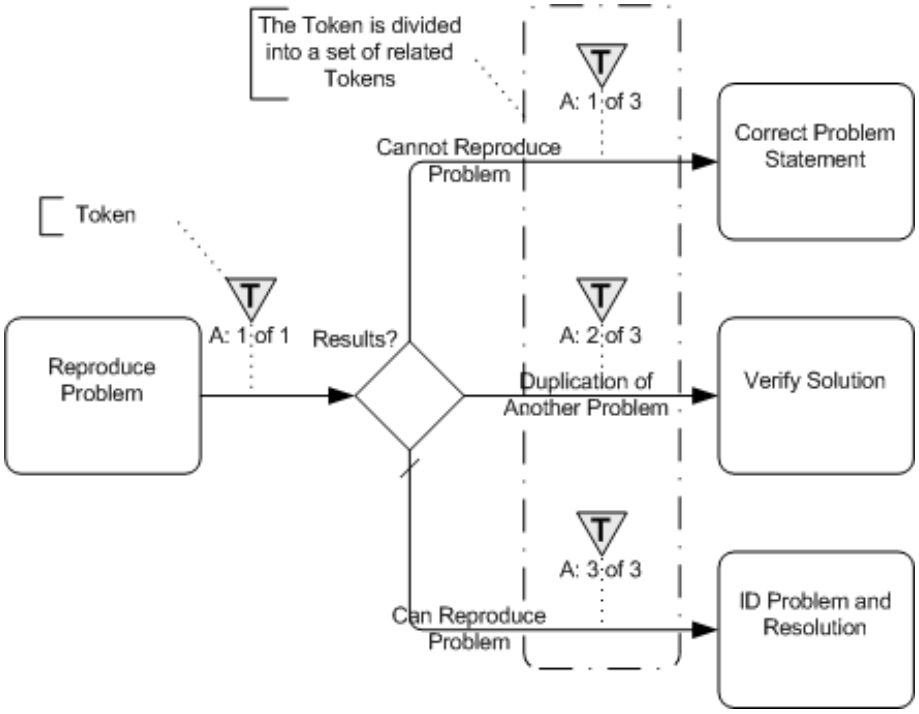

Figure 117 The Creation of Related Tokens

Finding the End of a BPEL4WS Element

The end of a BPEL4WS structured element will be found when all the paths, which were identified at the start of the element, have converged.

- Trace each path until there is a merge or join with all the other paths. When all the Tokens with the appropriate IDs arrive at the same BPMN object and can be recombined, then the structured element SHALL be closed (see Figure 118).

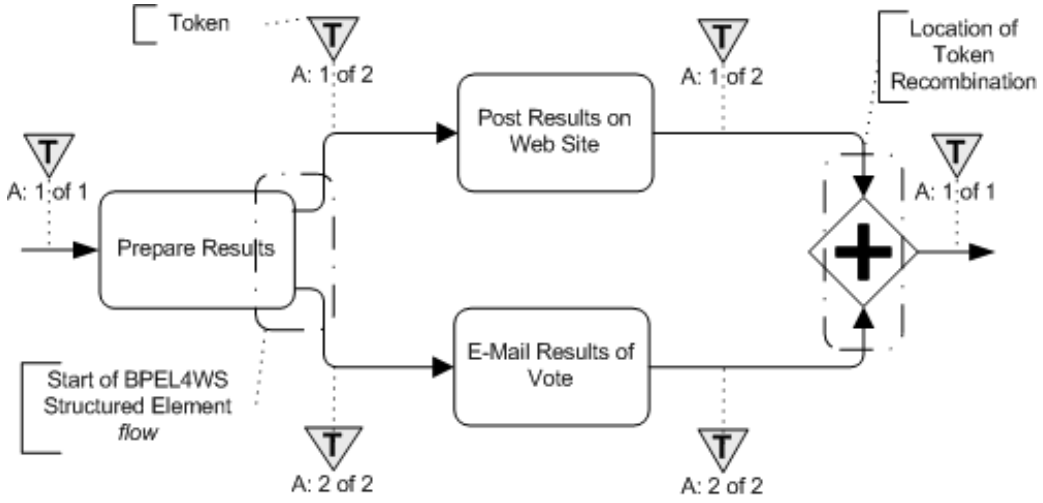

Figure 118 Example of Recombination of Tokens

- There MAY be partial recombinations of the Tokens prior to the final recombination. In this case, one Token will contain all the identities of the Tokens that have been merged (see Figure 119). Note that partial recombination of a Token creates another mapping issue that is described in the section entitled “BPMN Elements that Span Multiple BPEL4WS Sub-Elements” on page 212.

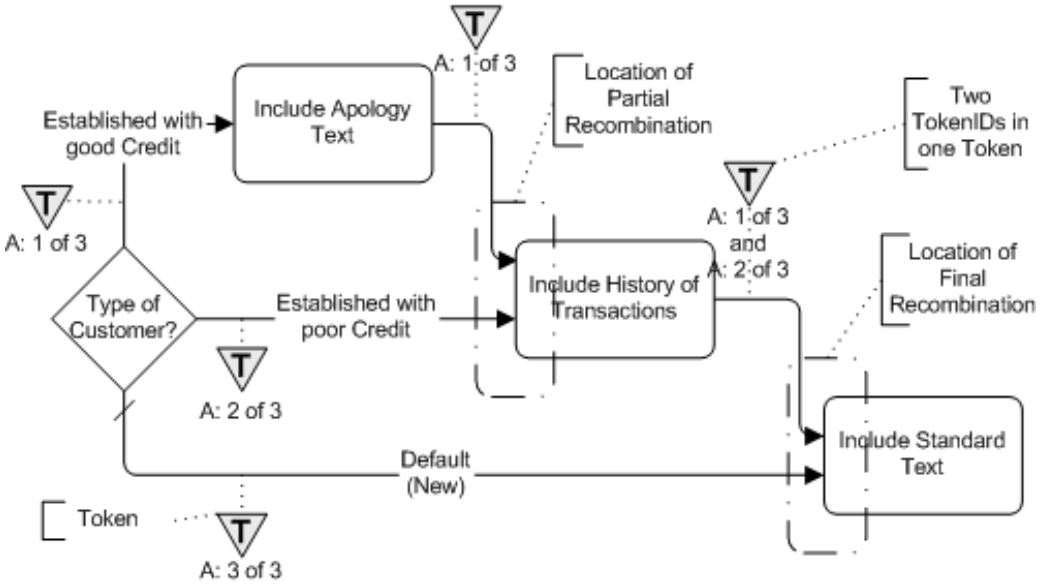

Figure 119 Example of Partial Recombination of Tokens

- End Events can be combined with other BPMN objects to complete the merging or joining of the paths of a BPEL4WS structured element (see Figure 120).

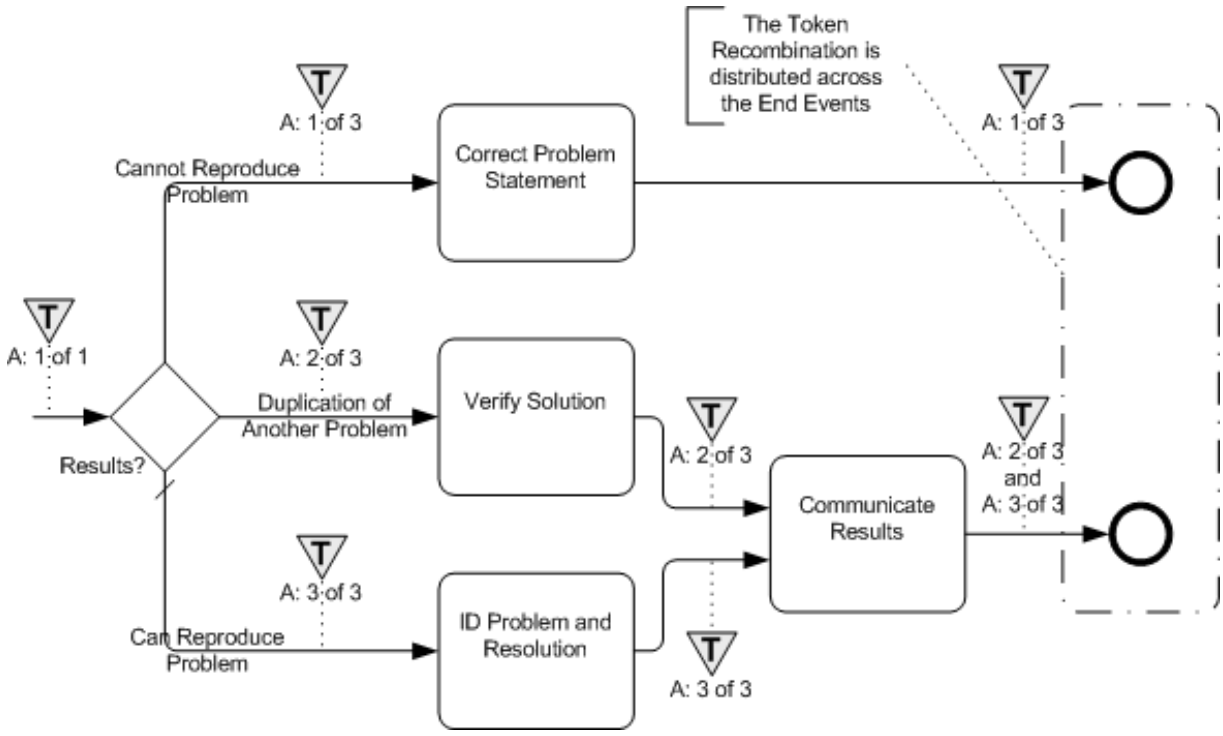

Figure 120 Example of Distributed Token Recombination

Nested Elements

Another structured element may occur before the first structure element is closed.

- If another structured element is encountered before all the paths are merged (see Figure 121), then the tracing of the first element MUST be stopped and the tracing of the paths of the second element MUST begin. The extent of the second element MUST be determined before the extent of the first element can be determined.
  - This process MUST be repeated if other structured elements are encountered during the tracing of any paths of structured elements.

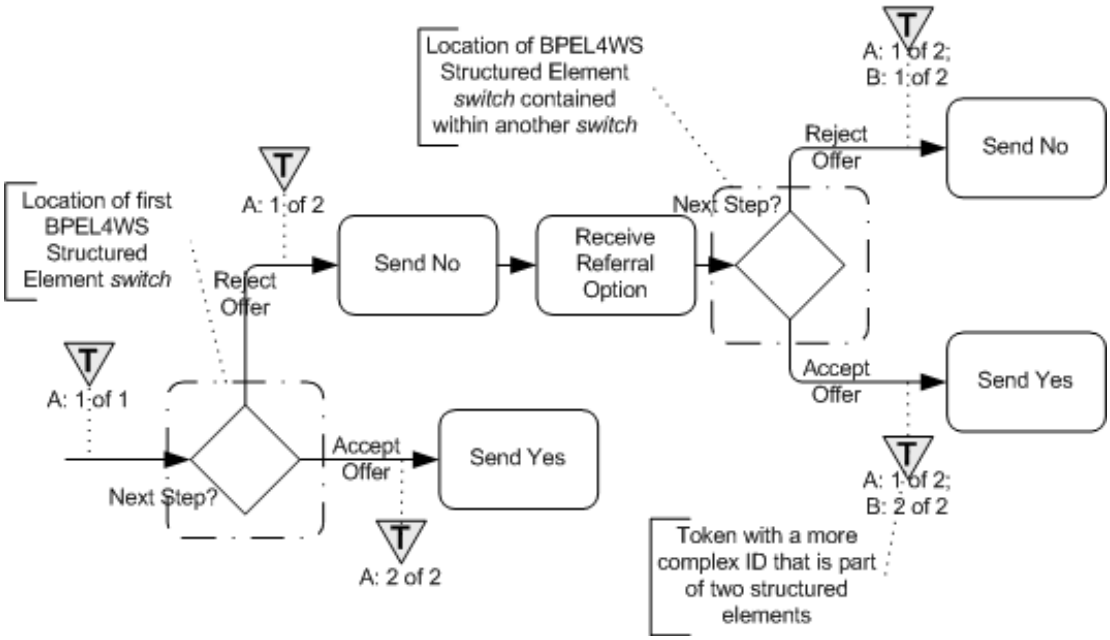

Figure 121 Example of nested BPEL4WS structural elements

Handling Loops

Loops are created when the flow of the Process moves from a downstream object to an upstream object.

- If one of the paths arrives at a BPMN object that is upstream from the source of the structured element, then this SHALL create a looping situation. How the loop is handled depends on the type structured element is being traced and how many paths are included in the element.

The following sections will describe the mapping for the different type of loop configurations.

Simple Loop From a Gateway

This type of loop is created by a Gateway that has only two outgoing Sequence Flow. One Sequence Flow continues downstream and the other loops back upstream (see Figure 122). Note that there might be intervening activities prior to when the Sequence Flow loops back upstream.

- This will map to a BPEL4WS *while* activity.
  - The Condition for the Squence Flow that loops back upstream will map to the *condition* of the *while*.
  - All the activities that span the distance between where the loop starts and where it ends, will be mapped and placed within the activity for the *while*, usually within a *sequence*.

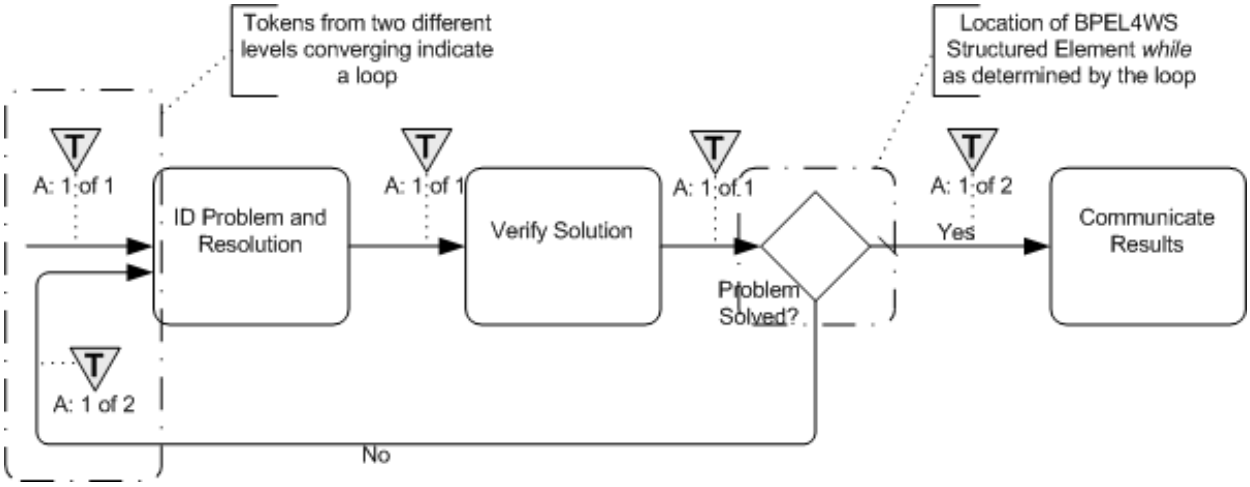

Figure 122 Example of a Loop from a Decision with Two Alternative Paths

Loop/Switch Combinations From a Gateway

This type of loop is created by a Gateway that has three or more outgoing Sequence Flow. One Sequence Flow loops back upstream while the others continue downstream (see Figure 123). Note that there might be intervening activities prior to when the Sequence Flow loops back upstream.

- This maps to both a BPEL4WS *while* and a *switch*. Both activities will be placed within a *sequence*, with the *while* preceding the *switch*.
  - For the *while*:
    - The Condition for the Squence Flow that loops back upstream will map to the *condition* of the *while*.
    - All the activities that span the distance between where the loop starts and where it ends, will be mapped and placed within the activity for the *while*, usually within a *sequence*.
  - For the *switch*:
    - For each additional outgoing Sequence Flow there will be a *case* for the *switch*. The details for mapping to a switch from a Gateway can be found in the section entitled “Gateways” on page 183.

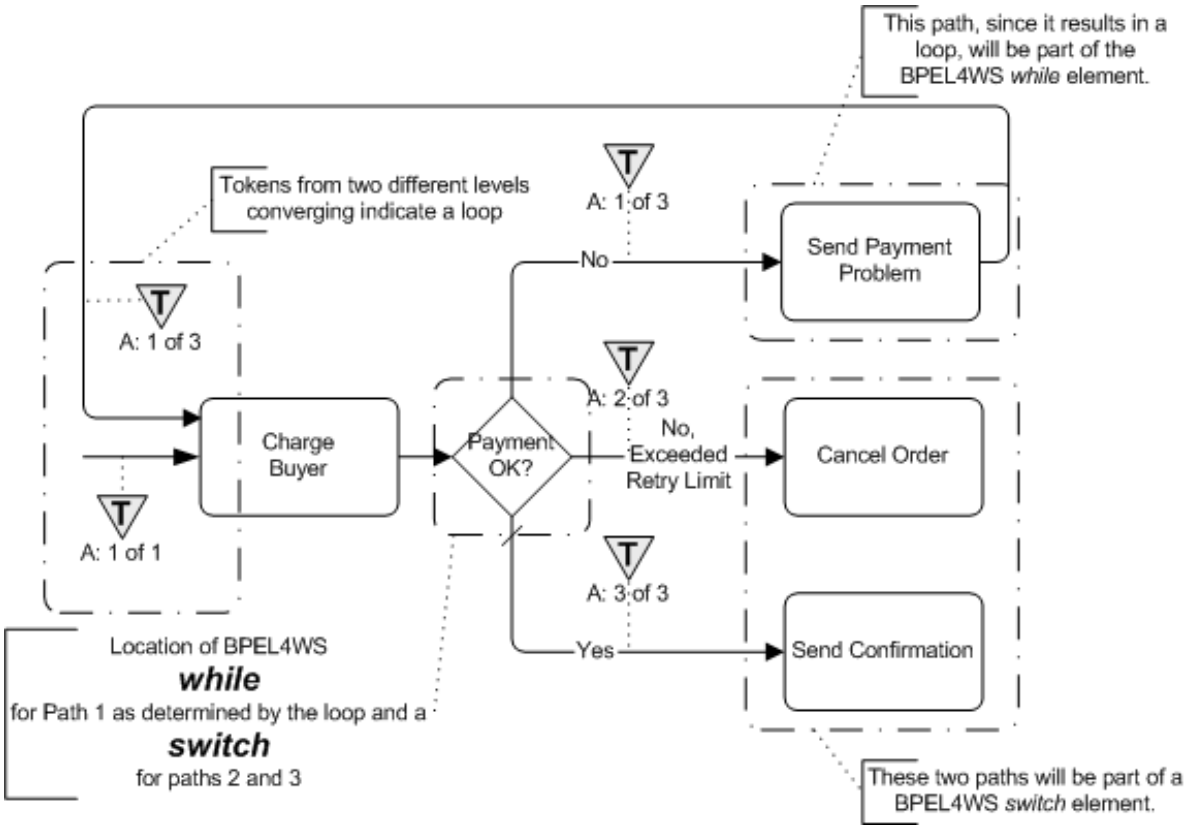

Figure 123 Example of a Loop from a Decision with more than Two Alternative Paths

Interleaved Loops

This is a situation where there at least two loops involved and they are not nested (see Figure 124). Multiple looping situations can map, as described above, if they are in a sequence or are fully nested (e.g., one *while* inside another *while*). However, if the loops overlap in a non-nested fashion, as shown in Figure 124, then the structured element *while* cannot be used to handle the situation. Also, since a *flow* is acyclic, it cannot handle the behavior either.

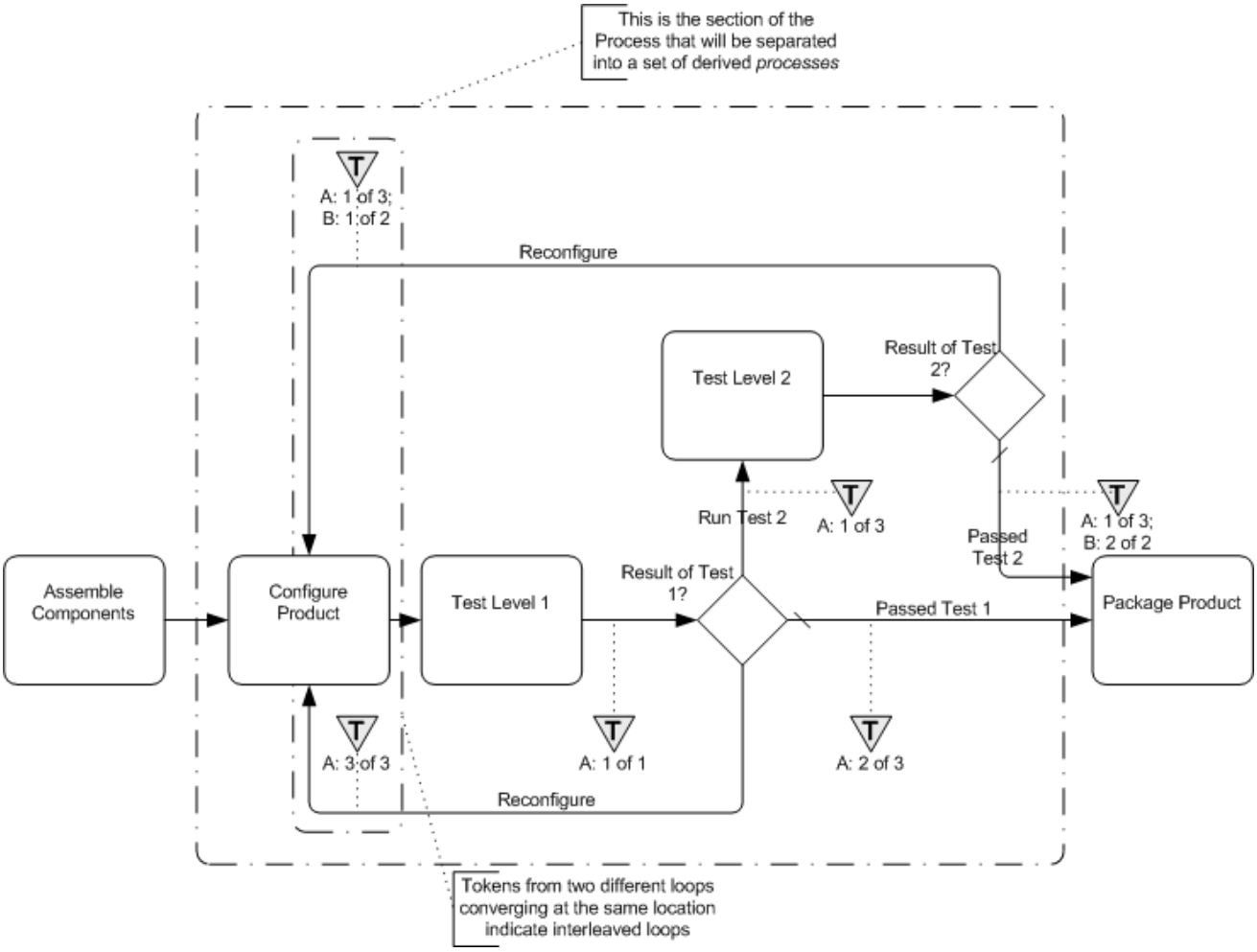

Figure 124 Example of Interleaved Loops

To handle this type of behavior, parts of the BPEL4WS *process* will have to be separated into one or more derived *processes* that are spawned from the main *process* and will also spawn or call each other (note that the examples below are using a spawning technique). Through this mechanism, the linear and structured elements of BPEL4WS can provide the same behavior that is shown through a set of cycles in a single BPMN diagram. To do this:

- The looping section of the process, where the loops first merge back (upstream) into the flow until all the paths have merged back to Normal Flow, shall be separated from the main *process* into a set of derived *processes* that will spawn each other until all the looping conditions are satisfied.
  - The section of the Process that is removed will be replaced by a (one-way) *invoke* to spawn the derived *process*, followed by a *receive* to accept the message that the looping sections have completed and the main *process* can continue (see Figure 125).
    - The name of the *invoke* will be in the form of:
      - “Spawn_[(loop target)activity.Name]_Derived_Process”
    - The name of the *receive* will be in the form of:
      - “[(loop target)activity.Name]_Derived_Process_Completed”

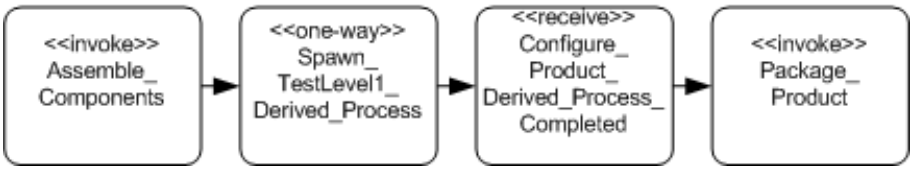

Figure 125 Example of the BPEL4WS Pattern for Substituting for the Derived *Process*

- For each location in the Process where a Sequence Flow connects upstream, there will be a separate derived BPEL4WS *process*.
  - The name of the derived process will be in the form of:
    - "[(loop target)activity.Name]_Derived_Process"
- All Gateways in this section will be mapped to *switch* elements, instead of *while* elements (see Figure 126).
- Each time there is a Sequence Flow that loops back upstream, the activity for the *switch case* will be a (one-way) *invoke* that will spawn the appropriate derived *process*, even if the *invoke* spawns the same *process* again.
  - The name of the *invoke* will the same as the one describe above.
- At the end of the derived *process* a (one-way) *invoke* will be used to signal the main process that all the derived activity has completed and the main *process* can continue.
  - The name of the *invoke* will be in the form of:
    - "[(loop target)activity.Name]_Derived_Process_Completed"

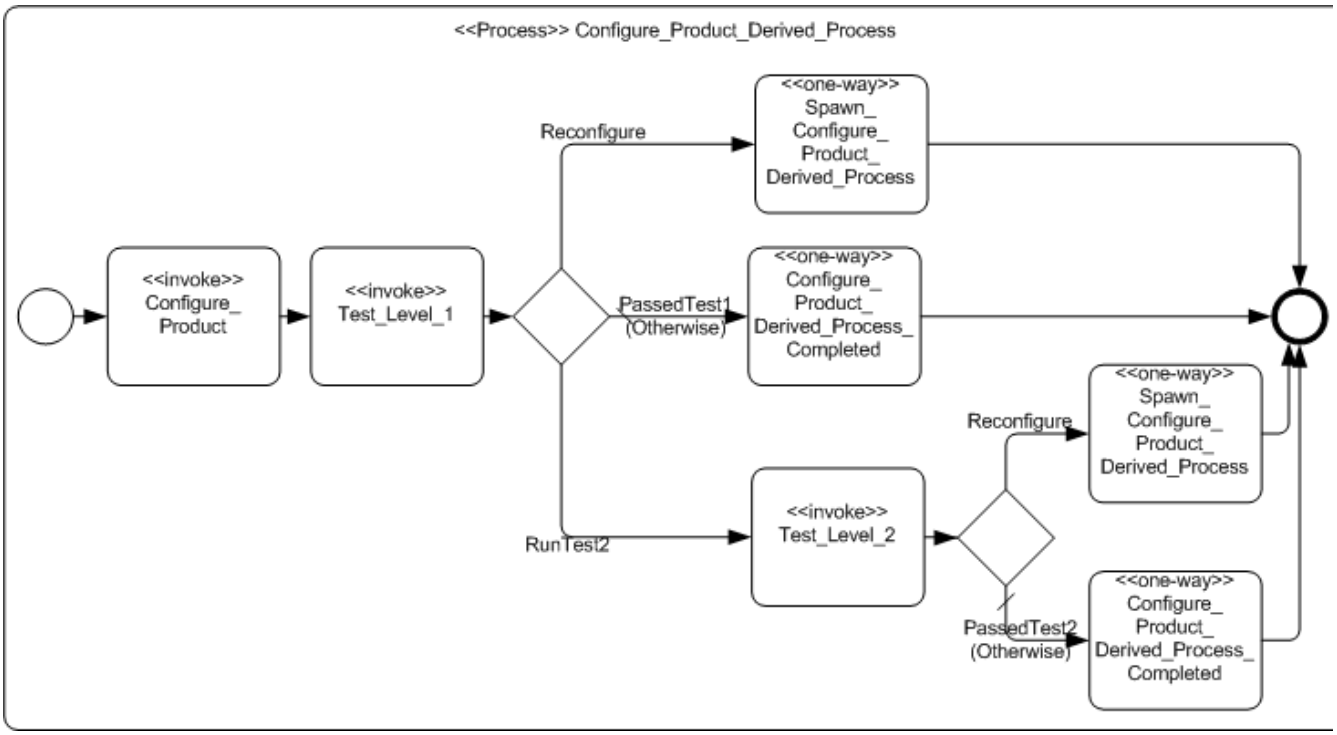

Figure 126 Example of a BPEL4WS Pattern for the Derived Process

Infinite Loops

This type of loop is created by a Sequence Flow that loops back without an intervening Gateway to create alternative paths (see Figure 127). While this may be a modeling error most of the time, there may be situations where this type of loop is desired, especially if it is placed within a larger activity that will eventually be interrupted.

- This will map to a *while* activity.
  - The condition of the while will be set to an expression that will never evaluate to True, such as condition ”1 = 0.”
  - All the activities that span the distance between where the loop starts and where it ends, will be mapped and placed within the activity for the *while*, usually within a *sequence*.

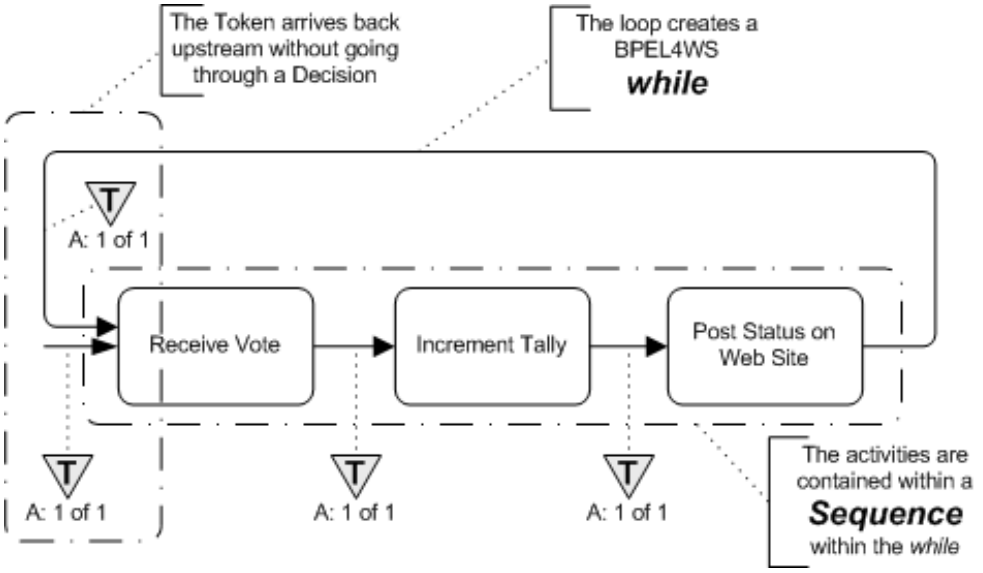

Figure 127 Example: An Infinite Loop

Handling Link Events as Go To Objects

As was seen in Figure 85, Figure 86, and Figure 87, Link Intermediate Events can be used as Go To Objects. The basic impact of using them in such a way is that they are a substitute using a single, longer Sequence Flow to make the same connection between two objects. Thus, the mapping to BPEL4WS should be done by considering them as just a single Sequence Flow. This means that the Intermediate Events are not mapped to any BPEL4WS element. Instead a conceptual Sequence Flow will be used, with the Source and Target of that Sequence Flow being the Source of the Sequence Flow going into the Source Link Event and the Target of the Sequence Flow coming out of the Target Link Event (see Figure 129). The mapping at this point can done using all the mapping consideration described in this Chapter.

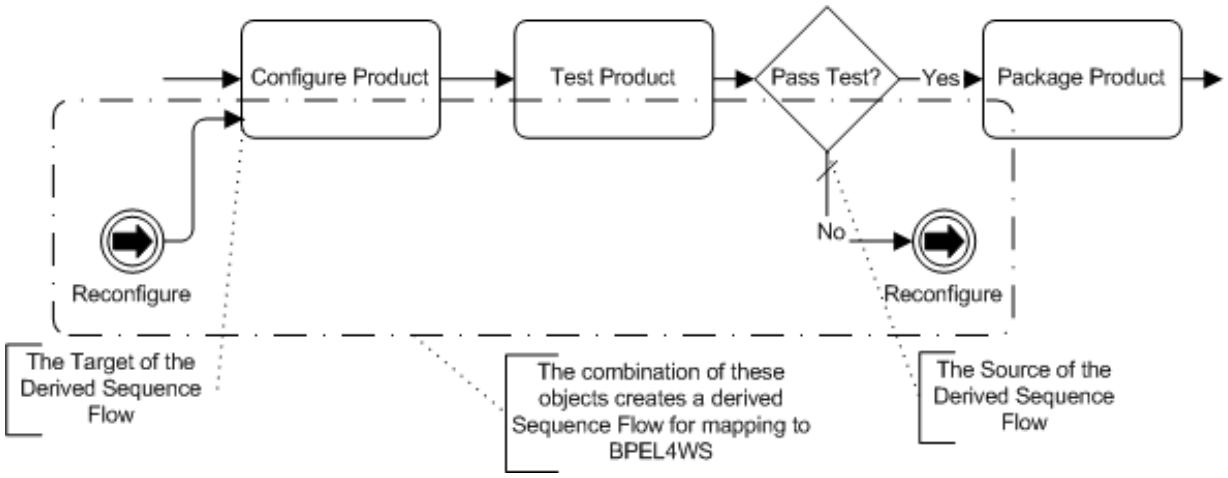

Figure 128 Example: A Pair of Go To Link Events are Treated as a Single Sequence Flow

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- The details of this section were added.

6.17.1 BPMN Elements that Span Multiple BPEL4WS Sub-Elements

Figure 119 is repeated below in Figure 129 to illustrate how BPMN objects may exist in two separate sub-elements of a BPEL4WS structured element at the same time. Since BPMN allows free form connections of activities and Sequence Flow, it is possible that two (or more) Sequence Flow will merge before all the Sequence Flow that map to a BPEL4WS structure element have merged. The sub-elements of a BPEL4WS structured elements are also self contained and there is no cross sub-element flow. For example, the *cases* of a *switch* cannot interact; that is, they cannot share activities. Thus, one BPMN activity will need to appear in two (or more) BPEL4WS structured elements.

There are two possible mechanisms to deal with the situation.

- First, the activities are simply duplicated in all appropriate BPEL4WS elements.
- Second, the activities that need to be duplicated can be removed from the main process and placed in a derived process that is called (*invoked*) from all locations in the BPEL4WS elements as required.
    - The name of the derived process will be in the form of:
        - “[(target)object.Name]_Derived_Process”

In Figure 129 displays this issue with an example. In that example, two Sequence Flow merge into the “Include History of Transactions” Task. However, the Decision that precedes the Task has three (3) alternatives. Thus, the Decision maps to a BPEL4WS *switch* with three (3) *cases*. The three cases are not closed until the “Include Standard Text” Task, downstream. This means that the “Include History of Transactions” Task will actually appear in two (2) of the three (3) *cases* of the *switch*.

Note: the use of a BPEL4WS *flow* will be able to handle the behavior without duplicating activities, but a *flow* will not always be available for use in these situations, particularly if a BPEL4WS *pick* is required.

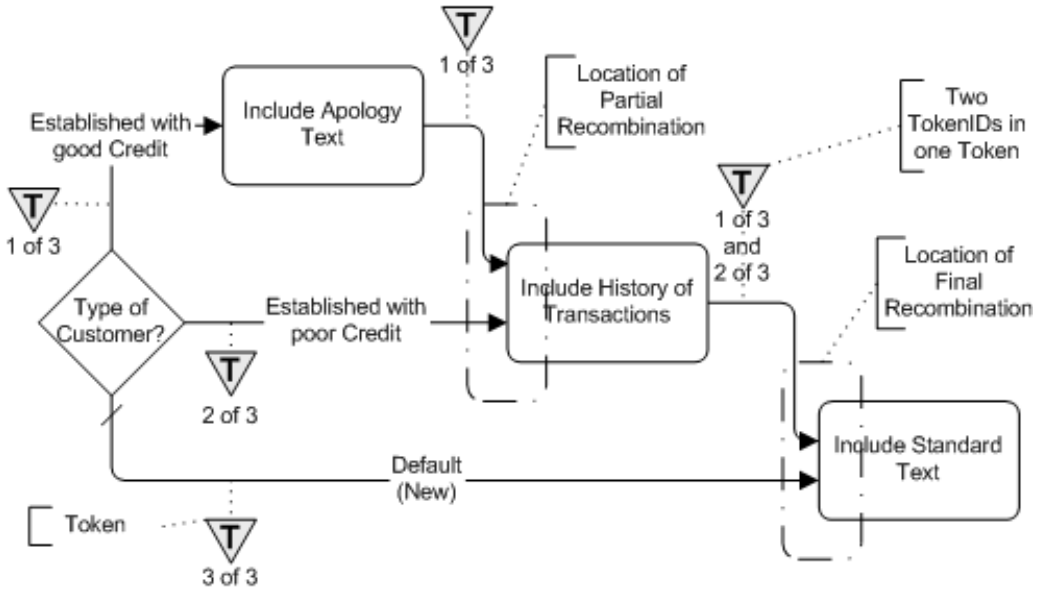

Figure 129 Example: Activity that spans two paths of a BPEL4WS Structured Element

Example 8 displays some sample BPEL4WS code that reflects the portion of the Process that was just discussed and is shown in Figure 129. Note that there are two *invoke* elements that have the same *name* attribute (“IncludeHistoryofTransactions”).

```
<!--Continue with the process-->
<switch name="TypeofCustomer">
  <!-- name="Established with Good Credit" -->
  <case condition="bpws:getVariableProperty(ProcessData,CreditType)>”Yes, Good””>
    <invoke name="IncludeApologyText" ...>
    <!--This also exists in the other case-->
    <invoke name="IncludeHistoryofTransactions" ...>
  </case>
  <!--name="Established with poor Credit" -->
  <case condition="bpws:getVariableProperty(ProcessData,CreditType)>”Yes, Poor””>
    <!--This also exists in the other case-->
    <invoke name="IncludeHistoryofTransactions" ...>
  </case>
  <!--name="Default (New)" -->
  <otherwise>
    <!--Nothing happens here-->
    <empty/>
  </otherwise>
</switch>
<invoke name="IncludeStandardText" ...>
<!--Continue with the process-->
```

Example 8 Example: BPMN Elements that Span Multiple BPEL4WS Sub-Elements

Changes Since 1.0 Draft Version

These are the changes since the last publicly release version:

- This section was added.