Designing event-driven business processes in IBM Business Process Manager

Fintan McElroy

June 27, 2012

IBM® Business Process Manager V7.5.1 introduced new support for BPMN 2.0 event subprocesses. This powerful construct provides the capability to design a business process to be able to handle planned and exceptional events with the flexibility to process events that interrupt normal processing or that need to be processed in parallel to the main logic flow. In this article, you'll learn how to use the event subprocesses feature in IBM Business Process Manager to distribute and process business events both for a specific process instance and for a set of qualifying instances.

Introduction

The Business Process Modeling and Notation (BPMN) 2.0 standard introduced a new event-handling mechanism called an event subprocess, which runs in the context of the process that contains it. Similar to a boundary event on an activity, an event subprocess listens for its trigger signal only when the containing process is active. Once the process instance is complete, the contained event subprocess stops listening for events. Event subprocesses are like other subprocesses in that they are contained within a parent process, and are not reusable outside of that process. They are unlike other subprocesses in that they are not connected to other activities in the process by incoming or outgoing connections, and are only triggered by events that either occur in the process or its subprocesses or by an external message.

IBM Business Process Manager 7.5.1 introduces support for the most common categories of event subprocesses: message-based events, timer-based events, and exception events.

• A message-based event might be used in a situation where a business message, such as an out-of-stock message, is received by the event subprocess.
• A timer-based event might be used to model the steps to take when an activity within the parent process is not completed after a specified amount of time.
• An exception event might be triggered when something has gone wrong in the process, for example, the order fulfillment system is non-responsive.

Of the three types, the message-based one provides the most flexibility because it allows for communication from outside the process as the trigger of the event (for example, where an existing system client also allows for processing of stock orders outside of the context of the IBM Business Process Manager process). This article focuses on message-based event subprocesses.
The BPMN_2.0_Event_Sub-Processes_1.0.twx file provided for download with this article contains a sample scenario that you can import and run in conjunction with the steps described in the article.

**Concurrent processing using message events**

This section introduces the example process for the scenario and shows how a message-based event subprocess is used to allow concurrent updates to be made to an existing process instance.

**Figure 1** shows an example order process that will process a customer request for some products, first checking that they have adequate credit and then calling a linked process to fulfill the requested order. If the order is successfully fulfilled, an invoice for the goods will be sent to the customer. **Fulfill order** has been explicitly modeled as a separate process because event subprocesses are scoped to a process and, in this scenario, we only want the event subprocess to be executed in the context of the set of activities within fulfillment. The event subprocess will allow order updates, however because it is not valid to update an order before it has been created or to allow for updates when the invoicing is being prepared, the set of activities is contained within the separate **Fulfill order** linked process.

**Figure 1. Order Process**

![Figure 1. Order Process](image)

(See a larger version of Figure 1.)

**Figure 2** shows the **Fulfill order** process. The process checks that there is sufficient stock of the requested products and then includes steps to pick the stock and ship it to the customer. The message-based event subprocess **Handle order updates** is a non-interruptible event subprocess meaning that the activities within it occur in parallel to the main processing path. You can tell it is non-interruptible from the dotted line around the message icon (highlighted below).
Double-clicking on the event subprocess expands it to show the detailed activities inside it, as shown in Figure 3. Once expanded, you'll see a breadcrumb trail at the top of the diagram that allows you to navigate back to the parent process. The `Update request` start event is shown also with a dotted line to signify that it has been set to non-interruptible. For the purposes of this simple example, we just encapsulate all the update processing in a single human service, but in reality there might be more complex processing to recheck the delta credit available and consolidate the updates with the original order.

The message start event is implemented with an Undercover Agent (UCA) as shown in Figure 4. Notice that the setting `Interrupt Parent Process?` is unchecked, which makes this event
subprocess non-interruptible. The **Repeatable?** setting has been checked to allow for multiple updates to the order to be sent while the fulfillment steps are active.

**Figure 4. Message event implementation settings**

The **Data Mapping** section of the properties of the message event is where the correlation is specified, along with any mapping of additional data provided in the message to the variables within the process, as shown in **Figure 5**. Correlation is required in order to identify the specific instance of the **Order** process that the update is for. In this case, the event message supplies an orderID parameter which contains the order identifier for the order to be updated. In addition, an updated Order data structure is supplied on the event, which is mapped to the process variable updatesToOrder.

**Figure 5. Message event data mapping**

To illustrate sending an event to be handled by the message-based event subprocess, let’s create an order instance. After starting the process, you can view the instance and work on the first task using the Process Inspector as shown in **Figure 6**.
Figure 6. Order Process in Process Inspector

(See a larger version of Figure 6.)

In the resulting Coach, shown in Figure 7, we will supply some example data. Take special note of the order ID value used as this identifier is needed to correlate on later when you emulate the update being performed outside the process.

Figure 7. Example order details

(See a larger version of Figure 7.)

Returning to the Process Inspector, as shown in Figure 8, you can see that the process has now moved into the Fulfill order linked process. The event subprocess Handle order updates is now in scope, so you'll notice that there is a token shown against the subprocess, which signifies that the event subprocess is ready to receive messages.
Figure 8. Event subprocess awaiting token

(See a larger version of Figure 8.)

For the example scenario, we'll provide a Coach to emulate the update of the order being performed by a client outside of the context of the process (the Send Order Update Event Client shown in Figure 9).

Figure 9. Order update client Coach

Figure 10 shows an example of a completed Coach for emulating the update. Notice that the Order ID matches the one used earlier to create the order. For this simple example we just show an amended order for illustration, but in reality we would expect this client to retrieve the existing order and allow updates to be made to it.
In the Process Inspector, you can confirm that the instance of the Order process has received the event, as shown in Figure 11, and the parallel path for processing it has been initiated.

If you drill into the event subprocess, as shown in Figure 12, you can see that the Process order update task is ready (the token with ID 21), but also there is an additional token on the message start event (with ID 20). This is due to the Repeatable setting on the message event; the token is there to await any further update UCAs to trigger the message while the Fulfill order process is active.
To illustrate the ability to have repeatable events, Figure 13 shows a situation in which work was completed on the original update and there has since been a further update made and that event has also been received by the event subprocess.

For this scenario, we complete this further update and also the tasks on the main processing path of Fulfill order. If you refresh the view in the Process Inspector, as shown in Figure 14, you can see that the "Shipped" path out of the decision gateway has been followed in the Order process and the activity Send invoice is now awaiting action.
The send invoice activity is completed, and this instance of the process is then complete. We now want to move on to illustrate an aspect of the implementation of event subprocesses in IBM Business Process Manager V7.5.1 that may lead to unexpected consequences. In order to illustrate this, let's imagine an unlikely scenario in which it is possible to create another order with the same order identifier, as shown in Figure 15.

After creating the new order and viewing the progress of the instance in Process Inspector, as shown in Figure 16, we now find we have an unexpected situation. Even though we have not performed an update on this order, we find that two instances of the update event subprocess have been activated. Recall that in the previous example, we performed two separate updates to the order, and now we seem to have received those updates again for this new instance.
Figure 16. Event subprocess activated unexpectedly

(See a larger version of Figure 16.)

The key to explaining this seeming anomaly is to look again at the message event settings, as shown in Figure 17. Notice that the **Durable Subscription** setting is disabled and set by default, which means that the message is retained and is available to any future instances. This may not happen if you can absolutely guarantee that the business data you choose for correlating upon will be unique for all time (not just for active instances).

Figure 17. Durable Subscription setting

In the next section, we'll look at amending our process design to use a safer guaranteed unique correlation mechanism. The importance of such a mechanism will become even more apparent when we look at how to distribute an event that has more than one potential matching instance.
Guaranteed unique correlation of events

In this modified design, we’ll use an internal IBM Business Process Manager correlation identifier that is guaranteed to be unique – the process instance ID itself. We add a private variable to the process, as shown in Figure 18, and set it by default to retrieve the instance ID at process start.

Figure 18. Default assignment of process instance ID

Because the event subprocess runs within the linked subprocess Fulfill order and will need access to the process instance ID to correlate against, you pass the ID to the linked process as a parameter, as shown in Figure 19.

Figure 19. Process instance ID passed to linked process

Figure 20 shows the amended message event, which now correlates to the process instance ID that has been passed to the Fulfill order process.
We still want to allow clients to use business data for identifying the order to update, rather than an internal identifier from IBM Business Process Manager, therefore some mediation is needed between the business identifier and the internal identifier. Figure 21 shows the UCA used for the message event and the attached service (Order Update Event Handler) it uses to pass and manipulate data parameters.

Figure 21. Order Update UCA

If you look at the variables of the attached service, shown in Figure 22, you see that an order identifier (orderID) is provided as input, but a process instance identifier (orderProcessInstanceID) is provided as output. You'll see in the logic of the service how this translation happens.
**Figure 22. Order update handler variables**

The **Order Update Event Handler** service, shown in Figure 23, first calls a service that retrieves all active process instances for the supplied order identifier. The called service has a default setting for the name of the process. We pass in the order identifier and the service returns an array of process instances (of which we expect there to be at most one member). If there is a match, then a simple script is used to return the first process instance identifier as the output of the **Order Update Event Handler** service.

**Figure 23. Contained service for retrieving matching instances**

If you look at the input variables of the service that retrieves instances, shown in Figure 24, note that the setting for the name of the process is the name of the linked process not the parent process. This is because this is the process instance name that will be active at the time we want to send an update event. You’ll see later in the search API that you need to pass this name to narrow the search to instances of this type.
Figure 24. Retrieve process instances variables

![Image](image1.png)

The body of the service contains a script element with JS API code to perform a search against the business data column, highlighted in Figure 25.

Figure 25. JS API showing business data condition

![Image](image2.png)

```javascript
// Add a search condition to search for a business data column
var colOrderID = new TWSearchColumn();
colOrderID.type = TWSearchColumn.Types.OrderBusinessData;
colOrderID.name = "orderID";
```

Figure 26 shows the corresponding business data column that has been made available as a business data search field on the process.
Figure 26. Business data exposed for searching

Returning to the rest of the JS API code, shown in Figure 27, the set of conditions including the search against the order identifier business data are set on the search and the search executed. The results are then parsed into an array of `OrderProcessInstance` objects to pass back as output to the caller of the service.

Figure 27. JS API search and results parsing

You can now re-run the earlier scenario to verify that the issue with events being delivered in error has been addressed in the revised design. Figure 28 shows another instance of an order in Process Inspector. The task subject has been modified to show the order identifier, an order update has been performed, and the update event has also been received.
After completing all the tasks in the first instance, we create a second instance using the same order identifier, as shown in Figure 29. This time no unexpected events have been received by this new instance and the token is shown on the event subprocess awaiting any valid event messages that we wish to generate for this order instance.

We'll expand on the adjustments we've made in the next section, where we look at how to distribute events to multiple matching process instances.

1-to-many event correlation

We now want to extend our scenario to allow for the cancellation of any outstanding orders for a particular customer. We could imagine a situation where perhaps a customer has not paid outstanding invoices and the finance department wants to prevent any further orders being
shipped for that customer. In this case, we want to send an event that could result in multiple potential matches.

**Figure 30** shows the revised *Fulfill order* process with an additional event subprocess *Handle order cancellation* added. In this case, the event subprocess is interruptible (denoted by the solid line around the message icon below) -- we don't want the main process path to continue, but instead for control to switch to the activities inside the event subprocess.

**Figure 30. Revised Fulfill Order linked process**

![Diagram of the revised Fulfill Order linked process](image)

*Figure 31* shows the expanded event subprocess with the *Interrupt Parent Process?* setting checked. The message event has a solid line around it consistent with the view in the collapsed event subprocess you saw in **Figure 30**.
Figure 31. Cancel order message event settings

For the data mapping, shown in Figure 32, the process identifier is again used for correlation, however in this case there is no additional business data passed in to be processed.

Figure 32. Cancel order message event correlation settings
The UCA used for this cancellation event is shown in **Figure 33**.

**Figure 33. Order Cancel UCA**

The variables for the event handler, shown in **Figure 34**, are the same for input and output in this case. You'll see shortly that a further UCA is used to translate between business data and the internal process identifiers to then distribute to the target UCA.

**Figure 34. Order Cancel handler variables**

The UCA event handler `Order Cancel Event Handler` used by the UCA is just a simple pass-through pattern, as shown in **Figure 35**.

**Figure 35. Order Cancel handler**
In order to distribute events, we'll create a proxy UCA -- that is, a UCA that is not directly connected to a message event, but is used to send events to one or more instances of the UCA for the cancel event, as shown in Figure 36.

**Figure 36. Order cancel distribution UCA**

![Order cancel distribution UCA](image)

(See a larger version of Figure 36.)

When you look at the variables for the event handler, shown in Figure 37, you can see that there is a customer identifier (customerID) provided as input, but there is no output variable. This is because the UCA is not going to be directly referenced by a message event, so does not need any output variables to be used in a mapping. Instead as you'll see in the logic flow of the service, the handler will call a helper service to invoke the real UCA.

**Figure 37. Event handler variables**

![Event handler variables](image)

**Figure 38** shows the logic flow for the service. The first step is a call to a similar service to that used in the update UCA handler, but this time the service searches on a customer identifier which has also been exposed for business data search on the process. If any matches are found, then the resulting array of matched instances is passed to another service, which will distribute the resulting order cancellation events.
If you look at the implementation of the service that distributes the events (Distribute Order Cancellation Events shown in Figure 39), you see that the service uses a visual looping pattern to process the array and invoke the order cancel UCA for each match, passing in the process instance identifier for each one.

To test out the distribution of a cancellation event, we create two instances of the order process for the same customer. Figure 40 shows the first order process instance in the Process Inspector. The task subject has been modified to show the customer identifier we will be using for the
emulation of a cancellation request. Also note that there are now active tokens for both of the event subprocesses that are now in scope.

**Figure 40. First order for customer in Process Inspector**

![Figure 40](image)

*(See a larger version of Figure 40.)*

**Figure 41** shows the second instance highlighted to show that it is for the same customer.

**Figure 41. Second order for customer in Process Inspector**

![Figure 41](image)

*(See a larger version of Figure 41.)*

In order to emulate the cancellation scenario, we will run a test Coach to allow for providing the customer identifier for the customer whose outstanding orders we wish to cancel. **Figure 42** shows the test Coach called `Send Order Cancellation Distribution Event Client`. Note that the service it calls, `Send Order Cancellation`, will call the proxy UCA, which will then use the event handler to search for matching process instances and distribute the events to trigger the real UCA that corresponds to the message event on the cancellation event subprocess.
Figure 42. Send order cancellation Coach

Figure 43 shows we are about to provide the relevant customer identifier in the running Coach to trigger the cancellation.

Figure 43. Specifying the customer to match on

If you switch back to the Process Inspector and refresh the instances, you can see that the first order instance has successfully received the cancellation event, as shown in Figure 44. Notice that there is no longer a token against the Handle order updates event subprocess. Also the Handle order cancellation event subprocess only has a token representing the received event and does not have a separate token awaiting any further events to be received. Because the cancellation is set to interrupt, it takes full control now of processing and no further events can be received.
Figure 44. Cancellation received for first order

(See a larger version of Figure 44.)

If you look at the second order instance, shown in Figure 45, you can confirm that it too has received the cancellation event and that is the only path that can now execute within the instance.

Figure 45. Cancellation received for second order

(See a larger version of Figure 45.)

Conclusion

This article introduced the concept of BPMN 2.0 event subprocesses and then examined in detail the support in IBM Business Process Manager V7.5.1 for message-based event subprocesses. You learned about correlation of events and some potential challenges when using business data for correlation. The article identified a pattern for using an internal unique correlating identifier and mediating between it and business data fields. Finally you saw how to support the pattern for a 1-to-many distribution of events to running process instances.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample scenario</td>
<td>BPMN_2.0_Event_Sub-Processes_1.0.twx</td>
<td>1.4MB</td>
</tr>
</tbody>
</table>
Related topics

- BPMN 2.0 Specification
- IBM Business Process Manager V7.5 Information Center
- IBM BPM Journal: Get the latest articles and columns on BPM solutions in this quarterly journal, also available in both Kindle and PDF versions.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)