7. BPMN by Example

This section will provide an example of a business process modeled with BPMN. The process that will be described is a process that BPMI has been using to develop this notation. It is a process for resolving issues through e-mail votes (see Figure 121). This Process is small, but fairly complex and will provide examples for many of the features of BPMN. There are some unusual features of this business process, such as infinite loops. Although not a typical process, it will help illustrate that BPMN can handle simple and unusual business processes and still be easily understandable for readers of the Diagram. The sections below will isolate segments of the Process and highlight the modeling features as the workings of the Process is described. In addition, samples of BPEL4WS code are provided to demonstrate how a BPMN Diagram maps to BPEL4WS.

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

The Process starts with Timer Start Event that is set to trigger the Process every Friday (see Figure 122).

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

7.1.1 Mapping to BPEL4WS

BPEL4WS processes must begin with a receive activity for instantiation (i.e., it “bootstraps” itself). The “E-Mail Voting Process” is scheduled to start every Friday as shown by the Timer Start Event. Therefore, an additional Process will have to be created and implemented that will run indefinitely and will send a starting message with the list of Issues to the “E-Mail Voting Process” every Friday. Figure 123 shows this Process as starting that the beginning of the Working Group and continuing until the end of the Working Group. Even this Process needs a message to be sent to it to signal the start of the Working Group. There may be another Process defined that sends that message, but that Process is not shown here. In addition, the mapping from the Starter Process to BPEL4WS is not shown here.
Within the main Process (see Figure 122), the “Receive Issue List” Task will map to a BPEL4WS receive that has its createInstance attribute set to “yes.” This will receive a starting message and start the process.

This receive will be placed inside a sequence since other activities follow the activity. The message to be received will contain all the variable parts that will be used in the process and their initialized values.

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

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

The “Review Issue List” Task will map to a BPEL4WS invoke. This TaskType is User, which means that the invoke will be synchronous and an outputVariable included.

Mapping an Exclusive Gateway (Decision)

The “Any Issues Ready?” Exclusive Gateway (Decision) will map to a BPEL4WS switch.

The Gate for the “No” Sequence Flow will map to the otherwise case of the switch. This otherwise will only contain an empty activity since there is nothing to do and the Process is over.

Note that empty does not have any corresponding activity in the BPMN Diagram, but is derived through the Diagram configuration.

The Gate for the “Yes” Sequence Flow will map to other case for the switch. This case will have a condition that checks the number of issues that are ready. This case will handle the remainder of the Process that is shown in Figure 121.
This is done because the switch is a block structure and needs a definitive ending point and since the otherwise is connected to the end of the Process, then the end of the Process is the ending point that the case must use. The actual activities that make up the rest of the Process will be distributed among a set of BPEL4WS processes instead of all being within the case. The case will only contain an invoke that will call another process (as a web service). The distribution of the Process activities is due to the overall Diagram configuration that includes three upstream Sequence Flow that define some interleaving loops.

The Impact of Interleaved Loops

If the loop shown in this section of the model were merely a simple loop, and perhaps the only loop, then a BPEL4WS while would be used to handle the loop. In this situation, though, the looping is handled through a set of derived processes that are accessed by invoking them (as a web service). There would no specific Diagram element to represent these derived processes; indeed, a modeler would not want to create a set of related Processes to handle complex looping. While an execution engine can easily handle a complex set of language documents and elements, a business person developing and monitoring this process will want to see the Process in an easy-to-read format (such as BPMN) that contains the information in a more comprehensive, less distributed format. Refer to the section entitled “Interleaved Loops” on page 213 for details about how interleaved loops are mapped to BPEL4WS.

In this example, all derived processes will be named “[(target of loop) activity.Name]_Derived_Process.” Any naming scheme will work as long as all the processes have unique names.

- Thus, to handle the rest of the Process, a derived nested process named “Discussion_Cycle_Derived_Process” is created and then
- A BPEL4WS invoke is used to access this process from the “Yes” case of the “Any issues ready?” switch.

We shall see that later in the Process the same process is accessed through another invoke, marking the source of the loop.

All the sub-processes and derived processes in the BPEL4WS documents must be started with the receive of a message and then a reply to send a message back to the calling process.

- This means that a receive will be the first activity inside a sequence that will be the main activity of these processes. These receive activities will have the createInstance attribute set to “Yes.” A named “internal,” a portType name “processPort” will be created to support all of these process to process communications. The WSDL operations that will support these communications will all be named “call_<process name>” (as noted above, the processes are actually spawned).

The “Discussion Cycle” Sub-Process shown in Figure 122 will continue the sequence (after the instantiating receive) for the “Discussion_Cycle_Derived_Process” process.

- Since “Discussion Cycle” is a Sub-Process it will map to a separate BPEL4WS process that is access through an invoke.

Mapping an Activity Loop Condition

The “Discussion Cycle” Process has a loop marker. In this situation, the looping mechanism is simple. The attributes of the Sub-Process will tell us the details. The “Discussion Cycle” Sub-
Process's relevant attributes are: LoopType: "Standard"; LoopCondition: DiscussionOver = "FALSE"; and TestTime: "After."

- This means that the invoke that calls the process will be enclosed within a while activity when the BPEL4WS is derived.
- The LoopType will map to a BPEL4WS while. The LoopCondition of the Process (as shown above) will map to the “DiscussionOver = False” will be the condition for the while.

The default value for the “DiscussionOver” property is False, thus an activity within the Sub-Process will have to change it to True before the while loop is over. The logical opposite of the expression that is shown in the Sub-Process attributes is used since the EvaluationCondition property is “after.” However, a while will test the condition prior to running the activity within. This means that to insure that the activity is always performed at least once (to mimic the behavior of an “until”) a LoopCounter variable will always be added to a the while condition for an BPMN activity that has its TestTime attribute set to “After.”

- The LoopCounter will be initialized to zero, and an assign will be added to the sequence prior to the while element.
- The activity of the while will be changed to a sequence, with the invoke for the Sub-Process, which is
  - Followed by an assign that will increment the LoopCounter variable, inside the sequence.

We will look into the details of the “Discussion Cycle” Sub-Process in the section entitled “The First Sub-Process” on page 208.
BPEL4WS Sample for the Beginning of the Process

Example 4 displays some sample BPEL4WS code that reflects the portion of the Process that was just discussed and is shown in Figure 122.

```xml
<process name="EMailVotingProcess">
  <!-- The Process data is defined first-->
  <sequence>
    <!-- This starts the beginning of the Process. The process that sends the starting message every Friday is related to the Timer Start Event and is not shown here.-->
    <receive partnerLink="Internal" portType="tns:processPort" operation="receiveIssueList" variable="processData" createInstance="Yes"/>
    <invoke name="ReviewIssueList" partnerLink="Internal"
      portType="tns:internalPort" operation="sendIssueList"
      inputVariable="processData" outputVariable="processData"/>
    <switch name="Anyissuesready">
      <!-- name="Yes" -->
      <case condition="bpws:getVariableProperty(ProcessData,NumIssues)>0">
        <!-- A chunk of this process is separated into a derived process so that it can be called from a complex loop. Thus, it is called from here and from "Collect Votes" as part of a loop-->
        <invoke name="Discussion_Cycle_Derived_Process" partnerLink="Internal"
          portType="tns:processPort"
          operation="call_Discussion_Cycle_Derived_Process" inputVariable="processData"
          outputVariable="processData"/>
      </case>
      <!-- name="No" -->
      <otherwise>
        <!-- This is one of the two ways to the end of the Process-->
        <empty/>
      </otherwise>
    </switch>
  </sequence>
</process>

<process name="Discussion_Cycle_Derived_Process">
  <!-- The Process data is defined first-->
  <sequence>
    <!-- The first Sub-Process has a loop condition, so it is within a while-->
    <receive partnerLink="Internal" portType="tns:processPort"
      operation="call_Discussion_Cycle_Derived_Process" variable="processData"
      createInstance="Yes"/>
    <assign name="Discussion_Cycle_initialize_loopCounter">
      <copy>
        <from expression="0"/>
        <to variable="Discussion_Cycle_loopCounter" part="loopCounter" />
      </copy>
    </assign>
    <invoke name="Discussion_Cycle" partnerLink="Internal"
      portType="tns:processPort" operation="call_Discussion_Cycle"
      inputVariable="processData" outputVariable="processData"/>
  </sequence>
</process>
```
Example 4 BPEL4WS Sample for Beginning of E-Mail Voting Process
7.2 The First Sub-Process

Figure 124 shows the details of the “Discussion Cycle” as an Expanded Sub-Process.

The Sub-Process starts with a Task for the Issue List Manager to send an e-mail to the working group that a set of Issues are now open for discussion through the working group’s message board. Since this Task sends a message to an outside Participant (the working group members), an outgoing Message Flow is seen from the “Discussion Cycle” Sub-Process to the “Voting Members” Pool in Figure 121. Basically, the working group will be discussing the issues for one week and proposing additional solutions to the issues. After the first Task, three separate parallel paths are followed, which are synchronized downstream. This is shown by the three outgoing Sequence Flow for that activity.

The top parallel path in the figure starts with a long-running Task, “Moderate E-mail Discussion,” that has a Timer Intermediate Event attached to its boundary. Although the “Moderate E-Mail Discussion” Task will never actually be completed normally in this model, there must be an outgoing Sequence Flow for the Task since Start and End Events are being used within the Process. This Sequence Flow will merged with the Sequence Flow that comes from the Timer Intermediate Event. A merging Exclusive Gateway is used in this situation because the next object is a joining Parallel Gateway (the diamond with the cross in the center) that is used to synchronize the three parallel paths. If the merging Gateway was not used and both Sequence Flow connected to the joining Gateway, the Process would have been stuck at the joining Gateway that would wait for a Token to arrive from each of the incoming Sequence Flow.

The middle parallel path of the fork contains an Intermediate Event and a Task. A Timer Intermediate Event used in the middle of the Process flow (not attached to the boundary of an activity) will cause a delay. This delay is set to 6 days. The “E-Mail Discussion Deadline Warning” Task will follow. Again, since this Task sends a message to an outside Participant, an
outgoing Message Flow is seen from the “Discussion Cycle” Sub-Process to the “Voting Members” Pool in Figure 121.

The bottom parallel path of the fork contains more than one object, first of which is Task where the issue list manager checks the calendar to see if there is a conference call this week. The output of the Task will be an update to the variable “ConCall,” which will be true or false. After the Task, an Exclusive Gateway with its two Gates follows. The Gate for labeled “default” flows directly to an merging Exclusive Gateway, for the same reason as in the top parallel path. The Gate for the “Yes” Sequence Flow will have a condition that checks the value of the “ConCall” variable (set in the previous Task) to see if there will be a conference call during the coming week. If so, the Timer Intermediate Event indicates delay, since all conference calls for the working group start at 9am PDT on Thursdays. The Task for moderating the conference call follows the delay, which is followed the merging Gateway.

The merging Gateways in the top and bottom paths and the “E-Mail Discussion Deadline Warning” Task all flow into a joining Gateway. This Gateway waits for all three paths to complete before the Process flows to the next Task, “Evaluate Discussion Progress.” The issue list manager will review the status of the issues and the discussions during the past week and decide if the discussions are over. The DiscussionOver variable will be set to TRUE or FALSE, depending on this evaluation. If the variable is set to FALSE, then the whole Sub-Process will be repeated, since it has looping set and the loop condition will test the DiscussionOver variable.

### 7.2.1 Mapping to BPEL4WS

- The “Discussion Cycle” Sub-Process itself maps to a BPEL4WS process.

Because it is a Sub-Process within a higher-level Process (the “E-Mail Voting” Process), it is invoked from the higher-level Process. The invoke sends a message from one (higher-level) BPEL4WS process to the other (lower-level) process for instantiation.

- This means that the process being instantiated must have a receive to start it off.
- The process being instantiated must have a reply to end it, since it is being synchronously called.

The receive and reply are not actually shown in the BPMN Diagram, but it is derived from this invoke relationship of “Discussion Cycle” Process being a Sub-Process to the “E-Mail Voting” Process.

- Given this, the activity of the BPEL4WS process will be a sequence with the derived receive as the first activity.

The Diagrams elements of Figure 124 will determine the remaining activity(ies) of the sequence.

- The Sub-Process starts off with a Task, which maps to a BPEL4WS invoke (which is after the automatically generated receive that starts the process).

- After the first Task, three separate parallel paths are followed. The forking of the flow marks the start of a BPEL4WS flow. The flow will extend until the Parallel Gateway, which joins the three paths.
The Upper Parallel Path

In the upper parallel path of the fork, the Task, “Moderate E-mail Discussion,” has a Timer Intermediate Event attached to its boundary. Because of this,

- the Task is placed in its own scope with a faultHandlers.
- The Task itself is mapped to a BPEL4WS invoke (synchronous), and will be placed in a lower-level flow, for reasons described below.

The Timer Intermediate Event must be set up to create a fault at the appropriate time. To do this,

- An eventHandlers is added to the scope.
  - An onAlarm is included in the eventHandlers and the for attribute is set to the duration that is defined in the Timer Intermediate Event.
  - The onAlarm contains a throw with a fault name after the Intermediate Event with “_Exit” appended.

The catch of a faultHandlers will be triggered by the fault generated by the above throw. Since the Timer Intermediate Event leads direction to the Exclusive Gateway, there is no specific activity that must be performed in response the to time-out. The main purpose is to exit the Task. Thus,

- A faultHandlers is added to the scope.
  - The catch in the faultHandlers has a faultName set to Intermediate Event with “_Exit” appended.
  - the catch will contain an empty activity.

The Middle Parallel Path

The middle parallel path of the fork has a string of two objects.

- Even though this series of objects appears in the middle of a BPEL4WS flow, they will be place within a sequence element.

In these situations, the sequence will continue until there is a location in the Diagram where there are multiple incoming Sequence Flow. When more than one Sequence Flow converge it marks the end of a BPEL4WS structure (as determined by structures that have been created by upstream objects). In this case, the Parallel Gateway also marks the end of the higher-level flow. The sequence will be listed in the higher-level flow without a source sub-element. This means that the sequence will be instantiated when the higher-level flow begins since it has no dependencies on any other activity. The sequence will have two activities:

- First, the Timer Intermediate Event used in this situation will map to a BPEL4WS wait (set to 6 days).
- Second, the “E-Mail Discussion Deadline Warning” Task will map to an invoke that follows the wait. In addition, this invoke can be asynchronous since a response is not required. This means that the outputVariable will not be included.

This middle path of the fork could have been configured in BPEL4WS without a sequence and with links instead. This is an example of a situation where a BPMN configuration may derive two possible BPEL4WS configurations. Since both BPEL4WS configurations will handle the
appropriate behavior, it is up to the implementation of the BPMN to BPEL4WS derivation to determine which configuration will be used. BPMN does not provide any specific recommendation in these situations. However, the lower parallel path of the Process can also be modeled with a sequence or with links, and, to show how links would be used, this section of the Process will be mapped to elements in a flow that have dependencies specified by links.

**The Lower Parallel Path**

The lower parallel path of the fork has a number of objects and, as just described above, will be mapped to BPEL4WS elements connected with links. The path also contains a Decision, which can map to a switch, as will happen later in the process, but in this situation the Decision is mapped to links controlled by transitionConditions.

- The first object is a Task, which will map to an invoke (synchronous) that has two source elements referring to two of the links. There are two Target links because the Task is followed by the Gateway with its two Gates. This is done instead of a switch with a case and an otherwise.

- The ConditionExpression for the Gate labeled “Yes” will map to the source element’s transitionCondition. The expression checks the value of the “ConCall” property (set in the previous Task) to see if there will be a conference call during the coming week.

- The Gate labeled “No” has a condition of default. For a switch, this would map to the otherwise element. However, since a switch is not being used, the source element’s transitionCondition must be the inverse of all the other transitionConditions for the activity. The expression of the other source will be placed inside a “not” function.

The invoke will be listed in the higher-level flow without a source sub-element. This means that the invoke will be instantiated when the higher-level flow begins since it has no dependencies on any other activity. The remaining elements of the higher-level flow will have a source element. Thus, they will not be instantiated until the source of the link has completed.

- The “Yes” Gate from the Gateway leads to a Timer Intermediate Event, which will map to a wait.
  - The for element of the wait will set to for 9am PDT on the next Thursday.
  - This wait will have a target element that corresponds to the target element from the previous invoke.
  - The wait will also have a target element to link to the following invoke.

- The “No” Gate from the Gateway leads to a merging Exclusive Gateway, which means that nothing is expected to happen down this path. Thus, this will map to an empty.
  - This empty will have a target element that corresponds to the target element from the previous invoke.

- The Task for moderating the conference call follows the wait, which will map to an invoke (synchronous).
  - This invoke will have a target element that corresponds to the target element from the previous wait.

There are three link elements in the flow:
7.2 The First Sub-Process

7.2.1 Mapping to BPEL4WS

- One link will have a source of the first invoke and a target of the wait.
- One link will have a source of the first invoke and a target of the empty.
- One link will have a source of the first wait and a target of the last invoke.

As mentioned above, the Parallel Gateway marks the end of the flow.

Finally, there will be a reply at the end of the sequence that corresponds to the initial receive and lets the parent process know that the (sub) process has been completed.

After the Parallel Paths are Joined

The Task “Evaluate Discussion Progress” is intended to occur only when all the parallel paths have completed, and thus, it will

- Map to an invoke that follows the closing of the flow.

BPEL4WS Sample for the First Sub-Process

Example 5 displays some sample BPEL4WS code that reflects the portion of the Process as described above and shown in Figure 124.

```xml
<process name="Discussion_Cycle">
<!-- The Process data is defined first-->
<sequence>
   <receive partnerLink="Internal" portType="tns:processPort" operation="call_Discussion_Cycle" variable="processData" createInstance="Yes"/>
   <invoke name="AnnounceIssuesforDiscussion" partnerLink="WGVoter" portType="tns:emailPort" operation="sendDiscussionAnnouncement" inputVariable="processData"/>
   <flow>
      <links>
         <link name="CheckCalendarforConferenceCalltoWaituntilThursday,9am"/>
         <link name="CheckCalendarforConferenceCalltoEmpty"/>
         <link name="WaituntilThursday9amtoModerateConferenceCallDiscussion"/>
      </links>
      <!-- This is the first of the three paths of the fork. -->
      <scope>
         <invoke name="ModerateEmailDiscussion" partnerLink="internal" portType="tns:internalPort" operation="sendDiscussion" inputVariable="processData" outputVariable="processData"/>
         <faultHandlers>
            <catch faultName="7Days_Exit"/>
            <empty/>
            </catch>
         </faultHandlers>
         <eventHandlers>
            <onAlarm for="tns:OneWeek">
               <throw faultName="7Days_Exit"/>
            </onAlarm>
         </eventHandlers>
      </scope>
      <!-- This is the second of the three paths of the fork. -->
   </flow>
</sequence>
</process>
```
Example 5 BPEL4WS Sample of “Discussion Cycle” Sub-Process Details

### 7.3 The Second Sub-Process

Figure 125 shows the next section of the Process, which includes the expanded details of the “Collect Votes” Sub-Process.
This part of the process starts out with a Task for the issue list manager to send out an e-mail to announce to the working group, and the voting members in particular, which lets them know that the issues are now ready for voting. Since this Task sends a message to an outside Participant (the working group members), an outgoing Message Flow is seen from the “Announce Issues” Task to the “Voting Members” Pool in Figure 121. This Task is also a target for one of the complex loops in the Process.

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

The first branch of the fork leads to a Decision that determines whether or not a conference call will occur during the upcoming week, after the Working Group’s schedule has been
checked. Basically, if there was a call last week, then there will not be a call this week and vice versa. The appropriate variable that was updated in the “Discussion Cycle” Process will be used again.

The second and third branches forks work the same way as the similar activities in the “Discussion Cycle” Sub-Process, except that the “Moderate E-Mail Discussion” Task does not have a Timer Intermediate Event attached. This is not necessary since the whole Sub-Process is interrupted after 7 days through the Intermediate Event attached to the Sub-Process boundary. The “E-Mail Vote Deadline Warning” Task sends a message to an outside Participant (the working group members), thus, an outgoing Message Flow is seen from the “Collect Votes” Sub-Process to the “Voting Members” Pool in Figure 121.

The fourth branch of the fork is rather unique in that the Diagram uses a loop that does not utilize a Decision. Thus, it is, as it is intended to be, an infinite loop. The policy of the working group is that voting members can vote more than once on an issue; that is, they can change their mind as many times as they want throughout the entire week. The first Task in the loop receives a message from the outside Participant (the working group members), thus, an incoming Message Flow is seen from the “Voting Members” Pool to the “Collect Votes” Sub-Process in Figure 121. The Timer Intermediate Event attached to the boundary of the Sub-Process is the mechanism that will end the infinite loop, since all work inside the Sub-Process will be ended when the time-out is triggered. All the remaining work of the Process is conducted after the time-out and flows from the Timer Intermediate Event.

Figure 125 shows that there are Two Tasks that follow the time-out. First, a Task will prepare all the voting results, then a Task will send the results to the voting members. A Document Object, “Issue Votes,” is shown in the Diagram to illustrate how one might be used, but it will not map to anything in the execution languages. The remaining activities of the Process will be described in the next section.

7.3.1 Mapping to BPEL4WS

The Loops Cause Derived Sub-Processes

- The first Task of this section of the Process is also a target for one of the complex loops in the Process, thus, it will map to an invoke (asynchronous) that is placed inside another derived process (“Announce_Issues_Derived_Process”).

- This derived process will be invoked from “Discussion_Cycle_Derived_Process,” after the “Discussion Cycle” process has been completed, as part of the normal flow and then from another part of the Process as part of the looping flow.

- Thus, “Announce_Issues_Derived_Process” will require a (instantiation) receive to accept the message from “Discussion_Cycle_Derived_Process” and from “Issues_wo_Majority_Derived_Process” (as we shall see later).

- The “Collect Votes” Sub-Process follows the Task, but is also a target of one of the looping Sequence Flows. Thus, it will also be set inside a derived process (“Collect_Votes_Derived_Process”).

- In addition, “Collect_Votes_Derived_Process” will require a (instantiation) receive to accept the message from “Announce_Issues_Derived_Process” and from the fault handler of “Collect Votes” (as we shall see later).

- The “Collect Votes” Sub-Process will map to an invoke (asynchronous) and the details will be in a process referenced through the invoke.
The BPEL4WS Sample of the Derived Sub-Processes

Example 6 shows sample BPEL4WS code that defines the two derived processes.

```xml
<process name="Announce_Issues_Derived_Process">
<!-- This starts the middle section of the Process and is call from
the first time and then from “Collect Votes” during a loop-->
<!-- The Process data is defined first-->
<sequence>
    <receive partnerLink="Internal" portType="tns:processPort"
        operation="call_Announce_Issues_Derived_Process"
        variable="processData" createInstance="Yes"/>
    <invoke name="AnnounceIssuesforVote" partnerLink="WGVoter" portType="tns:emailPort"
        operation="sendVoteAnnouncement" inputVariable="processData"/>
    <invoke name="Collect_Votes_Derived_Process" partnerLink="Internal"
        portType="tns:processPort"
        operation="call_Collect_Votes_Derived_Process" inputVariable="processData"/>
    <reply partnerLink="Internal" portType="tns:processPort"
        operation="call_Announce_Issues_Derived_Process"
        variable="processData" createInstance="Yes"/>
</sequence>
</process>

<process name="Collect_Votes_Derived_Process">
<!-- this calls the second Sub-Process and then continues. It is also
called from “Collect Votes” as part of a loop-->
<!-- The Process data is defined first-->
<sequence>
    <receive partnerLink="Internal" portType="tns:processPort"
        operation="call_Collect_Votes_Derived_Process"
        variable="processData" createInstance="Yes"/>
    <invoke name="Collect_Votes" partnerLink="Internal" portType="tns:processPort"
        operation="call_Collect_Votes" inputVariable="processData"/>
    <reply partnerLink="Internal" portType="tns:processPort"
        operation="call_Collect_Votes_Derived_Process"
        variable="processData" createInstance="Yes"/>
</sequence>
</process>
```

Example 6 BPEL4WS Sample that sets up the Access for the Second Sub-Process

The Paths of the Sub-Process

The “Collect Votes Sub-Process is basically a set of four parallel paths that extend from the beginning to the end of the Sub-Process.

- Thus, the activity for the process will be a flow.

The Upper Parallel Path

The first branch of this Sub-Process is basically the same as the upper parallel of the previous Sub-Process. An invokes, a wait, and an empty will be created. In addition, three links will be created to handle the dependencies between the elements, including the branching created by the Exclusive Gateway. Refer to the section entitled “The Lower Parallel Path” on page 211 for the details of the mappings.
The Middle Two Parallel Paths

The second and third branches of the fork are rather straightforward mappings of:

- Two Tasks to *invokes* (one synchronous and one asynchronous), and
- A Timer Intermediate Event to a *delay*.
- In addition, one *link* is created so that one of the *invokes* will wait for the *delay*.

The Lower Parallel Path

The fourth branch of the fork is the location the infinite loop.

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

Exiting the Second Sub-Process

To exit out of the infinite loop and the whole “Collect Votes” Sub-Process,

- A *scope* will be wrapped around the main *flow* of the *process*, which will include an *eventHandlers* and a *faultHandlers*.

The Timer Intermediate Event must be set up to create a *fault* at the appropriate time. To do this,

- An *onAlarm* will be placed inside the *eventHandlers*. The timing of the *onAlarm* will be determined by the time setting in the Intermediate Event.
  - Within the *onAlarm*, a *throw* will a fault name after the Intermediate Event with “_Exit” appended.
- The *catch* element of the *faultHandlers* will be triggered by the *fault* generated by the above *throw*.
  - The *activity* for the *catch* will be a *sequence* and will be the source of all the remaining activities of the Process, since all the remaining Sequence Flow begins from the Timer Intermediate Event.
    - The first three Tasks, as shown in the figure, will map to *invokes*. The latter two will be placed within a *flow*.

The Document Objects shown in the figure is not mapped into BPEL4WS. The remainder of the Process will be described in the next section.
Example 7 shows sample BPEL4WS code that defines the “Collect Votes” Sub-Process.

```xml
<process name="Collect_Votes">
  <!--This is a nested process for the E-Mail Voting collection. It consists of
  an all and a faultHandlers (for a timeout). The all will never complete
  normally since there is an infinite loop inside. The timeout is intended to
  be the normal way of ending the process-->
  <sequence>
    <receive partnerLink="Internal" portType="tns:processPort"
      operation="call_Collect_Votes" variable="processData" createInstance="Yes"/>
  </sequence>
  <scope>
    <flow>
      <links>
        <link name="Delay6daysfromVoteAnnouncementtoEMailVoteDeadlineWarning"/>
        <link name="CheckCalendarforConferenceCalltoWaituntilThursday9am"/>
        <link name="CheckCalendarforConferenceCalltoEmpty"/>
        <link name="WaituntilThursday9amtoModerateConferenceCallDiscussion"/>
      </links>
      <!-- This is the first of the four paths of the fork. -->
      <invoke name="CheckCalendarforConferenceCall" partnerLink="internal"
        portType="tns:internalPort" operation="receiveCallSchedule"
        inputVariable="processData" outputVariable="processData">
        <source linkName="CheckCalendarforConferenceCalltoWaituntilThursday9am"
          transitionCondition="bpws:getVariableProperty(processData,conCall)=true"/>
        <source linkName="CheckCalendarforConferenceCalltoEmpty"
          transitionCondition="not(bpws:getVariableProperty(processData,conCall)=true)"/>
      </invoke>
      <!-- name="Yes" -->
      <wait name="WaituntilThursday9am" for="P6DT9H">
        <target linkName="CheckCalendarforConferenceCalltoWaituntilThursday9am"/>
        <source linkName="WaituntilThursday9amtoModerateConferenceCallDiscussion"/>
      </wait>
      <invoke name="ModerateConferenceCallDiscussion" partnerLink="internal"
        portType="tns:internalPort" operation="sendConCall"
        inputVariable="processData" outputVariable="processData">
        <target linkName="WaituntilThursday9amtoModerateConferenceCallDiscussion"/>
      </invoke>
      <!-- name="otherwise" -->
      <empty>
        <target linkName="CheckCalendarforConferenceCalltoEmpty"/>
      </empty>
      <!-- This is the second of the four paths of the fork. -->
      <invoke name="ModerateEMailDiscussion" partnerLink="internal"
        portType="tns:internalPort" operation="sendDiscussion"
        inputVariable="processData" outputVariable="processData"/>
      <!-- This is the third of the four paths of the fork. -->
      <wait name="Delay6daysfromVoteAnnouncement" for="P6D">
        <source linkName="Delay6daysfromVoteAnnouncementtoEMailVoteDeadlineWarning"/>
      </wait>
    </flow>
  </scope>
</process>
```
Example 7 BPEL4WS Sample of the Second Sub-Process

```
<invoke name="EMailVoteDeadlineWarning" partnerLink="WGVoter"
    portType="tns:emailPort" operation="sendVoteWarning"
    inputVariable="processData">
    <target linkName="Delay6daysfromVoteAnnouncementtoEMailVoteDeadlineWarning"/>
</invoke>

<!--This is the fourth of the four paths of the fork. This branch of the all is intended to be an infinite loop that is eventually interrupted by the Time Out. This is necessary since any voter can change their vote until the deadline. -->
<while condition="1=0">
    <sequence>
        <receive name="ReceiveVote" partnerLink="WGVoter" portType="tns:emailPort"
            operation="receiveVote" variable="processData"/>
        <invoke name="IncrementTally" partnerLink="internal"
            portType="tns:internalPort" operation="sendReceiveTotal"
            inputVariable="processData" outputVariable="processData"/>
    </sequence>
</while>
</flow>
<eventHandlers>
    <onAlarm for="P7D">
        <throw faultName="7days_Exit"/>
    </onAlarm>
</eventHandlers>
<faultHandlers>
    <catch faultName="7days_Exit">
        <!-- The BPMN Diagram shows that the Timer Intermediate Event connects directly to the rest of the Process. Thus, they will show up in this activity set. -->
        <sequence>
            <invoke name="PrepareResults" partnerLink="internal"
                portType="tns:internalPort" operation="sendReceiveResults"
                inputVariable="processData" outputVariable="processData"/>
            <flow>
                <invoke name="PostResultsOnWebsite" partnerLink="internal"
                    portType="tns:internalPort" operation="postVotingResults"
                    inputVariable="processData"/>
                <invoke name="EMailResultsOfVote" partnerLink="WGVoter"
                    portType="tns:emailPort" operation="sendVotingResults"
                    inputVariable="processData"/>
            </flow>
        </sequence>
    </catch>
</faultHandlers>
</scope>
<reply partnerLink="Internal" portType="tns:processPort"
    operation="call_Collect_Votes" variable="processData" createInstance="Yes"/>
</sequence>
</process>
```
7.4 The End of the Process

Figure 126 shows the last section of the Process, which includes a complex set of Decisions and loops.

This segment of the Process continues from where the last segment left off (as described in the section above). It contains four Decisions that interact with each other and create loops to upstream activities.
The first Decision, “Did Enough Members Vote?,” is necessary since two-thirds of the voting members are required to approve any solution to an issue. If less than two-thirds of the voting members cast votes, which sometimes happens, the issues can’t be resolved. This Decision flows to another Decision for both of its Alternatives. The “No” Alternative is followed by the “Have the Members been Warned?” Decision. If a voting member misses a vote, they are warned. If they miss a second vote, they lose their status as a voting member and the voting percentages are recalculated through a Task (“Reduce number of Voting Members and Recalculate Vote”). If they haven’t yet been warned, then a warning is sent and the voting week is repeated.

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

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

7.4.1 Mapping to BPEL4WS

As mentioned above, the entire contents of this segment follow a Timer Intermediate Event, which means they are contained in the faultHandlers of the scope within the “Collect Votes” process.

- Each of the Decisions in this section will map to a BPEL4WS switch.

The First Decision

The first Decision, “Did Enough Members Vote?,” flows to another Decision for both of its Alternatives.

- Thus, each of the switch cases will contain another switch.

The “No” Alternative is followed by the “Have the Members been Warned?” Decision.

- Each Alternative from this Decision is followed by a Task, which maps to Invokes (one synchronous and the other asynchronous).

The “No (default)” Alternative leads to a loop.

- This looping is handled by using an invoke (asynchronous) to the “Collect_Votes_Derived_Process” process, which was created just for the purpose of this loop (since it is in the context of a more complex looping situation).

Notice that the “Issues w/o Majority?” Decision can be reached through the alternative paths from two different Decisions. This creates a situation that has two impacts on the Mapping to Execution Languages. First, it creates a section of the Process in which the Alternatives from two Decisions overlap. This is possible in a graph-structured Diagram like BPMN, but in a
block-structured (and acyclic) language like BPEL4WS, these two sections cannot overlap because they have different block boundaries. This means that this section must be repeated in some way in both of the appropriate switch case activities. All these elements could be actually duplicated or they can be separated into a derived process and then invoked from the appropriate place. The later method was used in this example (see Example 8 and Example 9).

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

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

A different type of looping mechanism is required. This method requires the creation of a set of derived processes that can reference each other and also themselves. In this way, a block-structured language can simulate a set of interleaving loops (as seen in a graph-structured Diagram).

- Thus, in this BPMN example, derived processes were created to mark places where loops can be targeted within the BPEL4WS code from the “downstream” elements.
- A BPEL4WS invoke is used to re-perform activities that had already been executed in the process.

**BPEL4WS Sample for the End of the Process**

Example 8 displays the BPEL4WS code for first part of the end of the “E-Mail Voting Process.”

```xml
<catch property="tns:OneWeek" type="duration">
  <!--The BPMN Diagram shows that the Timer Intermediate Event connects directly to the rest of the Process. Thus, they will show up in this activity set-->
  <!--The first two actions are not shown--> 
  <sequence>
    <invoke name="PrepareResults" partnerLink="internal" portType="tns:internalPort" operation="sendReceiveResults" inputVariable="processData" outputVariable="processData"/>
    <invoke name="EMailResultsofVote" partnerLink="WGVoter" portType="tns:emailPort" operation="sendVotingResults" inputVariable="processData"/>
  </sequence>
</catch>
```

```xml
<!--This segment of the code is within the context of the “Collect Votes” nested process-->
<!--The BPMN Diagram shows that the Timer Intermediate Event connects directly to the rest of the Process. Thus, they will show up in this activity set-->
<!--The first two actions are not shown--> 
<sequence>
  <invoke name="PrepareResults" partnerLink="internal" portType="tns:internalPort" operation="sendReceiveResults" inputVariable="processData" outputVariable="processData"/>
  <invoke name="EMailResultsofVote" partnerLink="WGVoter" portType="tns:emailPort" operation="sendVotingResults" inputVariable="processData"/>
</sequence>
```
Example 8 Sample BPEL4WS code for the last section of the Process

```xml
<switch name="DidEnoughMembersVote">
  <!-- name="No" -->
  <case condition="bpws:getVariableProperty(ProcessData,NumVoted) >
    (0.7)*(bpws:getVariableProperty(ProcessData,NumVWGM))">
    <switch name="Have the members been warned">
      <!-- name="Yes" -->
      <case condition="bpws:getVariableProperty(ProcessData,VotersWarned)=true">
        <sequence>
          <invoke name="Reduce numberOf Voting Members and Recalculate Vote" partnerLink="Internal" portType="tns:internalPort" operation="sendReceiveNumVoters" inputVariable="processData" outputVariable="processData"/>
          <!-- Some elements of the process were separated into a derived process since they would have been repeated. They would have been repeated because they are arrived by alternative paths that do not close a set of alternative paths. -->
          <invoke name="Issues wo Majority Derived Process" partnerLink="Internal" portType="tns:processPort" operation="call_Issues_wo_Majority_Derived_Process" inputVariable="processData" outputVariable="processData"/>
        </sequence>
      </case>
      <!-- name="No (otherwise)" -->
      <otherwise>
        <sequence>
          <invoke name="Reannounce Vote with warning to voting members" partnerLink="WG Voter" portType="tns:emailPort" operation="sendReannounceVote" inputVariable="processData" outputVariable="processData"/>
          <invoke name="Collect Votes Derived Process" partnerLink="Internal" portType="tns:processPort" operation="call_Collect_Votes_Derived_Process" inputVariable="processData" outputVariable="processData"/>
        </sequence>
      </otherwise>
    </switch>
  </case>
  <!-- name="Yes (otherwise)" -->
  <otherwise>
    <!-- Some elements of the process were separated into a derived process since they would have been repeated. They would have been repeated because they are arrived by alternative paths that do not close a set of alternative paths. -->
    <invoke process="Issues wo Majority Derived Process" partnerLink="Internal" portType="tns:processPort" operation="call_Issues_wo_Majority_Derived_Process" inputVariable="processData" outputVariable="processData"/>
  </otherwise>
</switch>
</case>
<!-- name="Yes (otherwise)" -->
<otherwise>
  <!-- Some elements of the process were separated into a derived process since they would have been repeated. They would have been repeated because they are arrived by alternative paths that do not close a set of alternative paths. -->
  <invoke process="Issues wo Majority Derived Process" partnerLink="Internal" portType="tns:processPort" operation="call_Issues_wo_Majority_Derived_Process" inputVariable="processData" outputVariable="processData"/>
</otherwise>
</catch>
```
Example 9 shows the BPEL4WS code for the Process from the “Issues w/o Majority?” Decision until the end of the Process or loops.

- The mappings are a fairly straightforward set of *switches*.

If all issues are resolved, then the Process is done. If not, then another Decision is required.

- The “parallel box,” as is any forking situation, will map to a BPEL4WS *flow*.

After the parallel box, the flow loops back to the “Collect Votes” Sub-Process.

- This looping is handled by using an *invoke* (asynchronous) to the “Announce_Issues_Derived_Process” process, which was created just for the purpose of this loop.

If there has already been two cycles of voting, then the process flows back to the “Decision Cycle” Sub-Process.

- This looping is handled by using an *invoke* (asynchronous) to the “Discussion_Cycle_Derived_Process” process, which was created just for the purpose of this loop.

Example 8 displays the BPEL4WS code for the final derived *process* of the “E-Mail Voting Process.”

```xml
<process name="Issues_wo_Majority_Derived_Process">
  <sequence>
    <receive partnerLink="Internal" portType="tns:processPort"
      operation="call_Issues_wo_Majority_Derived_Process" variable="processData"
      createInstance="Yes"/>
    <switch name="IssueswoMajority">
      <case name="Yes" condition="NoMajority=true">
        <switch name="2ndTime">
          <!-- name="Yes" -->
          <case condition="bpws:getVariableProperty(ProcessData,VotedOnce)=true">
            <!-- This is done to do the complex looping situation. -->
            <invoke name="Discussion_Cycle_Derived_Process" partnerLink="Internal"
              portType="tns:processPort"
              operation="call_Discussion_Cycle_Derived_Process"
              inputVariable="processData" outputVariable="processData"/>
          </case>
          <!-- name="No (otherwise)"-->
          <otherwise>
            <sequence>
              <flow>
                <invoke name="ReducetoTwoSolutions" partnerLink="internal"
                  portType="tns:internalPort" operation="sendReceiveSolutions"
                  inputVariable="processData" outputVariable="processData"/>
                <invoke name="EMailVotersthathavetoChangeVotes" partnerLink="WGVoter"
                  portType="tns:emailPort" operation="sendVoteWarning"
                  inputVariable="processData"/>
              </flow>
            </sequence>
          </otherwise>
        </switch>
      </case>
      <!-- name="No" (otherwise)"-->
    </otherwise>
  </sequence>
</process>
```
Example 9 Sample BPEL4WS code for derived process for repeated elements

```xml
<invoke process="Announce_Issues_Derived_Process" partnerLink="Internal"
         portType="tns:processPort"
         operation="call_Announce_Issues_Derived_Process"
         inputVariable="processData" outputVariable="processData"/>

</sequence>
</otherwise>
</case>
<otherwise name="Nootherwise">
    <!-- This is one of the two ways to the end of the Process. -->
    <empty/>
</otherwise>
</switch>
<reply partnerLink="Internal" portType="tns:processPort"
       operation="call_Issues_wo_Majority_Derived_Process" variable="processData"
       createInstance="Yes"/>
</sequence>
</process>
```