



BPMN and the Business Process Expert, Part 4: Mastering BPMN Events

Summary: The ability to describe event-triggered behavior directly in the diagram separates BPMN from traditional modeling notations. An event can start a process, resume a waiting process, or abort a process activity and redirect the flow. The BPMN spec describes many different event types, but learning just a few patterns is all you really need. Fourth of six parts.

Author: Bruce Silver

Company: Bruce Silver Associates

Created on: 3 December 2007

Author Bio



Dr Bruce Silver is an independent industry analyst and consultant focused on business process management software. He provides training on process modeling with BPMN through BPMessentials.com, the BPM Institute, and Gartner conferences, and is the author of The BPMS Report series of product evaluations available from the BPM Institute.

If you had to name the one thing that sets BPMN apart from traditional process modeling notations, it would be *events*. That is to say BPMN provides a logically consistent notation for representing event-triggered behavior directly in the diagram. BPMN lets you show, for example, that a process is started by a particular event, or is waiting someplace for an event, or is redirected from its normal flow to a special event-triggered exception flow. BPMN also lets you show how one process interacts with other processes, both internal and external, via events, and even how one part of a process can signal to another part of the same process using events. This article will show you how to use events correctly and effectively.¹

In BPMN, an event is a *signal that something happened*. It is not assigned to a resource, and it does not perform work – unless you consider sending and listening for signals “work.” In the notation, the event symbol – always a circle – does not represent the implementation of that signal, how it was generated or received, but rather the *interaction of that signal with the process*.

Not all modeling tools that claim to be BPMN-based support events. While there is nothing in the BPMN spec that says a tool must support events, let’s just say that tools that don’t are leaving out the good stuff.


¹ The notation for a few event types will change a bit in BPMN 1.1. Except for discussion of the Signal event, new in BPMN 1.1, the graphics in this article use the BPMN 1.0 notation. However, where the semantics of an event are clarified in BPMN 1.1, we use the newer interpretation.


BPMN 1.0 defines a wide variety of event *trigger types*, distinguished by their icons, and if your BPMN tool is able to make the model executable, it is extremely unlikely to support every single one of them. That's not so bad, since there are actually only a few event types that you need 99% of the time, and these are the ones provided by most BPMN tools.

Start Events


A start event, a circle with a thin border, indicates the start of a process or subprocess. A *none start* event, with no symbol inside, means the trigger is unspecified. In a subprocess, none start is the only type of start event allowed, since the sequence flow in to the subprocess is, by default, always the trigger. In the main, or top-level, process, you will also frequently see message and timer start events.


In BPMN, a signal received from outside the process is called a *message*, whether it's a SOAP message or a fax, phone call, or paper mail.

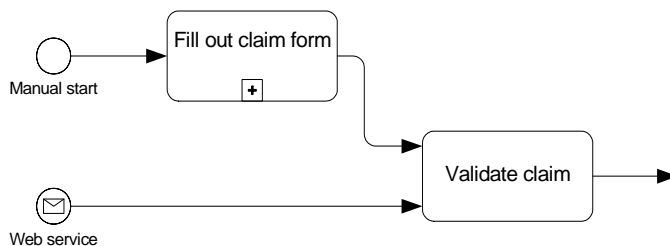
A *message start* event  indicates that the process is triggered by a particular message, identified by the label and/or the event's message attribute.

A *timer start* event  indicates that the process starts at a particular date and time or on a periodic date/time cycle.

None, message, and timer are the only start event types supported by most tools. However, the spec defines a few more.

A *rule* (renamed *conditional* in BPMN 1.1) start event  means the process is triggered when a data expression (within the process) becomes true. Say you want to trigger a process whenever a new customer is added to the ERP system. Is that a rule event or a message? In most tools this would be a message, as the ERP system would be considered external to the process. In fact, rule start is inherently ambiguous because what does "data within the process" mean when the process hasn't started yet? But there it is.

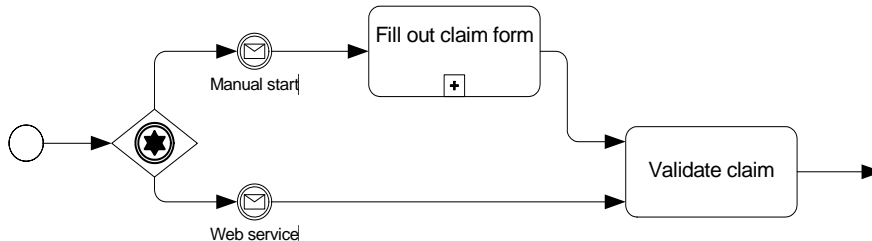
A *multiple start* event  signifies that any of N events (of the types described above) will trigger the process. You rarely see this, but you do see the need for a related use case, where the process can be triggered in different ways and the initial processing depends on which trigger it was. (Downstream activities may be common.) In BPMN 1.0, most users would diagram it this way:



But this is ambiguous, since the none start is undefined. The BPMN spec says that if a process or subprocess has multiple start events, they are all enabled when the process or subprocess starts, and that is not the intent here. BPMN 1.1 provides a less ambiguous notation, using the event gateway.

The event gateway uses the multiple event (actually an intermediate event, not start event) tucked inside a gateway diamond. Normally it means wait (within a started process) for one of N events, and take the path from the first event to occur, ignoring any others that occur


subsequently. But how did the process get started in the first place? When it immediately follows a start event, the event gateway is said to “bootstrap” the process. So you can think of the combination of the start event and the event gateway as the multistart notation we are looking for. Notice that “manual start” has been recategorized as a message event. BPMN should really add a new trigger type for user-initiated processes, and allow it in event gateways, for this pattern to be consistent with the rest of BPMN.





In BPMN 1.1 there is also *signal start*. We’ll talk about signal later on.


End Event





An end event, a circle with a thick border, denotes the end of a branch of a process or subprocess. A process or subprocess with parallel paths must allow *all* parallel paths to reach an end event in order to complete. While it is recommended to use a single start event in a process or subprocess, use of multiple end events is often best practice. Each separate end event can carry a label to indicate the end state (e.g., success or failure) of the process or subprocess. Also, some end events may throw an event signal, called a *result*, either to an external system or process or to another part of the same process, depending on the event type.

A *none end* event  simply ends the process path without throwing the result signal.

A *message end* event  throws a result signal to an external process, service, or system. It can be used for any purpose, to return data or simply to indicate completion.

An *error end* event  sends an error signal to another part of the same process, specifically an Error intermediate event attached to the subprocess containing the Error end event. The Error event that catches the thrown error signal will abort the subprocess immediately, even if parallel paths in the process or subprocess have not yet reached their end event.



Terminate  is a very useful end event. It does not throw a result signal but immediately ends the subprocess that contains it, without waiting for parallel paths to complete. As we shall see, you can use it to implement the equivalent of a non-aborting attached event.


These are the common ones. There is also *multiple end* , meaning multiple result signals will be thrown; *cancel*  and *compensate* , used for business transaction recovery, to be discussed in Part 5 of this series; and *signal* , discussed later on.


Intermediate Event in Sequence Flow



Intermediate events are the cool part of BPMN. Drawn with a double border, they signify events that occur after the process has started but before it has ended. Depending on the trigger type and where it is drawn in the diagram, an intermediate event can mean either pause and wait for the signal, then continue; immediately send the signal and continue; or immediately abort a running activity and redirect processing to an exception flow.


An intermediate event drawn with sequence flow in and out (called *in sequence flow* or *in normal flow*) can mean, depending on the trigger type, either wait for the event or throw the event. Some trigger types, like message, multiple, and signal, can do both, so in BPMN 1.1


the “throwing” variant is drawn with the icon filled  and the “catching” variant is drawn unfilled . In BPMN 1.0 they both are drawn unfilled, so best practice is to standardize its use only for one of those meanings and use a Send or Receive task for the other. A Send task is exactly the same as a message intermediate event (throwing) in sequence flow. A Receive task is exactly the same as a message intermediate event (catching) in sequence flow.

A *timer event*  in sequence flow means a delay, either wait for a specified duration or wait until a specified date/time.

A “catching” *message event*  in sequence flow means wait for a signal from outside the process, then resume. Remember, in BPMN a “message” really means any kind of signal from outside, even a phone call or web interaction. The key is that it comes from outside the process.

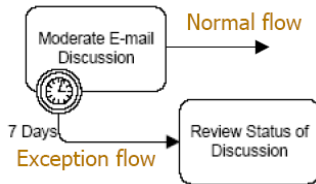
These are the common ones in sequence flow. The spec also defines a *rule (conditional) event* , meaning wait for some process data expression to become true; *multiple event* , meaning wait for any of N events to occur; and *signal*, both throwing and catching, discussed later.

A *none intermediate event* in sequence flow , while rarely used, has a valuable use case. It simply represents a *state* of the process; there is no signal sent or received. Such states are common in other modeling notations, and the none intermediate event could be used to translate them to BPMN.

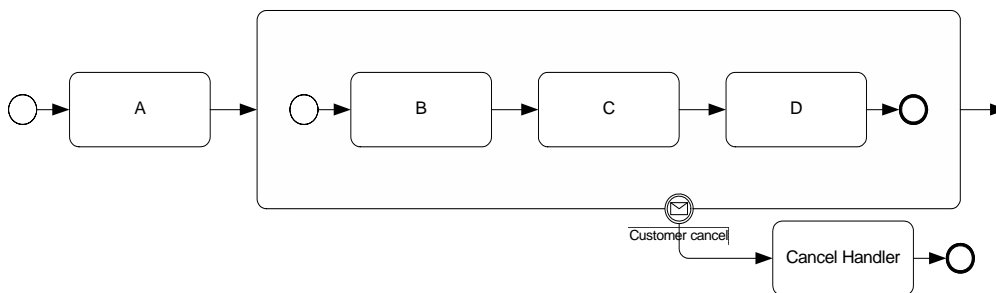
The *link event*  is like a goto on the diagram, typically used to connect parts of a single process spread across multiple pages. A link event drawn on one page with a sequence flow in but none out is logically connected to a paired link event on another page drawn with no sequence flow in but a sequence flow out. It’s just an off-page connector.

Attached Intermediate Event

An intermediate event drawn attached to the boundary of a process activity (*attached event*) means something completely different from the same intermediate event drawn in sequence flow. An attached event always means if the trigger occurs, abort the activity the event is attached to, and proceed down the sequence flow coming out of the event, called the *exception flow*. If the activity completes without the trigger occurring, take the sequence flow directly out of the activity, called the *normal flow*.



The activity defines the *scope* of the event. If the trigger signal occurs before the activity starts or after it ends, it is ignored. Thus, it is common to wrap a sequence of tasks within a subprocess activity solely for the purpose of defining that scope. For example, if in an order handling process having steps A to F, you allow the customer to cancel or change the order between steps B and D with a particular handling flow, you can draw a subprocess enclosing the portion from B to D and attach a message event to it. Order cancellation or change downstream, having a different consequence and handling flow, could have other message events placed there.



In this way, attached events provide a business-friendly notation for explicitly indicating both the scope and handling of events, be they external (message events), timeouts (timer events), system faults (error events), etc. In BPMN, the exception flow is allowed to go anywhere (within its enclosing subprocess, if any). It can go to a handler task, or loop back to the activity and restart it, or jump to an end event.

New Signal Event

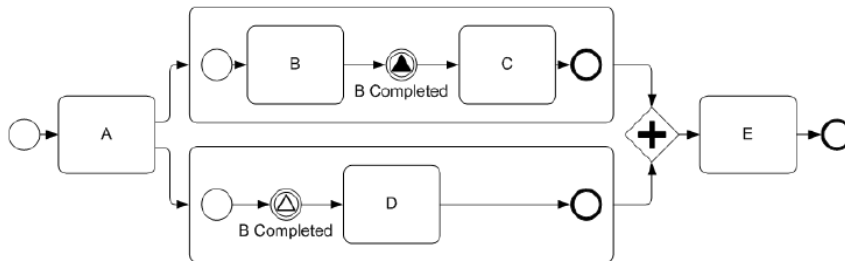
Certain event types, like message, error, compensate, and cancel, support both throwing and catching, so both the source of a signal and the triggered behavior can be shown within the same business process diagram. But each of these types is limited in its use.

A message event can only be used to signal across a process (pool) boundary; it cannot be used to signal from one subprocess to another within the same pool. Also, a message signal technically is addressed to a specific target, such as a message queue, rather than broadcast in publish/subscribe manner.

An error event can only be used to signal within a single process (pool). It can be broadcast, but you cannot have an error intermediate event in sequence flow, so you cannot use error event to wait for a signal. Also, fundamentally, it suggests an error condition has occurred. Compensate and cancel are specific to transaction recovery; we'll cover them in part 5.

BPMN 1.1 addresses the need for a more general purpose signaling mechanism with the new signal event. It can be broadcast to multiple catchers, in pub-sub style. It can work either within a pool or across pool boundaries. It can be used as a start event, throwing or catching intermediate event in sequence flow, attached intermediate event, or end event. It does not necessarily imply an error condition. Signal can do it all.

An important use case for the signal event is, from the *middle* of one activity triggering the start of a second parallel activity within the same process. BPMN has a hard time doing *anything* from the middle of an activity, since sequence flow presupposes that the activity is done.



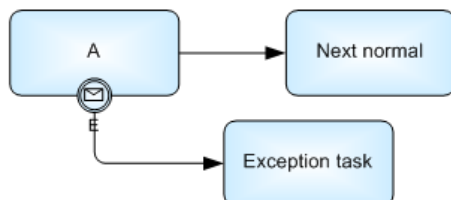
Consider the diagram above, from the BPMN 1.1 spec. It illustrates what the BPM academics call the milestone pattern. Two parallel activities follow A. One has parts B and C, and the other is just D. You want to say D can start once B is done. If the subprocess boundary didn't exist, you could just draw a sequence flows from A and B to an AND-join preceding D. But the subprocess boundary does exist, for many possible reasons – event scoping, service reuse, ownership and governance. It's a given, and you can't cross subprocess boundaries with a sequence flow. But you can with a signal event. A throwing signal event following B links to a catching signal event in sequence flow (wait-for) before D.

Another use case might be an exception process triggered by a variety of errors in the diagram. This should be drawn in a separate pool. Exceptions thrown by signal end events (or throwing intermediate events) can be caught by a signal start event triggering the exception process. Because of the pub-sub linking, the thrower is loosely coupled to the catching process.

Signal events will add a powerful new dimension to event-triggered behavior once BPMN 1.1-compliant tools emerge.

Non-Aborting Attached Events

One limitation of attached events in BPMN is they always abort the activity. It would be nice to have a variant that preserves the scoping and triggered exception flow behavior without aborting the original activity. Some BPM Suites, like TIBCO and Lombardi, offer this, but the BPMN spec does not sanction it. Perhaps a future version of BPMN will support it directly in the attached event notation, but as it stands currently, non-aborting attached events takes a workaround using Terminate and Signal events.

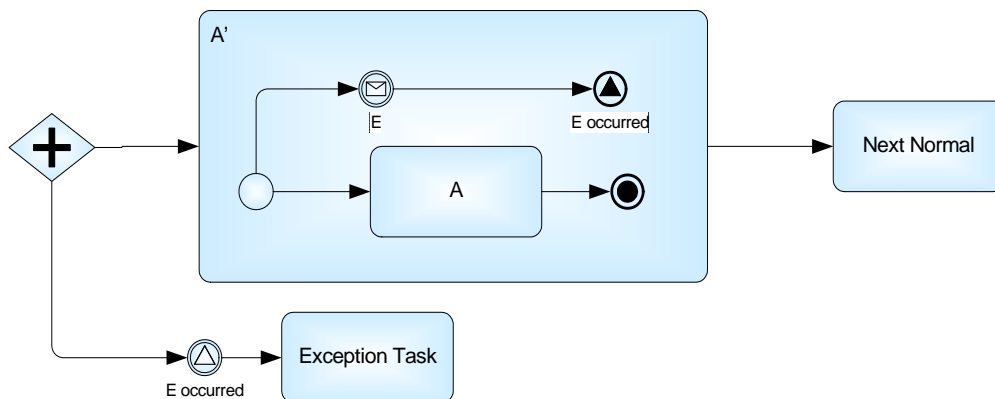


Consider activity A with an attached event E, as drawn above. In standard BPMN, if A completes without the E signal occurring, the task *Next normal* starts. If E occurs before A completes, *Exception task* starts. Exception task and Next normal are exclusive alternatives.

But suppose you wanted a non-aborting version of this, i.e., if E occurs while A is incomplete, trigger Exception task but don't abort A. Let it continue, and when it completes go to Next normal. For example, maybe Exception task is just a notification. Thus the normal flow and exception flow are parallel threads of this process or subprocess.

One way to do it uses the Terminate end event in combination with Signal. Instead of attaching E to the boundary of A, wrap A in a subprocess A', and make E an intermediate event in a sequence flow in parallel with A, as in the diagram below. Now E is a wait-for event, not an attached event, and the sequence flow out of E leads to a Signal end event. The Signal informs any paired listening events that the event E has occurred. Note that if A finishes before E occurs, the Terminate ends A' and the Signal event is never thrown. But if E occurs before A is done, the Signal event is thrown. A' does not complete until A is done, but the Signal is thrown before that.

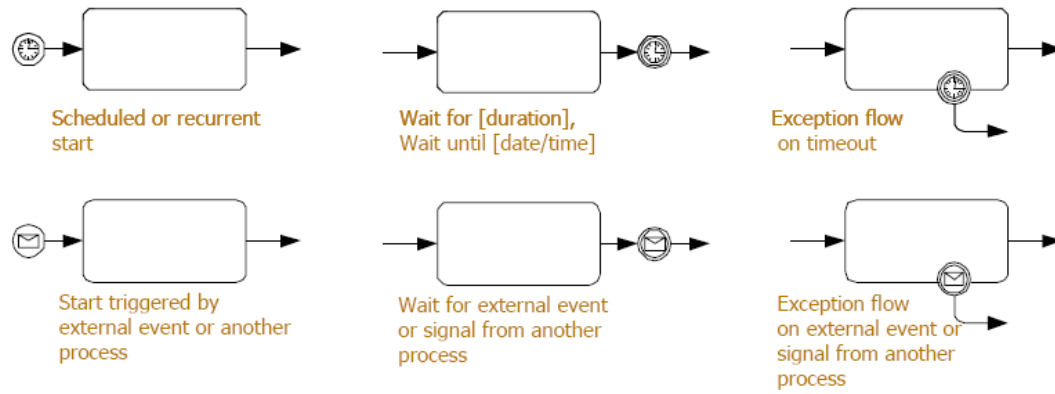
In parallel with A', there is a paired wait-for Signal event, listening for the signal that the event E has occurred. If this occurs, Exception task runs in parallel with A, and both the normal flow out of A' and the exception flow are enabled in parallel. This is just what we want. Merging of the normal and exception flows downstream in a join is tricky, since if E does not occur, the join cannot complete, since one input is still waiting at the Signal event. The solution is to let the normal flow downstream from Next Normal end in another Terminate event, which avoids the deadlock.



Getting Started: Focus on Basic Patterns

The variety of BPMN events can seem overwhelming when you get started, but in practice you can do most of what you need with just message and timer events. The important thing is to understand the differences between each of the basic patterns. Use the cheat sheet below, and you're on your way to becoming an BPMN event expert.

BPMN – Mastering BPMN Events



Bruce Silver