Business Process with BPEL4WS: Learning BPEL4WS, Part 8

Using switch, pick, and compensate

Level: Introductory

Rania Khalaf (rkhalaf@watson.ibm.com), Software Engineer, IBM TJ Watson Research Center
Nirmal Mukhi (nmukhi@us.ibm.com), Software Engineer, IBM TJ Watson Research Center

16 May 2003

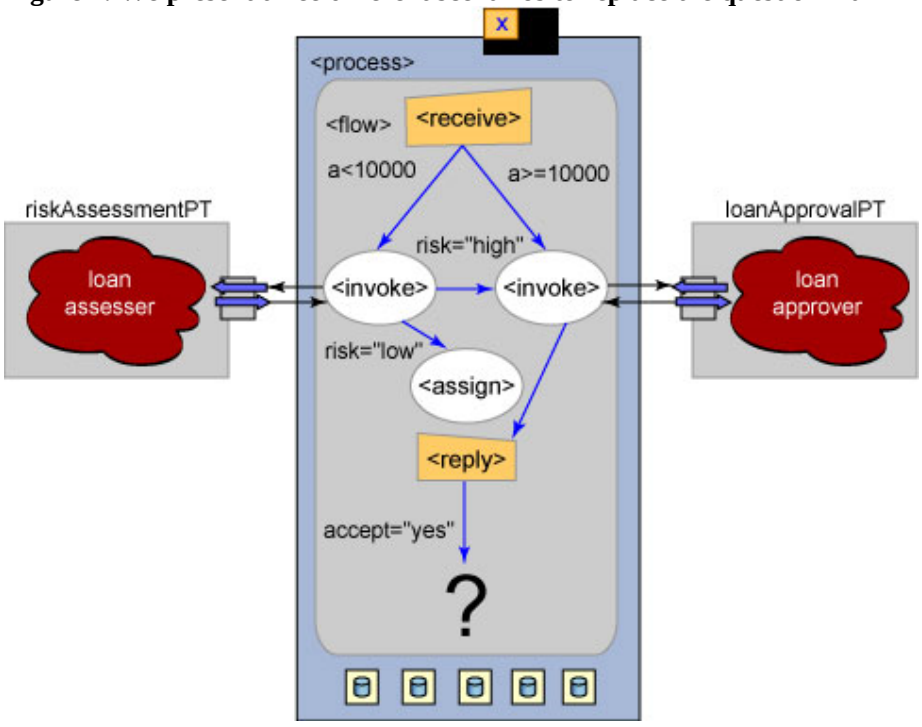
This article illustrates the use of three more BPEL activities: switch, pick, and compensate. In addition to showing how you can branch on conditionals using `<switch>`, we show how you can use `<pick>` to branch based on incoming messages or timeouts. A simple explicit compensation example is also presented to show how committed actions may later be undone.

Introduction

In the previous articles we took a simple flow that can invoke one Web service, and added another partner, control logic through links and conditions, data manipulation using assignment, nested activities and scopes, and finally correlation sets and fault handlers. In this article, we take our loan approval example and present three different scenarios showing the use of the compound activities `<switch>` and `<pick>`, as well as a simple compensation handler.

Let's start with the process we have created so far, and cut out the last few activities. Figure 1 replaces these activities with a big bold question mark which we will experiment with. To recap, we have a customer who wants to request a loan. The customer sends a message to the process, which then checks whether the amount is above or below a certain threshold. If the amount is low, then a Web service is invoked to check whether the applicant is low risk, in which case the loan is granted. Otherwise, the loan approver is invoked to check into the applicant's information and decide whether or not the loan should be granted. This winds up looking a lot like the flow from Part 5 of this series.

Figure 1. We present three different scenarios to replace the question mark



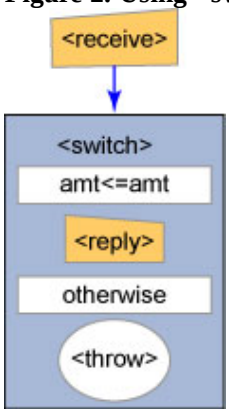
Scenario 1: Branching with <switch>

In this first scenario, the <switch> activity is used to perform the same branching we saw in the correlation article, but without using links. The idea was that if the applicant is eligible for the loan, he sends another message asking for the loan to be processed. If the amount in this new request is more than the amount initially asked for, the process faults. If not, a reply informing the applicant that the loan has been processed is sent back.

In the previous example this was done by a <receive> activity, followed by two links to either a <throw> or a <reply>. Now, we will keep the <receive>, but use a <switch> activity for the rest. This activity is similar to the switch programming construct in a Java environment, except that, at most, one of the branches can be taken. It contains a number of case conditionals, each followed by an activity. The case statements are evaluated in order, and the first one to evaluate to true will have its activity performed.

The <switch> we will add will contain two branches: one to check whether the amount is less than or equal to the initially requested one and containing a <reply>; another to run in all other cases, defined using the <otherwise> construct and containing a <throw> activity. The second branch can, of course, consist of a conditional to check whether the amount is too great. The <otherwise> construct is illustrated here to remind you that a <switch> might not always be replaced by a set of linked activities, because it might not be possible or practical to enumerate each possible scenario with a boolean condition. The <receive>, <reply>, and <throw> are copied exactly from the previous article. The section we have added to replace the question mark above is shown in Figure 2, with the corresponding BPEL4WS syntax in Listing 1.

Figure 2. Using <switch>



Listing 1. Using <switch>

```
<receive name="acceptance-receive"
     partner="customer"
     portType="lns:loanApprovalPT"
     operation="obtain"
     container="acceptanceRequest">
  <target linkName="reply-to-receive"/>
  <source linkName="receive-to-switch"/>
  <correlations>
    <correlation set="loanIdentifier"/>
  </correlations>
</receive>
<switch name="check-final-amount">
  <target linkName="receive-to-switch"/>
  <case condition="bpws:getContainerData('acceptanceRequest', 'amount') <=
       bpws:getContainerData('request', 'amount')">
    <reply name="grant-reply"
         partner="customer"
         portType="lns:loanApprovalPT"
         operation="obtain"
         container="approvalInfo"/>
  </case>
  <otherwise>
    <throw name="grant-failure" faultName="lns:loanProcessFault"/>
  </otherwise>
</switch>
```

Scenario 2: Responding to events with <pick>

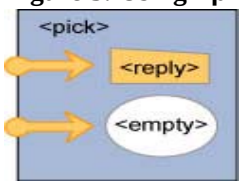
This scenario describes a different way of implementing the same functionality. As before, the applicant is expected to send a message asking that the loan request be processed. A message arrival can be modeled through a <receive> activity as you have seen. Alternatively, you can also view the arrival of a message as an event occurrence. BPEL4WS provides a specific activity that is designed to all the definition and processing of asynchronous events. This is the <pick> activity, which can handle two kinds of event occurrences:

1. **Message arrival**: This is modeled using the <onMessage> element within the <pick> activity. The syntax looks very much like a receive. <onMessage> also lets you define an activity to handle the message arrival.
2. **Expiration of a preset timer**: This is modeled using the <onAlarm> element within the <pick>. You can specify a duration period at the end of which the alarm goes off, executing the associated activity.

The <pick> executes the activity associated with the event that occurs first and then completes. In our specific example, we model the arrival of the applicant's message using the <onMessage> within the pick. The associated activity is simply a reply to the onMessage. In addition, the <pick> also has an <onAlarm> element. This is used as a timeout for the applicant's response. In our earlier scenario, the process remains in the same unfinished state if the applicant never responds. In this version, with the <onAlarm>, the <pick> activity executes the <onAlarm> section (which just contains an <empty> activity since we don't want to do anything other than timeout) and completes, thus resulting in the completion of the process instance if the applicant's message doesn't arrive within 30 seconds of the start of the <pick>. So if the message arrival event precedes the timer going off, the applicant's message is processed and the <switch> gets executed, otherwise the <empty> is executed and the pick terminates that way.

The section we have added to replace the question mark above is shown in Figure 3, with the corresponding BPEL4WS syntax in Listing 2.

Figure 3. Using <pick> with an <onMessage> and an <onAlarm>



Listing 2. Using <pick> with an <onMessage> and an <onAlarm>

```
<pick createInstance="no" name="check-acceptance">
      <target linkName="reply-to-pick"/>
      <onMessage partner="customer"
           portType="lns:loanApprovalPT"
           operation="obtain"
           container="acceptanceRequest">
        <correlations>
          <correlation set="loanIdentifier"/>
        </correlations>
        <switch name="check-final-amount">
          <case condition="bpws:getContainerData('acceptanceRequest', 'amount') <=
               bpws:getContainerData('request', 'amount')">
            <reply name="grant-reply"
                 partner="customer"
                 portType="lns:loanApprovalPT"
                 operation="obtain"
                 container="approvalInfo"/>
          </case>
          <otherwise>
            <throw name="grant-failure" faultName="lns:loanProcessFault"/>
          </otherwise>
        </switch>
      </onMessage>
      <onAlarm for="'PT30S'">
        <empty/>
```

```
      </onAlarm>
    </pick>
```

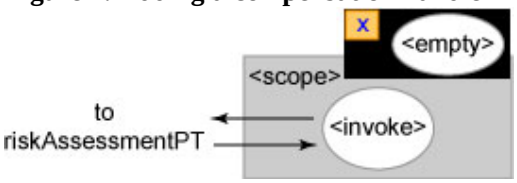
Scenario 3: Undoing with <compensate>

Our last scenario will ask the process to compensate the risk assessment invocation in case the customer sends an amount that is higher than the one already approved. The first step is to surround that <invoke> activity (Figure 1, left side) with a scope that has a compensation handler. As this is mainly for illustration purposes, we simply put an <empty> inside the handler. After the scope has run successfully, its compensation handler waits for a signal to actually run. This signal can be either explicit, arising from a running <compensate> activity, or implicit due to a fault. Note that the scope to be explicitly compensated must be named.

A more useful handler could contain an <invoke> to a *cancel* operation on the Loan Assessor. We simply use an <empty> here so you don't have to change the Java code and WSDL file of the Loan Assessor service.

In the code snippet shown in Listing 3, the additions are shown in bold. See also Figure 4.

Figure 4. Adding a compensation handler

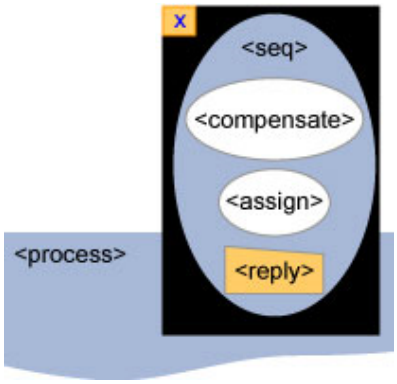


Listing 3. Adding a compensation handler

```
              <scope name="assessor-scope">
   <target linkName="receive-to-assess"/>
   <compensationHandler>
      <empty/>
    </compensationHandler>
    <invoke name="invokeAssessor" partner="assessor"
               portType="asns:riskAssessmentPT"  operation="check"
               inputContainer="request"  outputContainer="riskAssessment">
      <source linkName="assess-to-setMessage"
                  transitionCondition=
                  "bpws:getContainerData('riskAssessment', 'risk')='low'"/>
      <source linkName="assess-to-approval"
                  transitionCondition=
                  "bpws:getContainerData('riskAssessment', 'risk')!='low'"/>
    </invoke>
</scope>
```

In BPEL, you can only invoke explicit compensation from inside a fault or compensation handler in a scope above the one to be compensated. Therefore, we will modify the fault handler on the process itself to include a <compensate> activity referring to the *assessor-scope*; so, going back to the process, change the definition of the fault handler to include the compensate activity. Now if there's a fault, we will first compensate the scope mentioned earlier, put a message saying the request is too high in a container, and send that back to the customer:

Figure 5. Adding a fault handler on the scope

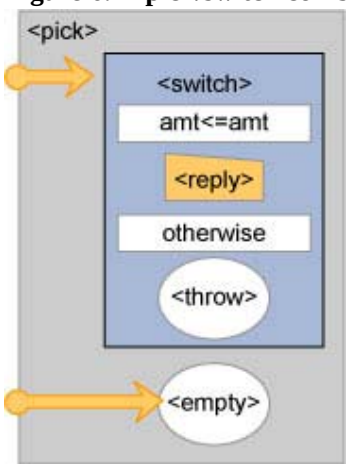


Listing 4. Adding a compensation handler

```
<process .... >
  ....
<faultHandlers>
  <catch faultName="lns:loanProcessFault"
         faultContainer="error">
     <sequence name="fault-sequence">
       <compensate scope="assessor-scope"/>
       <assign name="assigninvalid">
         <copy>
           <from expression="'invalid request: amount too high'"/>
           <to container="approvalInfo" part="accept"/>
         </copy>
       </assign>
       <reply name="grant-reply" partner="customer"
              portType="lns:loanApprovalPT"
              operation="obtain"  container="approvalInfo"/>
     </sequence>
  </catch>
</faultHandlers>
```

Now we move back to our question mark. Let's re-use both the `<pick>` and `<switch>` constructs. Again we use the `<pick>` to wait for a message for a certain time, after which we time out and end the process. Now, instead of simply replying, we handle an incoming message by a `<switch>` similar to the one you saw above. It checks whether the amount requested is less than or equal to the one the customer was approved for. If it is lower, we reply to the customer. If not, we want to end the process but first we want to compensate the scope with the compensation handler and send back the message saying the request was too high. To do that, we throw the fault with a *throw* activity that will trigger the handler we've just defined.The handler will then perform the compensation and the reply. If the 30 seconds end before we hear anything, the alarm is triggered and the process ends.

Figure 6. A pick/switch combination, and explicit compensation using `<compensate>`



Listing 5. A pick/switch combination, and explicit compensation using <compensate>

```
<pick createInstance="no" name="check-acceptance">
    <target linkName="reply-to-pick"/>
    <onMessage partner="customer"
         portType="lns:loanApprovalPT"
         operation="obtain"
         container="acceptanceRequest">
      <correlations>
        <correlation set="loanIdentifier"/>
      </correlations>
      <switch name="check-final-amount">
        <case condition="bpws:getContainerData('acceptanceRequest', 'amount') <=
             bpws:getContainerData('request', 'amount')">
          <reply name="grant-reply"
               partner="customer"
               portType="lns:loanApprovalPT"
               operation="obtain"
               container="approvalInfo"/>
        </case>
        <otherwise>
           <throw faultName="lns:loanProcessFault"/>
        </otherwise>
      </switch>
    </onMessage>
    <onAlarm for="'PT30S'">
      <empty name="alarm-empty"/>
    </onAlarm>
</pick>
```

Running the three scenarios in BPWS4J

Read the section on running the process from Part 7 of this series. Note that our scenarios use the same services (for example, all the WSDL files, including the process interface, remain as described here) and the same client used to test the functionality. The only difference is that the BPEL file used for each of our scenarios is slightly different from the one used here, since the process logic for each scenario is different.

Each of the scenarios describes a modified version of a process with the same name. This name is used to identify a deployed process. So if you run one scenario and want to try another one, you will need to undeploy the first process in order for the next scenario's deployment to work.

To run each scenario, deploy the process (use the BPEL file for that scenario) as described in the previous article. Here's a list of the BPEL files for each scenario, along with some comments on how to test the functionality using the client. These BPEL files and all the WSDL files required for deployment are included in the zip file, which you can download from the Resources section.

Scenario 1: Branching with `<switch>`

Use the file loanapproval-switch.bpel.

You can run the client as described in the previous article in this series, since the features of the process remain identical. As explained above, this is just an application of switch to demonstrate an alternative way of expressing the logic.

Scenario 2: Responding to events with `<pick>`

Use the file loanapproval-pick.bpel.

Here, our process has one additional feature not seen in the previous scenario, and that is the use of a timeout. To test this, you can use the client to request a loan approval. Once the process replies, it then expects the client to ask that the loan be obtained. However, it waits exactly thirty seconds for the client to send the message, at the end of which, if the message has not arrived, the process completes. To test this, once the loan approval is complete, do not send the request to obtain the loan by running the client again; instead, just do nothing. On your server console, you will observe the completion of the process (for more details you can turn logging on). If the client sends the *obtain* message after the process has completed, the BPWS4J

engine is unable to match it with a running process instance and returns with an error.

Scenario 3: Undoing with <compensate>

Use the file loanapproval-compensation.bpel.

This has features similar to the pick scenario, with the additional step that the BPEL file describes for situations in which a compensation block is to be run. To test this out, you should first turn logging on. The logs will let you know that the compensation block executes, since the client is not informed of this activity. Run your client (make sure you use the scripts from Part 7) asking that a loan be approved. Then, run the client again asking that the loan be obtained, but specify a larger loan amount than the one approved. You will see the compensation block execute on the log messages, and receive an *amount too high* response.

Conclusion

In these scenarios, we have brought together most of the main concepts of BPEL4WS. As the last tutorial-type article in this series, it provides you with a better feel for how to bring activities together to create simple as well as complex processes. Feel free to do what we have done here to further your understanding of the language: take the resulting .bpel file and play around with the syntax to try out the different constructs in the language and see how they affect the flow of control as well as the complexity of what you would like to express in your Web services compositions. You can use the Eclipse editor available with BPWS4J on alphaWorks (see Resources), and graphically substitute an activity for another or change its properties without having to worry about the rest of the process. For example, you can take a nested activity and request that it be wrapped in a scope and then add handlers to it.

Download

Name	Size	Download method
ws-bpelcol8.zip	7 KB	**HTTP**

→ Information about download methods

Resources

- Participate in the discussion forum.
- Please note that this articles refers to version 1.0 of the BPEL4WS specification. The latest version, BPEL4WS1.1, is now available, and an article describing the key differences between the two specifications will be available shortly.
- Download the BPWS4J engine from alphaWorks.
- Download the zip file with the code samples.

About the authors

Rania Khalaf is a software engineer in the Component Systems group at the IBM T.J. Watson Research Center. She joined IBM in 2001 after having completed her Bachelors and MEng degrees from MIT. Rania is a co-author of the IBM BPEL4WS engine, BPWS4J, available from alphaWorks. You can contact Rania at rkhalaf@watson.ibm.com.

Nirmal Mukhi is a Software Engineer in the Component Systems Group at the IBM T.J. Watson Research Center. He holds a Masters Degree from Indiana University. At IBM, he conducts research on Web services and has been involved with WSIF, the Web Services Gateway, and BPWS4J projects. You can contact Nirmal at nmukhi@us.ibm.com.

Share this....