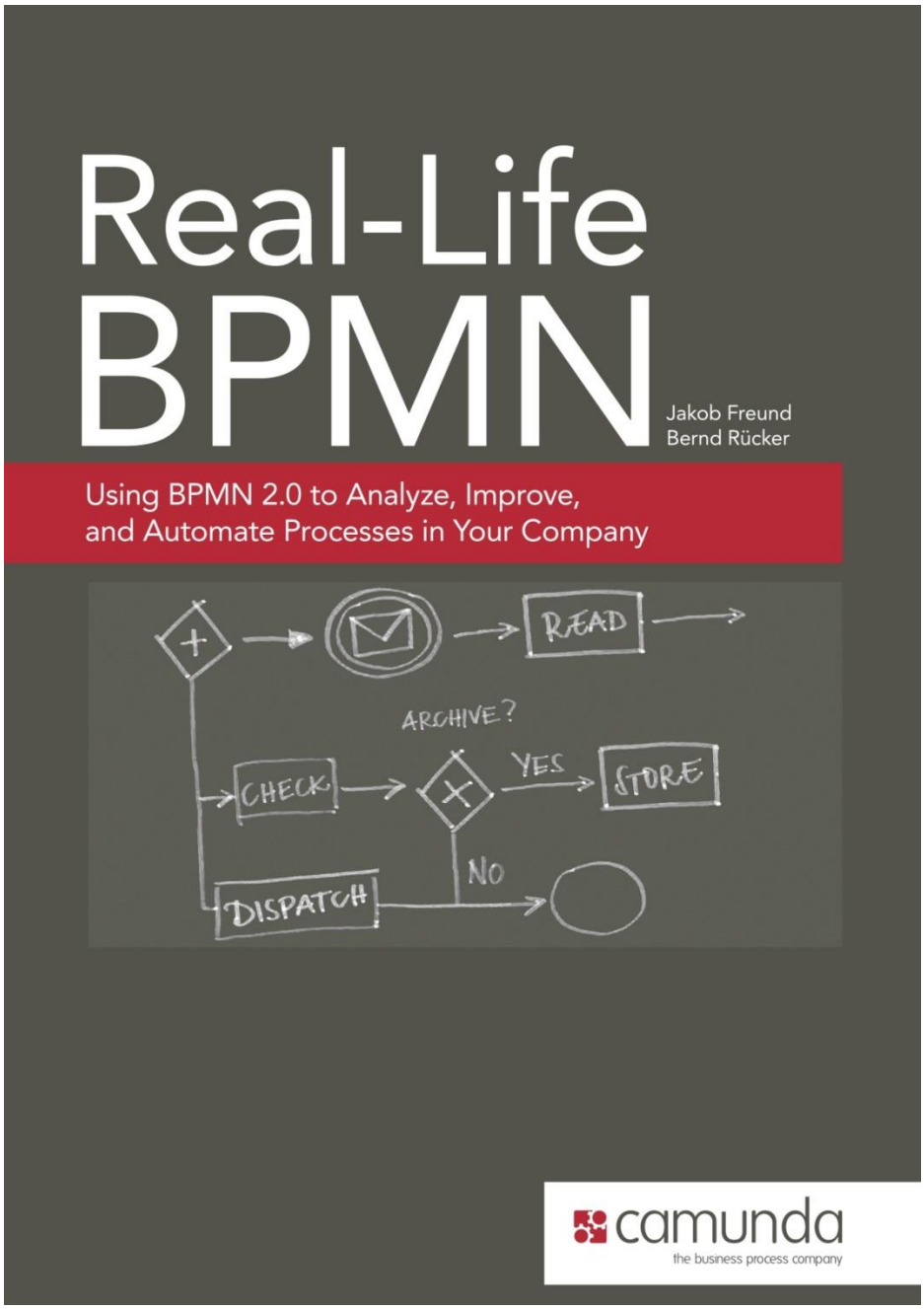
The following is a free excerpt from the book *Real-Life BPMN*.
It can be used as study material in preparation for the training for certification according to OCEB. *



*Disclaimer: Please note that this excerpt does not represent the complete study material required in preparation for the training for certification according to OCEB.

Real-Life BPMN

Jakob Freund
Bernd Rücker

Real-Life BPMN

Using BPMN 2.0 to Analyze, Improve, and Automate Processes in Your Company

Jakob Freund, Bernd Rücker
Founders of camunda services GmbH, Berlin, Germany
www.camunda.com

This first edition in English is based on the successful third German edition.
Also available in Spanish.

Editing for the English-language version of this book provided by James Venis of Lakewood, Colorado, USA, with assistance from Kristen Hannum.
www.jvenis.net

ISBN-10: 1480034983
ISBN-13: 978-1480034983

Contents

" "
...

2.3.4 Default flow and getting stuck

There's another aspect to working with XOR and OR gateways. (To simplify matters, let's set the salad aside for now and focus on real meals.) What happens if we want neither pasta nor steak? In the previous models, this situation meant that our token could never get beyond the XOR split for "desired dish." According to the BPMN specification, that "throws an exception." In other words, a runtime error occurs.

Don't get angry because we are talking about throwing exceptions! (We'll come back to this issue and show why it doesn't concern only IT.)

The so-called default flow protects us from runtime errors. We indicate the default flow with the small slash shown in figure 2.15. The principle behind default flows is simply that all outgoing paths are examined; when none of the other paths apply, the process uses the default. Don't mistake the default flow for the usual flow, however. The symbol does not mean that the default applies most of the time. That's a different question.

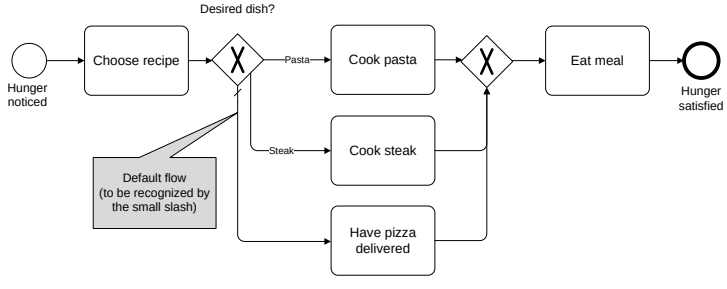


FIGURE 2.15 The default flow.

Our BPMN Etiquette

You don't have to use a default flow, of course. You can draw a normal sequence flow instead and label it "other" or whatever you like. We use the default flow any time there's a risk of getting stuck, and we want to avoid disruption to the organization. If a diagrammed decision has Yes or No outflows only, risk is zero; more complex decisions present greater risk.

In our models, default flows help us to determine if we have limited the risk of getting stuck. In terms of aligning business and IT goals, this is certainly good business practice.

2.3.5 Complex gateway

The complex gateway is a category apart. While it isn't used often, there are situations that justify its use. An example: we want to order pizza. We peruse the menu of our favorite supplier, but just for a change, we also look on the Internet. Once we find something we want to try after researching *both* sources, we order the pizza.

" "
...

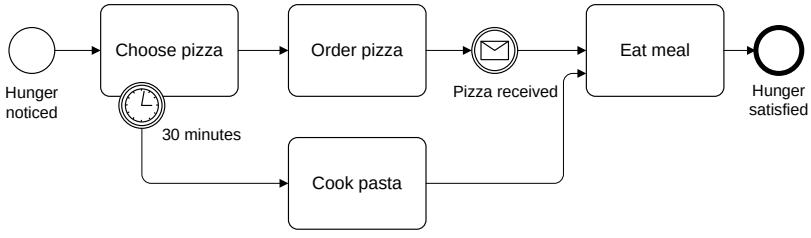


FIGURE 2.44 The timeout for the "choose pizza" task is 30 minutes.

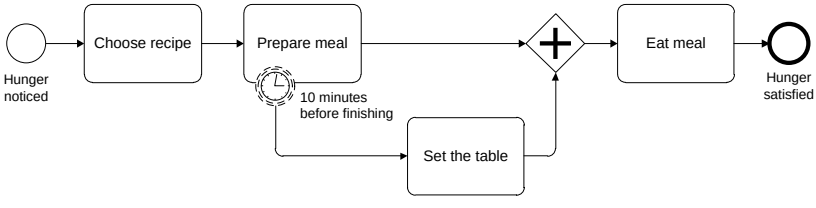


FIGURE 2.45 You can attach timer events that do not lead to cancellation, but instead generate another token.

2.6.4 Error events

Do your processes run completely error-free? If not, you can identify potential errors in your models as a step toward eliminating them, or as part of modeling escalation processes. In BPMN, error events are represented by a flash symbol. Apply them as shown in figure 2.46.

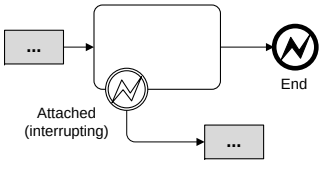


FIGURE 2.46 Applying an error event.

The BPMN specification does not specify what an error may be. As the modeler, you have to decide that. Section 4.5.1 on page 127 has some hands-on tips.

An error is a serious event in BPMN, so it can be modeled only as an attached intermediate event. This means that an error during task execution must be handled in a specific way: As a throwing event, it can be modeled only at the end of a process path so that the participant knows the process has failed. The parent process should likewise recognize the failure. (We explain the interaction between parent and subprocesses in section 2.8 on page 57. You'll also find an example of applying an error event there.)

2.6.5 Conditional

Sometimes we only want a process to start or to continue if a certain condition is true. Anything can be a condition, and conditions are independent of processes, which is why the condition (like the timer event) can only exist as a catching event (figure 2.47).

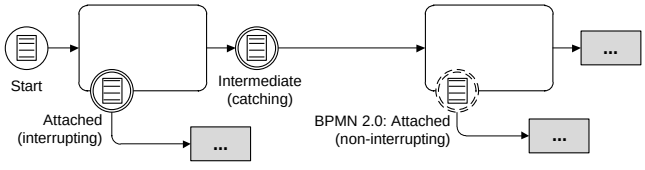


FIGURE 2.47 Applying a conditional event.

We can enhance our pizza process with conditions. If we want to have frozen pizza, the process starts as shown in figure 2.48. We take the pizza from the freezer and turn on the oven. But only after the temperature in the oven reaches 180°C do we put the pizza in, and only after it is done do we take it out to eat.

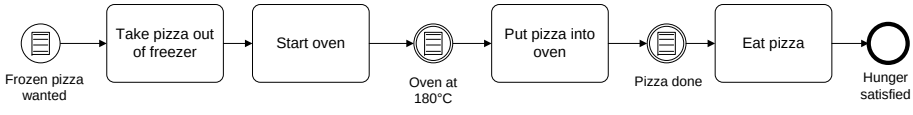


FIGURE 2.48 Baking pizza under fully modeled conditions.

If we know how long the pizza needs to cook, we can specify this in the process model by substituting a timer event for the last conditional event. The whole thing would then look as shown in figure 2.49.

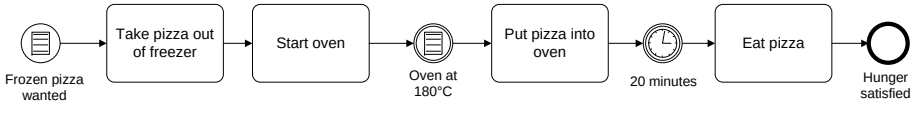


FIGURE 2.49 Baking pizza with indicated baking time.

2.6.6 Signal events

Signals are similar to messages, which is why you can model them in BPMN as events just as you can with messages (figure 2.50 on the facing page). The symbol for a signal is a triangle. The essential difference between a signal and a message is that that latter is always addressed to a specific recipient. (An e-mail contains the e-mail address of the recipient, a call starts with dialing the telephone number, and so on.) In contrast, a signal is more like a newspaper advertisement or a television commercial. It is relatively undirected. Anyone who receives the signal and wants to react may do so.

We saw a new frozen pizza on TV, and we are keen to try it. Figure 2.51 on the next page illustrates this new situation. We buy the pizza, but we keep it in the freezer until we're

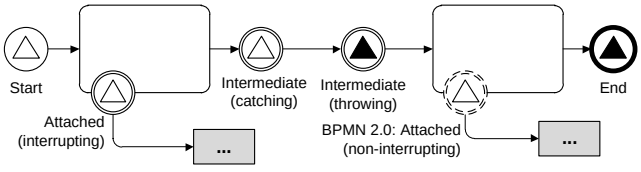


FIGURE 2.50 Applying the signal event.

really hungry for pizza. That's a conditional event. After trying the new pizza, we go to Pizzatest.de to rate the new product. That's a signal. It is a signal for the general public too. (Pizzatest.de actually exists, by the way, which proves again that you can find simply *everything* on the Internet!)

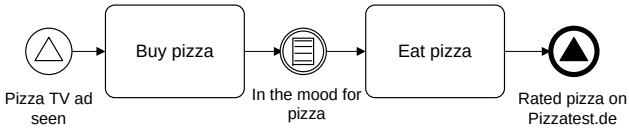


FIGURE 2.51 Pizza signals.

2.6.7 Terminate events

Let's look at the abstract example in figure 2.52. We already discussed (simple) Key Performance Indicator (KPI) analysis in section 2.3.2 on page 20, and we therefore know that this process always takes 55 minutes. After task 1, tasks 2 and 3 can be processed simultaneously. Processing task 2 takes more time than does processing task 3, which is why it determines the runtime of the process. A token that runs through the process is cloned in the AND split. The first token stays in task 2 for 45 minutes; the second token stays in task 3 for 30 minutes. The second token arrives at the none event first, where it is consumed. After 15 more minutes, the first token arrives at the upper none event, where it is consumed too. Since no more tokens are available, the process instance finishes after 55 minutes.

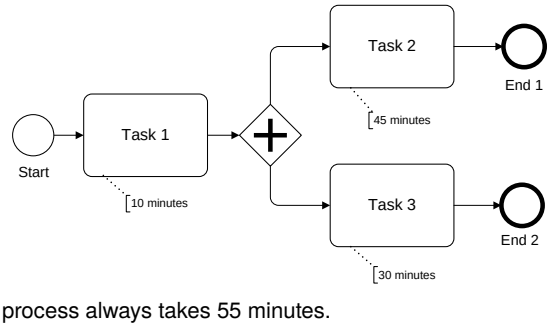


FIGURE 2.52 The process always takes 55 minutes.

So far, so good, but what happens if we already know that, after having completed task 3, task 2 has become redundant? This is a frequent situation with parallel task executions related to content. In such cases, we can apply the pattern shown in figure 2.53.

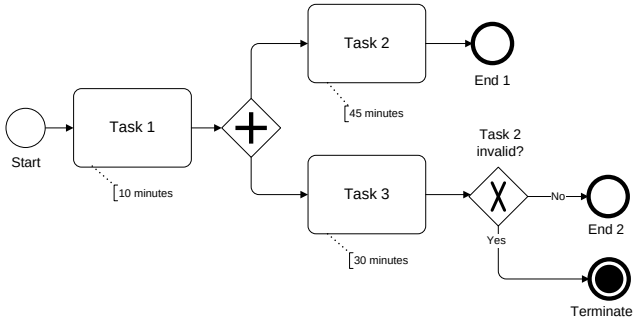


FIGURE 2.53 Potentially, the process terminates immediately after task 3 completes.

We use the terminate event to make sure that *all* available tokens are consumed immediately. That leads to the termination of the process instance, consequently, you can use the terminate event as an end event only. (See figure 2.54.)

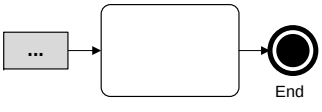


FIGURE 2.54 Applying a terminate event.

2.6.8 Link events

The link event is a special case. It has no significance related to content, but it facilitates the diagram-creation process. As shown in figure 2.55, you can draw two associated links as an alternative to a sequence flow. Here, "associated" means there is a throwing link event as the "exit point," and a catching link event as the "entrance point," and the two events are marked as a pair —in our example by the designation "A." Sometimes we use color coding to mark the association.

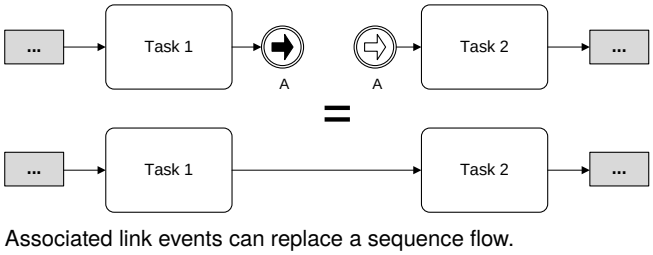


FIGURE 2.55 Associated link events can replace a sequence flow.

Link events can be very useful if:

- You have to distribute a process diagram across several pages. Links orient the reader from one page to the next.
- You draw comprehensive process diagrams with many sequence flows. Links help avoid what otherwise might look like a "spaghetti" diagram.

Link events can be used as intermediate events only (figure 2.56).

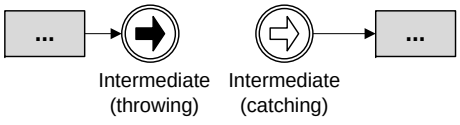


FIGURE 2.56 Applying a link event.

2.6.9 Compensation events

In practice, we apply compensation icons (see figure 2.57) only to transactions even though BPMN permits other uses. (See section 2.8.5 on page 65.) We execute tasks in our processes that sometimes have to be canceled later under certain circumstances.

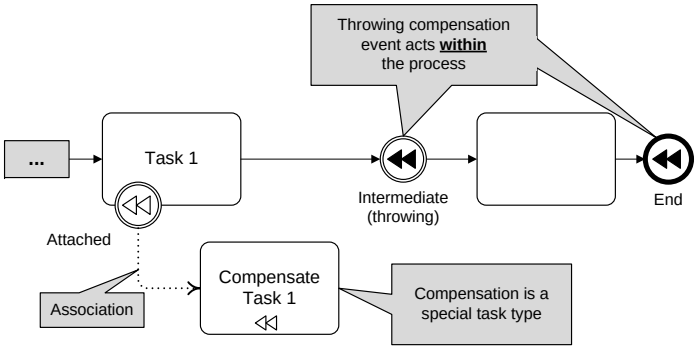


FIGURE 2.57 Applying a compensation event.

Typical examples are:

- Booking a train or airline ticket
- Reserving a rental car
- Charging a credit card
- Commissioning a service provider

In figure 2.58 on the following page, we see this process: On Friday at 1 p.m. we agree with our partner either to go to the theater or to spend the evening with friends. In both cases, we have to do something binding, either to reserve the theater tickets or make the arrangements with our friends. When evening arrives, perhaps we no longer feel like going out at all. We then have to cancel the arrangements we made with the theater or our friends before we can collapse in front of the TV in peace.

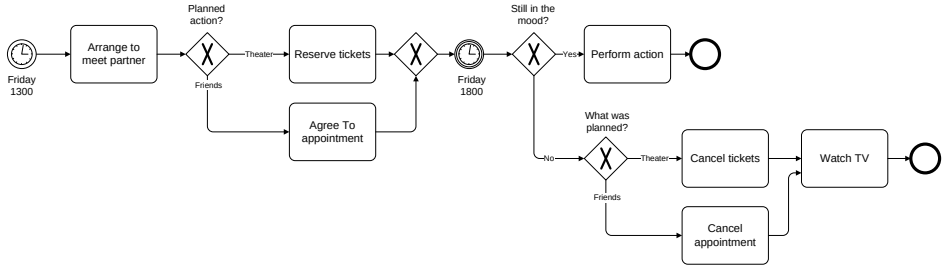


FIGURE 2.58 A possible process for the weekend.

We can represent the latter part of the model more compactly with a compensation event, as shown in figure 2.59. If we don't feel like going out, we have to cancel all our arrangements; we don't have to check which ones to cancel.

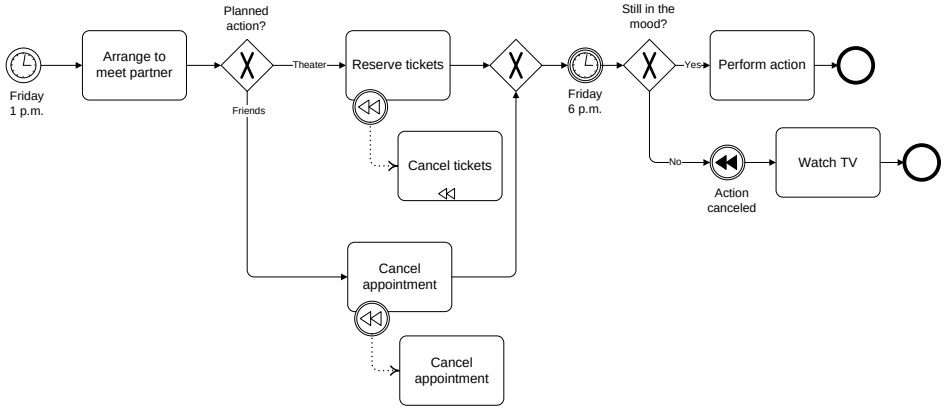


FIGURE 2.59 The same process as shown in figure 2.58, applied to the compensation event.

There are special rules for handling compensations:

- Throwing compensations refer to their own processes, so the event is effective within the pool. This shows how this event type differs from a throwing message event.
- Other attached events can take effect only while the activities to which they are attached remain active. In contrast, an attached compensation takes effect only if the process triggers a compensation *and* the activity to which the compensation is attached successfully completes.
- Attached compensation events connect to compensation tasks through associations, and *not* through sequence flows, which would otherwise be common usage. BPMN thus emphasizes that compensations are beyond the regular process sequence; executing one is an exception.
- The obligatory compensation task is a special task type that we explain with other task types in section 2.7 on page 52.

Our BPMN Etiquette

This example may be too simple to illustrate how much work this construct can save you. If you think of the complex business processes that frequently require compensations, however, you'll see how much leaner your models can be. You'll also be quick to spot the circumstances that demand compensations. We use compensation events only occasionally to describe complex processes.

2.6.10 Multiple events

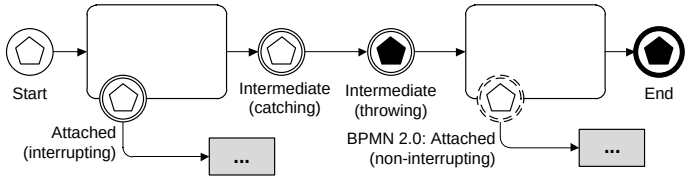


FIGURE 2.60 Application of the multiple event.

We can use the multiple event (figure 2.60) to summarize several events with a single symbol. The semantics are simple:

- If we model the multiple event as a catching event, *only one* of the summarized events has to occur to start or continue the process or to cancel the task.
- If we model a multiple event as a throwing event, it means that *all* of the summarized events are triggered.

Figure 2.61 applies the multiple event to our pizza scenario. In the example, we try a new pizza after having seen it on TV or after a friend recommended it. After eating it, we will rate the pizza on Pizzatest.de and in turn inform our friend if we also recommend this pizza.

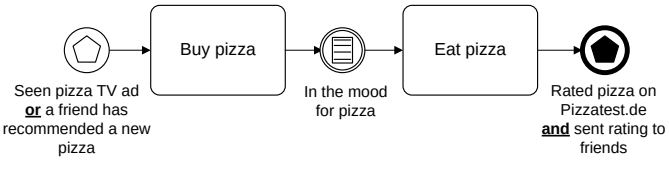


FIGURE 2.61 The multiple event summarizes events.

The model in figure 2.62 on the following page describes the same process, but the events are fully modeled.

Our BPMN Etiquette

You have to decide if multiple events serve your purposes. We concede their benefit in rough functional process descriptions, but they cease to be as useful in the

" "
...

many matters that seem self-evident, or that remained unconscious or forgotten in the process design. Formalization is our best chance of keeping up in a fast-changing environment with ever more complex processes.

2.8.3 Attached events

We already learned about intermediate events that can be attached to tasks. The same events can be attached to subprocesses as well, which opens up a wide range of opportunity in process modeling. As shown in figure 2.83, we can represent how a spontaneous dinner invitation leads to canceling our cooking process. In the process shown, however, we could ignore the invitation if our meal had already been prepared and we already ate it.

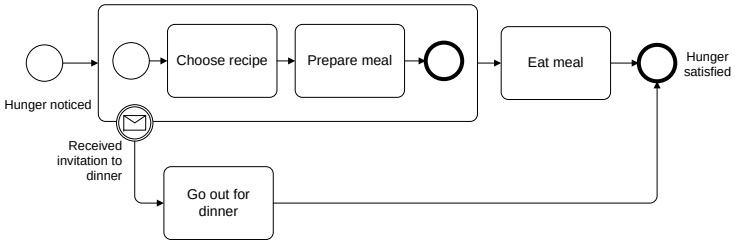


FIGURE 2.83 The catching event cancels the entire subprocess.

Where message, timer, and conditional events are involved, the parent process always aborts the subprocess when reacting to external circumstances. With error, cancellation, and escalation events, however, the subprocess reports these events to the parent process. This isn't as abstract as it may sound.

In the bottom right of figure 2.84 on the facing page, the item procurement task can fail because the item is no longer available. Because item procurement is a global subprocess, it triggers an error event to tell the parent process that something went wrong. In business terms, this may mean that the customer who wanted to buy the item tells a salesperson that his or her order failed because the item is out of stock. A clerk then orders more of the item to replenish inventory.

It is interesting that parent processes can handle the error message differently. While the disappointed customer must be informed within the scope of the order process, it is sufficient for the stock maintenance process to delete the item from the catalog. The respective parent processes decide what circumstances require canceling the subprocess and what happens next. That's a principle that you can use to build flexible and modular process landscapes.

The signal event serves two functions. A parent process can react to a signal received from the outside while it executes a subprocess —this is much like a message event. But we also use the signal event to let the subprocess communicate things other than errors to the parent process. Primarily, this is because we can't model this type of communication with message events. BPMN assumes that we always send messages to other participants who are outside of our pool boundaries; the communication between parent and subprocess

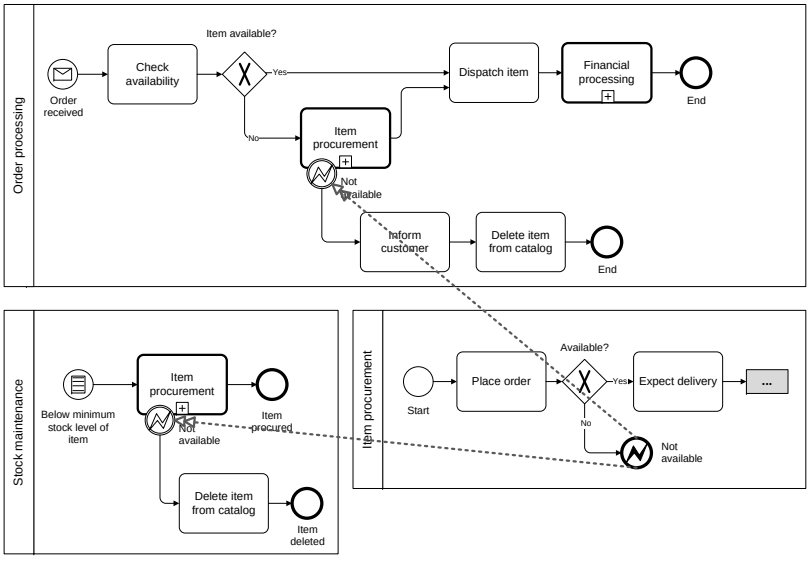


FIGURE 2.84 The subprocess reports an error to its parent.

doesn't fit that mold. We don't use signal events for directed communication, but rather to broadcast information akin to advertisements on the radio.

A better alternative provided in BPMN 2.0 is the escalation event (see figure 2.85).

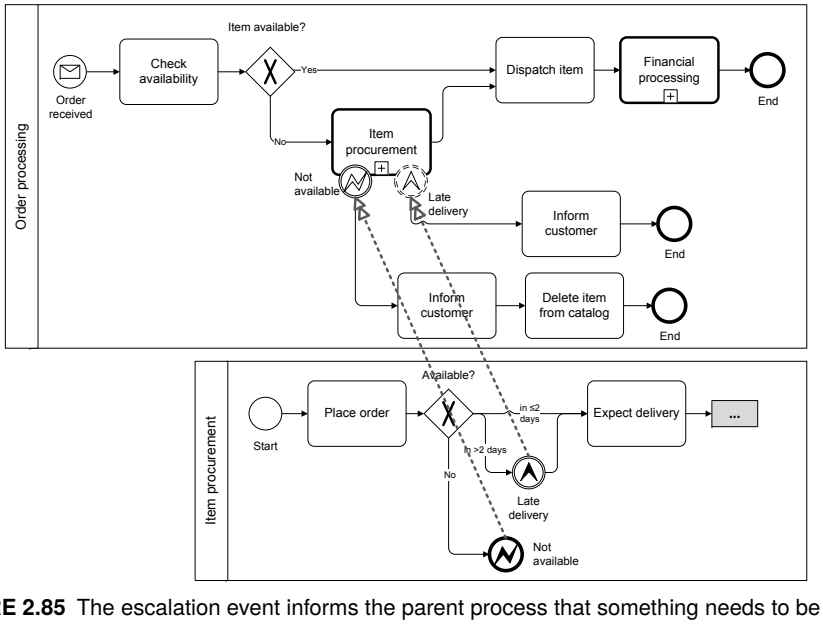


FIGURE 2.85 The escalation event informs the parent process that something needs to be done (available as of BPMN 2.0).

The subprocess can use an escalation event to report directly to the parent process, and the message won't be regarded as an error message. Also, the parent process can receive and process messages from escalation events without canceling the subprocess because non-interrupting intermediate events can be attached .

2.8.4 Markers

You can apply the loop, multiple instance, and compensation task markers that we described in section 2.7.2 on page 54 in a fashion similar to the way you apply subprocesses. You can use them to model even complex loops as shown in figure 2.86. The top and bottom parts of this diagram are equivalent.

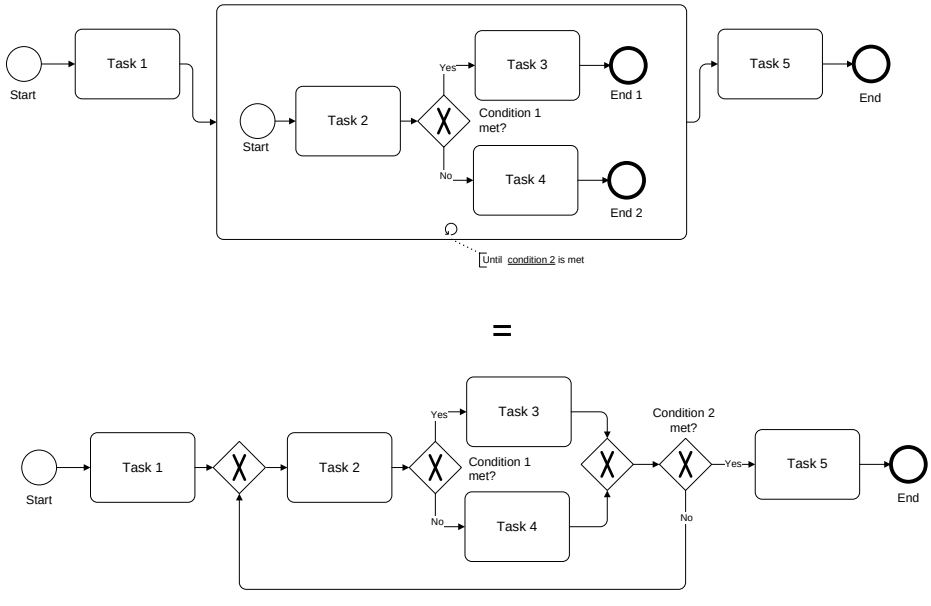


FIGURE 2.86 Subprocesses can be defined as loops.

One marker available only for subprocesses is called ad hoc. Recognize it by the tilde character as shown in (figure 2.87 on the facing page). Use the ad hoc subprocess to mark a segment in which the contained activities (tasks or subprocesses) can be:

- Executed in any order,
- Executed several times, or
- Skipped.

Any party who executes this subprocess decides what to do and when to do it. You could say that the "barely structured" nature of what happens inside this subprocess reduces the whole idea of process modeling to an absurdity because what happens and when are the things we most want to control. On the other hand, this is the reality of many processes, and you can't model them without representing their free-form character. Frequent examples are when a process relies largely on implicit knowledge or creativity, or when

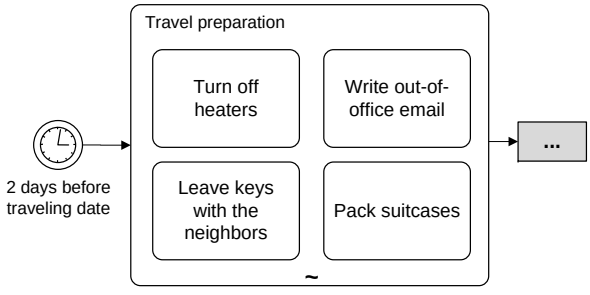


FIGURE 2.87 Travel preparation may include, but does not have to include, these tasks.

different employees carry out a process differently. You can use the ad hoc subprocess to flag what may be an undesirable actual state. Doing so could be a step on the path to a more standardized procedure.

BPMN 2.0 specifies which symbols must, which may, and which are forbidden to occur within an ad hoc subprocess. They are:

- **Must**: Activities
- **May**: Data objects, sequence flows, associations, groups, message flows, gateways, and intermediate events
- **Forbidden**: Start and end events, symbols for conversations and choreographies (discussed later)

By means of the specification, mixed forms —so-called weakly structured processes —can be modeled as shown in figure 2.88.

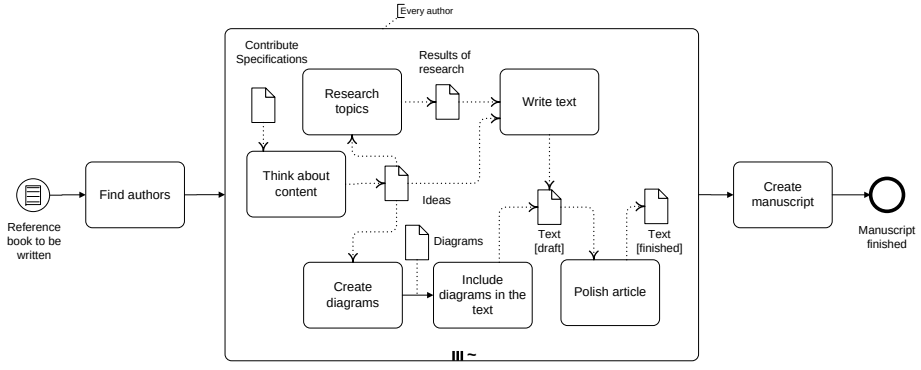


FIGURE 2.88 The processes of the individual authors are not subject to a predefined structure.

2.8.5 Transactions

Many processes work in an all-or-nothing fashion: either all steps must be carried out successfully or nothing must be done at all. The compensation event, which we discussed in section 2.6.9 on page 45, can undo tasks already completed without having to model

the undoing in detail. The transaction is a special subprocess, which also helps us in such cases. We explain this in figure 2.89 using the following example:

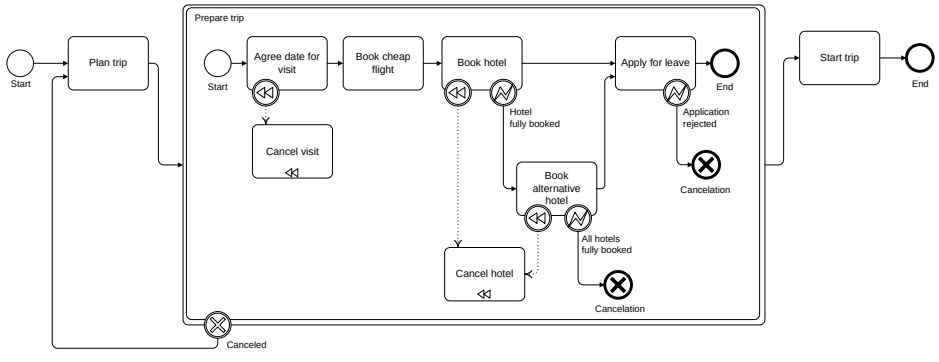


FIGURE 2.89 The double border marks a transaction, in this case the expanded subprocess "travel preparation."

Suppose you want to visit relatives overseas. After deciding to make the trip, you start preparing for it. First, you make a firm plan with your relatives regarding the date and length of your visit. Second, you book a discount flight, and you reserve a hotel room (to avoid being an excessive burden on your hosts, despite their protests to the contrary). Third, you schedule vacation time with your boss. If all goes well, you can start the trip.

What happens, however, if the hotel you wanted is booked and you can't find another? What if the boss denies your vacation request? You have to cancel the travel preparation transaction. The cancel event exists for this purpose. You can only use it within transactions. If you cancel a transaction, it triggers a compensation of all tasks to which corresponding compensation tasks were assigned. You therefore sadly inform your hosts that you won't be able to come at the agreed date, and you cancel the hotel reservation, if any. Because you booked a discount flight, the airline will not refund the ticket price. (You curse them silently.) After compensating for all the tasks already executed, you leave the transaction through the attached cancel event, and you start over with preparations for a different travel plan.

This process is flawed. Because of the cursed airline's poor refund policy, it would make more sense to book the flight only after the other details are confirmed. Book it at the end of, or outside of, the transaction to minimize risk. That's the point: transactions are for critical processes in which even the smallest risk has to be taken into account. If you arrange your vacation time with the boss in advance, the risk of having your request rejected seems small, but it hasn't been reduced to zero, has it? An important bit of work may come in, and your non-binding agreement with the boss may evaporate. Transactions provide security for scenarios like this.

2.8.6 Event subprocesses

BPMN 2.0 introduced a completely new construct, the event subprocess. We locate an event subprocess within another process or subprocess. Recognize them by their dotted-line frames.

A single start event triggers an event subprocess, and this can only happen while the enclosing process or subprocess remains active. For event subprocesses, there can be interrupting (continuous line) and non-interrupting (dashed line) events. This is the same differentiation made as for attached intermediate events. Depending on the type of start event, the event subprocess will cancel the enclosing subprocess, or it will execute simultaneously. You can trigger non-interrupting event subprocesses as often as you wish, as long as the enclosing subprocess remains active.

Okay, that's pretty abstract, but we can demonstrate how an event subprocess works with an example. (See figure 2.90.)

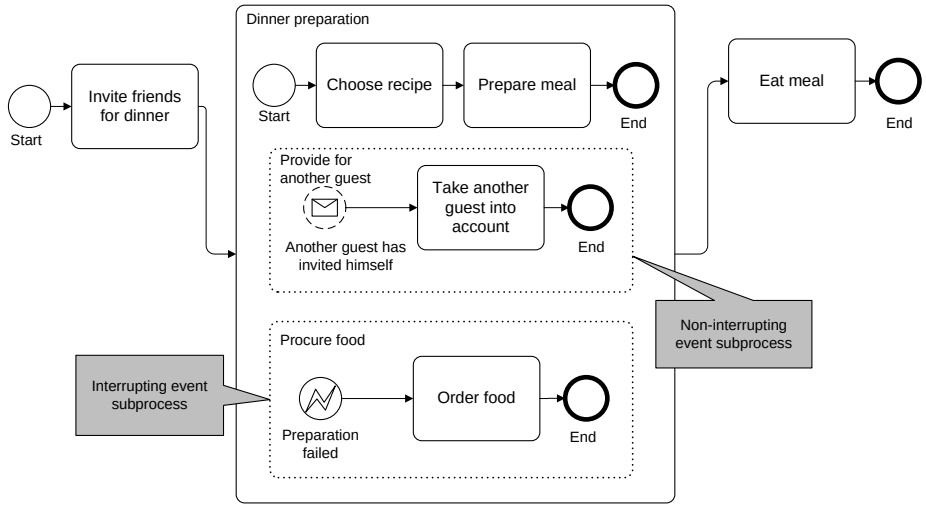


FIGURE 2.90 Event subprocess examples.

We invited a couple of friends for dinner. This starts the "dinner preparation" subprocess of choosing a recipe and then preparing the meal. While we are doing that, the telephone rings. Another guest invites himself to dinner. Spontaneous as we are, we just increase the amount of food or set another place at the table without interrupting the meal preparation. If an accident happens during preparation, however, the error immediately triggers the interrupting event subprocess for remedial action. We order food for delivery. When this event subprocess completes, we exit the enclosing subprocess through the regular exit and attend to eating the meal.

You can see in figure 2.91 on the next page how event subprocesses are represented in collapsed state: The frame is a dotted line, and we have again used the plus sign to represent collapsed subprocesses. In the top left corner, we also have the start event triggering the subprocess.

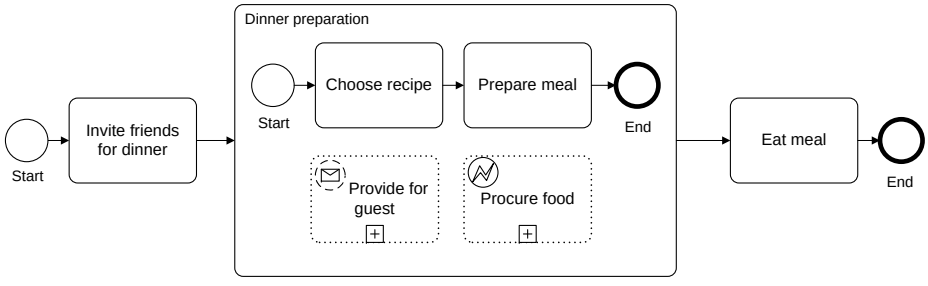


FIGURE 2.91 Collapsed event subprocesses.

The event types that can trigger non-interrupting event subprocesses are:

- Message
- Timer
- Escalation
- Conditional
- Signal
- Multiple
- Multiple parallel

There are two more types for the interrupting event subprocesses:

- Error
- Compensation

Are you wondering if you could model the example without event subprocesses and instead just attach events? Yes, you could. We did it that way in figure 2.92. In terms of sequence, the process works identically to the one shown in figure 2.90 on the preceding page. There is, however, a small but important difference: In the second model, adding an additional guest and ordering the alternative meal do not take place within the "dinner preparation" subprocess, but within the parent process instead. This has the following consequences (which apply particularly to global subprocesses):

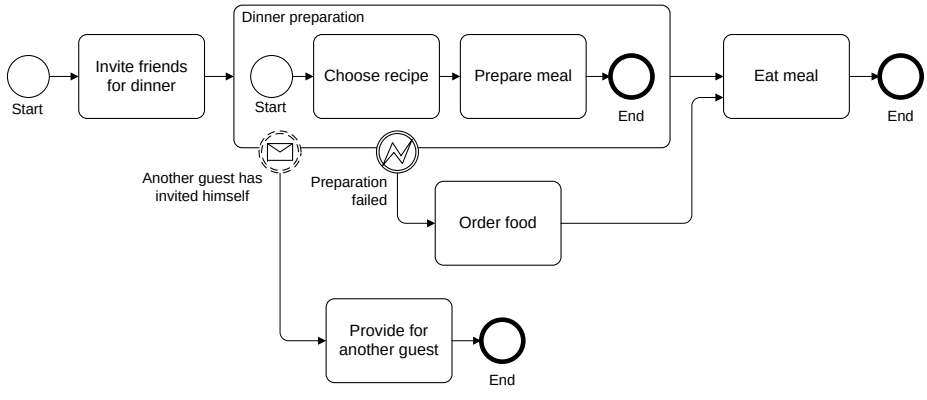


FIGURE 2.92 Compare this process to the one shown in figure 2.90 on the preceding page.

- If responsibility for the subprocess lies with another parent process, two different roles take care of executing the subprocess and handling the related events. If the handling takes place within the subprocess, the same role has to take care of it.
- If the subprocess is global and thus reusable, each parent process must specify how it reacts to both events. On the other hand, if handling takes places within the subprocess, it is reused as well —for good or ill.
- Global subprocesses cannot access directly the data of the top-level process (or their parent processes); some data mapping is required. Data mapping is not required with an event subprocess.

2.9 Pools and message flows

2.9.1 The conductor and the orchestra

In section 2.5 on page 30, we described using lanes to assign responsibility for tasks or subprocesses to different task workers. Lanes always exist in a pool, and the pool boundaries represent process boundaries from start to end. To BPMN, the pool represents a higher-ranking instance compared to its lanes. The pool assumes process control —in other words, it assigns the tasks. It behaves like the conductor of an orchestra, and so this type of process is called "orchestration."

In figure 2.93, the "conductor" arranges for Falko to process task 2 as soon as Robert completes task 1. The conductor has the highest-level control of the process, and each instrument in the orchestra plays the tune the conductor decides upon.

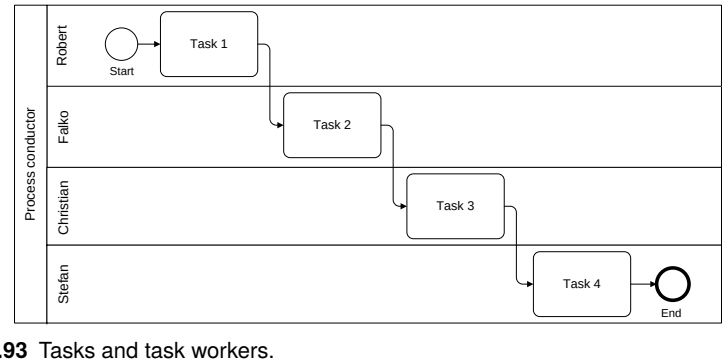


FIGURE 2.93 Tasks and task workers.

Do you think this is unrealistic? Many experienced process modelers have problems with this way of thinking. They would prefer to model a process sequence like that shown in figure 2.94 on the next page on the assumption that no almighty conductor exists in their company, and that individual task workers have to coordinate and cooperate on their own.

To coordinate cooperation with BPMN requires explicit modeling. You assign each task worker a separate pool, and the process passes from one to the next as a message flow (as shown in figure 2.95 on the following page). In principle, this creates four independent

" "
...

The collaboration diagram (figure 4.13 on page 121) shows clearly which parts of the process are directly captured by the process engine's measurements and which are not. We derive this knowledge from the process modeled in its pool:

1. The process engine can measure the running time of these tasks: "write job description," "check job description," "correct job description," and "trigger advertisement"; and of the "advertisement publishing" subprocess.
2. It can also measure the number of times the job description needs to be corrected.

You can analyze these key figures with an appropriate reporting engine. (Or, when in doubt, just use Microsoft Excel for a defined number of process instances, average them, and create colorful diagrams to make your top managers happy.) But we also see which steps the process engine cannot measure. It doesn't see the times that Christine has to repeat her request for clarification to Falko. It therefore can't record the rate of occurrence for these necessary correction loops nor can it store that information for analysis. It also does not know how long a clarification takes. From the engine's point of view, all this is part of the "write job description" task, which it assigned to Christine. This may lead to distorted measurements. You need to be aware of distorted measurements, and of the three ways to handle them:

1. Accept the distortion. After all, you know it is limited to the "write job description" task only.
2. Estimate the rate of necessary clarifications and their average time. Enter these estimates by hand into the database. (Of course, now you have adopted the same practice and the same disadvantage as conventional organizational process management.)
3. Represent Christine's inquiries as a human workflow in the process engine. Then you can measure and analyze the respective rates and times in a differentiated manner. The risk is that neither Christine nor Falko will be thrilled by this idea, and they may simply bypass the process engine and settle their questions with a more efficient phone call.

As you can see, process automation is a powerful instrument for process control, but you should be wary of overusing it. We hope you'll also see that BPMN helps us to recognize these limits in time and to prepare for them.

On the other hand, we have to understand that BPMN in its "raw" form does not provide sufficient support for process analysis based on key figures. You can only do this with a powerful BPMN tool that takes in the key figures —ideally from the process engine —and aggregates them for functional analysis. It can do this usefully because you are a master at producing consistent process models.

4.5.1.2 Explicit modeling of errors

Unlike other notations, BPMN explicitly models errors with event types (section 2.6.4 on page 41). It is only a question of when to use them. In the last section, we discussed correction loops that apply only in case of errors. You would not want to use error events in those cases, because you want to reserve error events for representing activities that *cannot* complete successfully. If an activity can complete, but the *result* differs from what was expected, that's different. It isn't always clear what every situation calls for, and there can be gray areas. Let's look at a simple example.

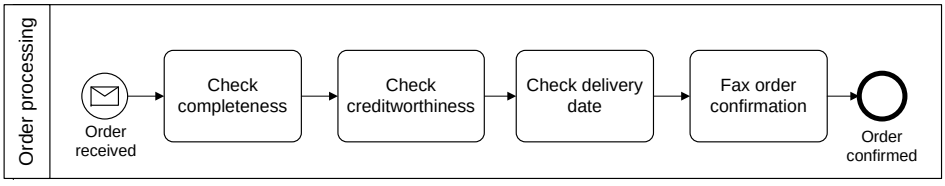


FIGURE 4.20 Happy path order processing.

Figure 4.20 shows the happy path for order processing. There are four steps: For a new order, the order processor checks the order form for completeness, then he or she checks the customer's credit. The date of delivery is checked, and finally, the processor faxes a confirmation.

Now we imagine what can go wrong, what the response is going to be in each case, and how to represent these possibilities in the process model. To think it through, we start at the end and work backwards. The happy path result was that the order was confirmed, so what could lead to the order *not* being confirmed? Yes, theoretically, anything can happen —even an earthquake —but practicality suggests that we deal in events with higher levels of probability. We decide on the following:

1. The order details are incomplete.
2. The order details are illegible.
3. The customer ID is wrong.
4. The customer has insufficient credit.
5. The ordered item is not available.
6. When faxing the order confirmation, someone answers the phone and asks our fax machine if the call is supposed to be a bad joke.

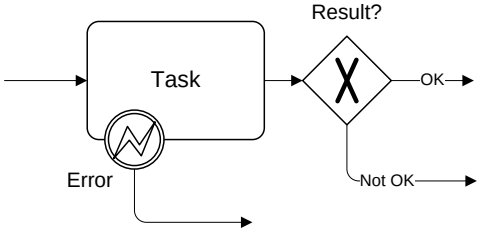


FIGURE 4.21 Representing possible (and probable) problems in the process.

How would we model these contingencies in the process? As shown in abstract in figure 4.21, a task either provides us with a result, that is, information that we can assess as OK or not OK, or there is no result, and the task can't complete at all. If we have the information to assess, we can model an XOR split after the task. If the task can't complete, this is the time for an error event. For each of the possible problems we've defined, we can now construct error-handling solutions. See the fully modeled process in figure 4.22 on the next page.

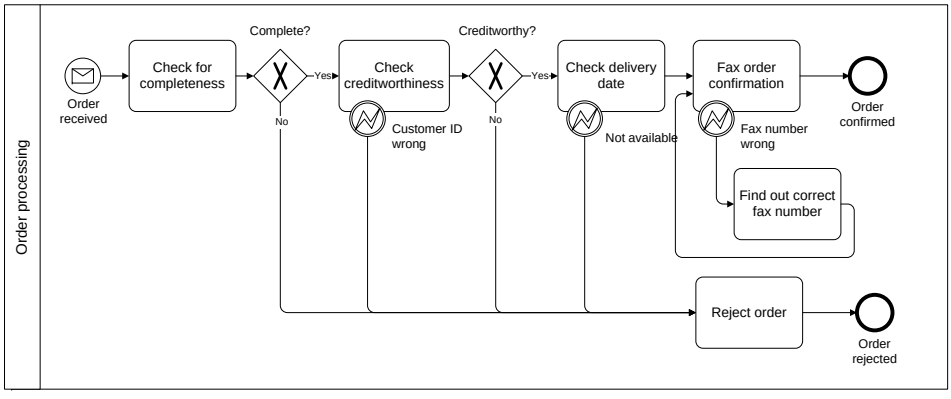


FIGURE 4.22 Representing possible alternatives to the happy path.

- **The order details are incomplete.**

  This is simple: The check for completeness succeeded, but the result is that the order is incomplete, so processing follows the XOR split after the task to "reject order."
- **The order details are illegible.**

  How can you check for completeness when the order is illegible? This isn't as obvious as when the order details are plainly incomplete, but the result is the same. If we cannot read the details, they don't exist for us. The order is still incomplete. (Though may be helpful to explain to the customer *why* his or her order was rejected.)
- **The customer ID is wrong.**

  Unless we can validate that the customer has the correct customer ID, we can't make much more progress on the order. This is a clear case for an attached error event.
- **The customer has insufficient credit.**

  If our process successfully checks credit, but the result is unfavorable, the result prevents the order confirmation. The XOR split after the task sends the order to "reject order."
- **The ordered item is not available.**

  This is not so easy, which is why you need to be pedantic. If the item is not available, no delivery date can be established, so no check of the delivery date can succeed if the item is not available. We have to attach an error event.
- **When faxing the order confirmation, someone answers the phone and asks our fax machine if the call is supposed to be a bad joke.**

  You can probably guess how to represent this.

You may be asking yourself why it's necessary to differentiate between error events and XOR gateways at all. Why not just show all error cases by means of XOR gateways as other process notations do? BPMN cannot keep you from doing that, but we recommend the differentiation because:

- Most people who design processes in the target state consider only some of the possible problems. They model these in "check tasks" and downstream XOR gateways. But when it comes time to implement the process, the IT people come back with questions no

one has taken into account, very often concerning situations that keep the check tasks from completing successfully. As part of the required ping-pong between business and IT, we document these questions with attached error events, and then we answer them specifically. Frequently, when obtaining the answer to such a question, new check tasks develop upstream, along with additional XOR gateways. If you placed a "check availability" task before the "check delivery date" task, for example, you could make the error event at "check delivery date" obsolete.

- Can you ensure processes with error events in case *anything* goes wrong? Yes. You can define this kind of extra safety net with an attached error event for a section within a process, or for an entire process.
- In general, XOR gateways differentiate between cases. They can differentiate between error conditions, and they can differentiate also between or among positive outcomes. An example would be the different steps to determine correct size depending on if the customer has ordered trousers versus a hat. So a happy path may not exist without XOR gateways, but positive XOR gateways cannot be differentiated visually or syntactically from the error XOR gateways. Error events are visually less ambiguous. Provided you have the appropriate tooling, you can even use them to switch between a simplified, happy path view and the complete view of the process.

The bottom line: Error events can be a good aid when modeling processes, and you should make use of them.

4.5.2 The true benefit of subprocesses

By now, you know the several levels of our BPMN framework, and how the levels use models with different amounts of detail. At level 2, we also work with different views of the same process to show only the aspect of the model that is most useful to the people affected.

Have you wondered what role the BPMN symbol for subprocess plays in this framework? We show in section 2.8 on page 57 that subprocesses have three main purposes in BPMN:

- To hide the complexity of detailed sequences. Collapsed subprocesses make diagrams more clear.
- To make sequences modular, and thus reusable.
- To define a scope within a process and then define how the process reacts to catching events for the scope by attaching the events.

You can benefit from all these advantages at all three levels of our framework. For instance, in the process model for "Job advertisement," we defined the "publish advertisement" subprocess in the process engine's pool to avoid overloading the diagram with interface calls (figure 4.13 on page 121). Another option would be to define error processing for the entire subprocess by attaching an error event. Because we likely won't need this subprocess later, we should not define it as global in our BPMN tool. Because the subprocess is embedded, and no longer an independent module, it can be collapsed for clarity.

In BPMN, subprocesses are seldom used to indicate organizational refinements of a process. This is why we often mix tasks and subprocesses in the same diagram. Some

" "
...

5 Level 3: Executable process models

5.1 About this level

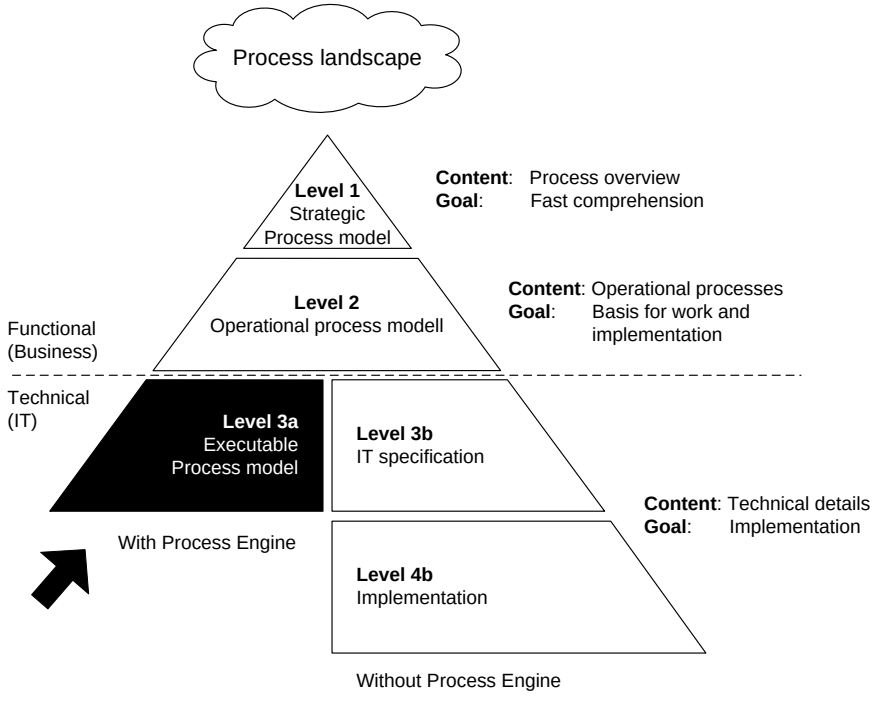


FIGURE 5.1 camunda BPMN framework: level 3.

5.1.1 Purpose and benefit

The third level deals with automating processes with software. As we pointed out in the last chapter, you can use conventional software development for this, but it is much more

interesting to apply a process engine, especially when you're trying to align business with technology.

There are process models produced at level 3 that can be executed directly. In other words, the model serves as the source code for a software solution. If that's your intention, you must remember that the models have to be defined in a very exact and detailed way, because process engines do not understand what "room for interpretation" means!

A major advantage of the model-as-source-code concept is that, at minimum, executable process models always show the actual state of those parts of a process that are automated by a process engine. To change those parts, you have to adapt the model.

This brings up a second important aspect of the caBPMN framework: if the process models at the second and third levels are linked intelligently with suitable tools, you can keep the operational process model at level 2 up to date! You can display the key performance indicators, as measured by the process engine, in the operational model as process control. Yes, many others have claimed this power, but our framework differs from previous approaches in that the operational model at level 2 must be modeled with precision. That's a major challenge for the process analyst. Many suppliers have tried to hide the complexity in their operational models, but regrettably, that has never worked out. We therefore believe that levels 2 and 3 can facilitate the alignment of business and technology —providing that an appropriate tool is available. (We talk about tools in section 6.4.2 on page 205.)

5.1.2 Model requirements

Process diagrams at level 3 must be syntactically and semantically correct, but they also must contain all the technical details necessary for automation with the process engine. The model must be precisely built. It must leave no room for interpretation. And all cases of technical error or exceptions have to be dealt with too. After all, this is source code!

5.1.3 Procedure

The procedure for developing a level 3 process model determines a successful implementation. This is where business collides with IT. In our experience, prospects for success depend not only on the process analyst's skill, but also on his or her collaboration and communication with the process engineer.

The procedure typically has the following steps:

1. Clarify the target state process at level 2, which we discussed in the last chapter.
2. Decide on the technology, language, and process engine. (See section 5.2 on the next page and section 5.8 on page 184.)
3. If you apply a BPMN 2.0 engine (see section 5.3 on page 150) all you need to do is refine and detail the operational model with technical details. If you apply another modeling language, you need to map the level 2 model to the executable model.
4. Iteratively refine and specify the level 2 model if new questions arise.
5. Test and execute the process with established methods of software development.

So far, we've examined only the sequence flow aspect of process control. For technical implementation, it is essential to reconcile other aspects of software technology. (See figure 4.14 on page 123 especially.) In section 5.3 on page 150, we examine some technical aspects still missing in the operational model, such as:

- Specifying data in the desired technology such as XML or Java.
- Defining service calls in the desired technology, for example, Web Services.
- Detailing human tasks such as assigning users to groups or determining the forms to be displayed.

Before diving into the details of creating level 3 diagrams, we want to provide some notes on reading this chapter, and also to give a general introduction to process automation with process engines.

5.1.4 Notes on how to read this chapter

This chapter goes deeper into the technology than does any other chapter. One effect of this is that we present sections of source code that express BPMN 2.0 models as XML. Though we tried to keep the text comprehensible even without the XML, we figured that some readers would find the source code examples interesting and useful. If the XML is too much information for you, you can safely skip over it.

At the time we wrote the original version of this book, there were almost no examples of BPMN 2.0 XML available. This changed when the OMG supplemented the specification with an official sample document. You can access this at http://www.omg.org/cgi-bin/doc?dtc/10-06-02. If you want to study an example that shows modeling integrated across all three levels, we contributed to one implemented in camunda fox. It includes complete source code. You can see it at www.bpm-guide.de/bpm-java/.

If the gritty details aren't your concern, you can skim the rest of this chapter for general comprehension, and then pass the book to your IT department.

5.2 The basics

5.2.1 Process automation with process engine

The process engine (sometimes called a business process or workflow engine) is software that carries out business processes. You also may hear this called process execution or process automation. The engine requires a model of the business process that contains all technical details necessary for execution. While the process runs, it creates process instances for individual process cycles; the process engine computes the control flow and always knows what to do next. The token concept introduced in section 2.1.4 on page 16 may be useful to recall here. While not every process engine works with tokens, it is not unusual for them to do so.

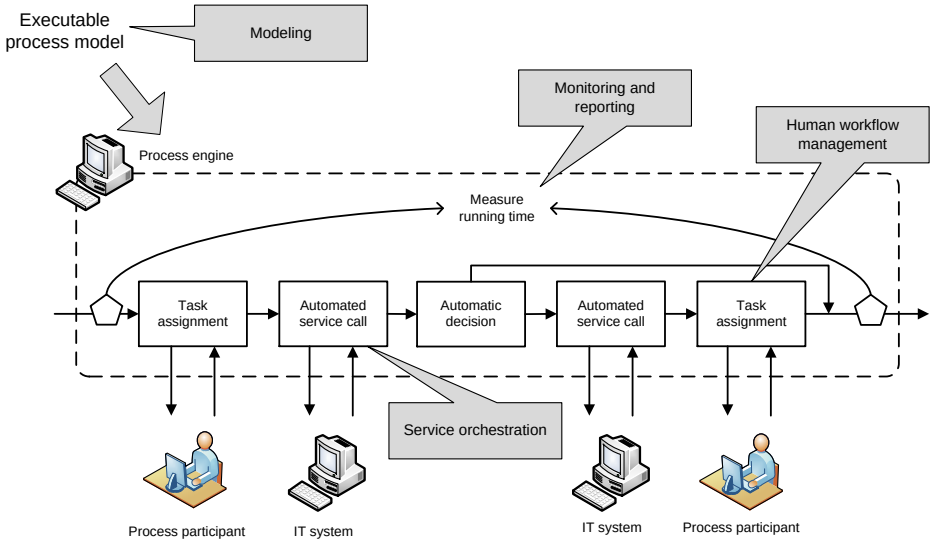


FIGURE 5.2 Process engine principle of operation.

The process engine recognizes two types of activities: automated activities and those that require human interaction. The automated activities may be service calls, for example, but they also may analyze gateways and events. Human interactions are applied through user tasks, usually by means of a task list comparable to an email in-box. The list items indicate to the user which tasks have yet to be executed. When the user opens a task from the list, he or she views it through a pre-configured screen mask to view or edit data or to make a decision. Figure 5.2 provides an overview of process engine operations.

As the user tasks make clear, a process engine has a lot more to do than just to automate a control flow. It has to consider data flow as well. Data can be added to a process instance while the process runs, and the engine manages the data as well as the process state. The data is usually persistent and stored in a database so that it remains available even in case of system failure. A process engine typically provides other services too:

- **Versioning of process models**: Business processes take a long time. An ordering process, for example, may take a couple of days or even several months. This means that when a process needs to change, there are likely to be instances running, and that is why most process engines can process different versions of a process model simultaneously. Transition to a new process version happens by letting the instances of the old version run down.
- **Data collection, performance indicators, and analyses**: The engine can collect data automatically while controlling process instances. We can know exactly when an order was released, for example, and when delivery completed, and so on. We can aggregate the data and use it to analyze efficiency or to identify bottlenecks in the process. This also provides a well-aggregated overview of the entire process landscape. Another option is business activity monitoring (BAM), which aims to recognize patterns in real time and issue warnings or intervene with autonomous controls.

- **Monitoring and administration**: Process engines provide options for viewing the status of process instances. This creates intervention opportunities such as aborting or restarting faulty instances.

Process models must be available in a language suitable for the process engine. We discuss the extent to which these executable models can be synchronized with operational BPMN models, or replaced by executable BPMN models, in the next sections.

5.2.2 Execute process models – is that possible?

Once the idea of a process engine has taken root, we often face one problem: the expectation that a magical BPM suite solves all problems. Figure 5.3 illustrates this wishful thinking. If you feed the suite a purely operational model, it integrates IT systems automatically and takes care of human workflow management. At the end, functional performance indicators tumble out through a dashboard, which the business division uses to recognize problems in real time, and then the problems resolve themselves.

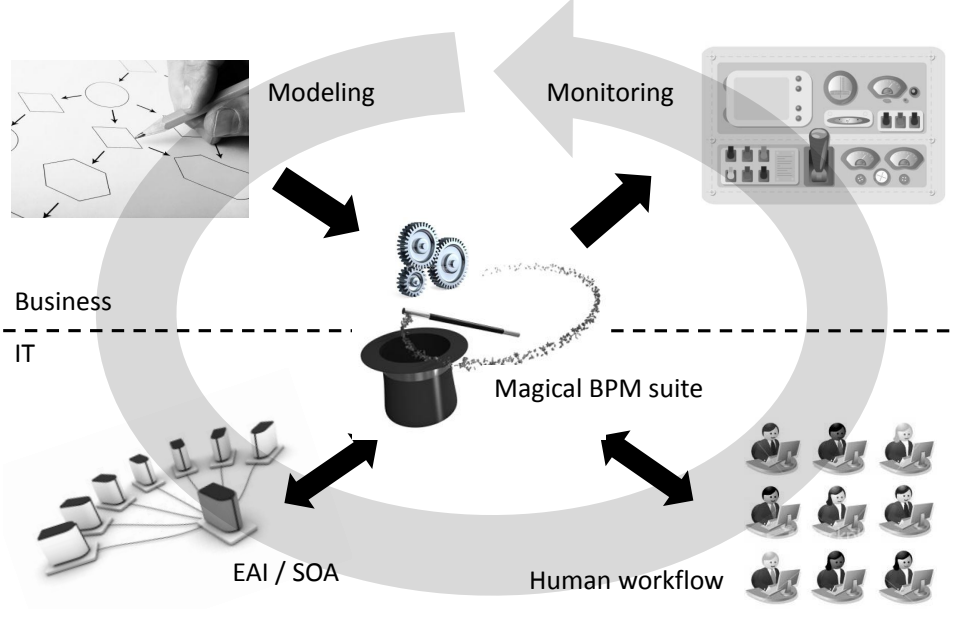


FIGURE 5.3 The magical BPM suite.

The scenario is too good to be true. While the magical suite may illustrate what an organization is striving for (and suppliers of BPM tools don't exactly discourage this kind of thinking), reality may not meet expectations. The resulting disappointment can color an organization's appreciation for BPM methodology.

Do any parts of this vision work? Are operational process models ever executable? If you need to use your own technical models for the execution, how do you synchronize them with the operational models? Is the synchronization achievable by the often-cited BPMN-to-BPEL (Business Process Execution Language) round-trip? All these are

questions to be answered in this chapter. To begin with, we examine the general executability of process models.

There are efficient process engines that can execute process models. It is a misconception, however, to think that the business creates these models directly. An executable process model must incorporate loads of details, and, as source code, it can leave no room for interpretation. Some of that detail is not strictly relevant to a purely operational model; it may even interfere with the job that an operational model is supposed to do, which is to communicate. These things, and the often deliberate informality of an operational model, mitigate against its success as something a process engine can execute. Examples of the things an operational model usually lacks, but which it must have to be executable, include:

- **Data modeling**: Operational models do not require detailed specification of data types; the model may not even mention all the data contained in the process. Exact technical definition, however, is required for the execution. (The definition depends on the target technology, but one example would be an XML schema.) Similarly, the data flow in the process must be specified exactly, including when particular information must be sent to particular systems, as well as any steps necessary in transforming the data.
- **Error cases**: If you include only a certain number of errors in your operational model, technical errors must be taken into account to make it executable. Depending on the architecture and technology, this may affect the process model.
- **Correlation conditions**: As with data modeling, you must define precisely the correlation of messages with process instances. After all, the process engine must be able to forward messages to the correct process instance even when they arrive while the process is running.
- **Process instantiation**: A start event that looks simple in the operational model may not be at all trivial to IT. Say a message starts a new process instance. Technically, this message may need to be picked up somewhere in reality. Or maybe a process instance is supposed to start only after several events have occurred at the same time, or perhaps they must occur in a certain sequence.

These few examples should illustrate why executing a process requires precise and highly detailed models. Our experience confirms that models fit for execution are too detailed to serve as operational models. They simply aren't suitable for creating understanding between users. Usually, a custom executable model has to be developed.

An operational model that can't be executed isn't all bad, even from an IT perspective. At least it is a graphical representation of the process, and compared to conventional programming practice, that's a huge advantage. The use of BPMN, which uses the same notation for operational and executable modeling, gives another advantage. It facilitates the communication between business and IT considerably, even though there is no *single* model. There is simply a need for different models, and an obvious idea would then be to link the models or, even better, to generate one from the other.

You might "forward engineer" an executable model out of the operational one. That's no simple matter, and real problems can arise if the content of one model changes. If the executable model changes, the change may take place in some detail that's invisible to the operational level. If the change in the executable model *is* visible to the operational level, then the operational model has to be changed manually to reflect it. If the operational

model changes, this *always* affects the executable model, and the change either has to be done manually or the executable model has to be re-generated from the operational one. In the latter case, how do you maintain technical changes or amendments? One approach may be to use protected regions in the source code so they are not touched during regeneration. This technique, however, has its problems.

"Round-trip engineering" is supposed to mean that changes to the executable model are represented in the operational model. Their worlds remain synchronized. It is a procedure recommended for applying BPMN and BPEL. Unfortunately, round-trips between these languages don't work with real projects. (See section 5.7.1 on page 179.) Real-world lack of synchronization is a problem common to other approaches of model-driven software development such as Model Driven Architecture (MDA) by the OMG. Interestingly, these approaches have been available for years but have never achieved wide acceptance. We think the issue is linked to the granularity of the modeling: What is *modeled* and what is *programmed*? It is a question we explore more in the in the next section.

Hint: Business-IT-Alignment

You just have to accept that operational models cannot be executed directly. Also, the likely success of generating an executable model from an operational one is overrated. It doesn't work in practice. Alignment between business and technology is achievable, however, if both sides work with process diagrams that are similar. BPMN is a good choice for creating the similar diagrams because it can represent operational as well as executable processes. The caBPMN framework shows how these process models can be aligned in real life.

5.2.3 Modeling or programming?

Perhaps this section title should read *Process engine or classical software development?*, though it's not as nicely provocative. The differentiation between model and program is actually somewhat vague since, strictly speaking, a process model for a process engine is a kind of program code. Our purpose here isn't to debate this. We want to pose a question: is the purpose of the model to work with a process engine, or is it just to create a requirements artifact for classical software development? Of course, there are processes that don't need automation, but to determine if process automation is even worthwhile, ask if the process has the following characteristics:

- **High repetition rates**: The effort to automate is worthwhile only if an appropriate number of process instances execute. Otherwise, development costs may exceed the costs that automation would avoid.
- **Standardization**: If most process instances follow the same pattern, automation offers an advantage.
- **Information-intensive**: Generally, information-intensive processes are most suitable for process automation. The need to move physical objects frequently —including paper —makes automation more difficult to justify and less exciting.

- **High level of automation**: Automating tasks can increase process efficiency. Tasks such as the booking in an ERP system are well suited for this because you don't have to enter data manually. Manual tasks such as calling a customer, however, are not as well suited to automation.

Automation projects don't necessarily save money immediately. And while increased efficiencies are possible, they may be difficult to achieve. To us, however, there remain three more, and much more important, reasons to automate processes:

- **Transparency**: One huge advantage of applying a process engine is that the process is visible as a graphic diagram; it isn't buried deep in the software. This makes it comparatively easy to understand a process. It also facilitates discussions about weak points, possible improvements, and changes. Without a process engine, computer scientists seem like archaeologists: they have to dig to uncover how a process is actually implemented. Only process automation provides the transparency necessary to enable reasonable re-engineering. When an automated process runs exactly as described —and this can be evidenced by log files —it is often a great help to compliance requirements.
- **Agility**: BPM and SOA are often compared to Legos, the interlocking plastic toy bricks. As with Legos, you build your process out of available services. As one of our customers observed, however, "Look at today's Lego kits. They contain highly complex and specific bricks, which you would by no means reuse in other places." To build a complex end product like a Lego spaceship, the simple bricks won't do, nor can you put any brick in any position. Does this mean no agility? No, because the increased transparency makes the adjustment of processes feasible in the first place. It doesn't happen at the push of a button, but at least we can assess the effects of process changes and see where we have to intervene. Furthermore, the visibility of the process leads to services of suitable granularity, and so it supports reasonable modularization of IT systems.
- **Quality**: Imagine any mail-order company. When you inquire about an order, there are two possible answers: delayed ("We have to find out first, please wait a moment") or immediate ("Your order is waiting for xy.") A company that can answer you immediately probably uses a process engine that enables an exact view into the process definitions and process instances. Escalation situations due to long waits are thus not triggered by exasperated customers, but by the process engine itself, and perhaps before the customer relationship is damaged.

All right, so you are convinced to apply a process engine. The next consideration is to determine which aspects are expressed by executable process models and which aspects may be addressed better by classical software development. You can't make that determination without knowing the specifics, but there are several factors to think through:

- **Technology and architecture**: Depending on the process engine used and the overall architecture, it can be easy or difficult to implement certain requirements within the process. Some process engines, for example, can implement simple scripts directly.
- **Available infrastructure**: Few projects are developed with a greenfield approach. Of course you should reuse or integrate available systems and services. Then, for instance, you can trigger processes with an existing scheduler infrastructure and not with the process engine. Marginal conditions must be taken into account.

- **Roles in the project**: It is important not to underestimate existing roles and know-how. Projects often involve developers who are able to implement certain functionality quickly with classical programming, but they need a long time to do it with a process engine. On the other hand, there may be qualified process engineers who can master process models better than they can programming languages.

If you fail to take these factors into account, it can lead to less-than-optimal solutions. It can even lead to solutions that make no technical sense for the target architecture.

Hint: Business-IT-Alignment

The alignment of business and technology does not mean that software can no longer be developed conventionally. It means integrating the process engine and graphical views of processes as additional tools within your architecture. Beware of finely granular process models. Be guided by the operational diagrams, and you'll be pleased to find that your executable models may even be understood by accomplished business users.

In our practice, we often find that the impulse, once the process engine has been procured, is to use it for everything. Overly detailed process models often emerge, in which you can't distinguish the forest from the trees. Such models neither help in communicating with the business, nor are they easier to maintain than conventional program code. IT will also hate the models, which doesn't help anybody. The point is to strive for just the right granularity. Modeled processes are just one piece of the puzzle.

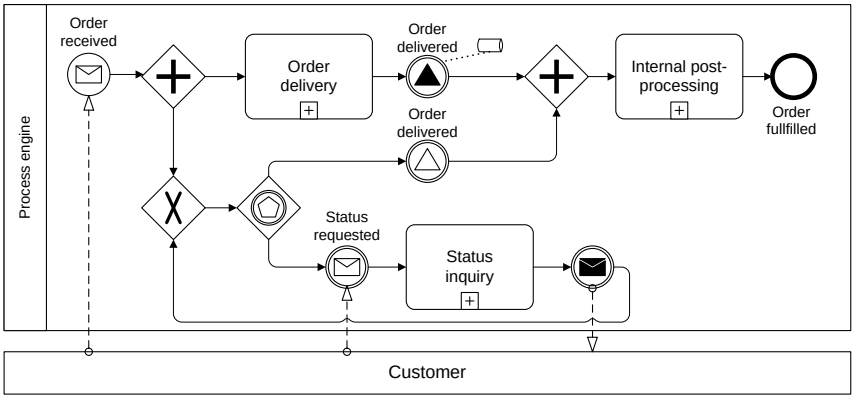


FIGURE 5.4 Bad example: modeling as many aspects as possible in the process.

Figure 5.4 shows an unfortunate example. It models explicitly how a customer inquiry for his current process state is answered. It's irrelevant if we model this process with a signal event (as shown), or a conditional event, or a terminate end event. The process becomes too complicated no matter what. Also, it may be a bad idea to integrate the customer inquiry directly into the order process. Wouldn't it be better to model the inquiry as a separate process or to use a simple service to get the status from the process engine? The requirement we place on the process engine is then: the status of a running process instance must be easy to discover. See the improvement in figure 5.5 on the next page.

Whether the inquiry is realized in a separate process or as a simple software service depends on the architecture.

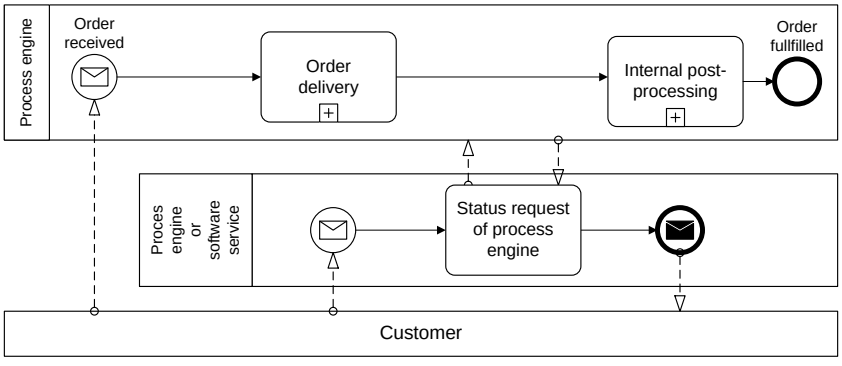


FIGURE 5.5 Better example: the status inquiry separated from the process.

5.3 Process automation with BPMN 2.0

One major innovation of BPMN version 2.0 is that it introduced a defined execution semantics as well as an XML serialization format. What does that mean? This is easily explained. Models can be stored as XML files, and the specification says exactly how to do it. There are two defined XML schemas for this:

- **Diagram interchange** contains all relevant information, such as layout information, for transferring the model to another tool.
- **Execution semantics** describe how technical details of the process are stored.

We take a closer look at execution semantics in this section. In section 5.5 on page 177 we explore model exchange. Aside from using XML schema for the storage syntax, the BPMN 2.0 specification also describes the execution semantics and the meta-model. This was missing in previous BPMN versions.

To be precise, any process engine compatible with BPMN 2.0 can execute BPMN models. If you are lucky, you can execute BPMN models without adjusting any proprietary extensions, but we discuss typical problems in section 5.6 on page 177. This issue formerly was the domain of the Business Process Execution Language (BPEL), which was regarded as *the* standard of automation, but it has now been replaced by BPMN 2.0. What distinguishes these languages, what the differences are, which standard is suitable for which application, and how much sense there is in a round trip between BPMN and BPEL are all interesting questions. Our opinion is in section 5.7 on page 178. The following sections describe a simple example of an executable process with BPMN 2.0.

Unfortunately, it is impossible here to provide a comprehensive introduction to process execution with BPMN 2.0. This section merely gives a taste. See more examples in the official sample document for BPMN 2.0 that we mentioned earlier in this chapter.

5.3.1 The executable process model

Starting with the level 2 model, we use it as input for level 3 (see figure 4.13 on page 121), but we only consider the process engine's pool. Figure 5.6 represents this process. Before, we did not show that the job advertisement is executed on different platforms as subprocesses. For simplicity, we embedded it. In a real project, a subprocess would be a better solution because it corresponds to the level 2 model.

For illustration, we also show "MajorJobs.com" as a separate pool, mainly to show the message flow. We will specify the exact content of the message later.

So far, the process doesn't look all that technical, does it? Many of the details necessary for automation are hidden in the underlying model, which is available as an XML file. For now, we want to examine several aspects while introducing the XML format step by step. Because of limited space, we do not cover the whole process, but you can find the full example at http://www.bpm-guide.de/bpm-java/.

From a merely visual point of view, the executable process in this example corresponds to the operational process from level 2. Isn't that brilliant? For those concerned with details, note that we changed two more things:

- We added the "job advertisement" data object because we have to store certain data in running process instances.
- We changed "send confirmation of success" from a send task to a script task. Why? While the confirmation could be sent by different means, it probably will be by email, and the email could come through an automated service or some built-in capability of the process engine. For our example, we assumed the latter, and this leads to the use of a script task in the process. This deviation should be synchronized with the level 2 model for consistency between the models. This type of iterative change happens frequently as the executable model develops.

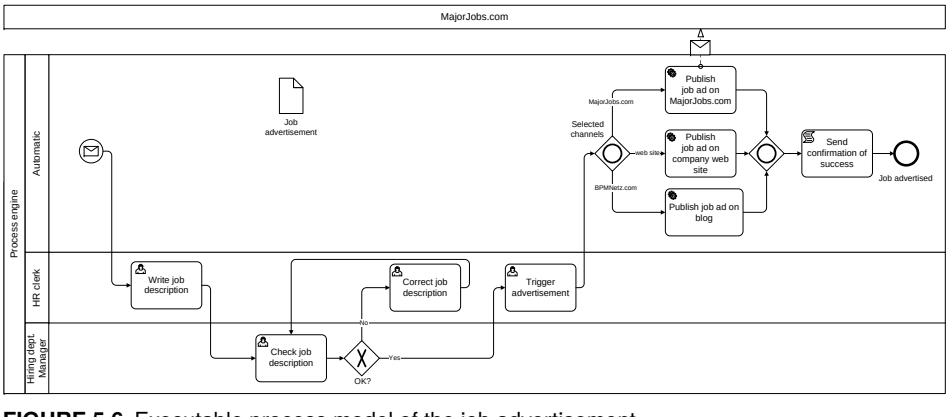


FIGURE 5.6 Executable process model of the job advertisement.

5.3.2 Data modeling and expressions

In the process diagram, the job advertisement is represented as a data object. There is also the real modeling of the data object in the process, but BPMN refrains from implementing detailed technical data modeling. Instead, it provides expansion points to accommodate diverse technologies. The default setting for this is XML schema, although you could use Java or .NET data types just as well.

What does the data definition look like? For this example, we apply the XML code directly. We use the XML schema as the "type language," then we import a schema file to define a data type ("advertisementDef"). This refers to an XML element of the schema. We can then use this type in the data object as a kind of a process variable:

```
<definitions ...
 typeLanguage="http://www.w3.org/2001/XMLSchema"
 expressionLanguage="http://www.w3.org/1999/XPath"
 xmlns:sample="http://sample.bpmn.camunda.com/">
 ...
 <import namespace="http://sample.bpmn.camunda.com/"
   location="SampleService.xsd"
   importType="http://www.w3.org/2001/XMLSchema" />
 <itemDefinition id="advertisementDef" itemKind="Information"
     structureRef="sample:advertisement" />
 ...
 <process id="JobAdvertisementProcess">
 <dataObject id="advertisementVariable" name="Job advertisement"
    itemSubjectRef="advertisementDef" />
```

Note that not all process variables must be graphically represented in the diagram. They can be invisible.

The specification also recognizes a formal language to represent conditions called expression language. Expressions, well known in the IT world, can extract information from available data. In the simplest cases, they check for true or false. With a data-based exclusive gateway (XOR split), for instance, a token will leave the gateway through an exit at the time of execution. The exact exit depends on the data. Suppose the job advertisement is corrected only if it was marked accordingly:

```
<sequenceFlow id="flow4" sourceRef="CheckJobDescription"
       targetRef="CorrectJobDescription">
 <conditionExpression xsi:type="tFormalExpression">
 getDataObject("advertisementVariable")/needCorrection=true
 </conditionExpression>
</sequenceFlow>
```

By default, BPMN uses XPath as expression language. You can see this in the XML code. XPath is a query language that works directly with XML data. But note that the language can be changed and you could, for example, use Java Expression Language in connection with Java data types. In that case, the process could look like:

```
<definitions ...
 typeLanguage="http://java.sun.com/"
```

```
expressionLanguage="http://java.sun.com/j2ee/1.4/ExpressionLanguage">
...
<import namespace="http://sample.bpmn.camunda.com/"
  location="types.jar"
  importType="http://java.sun.com/" />
<itemDefinition id="advertisementDef" itemKind="Information"
    structureRef="com.camunda.bpmn.sample.Advertisement" />
...
<process id="Stellenausschreibungsprozess">
<dataObject id="advertisementVariable" name="Job advertisement"
   itemSubjectRef="advertisementDef" />
                ...
<sequenceFlow id="flow4" sourceRef="CheckJobDescription"
       targetRef="CorrectJobDescription">
 <conditionExpression xsi:type="tFormalExpression">
 #{advertisementVariable.needCorrection}
 </conditionExpression>
</sequenceFlow>
...
```

This ability to configure data types means great flexibility when you implement the process engine. Manufacturers can implement tools similar to the current BPEL engine, or they can develop versions that are lightweight and similar to programming languages, as is the case with current Java engines in the open-source market. On the other hand, this means that not every process is executable by every engine. The target engine has to support the chosen technology.

5.3.3 Service calls – synchronous or asynchronous?

Asynchronism is understood to be that the answer to an inquiry can be delayed while the sender does other things. Synchronism means that the sender has to wait for an answer, but the answer arrives immediately or without significant delay. This applies to software function calls as well as to human communication.

In BPMN models, we differentiate between synchronous and asynchronous communication. Consider the difference in the software function call (also known as a service call) in figure 5.7. If the answering message flow arrives at the sending activity, this is synchronous communication. If the answer arrives at a later activity, the communication is asynchronous. You can also use message events to represent

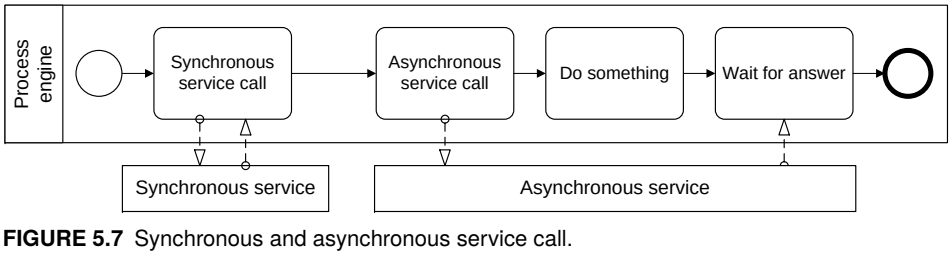


FIGURE 5.7 Synchronous and asynchronous service call.

asynchronism, but you can't do so for synchronous communication because events always differentiate between catching and throwing. They can't cover both cases.

When implementing a process in software, synchronous communication is simple. The answer is given directly in a technical manner —comparable to programming in which a simple function call returns a value. It's more complicated with asynchronous communication. Because the answer is delayed, it must be assigned to the correct waiting process instance. Now we have to correlate the answer, and the correlation condition has to be defined. In section 5.4.3 on page 165, we discuss correlations in detail.

In our example, we've implemented the call of the online job exchange as a Web Service, and we've modeled it as a service task. The Web Service returns no relevant result, but because it is synchronous, the data enters the job exchange immediately. If an error occurs, the error could be handled in the job advertisement process, and so the process waits for the service task to complete.

We also could incorporate the job exchanges asynchronously. In that case, we forward the message to the system, but the process can continue regardless of whether the message has arrived or if the processing initiated successfully. The IT infrastructure usually guarantees only that the message was successfully sent and that it cannot be lost, so the job could be advertised later. BPMN provides the send task for such cases.

Often, however, there's one problem: communication that looks asynchronous at the business level may be implemented with synchronous services at the technical level. So speaking about synchronous and asynchronous communication can lead to misunderstandings. Figure 5.8 shows a real-world illustration of this. Assume you want pizza. You grab the phone and call your trusted pizzeria. Technically speaking, the call is synchronous: a staff member answers the phone and provides feedback that your order was received. The order processing, however —the actual preparation and delivery —is carried out asynchronously. You receive the result half an hour later.

Suppose you send an email to the pizzeria. Technically, this is asynchronous because you don't know if or when the provider receives the mail, let alone when it reads your order. Things are less certain.

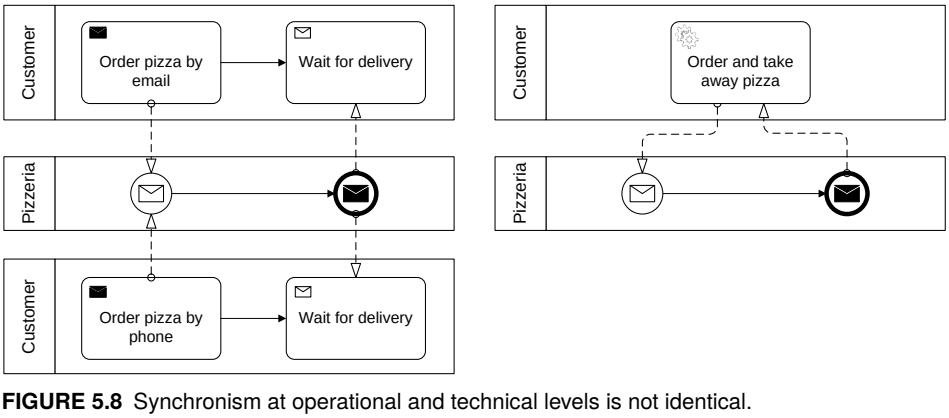


FIGURE 5.8 Synchronism at operational and technical levels is not identical.

A final alternative would be to walk half a block to the pizzeria, place your order in person, and then wait for the pizza. This corresponds to a synchronous call, because you leave the shop only after receiving the result. While waiting, you can't really do much else, so it equates to a waiting process.

Why are we going on about synchronism and asynchronism? Well, the interesting thing about figure 5.8 is that the synchronous service tasks and the asynchronous send tasks are applied in the technical sense. The executable model may differ in this from the operational model, which presumably provided for a send task in both cases because functionally, it is an asynchronous service. We can't yet anticipate how a process engine will implement this difference in practice. In other words, will it hide a synchronous Web Service call behind a send task if the result is ignored?

Hint: Business-IT-Alignment

Regrettably, the operational and technical views on synchronism and asynchronism differ. Situations that are functionally asynchronous may need to be represented in the executable process model as synchronous services and vice versa. Remember that.

5.3.4 Call IT systems

How can you call an IT system from the process automatically? BPMN's default is Web Services. As we said in section 5.3.3 on page 153, you need to decide from the context if synchronous or asynchronous service calls are involved. In practice, these are mostly technically synchronous calls (like the phone conversation), even though the result (like the pizza) is delivered later.

In recruiting process example, we used a service task to call the job exchange. To call a Web Service, you would define an input message that includes parameters and, if required, an output message containing the result. BPMN provides a message construct appropriate for this purpose.

BPMN is not tightly bound to Web Services, so you can use other technologies such as Java or REST-based services. You achieve this by means of an indirection. The interface is defined with parameters and return values regardless of the technology, applying the previously set technology for the data types. The interface is bound to a specific implementation only in our own mapping. The following shows the source code for defining the messages, the interface, and the application in the service task with Web Services.

```
...
<!-- import wsdl -->
<import
 namespace="http://sample.bpmn.camunda.com/"
 location="SampleService.wsdl"
 importType="http://schemas.xmlsoap.org/wsdl/soap/" />
```

```
<!-- Data structure of the message -->
<itemDefinition id="CreateAdvertisementDef" itemKind="Information"
    structureRef="sample:CreateAdvertisement" />
<!-- Nachricht -->
<message name="CreateAdvertisement" id="CreateAdvertisementMessage"
  structureRef="CreateAdvertisementDef" />

<!-- Interface -->
<interface id="Job ExchangeInterface" name="Job Exchange Interface">
<operation name="CreateAdvertisement">
 <inMessageRef>CreateAdvertisementMessage</inMessageRef>
</operation>
</interface>
...
<process id="JobAdvertisement" name="Job advertisement">
<serviceTask id="CreateAdvertisementOnJobExchange"
    name="Publish job ad on MajorJobs.com"
    implementation="WebService"
    operationRef="CreateAdvertisement">

 <ioSpecification>
 <dataInput id="CreateAdvertisementInput" isCollection="false"
    itemSubjectRef="CreateAdvertisementMessage" />
 <inputSet>
  <dataInputRefs>CreateAdvertisementInput</dataInputRefs>
 </inputSet>
 <outputSet />
 </ioSpecification>

 <dataInputAssociation>
 <assignment>
  <from xsi:type="tFormalExpression">
  getDataObject("advertisementVariable")
  </from>
  <to xsi:type="tFormalExpression">
  getDataInput("CreateAdvertisementInput")/advertisement/
  </to>
 </assignment>
 <sourceRef>advertisementVariable</sourceRef>
 <targetRef>SCreateAdvertisementInput</targetRef>
 </dataInputAssociation>

</serviceTask>
...
```

This binds the service task to the defined interface. It is realized by means of end points, though BPMN leaves these open. The specification defines only the extension points to which you can link the respective implementation. They can be Web Service end points. A connection to Java would be possible as well, as the following example illustrates.

```
...
<import
```

```
namespace="http://sample.bpmn.camunda.com/"
location="services.jar"
importType="http://java.sun.com/" />

<!-- Data structure -->
<itemDefinition id="CreateAdvertisementDef" itemKind="Information"
    structureRef="com.camunda.bpmn.sample.Job" />
<message name="CreateAdvertisement" id="CreateAdvertisementMessage"
  structureRef="CreateAdvertisementDef" />

<!-- Interface -->
<interface id="CreateAdvertisementInterface" name="Job Exchange
    Interface">
<operation name="createAdvertisement">
 <inMessageRef>CreateAdvertisementMessage</inMessageRef>
</operation>
</interface>

...
<process id="JobAdvertisement" name="Job advertisement">
<serviceTask id="CreateAdvertisementOnJobExchange"
    name="Publish job ad on MajorJobs.com"
    implementation="WebService"
    operationRef="createAdvertisement">

 <ioSpecification>
  <dataInput id="CreateAdvertisementInput" isCollection="false"
    itemSubjectRef="CreateAdvertisementMessage" />
 <inputSet>
  <dataInputRefs>CreateAdvertisementInput</dataInputRefs>
 </inputSet>
 <outputSet />
 </ioSpecification>
 <dataInputAssociation>
 <assignment>
  <from xsi:type="tFormalExpression">
   #{advertisementVariable}</from>
  <to xsi:type="tFormalExpression">
   #{CreateAdvertisementInput.advertisement}</to>
 </assignment>
 <sourceRef>advertisementVariable</sourceRef>
 <targetRef>CreateAdvertisementInput</targetRef>
 </dataInputAssociation>
</serviceTask>
...
```

In section 6.4.3 on page 207, we discuss the open source process engine Activiti, which we have used in several projects and "productized" as camunda fox. This takes another direction with service calls: it provides extensions to link a service task with Java code or corresponding expressions. This is illustrated in the next code sample. While this is a deliberate deviation from the standard, it's not a violation because extensions are permitted. It is a lot easier for Java developers to handle the process definition, and we

have had good experience with this approach in real-life projects. Section 6.4.3 on page 207 discusses cooperation with developers.

```
...
<process id="JobAdvertisement" name="Job advertisement">
 <serviceTask id="CreateAdvertisementOnJobExchange"
    name="Publish job ad on MajorJobs.com"
    activiti:class="com.camunda.bpmn.CreateAdvertisementDelegate" />
...
```

5.3.5 Start events and receive tasks

Not only can a process call a system, but you may also want to communicate with a process from the outside. In our example, this is true when a new process instance starts. BPMN's default assumption is for Web Service technology. You can therefore provide a Web Service to start a new process instance. This is like calling a service, except that the only data output from the event is the message; it requires no data input. The start event looks like this:

```
<startEvent id="Start">
 <dataOutput id="StartProcessOutput"
    itemSubjectRef="StartProcessItem" />
 <dataOutputAssociation>
 <assignment>
  <from xsi:type="tFormalExpression">
  getDataOutput("StartProcessOutput")/advertisement
  </from>
  <to xsi:type="tFormalExpression">
  getDataObject("advertisementVariable")
  </to>
 </assignment>
 <sourceRef>StartProcessOutput</sourceRef>
 <targetRef>advertisementVariable</targetRef>
 </dataOutputAssociation>
 <messageEventDefinition messageRef="StartProcessMessage">
 <operationRef>startAdvertisementProcess</operationRef>
 </messageEventDefinition>
</startEvent>
```

As with the service tasks, you have to define data structures and messages.

5.3.6 User tasks

The last problem in the example is the human interaction. The user task of the process leads to an entry in a task-management system. To the user, it appears as an item in a task list. The process only continues once the user completes the task.

In the area of BPMN engines, you find three options for handling tasks. The process engine calls some Web Service, which, for example, uses either the process engine's

proprietary task management or some internal implementation. The engine can also apply another technology, perhaps incorporating Java-based task management. That's a straightforward approach that often makes sense if the process engine provides its own task management. These proprietary choices may limit the interchangeability of different engines, however.

The third option is standard: WS HumanTask. This is a comprehensive specification that allows you to define user tasks in great detail. It is powerful enough to let you control responsibilities, delegation, escalation, and even the meta information to be displayed. You can define a subject, for instance, for a task. WS-HT is new, and it is complex, and support by manufacturers is growing slowly. But if you have WS-HT-compatible task management available to you, it may be a good choice.

Here we want to show an example of the simplest case, when the process engine provides its own task-management and does all the work under the hood. A human task could be as follows:

```
...
<resource id="ClerkResource" name="HR Clerk"/>
...
<process id="JobAdvertisement" name="Job advertisement">
<userTask id="writeJobDescription" name="WriteJobDescription">
 <potentialOwner resourceRef="ClerkResource"/>
</userTask>
...
```

5.4 One more word on execution semantics

After skimming the first example, we want to take a closer look at some of its aspects. This book cannot provide an understanding of the execution semantics complete in all details. What we have done is to pick the issues we think are the most interesting, and to discuss the core elements of BPMN within the scope of automation.

5.4.1 Start events and process instantiation

We already know that start events start new process instances, but how do they do it? If you are an IT person, you may think of the following possibilities:

- The instantiation is modeled in the process or as a separate process; either way, the process engine starts it.
- Another IT component starts the instantiation externally.

Imagine that a new process instance is initiated as soon as an email is received. Somehow this email must be technically retrieved, the data read from it, and the respective process

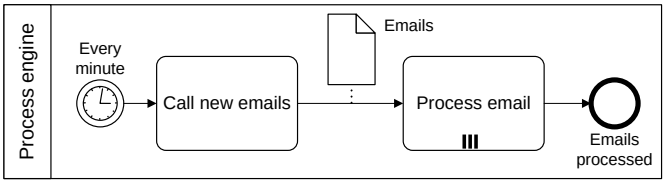


FIGURE 5.9 Modeling process instantiation in the process by means of regular email retrieval.

instance started. You can model these things as a process as shown in figure 5.9. The process engine would retrieve and process new emails every minute.

Alternatively, if IT components capable of retrieving and processing emails are already available in the organization, those components can receive new email and start a process instance that will work with the email directly. We show this in figure 5.10. If no email arrives, the process never instantiates. In practice, an upstream Enterprise Service Bus (ESB) or a similar component often assumes this task.

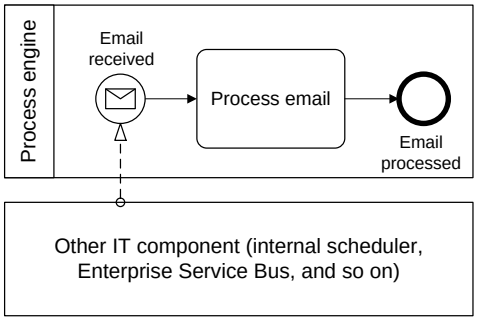


FIGURE 5.10 Process instantiation takes place outside of the process engine.

Both instantiation paradigms occur in our projects. Depending on the process engine, project size, project environment, and technology, one variant of both or maybe even a mixture of both variants is chosen. In larger projects, in which the process engine is a component of the IT architecture, maybe within the scope of Serviceorientied Architecture (SOA), the external way often is taken; in smaller projects, which are very process-engine-driven, the engine handles all the tasks.

Both alternatives have pros and cons. Modeling in the process is often easier, and it may mean that more aspects of the process can be handled from a central place. On the other hand, external instantiation is more flexible and places less demand on the process engine. External instantiation also frees the process from technical details such as whether the trigger is an email, JMS, or X.400 message. Make such decisions in the context of each situation. Do it deliberately, and with the required target architecture in mind.

One final note: the timer event is a special case because it is an "active" event. (The pure message event, as a counter-example, cannot itself become active.) Nevertheless, the event can be modeled in the process. The arrival of a message may signify that a Web Service call is accepted. This is supported explicitly by BPMN as shown in section 5.3.5 on page 158.

5.4.1.1 Multiple start events

Section 2.6 on page 34 suggested situations in which multiple start events must be provided to start a process. This problem is worse in process automation than in operational modeling, because the semantics need to be defined precisely for technological implementation.

Figure 5.11 contains an example. A broker accepts sell orders and buy orders. Assume that buyer and seller do not know each other and also that the product (shares, for example) need not be auctioned before they can be sold. If a buy order arrives first, the upper token runs to the AND merge. A process instance can complete successfully only if a sell order is also available. The trick of the parallel event-based gateway is that it can correlate the following events: if an offer is received, there's a check to see if the offer matches the existing process instance containing our order. A new token then goes to the existing process instance, and our process in the AND merge can continue. If the offer was not a match, a new process instance starts, and it then waits for the order.

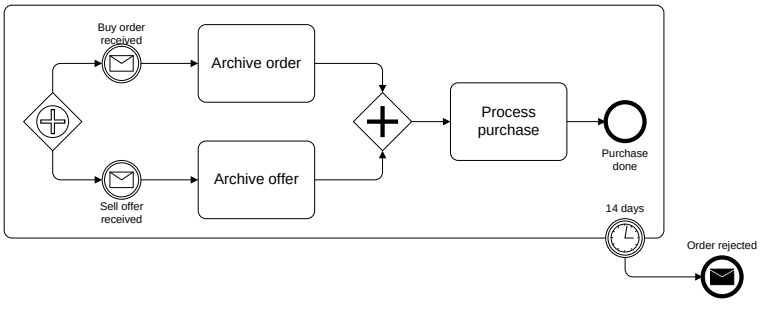


FIGURE 5.11 Example of a parallel, event-based gateway for process instantiation.

From the perspective of the process engine, a major problem is that the process instance may "starve" if a second message never arrives. You should take this kind of circumstance into account. Our approach was to attach a timer event.

Figure 5.12 shows different possibilities for multiple start events. Our example belongs to group (d). Group (c) should, by the way, represent the same situation regarding content, and we allowed for this in the operational model. This pattern is invalid for automation, however, since the AND merge cannot correlate. Besides, the result was two process instances, neither of which could end even if offer and order matched. You have to model with considerable precision in process automation so as not to confuse the process engine.

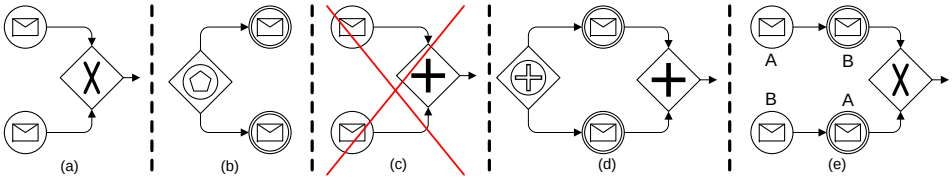


FIGURE 5.12 Start of a process instance by a multiple start event.

Bear in mind that groups (a) and (b) in figure 5.12 on the previous page show that either one or the other event starts the process. It does not matter if the exclusive event-based gateway is applied or not. So why does this construct exist? Surveys show that some people understand process models better if they have exactly one start node. The gateway serves that purpose.

Regarding group (e), this is another possibility to start a process with two events. You have to model the various combinations in the order of events separately. That way, either offer or order can start the process and the other message can be correlated.

You can find a theoretical discussion of the problems of process instantiation (with BPMN 1.2) in [DM08], which suggests a corresponding external process factory. The process is therefore not instantiated by the process engine itself.

5.4.2 Events and their implementation in IT

Events deserve a closer look from the automation perspective. First, we examine the explosion semantics mentioned in section 2.6 on page 34. Remember that arriving events "explode" if no process instance waits for them. Consider the upper portion of figure 5.13. An order is forwarded for delivery scheduling, but the invoice is sent first. Suppose that although the order itself is ready for immediate delivery, invoicing is a manual task that takes a while. In a case like this, we don't want the delivery event to vanish just because it was ready before anyone wrote out the invoice.

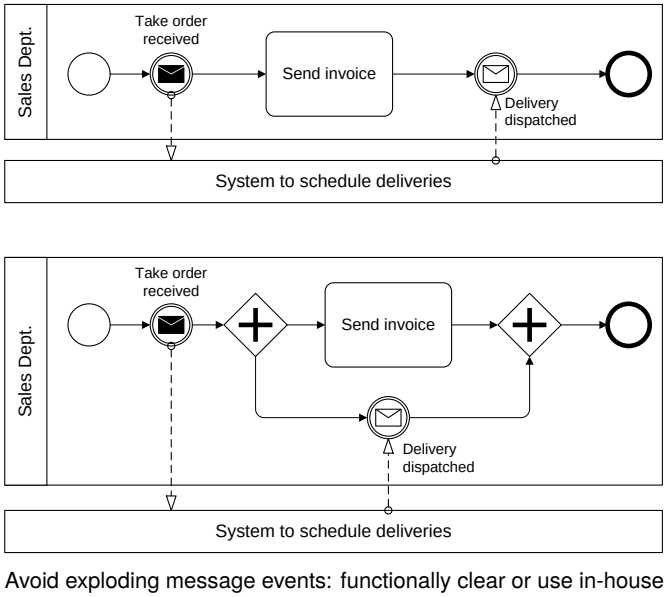


FIGURE 5.13 Avoid exploding message events: functionally clear or use in-house means?

But the BPMN 2.0 execution semantics are strict: events explode if nothing is waiting for them. Process engines have different ways to cope with that, though exactly how it is done depends on the technical implementation. How can you solve this now?

If you intend to implement a process with a process engine, you may have no choice other than to rebuild the model so as to have a parallel process path to wait for the event. This alternative is not necessarily difficult; we simply parallelize the process sequence as shown in the bottom part of figure 5.13 on the facing page. From the IT perspective, this rearrangement is no problem, but it would mean that the operational diagram has to become more complicated. It is another reason that executable models may differ from the business view of what is happening. And even this model need not always be coherent because, depending on transaction management, the answer may arrive before the process moves to the receive event. As always, the devil is in the details.

By the way, if process engines implement a queuing mechanism for messages, which is technically possible, this raises the question of what happens if no process ever retrieves the event from the queue? The sender of the message can no longer be informed, so the process engine will need a sophisticated error-handling mechanism. The mechanism will have to recognize that the process instance, the one to which the orphaned event could be correlated, has ended. If you think that's not easy, you're right. Sometimes this functionality is moved to an Enterprise Service Bus (ESB).

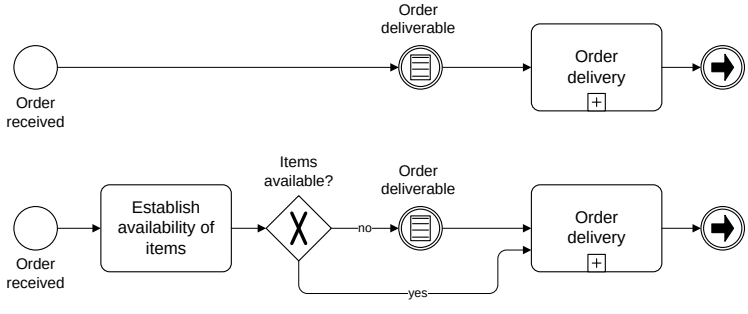


FIGURE 5.14 Conditional event: functionally clear (top) and technically correct (bottom).

What about conditional events? Using figure 5.14 as an example, an order can be delivered only if all the ordered items are available. If one item is missing, the process has to wait until the item is provided —this is what the "item available" condition implies. The operational model is clear, as shown in the top part of the figure. In automation, however, this construct causes problems:

- **Formulation of the condition**: The process engine has to know what the condition is in a formally unambiguous way. BPMN provides XPath as the default setting. But wait, didn't section 2.6.5 on page 42 say that the condition was met regardless of the process? How can this condition be analyzed by an expression *in* the process now? We regard this as a flaw in the specification. There is still a loophole: XPath (or other expression languages) can be extended so that access remains possible from the expression to data and services beyond the process. Thus you can imagine an expression that asks a rules engine if a certain condition is met.
- **Explosion**: The explosion problem might also happen with conditional events. Look at the operational model of the process in figure 5.14. If the items become available when a process instance waits for them, the order is delivered. But what happens if the token enters the conditional event after that event was already raised and exploded? Strictly

speaking, the token had to wait —possibly forever —because "items available" may never trigger again. From a practical point of view, the conditional event might not be implemented that way, but this is how the specification expects it to be. So it would be better to wait for the conditional event only if items are *not* available. The lower part of the figure shows checking for availability first. While this may be overly detailed from an operational point of view, it might be needed to be able to automate the process.

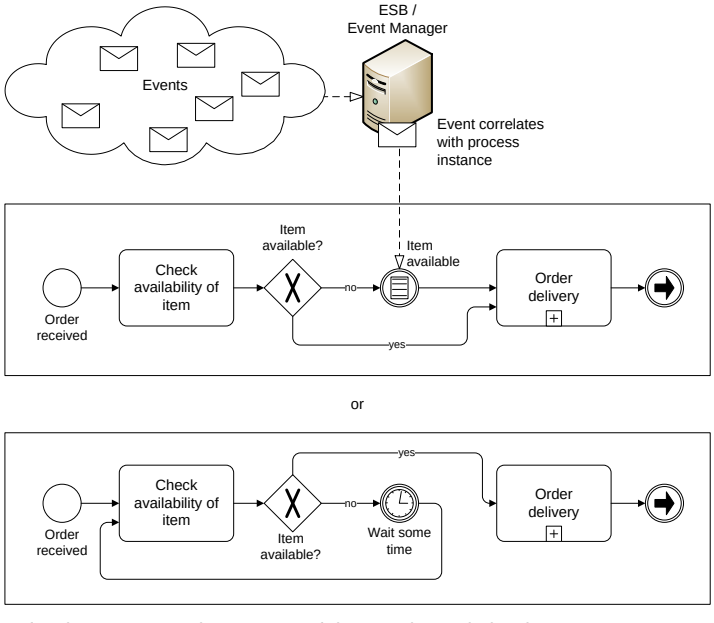


FIGURE 5.15 Implementation alternatives of the conditional check.

- **Who checks the condition and when**: The technical question arises —similar to that of the process start —of who performs the checks, and when do they do so? How does the process engine notice the result?

  In our discussion of explosions, we said that the conditional event does not check the condition when the token arrives. This is in accordance with the specification. Some process engines may implement it this way, but none that we know of. We see two options for waiting for the event: either check after a timed interval, or obtain notification through an external IT component.

  Using a timed interval, the process engine independently checks at the expiration of a specified period (see the lower portion of figure 5.15). The process engine could check the expression every 10 seconds, for instance. On the down side, this procedure is inefficient. It produces many unnecessary checks.

  With notification, it is an IT component separate from the process that checks and then actively informs the process. This is the first of two possibilities shown in figure 5.15. The checking and notifying could be undertaken by an ESB, for example, in which events like checking for items remains centrally accessible and can be analyzed. The process engine can obtain its information at the same time that the condition is met. This raises the question of how to formulate the expression in this case. The so-called

event-driven architecture (EDA) even has internal query languages for events. These problems are not part of the BPMN specification, so approaches to this issue are left to the manufacturers, which means that a process may not execute on different process engines.

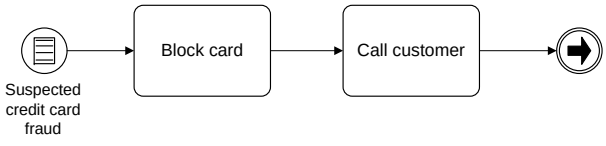


FIGURE 5.16 Conditional event to start the process.

Another complication is the question of how to start a process by means of a condition. In the example shown in figure 5.16, the credit card is blocked as soon as fraud is suspected. But what creates the suspicion? Typically, it is if a particularly large amount of money is withdrawn, or the card is used unusually often, or even if it is used in certain countries. According to the specification, to instantiate a process, the expression must become true within the event. Before another process can start, the condition must become false. The technical solution by the engine remains open.

5.4.3 Correlation

From reading the preceding section, you may have seen that correlating catching events with existing process instances is important. The question now becomes, to what instance or instances must you forward a caught event? There are two general approaches:

- **Technical keys**: An artificial key value is generated for conversations, and the key must be contained in call and response messages. It's then easy for the engine to correlate the messages with the correct process instances based on the key. This approach has the advantage that a key typically is valid for only one message exchange. It is thus unambiguous. That's not always possible in practice, however, because all the participants must support working with these keys.
- **Functional keys**: The alternative to artificial technical keys is correlation by means of context information, for example, an order number. You can also use properties of process variables or messages. You can specify expressions that define how to find these properties in the process variables or incoming messages, with the advantage that no new requirements are placed on messages or external systems. The disadvantage is that the correlation may be ambiguous.

Define the correlation correctly, and the process engine can allocate incoming messages among process instances by means of the correlation conditions. The specification talks about "instance routing" without additional infrastructure.

Unfortunately, correlation isn't easy. Problems remaining to be solved include:

- **What happens if the event cannot be correlated?** As in the case of explosions, when events occur too early, the event usually enters an error queue, and it requires further processing by hand.

- **What happens if the process changed state?** A process instance can leave the expected state —perhaps because an administrator intervened or there was a timeout —and so the event cannot be submitted. Even worse, the process instance is already waiting for the next event and might be incorrectly triggered by it. To prevent these situations, you have to detail the correlation so that it contains the expected process state. That isn't always easy.
- **Wrong correlation conditions**: The worst case involves wrong correlation conditions. These may trigger the wrong process instances. Such errors in modeling are hard to find, but they can be detected by good, up-front testing.

5.4.4 Gateways

We introduced the types of gateways in section 2.3 on page 18. BPMN 2.0 also defines an exact execution semantics, which we explain briefly here. "Execution semantics" determines the answers to questions such as: How are which tokens consumed? When are tokens generated in the gateway? Which tokens are generated? Which flow should they follow?

Figure 5.17 shows two simple cases. The **parallel gateway** (split and merge) waits for a token on all incoming flows, and it creates new tokens on all outgoing flows. One incoming token only is consumed in each flow. If several tokens are available in one flow (which usually doesn't make much sense in practice), the excess tokens remain in the flow, waiting for a subsequent activation of the gateway.

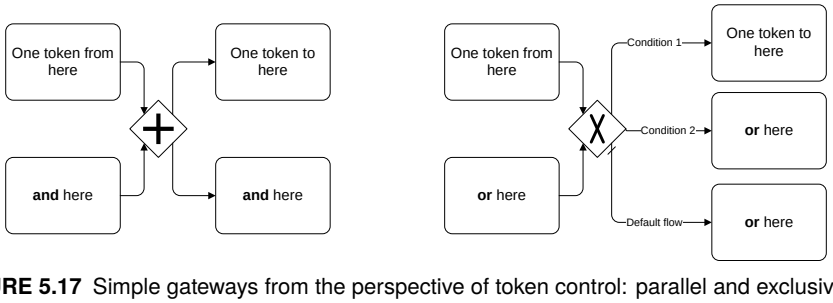


FIGURE 5.17 Simple gateways from the perspective of token control: parallel and exclusive (XOR).

In case of the **exclusive gateway** (XOR merge and XOR split), each incoming token is routed through a single outgoing flow. For this purpose, conditions available for the flow —these are provided as expressions —are analyzed in sequence. The token follows the flow with the first matching condition. The order, by the way, is specified by the XML. If no condition is true, only then is the default flow used. If no condition is true and no default flow is configured, this throws a runtime exception because an invalid situation exists. So far, this is simple.

The **inclusive gateway** (see figure 5.18 on the next page) is more complicated. Consider first the behavior with incoming tokens (OR merge). The gateway activates as soon as tokens are applied to all incoming flows or else no token can arrive on that flow. What? No token can arrive? That's a subtlety of the construct that is going to cause problems for

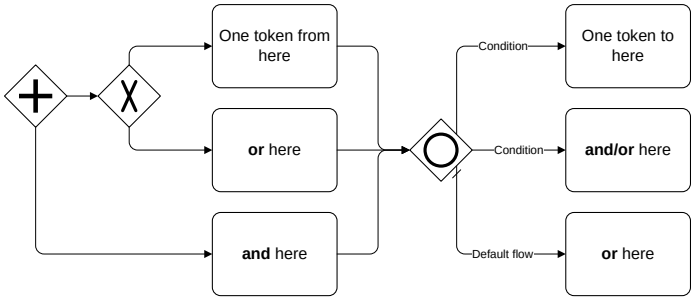


FIGURE 5.18 Inclusive gateway (OR).

process engine suppliers! Figure 5.19 shows an example of a more complex situation: only after task 2 completes can the OR split know if the lower flow can still arrive or not. We can imagine even more complicated constructs, such as loops, but fortunately this is a problem for the engine suppliers, and not for us users. Nevertheless, use this construct carefully, because it quickly becomes confusing.

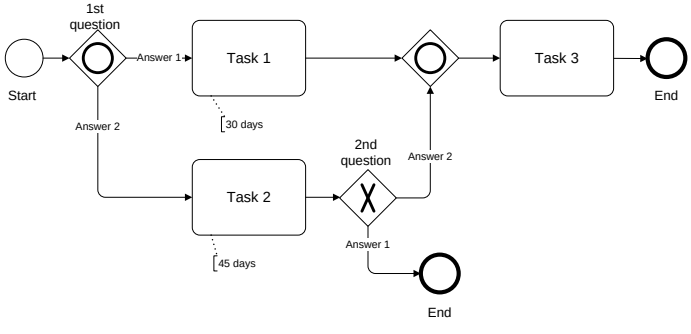


FIGURE 5.19 How long does the second OR gateway have to wait?

The OR split is easier. Each outgoing flow is checked for the set conditions and for expressions as well. If a condition is true, a token is created on this flow. Any number of tokens can be created in case of the OR split. If none of the conditions applies, a token follows the default flow, but if no condition is true and no default flow exists, this throws a runtime exception.

The **complex gateway** can serve multiple requirements, and you can configure it comprehensively. In figure 5.20 on the next page, a token arrives through flow a or b, and another one through c, which might activate the gateway depending on the conditions configured. If a token arrives through d later, it can be ignored (although that's not completely true because the token from d resets the gateway, which allows it to fire again later). The configuration is put into operation by an expression. While the process engine waits in the complex gateway it even counts incoming tokens. You can use this counter in expressions, which is a major difference from the OR merge. This makes it easy to express: *(1 x a or 1 x b) and 1 x c*.

The splitting behavior of the complex gateway corresponds to that of the OR split, and so we don't detail it further at this point.

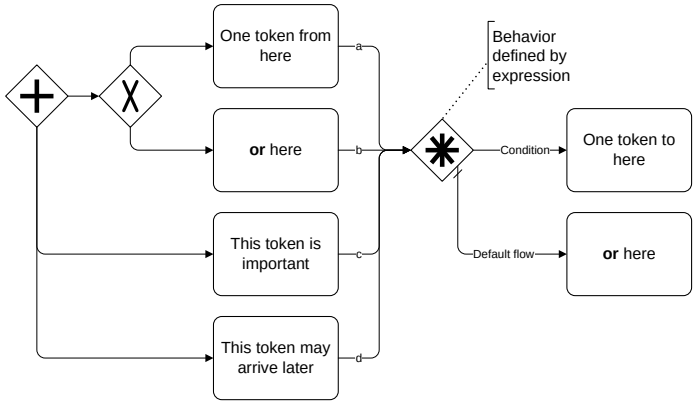


FIGURE 5.20 Complex gateway.

But for the end of this section, we want to present an easier gateway: the **event-based gateway**. A token simply stops in the gateway, waiting for one of the events to occur at the outgoing flows. It then leaves the gateway through that flow.

5.4.5 Termination of a process instance

A process instance can terminate under a variety of circumstances. It can terminate normally, or it can be canceled. It can also terminate or escalate with or without an error.

Normally, a process instance is regarded as complete if all tokens have reached an end state. An end state is either a dedicated event or a task without an outgoing sequence flow. This is illustrated in figure 5.21.

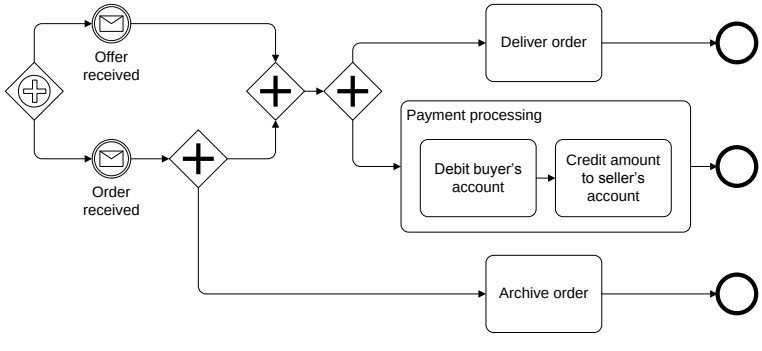


FIGURE 5.21 Normal termination of a process with parallel gateway to start the process.

The process contains several none end events that consume incoming tokens. If the last token arrives at an end state, the entire process instance terminates. This example has one special feature: a parallel gateway starts the process instance, so the process can complete only if all the end events have occurred. In this case, the offer and the order must both have arrived, and even though the order may be archived before delivery takes place or

payment completes, the process won't complete until those other tokens arrive at their respective end states.

Another feature is shown in the "process trust payment" subprocess. A task without any outgoing sequence flow also represents an end state, and the process instance can terminate there if no more active tokens are available. A third important feature is that subprocesses always have their own endings. That is, termination of a process instance is always determined at the same hierarchical level, so a subprocess that has not ended prevents its parent process from terminating.

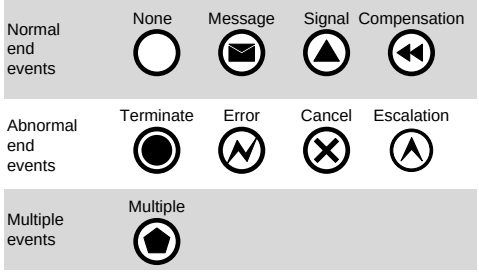


FIGURE 5.22 End events associated with normal and abnormal process terminations.

Normal end events end the token, and they may trigger additional actions (see figure 5.22). These are:

- Send message (message end event)
- Send signal (signal end event)
- Compensation required (compensation end event)

Normal end events terminate the incoming tokens only. A process instance ends only if no active token exists in any subprocess. This contrasts with so-called abnormal terminations:

- Termination (terminate end event): All the tokens in the current subprocess terminate, and the process instance terminates.
- Error (error end event): All the tokens terminate, and an error is forwarded to the parent process. If no parent process exists or if it does not react to the error, the behavior is not specified.
- Cancel (cancel end event): The behavior compares to the error event, but in addition, the current transaction is rolled back and available compensation tasks execute.
- Escalation (escalation end event): Other tokens in the current subprocess are not affected, but the current token terminates, throwing an escalation event to the parent process. But if the parent process absorbs the escalation with an interrupting intermediate event, the other tokens of the subprocess terminate too.

Only the multiple event remains to be covered. It has consequences that can encompass all the effects above. What's required with a multiple event is to define its behavior for the process engine, as in the XML below:

```
<bpmn:endEvent id="End" name="Multiple end event">
   <bpmn:messageEventDefinition id="Message" messageRef="messageDef" />
   <bpmn:signalEventDefinition id="Signal" signalRef="signalDef" />
```

5.4.6 Business vs. technical transactions

We've already discussed transactions, errors, and compensation from the business perspective. Now we want to examine these things from a technical point of view.

Computer scientists see transactions primarily in ACID terms. (ACID is an acronym for atomicity, consistency, isolation, and durability.) The relevant point is that they want to see a collection of actions carried out either simultaneously or not at all. While those actions take place, the state of the affected data remains isolated from other changes. Writing into a database, for instance, is an atomic action.

Booking a trip is an easily understood example. Booking the flight, the hotel, and charging your credit card take place at the same time, and either everything is booked or nothing is. Wouldn't that be great? Unfortunately, ACID transactions hardly ever work in the world of business processes, or they work only at a low level. In reality, we have to deal with waiting states, human interactions, and asynchronous service calls. A credit card firm can't wait hours or even minutes to complete a transaction because the required isolation may mean that the customer can't use his or her credit card for other purposes.

When we talk about transactions in BPMN, we mean business transactions. It is important, however, for you to separate these from technical transactions under the ACID paradigm. Figure 5.23 compares the concepts.

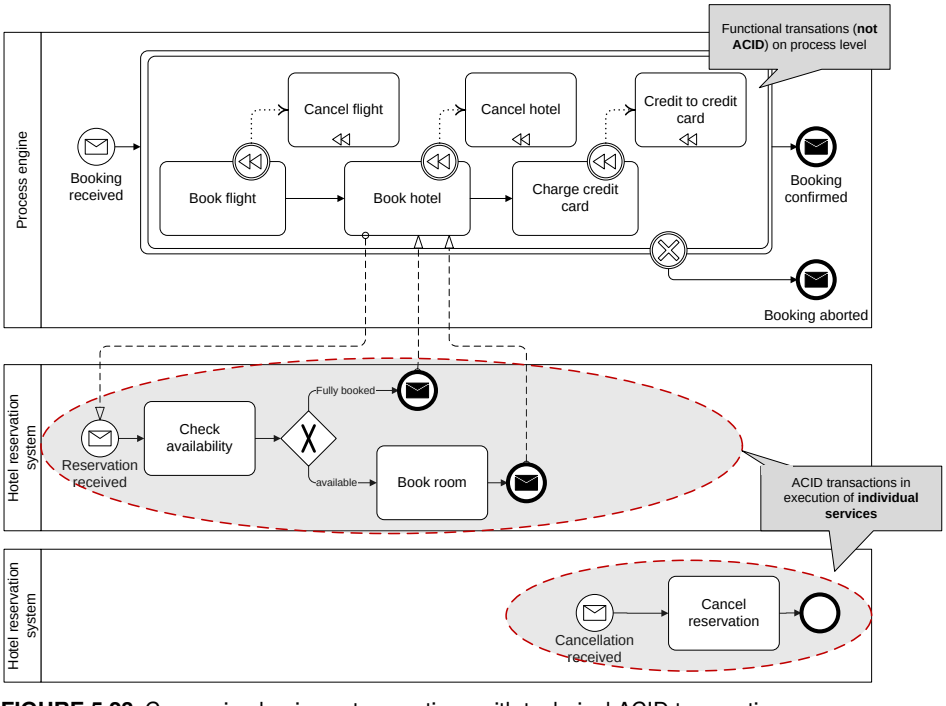


FIGURE 5.23 Comparing business transactions with technical ACID transactions.

The business transaction comprises the entire booking. Technically, ACID transactions normally take place only on the level of the called services —in the hotel reservation system, for instance. Once you make the reservation, it cannot be rolled back technically,

but from a business point of view it *can* be rolled back. This is why the compensation can call the cancellation service, which in its turn can be carried out by means of an ACID transaction.

When automating with a process engine, one problem usually arises swiftly: many systems either do not provide compensation services, or a compensation service can itself throw an error. In figure 5.24, what happens if the tour is already scheduled or the truck has already left? Some sort of error handling should be represented so that when it's time to automate the BPMN model, the details have been modeled accurately.

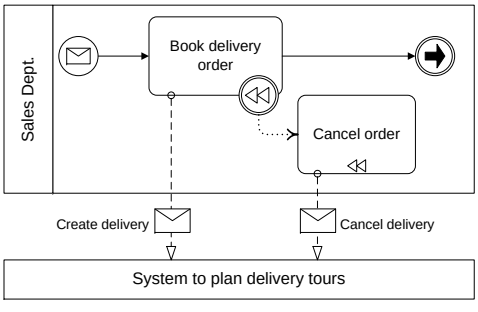


FIGURE 5.24 Compensation in a service call.

By the way, we don't have a good, standard approach to dealing with missing compensation services either. Usually we have to work around the problem, or the software manufacturer has to give in and integrate the additional functionality. We hope that manufacturers will begin to provide more and more compensation services based on the current discussion of BPM and SOA.

If you ask around among scientists for business transactions and compensation problems, you'll shortly hear a term that's also mentioned in the BPMN 2.0 specification: WS-Transaction (see http://en.wikipedia.org/wiki/WS-Transaction). This is a standard that defines two coordination types, and these correspond to our definitions for business and technical transactions:

- WS-AtomicTransaction transfers ACID transactions into the world of Web Services.
- In contrast, WS-BusinessActivity deals with long-term business transactions.

These standards are generally a good idea, but they haven't been well adopted in practice. We know of hardly any projects in which Web Services transactions have been applied successfully. What's often lacking is cooperation between different suppliers —the elementary vision of Web Services. If you are interested in more technical details on transaction management, you can visit our blog, where we discuss this in more detail: http://www.bpm-guide.de/2012/06/19/how-do-you-make-a-cappuccino-in-a-single-transaction/.

5.4.7 Subprocesses

BPMN knows two fundamentally different variants of subprocesses: embedded and reusable. Embedded subprocesses correspond to a scope, and within the scope,

properties or data objects are only visible locally. In other words, you can see them only within the scope itself or in embedded subprocesses. Subprocesses are also in scope for error treatment, compensation, or transactions.

An embedded subprocess typically has no large overhead. From the process engine's point of view, you can imagine that the content of the subprocess is simply inserted into the parent process.

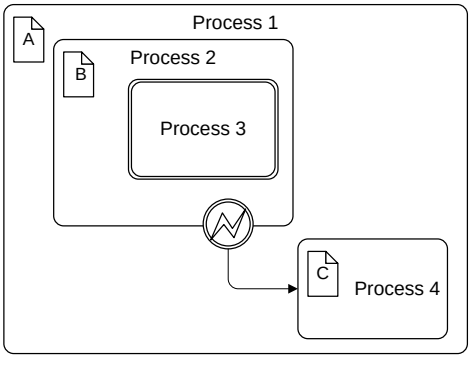


FIGURE 5.25 Subprocesses, scopes, and data visibility.

In figure 5.25, process 1 obviously sees data object A, but not data object B or C. In contrast, process 2 sees A and B, but not C. Process 4 sees A and C only. The error event at process 2 catches errors from processes 2 and 3.

This looks different with reusable subprocesses. These processes are entirely autonomous and have entirely separate data objects. If you want to use data from the parent process, these data must be transferred explicitly. This is done exactly in the same manner as a service call by means of an "InputOutputSpecification" as contained in the listing in section 5.3.4 on page 155.

Here's some advice that may not be obvious: even though BPMN is a standard, the behavior of process engines varies greatly. When a BPEL engine calls a subordinate process (in other words, a reusable subprocess), this is a normal Web Service call according to the service orchestration —strictly speaking, BPEL recognizes no subprocesses. This can reduce performance because such calls always add overhead, even if they do not leave the process engine space. We have seen customers subdivide their business processes reasonably into subprocesses, but who then shipwrecked in the technical implementation for performance reasons.

BPMN 2.0 engines can now do better; they make it possible to call a reusable subprocess without causing more trouble than would its embedded counterpart.

The parent process continues as soon as the subprocess terminates, as we explained in section 5.4.5 on page 168. The same applies to subprocesses. In the parent process, one token is generated for each outgoing sequence flow. In case of an error, the respective error event is called.

5.4.8 Loops and multiple instances

Loops and multiple instances are also interesting subjects. They sound simple from the business perspective, but you can set so many minor parameters that the results are powerful constructs in BPMN. What's more, no one can deny certain similarities to programming languages.

The loop activity is simple. The logic it contains executes repeatedly in succession. The "testBefore" attribute determines if the expression in "loopCondition" is checked before or after the loop cycle. If before, this corresponds to a "while" loop in a conventional programming language; if after, this corresponds to a "repeat until" loop. As soon as the expression —given in the expression language of the process engine —returns false, the loop terminates, and the sequence flow leaves the activity. You can specify a maximum number of loop cycles with "loopMaximum."

The multiple instance construct can execute sequentially, which makes it function much as the loop does, but you have to specify the number of instances in advance, either with an expression or by transferring a "collection" as a data object. One instance is created, and it executes, for each data object in the collection. An even more powerful capability is that it can also execute the logic it contains in parallel (meaning simultaneously).

It is a characteristic of the multiple instance activity that you can specify boundary events to be thrown when one instance ends. Usually, a token simply leaves the multiple instance after all the instances complete. You can instead apply events and specify which event to throw. You can even define complex event semantics to check any expression and to throw one or more events —though this is pretty rare in real-life projects.

What does it mean to throw an event in a multiple instance activity? The answer isn't necessarily intuitive. Consider an interrupting event such as an error. If an intermediate event catches the error in a multiple instance activity, all the instances of the activity cancel. In figure 5.26 on the next page, for instance, the top-left version shows that an error involving one invoice item results in all items being canceled. That may not be what you want!

What you may want is to cancel just the one instance. To make that happen, you may have to execute the error handling in the activity yourself. Suppose, however, that you don't want to process the functional error handling within the activity. Perhaps a participant in another swim lane is responsible for that. There's a trick: catch the error in the activity but pass a non-interrupting escalation event on. This way, the error can be handled at the top level without affecting the other instances.

What about the token flow? Fortunately, BPMN knows the magic OR merge, which waits for any kind of incoming token. In other words, it would have exactly the required effect.

Is it complicated? Yes . . . perhaps. But we *want* to handle complicated problems, and if we want to execute the model with a process engine, we must be precise with our models. All the possibilities we've outlined may make sense technically, but you still have to decide what you want.

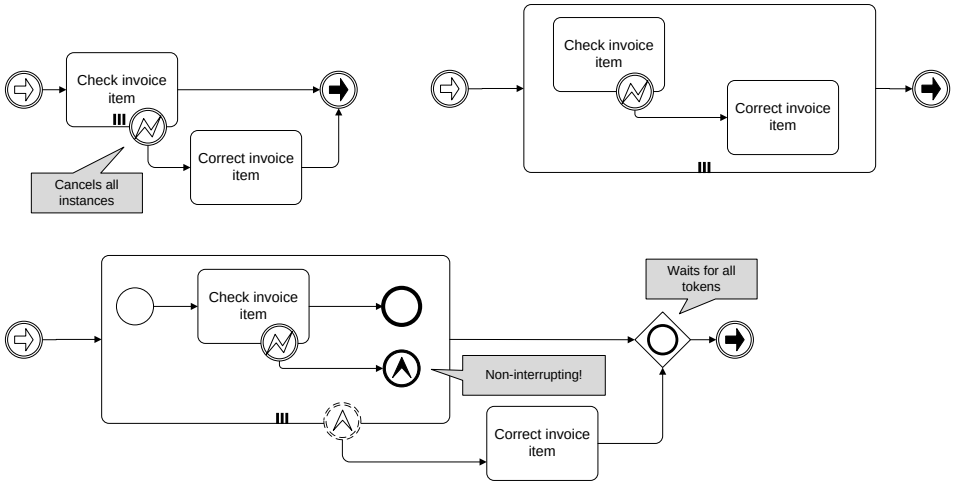


FIGURE 5.26 Multiple instances, termination, and error events.

5.4.9 Life cycle of an activity

As the final aspect of execution semantics, we want to discuss the activity itself: the tasks or subprocesses. Activities have a defined life cycle, as shown in figure 5.27 on the facing page. The life cycle is run through for each activity in a process. As soon as a token arrives in the activity, the activity is in a "ready" state. Many of the subsequent states respond to interrupting or non-interrupting events, but we want to ignore that for now.

When an activity switches to "active" state, it becomes much more interesting. This is precisely the case if you apply required input data. Remember the service task in section 5.3.4 on page 155: the XML representation contained a DataInput element, which expresses data from the process context and provides the input. If these data are not available, the activity waits for them. This gets checked, strictly speaking, even if one of several possible DataInputs is applied. Regrettably, the specification does not detail how this works in practice.

If the activity is being completed, the state switches to "completing," and the activity waits for possible dependencies, that is, as-yet-uncompleted, non-interrupting intermediate events. Then it switches to the "completed" state, which routes the token into the respective transitions, and it transfers the required data that were configured in the DataOutput elements.

As soon as the process instance finishes, the state changes to "closed." The process engine, by the way, does not need to represent these states, but it must implement the required state transitions.

5.4.10 Auditing and monitoring

Logging the process sequence is called auditing. Detailed data about the execution of a process instance are written to a log for analysis. Auditing and audit logs are an important

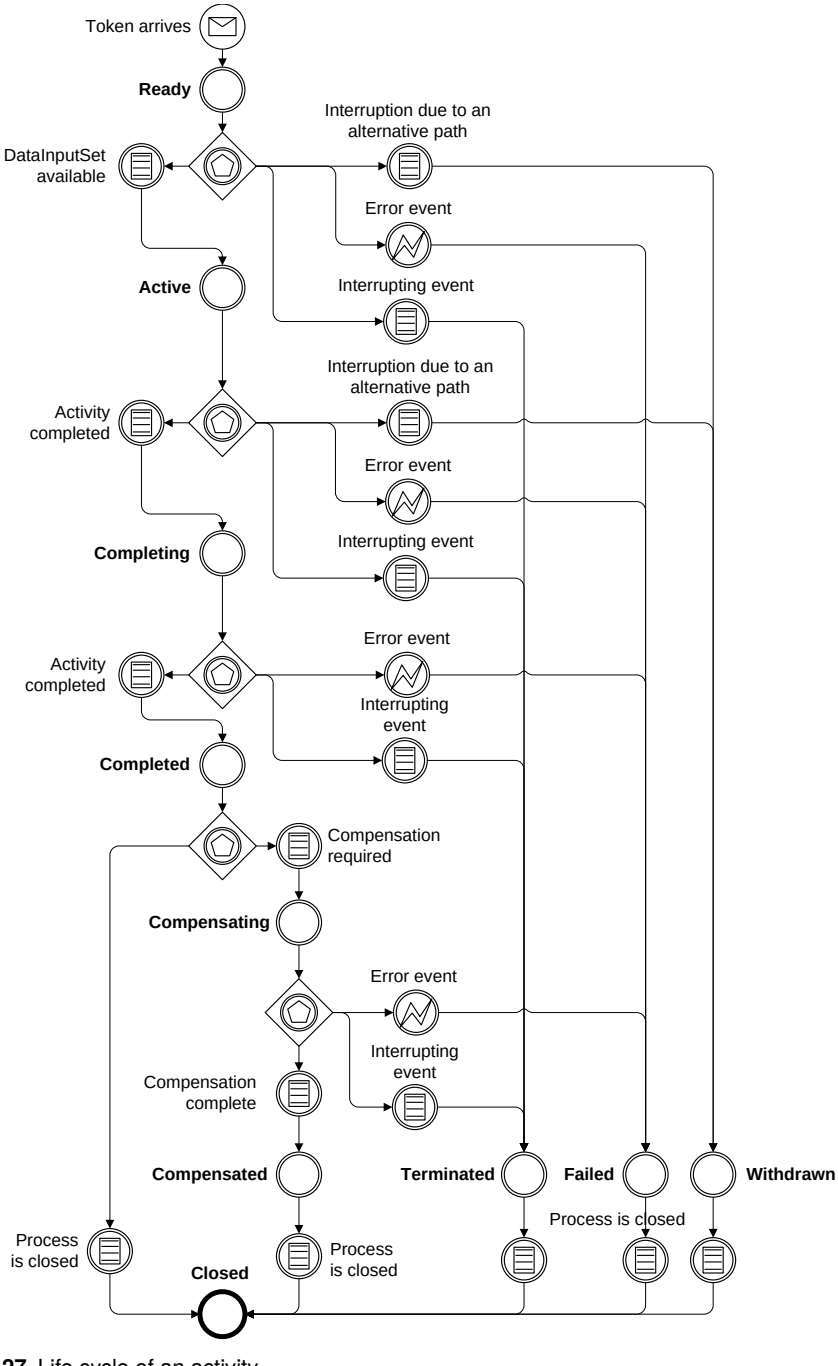


FIGURE 5.27 Life cycle of an activity.

achievement of process engine automation because in the past, writing such logs had to be programmed explicitly. Normally, engines generically support logging, so the log can be

written without knowing the process details. If you want to set additional configurations, BPMN provides an extension point in the form of the "auditing" element. What exactly the engine does with the auditing element isn't specified, hence it is a proprietary extension.

Monitoring covers several things. Technical monitoring of individual process instances is interesting. At what point in the process is the instance? Which data are contained in the process? Why did a certain error occur? There is also Business Activity Monitoring, which enables monitoring of all processes in real time, and it provides proactive warnings when problems are detected. Also, business monitoring is often intended to take key performance indicators into account that cannot be generically provided by the engine. Accordingly, BPMN provides an extension point called the "monitoring" element. As with auditing, there are no details given, so the engines therefore can anchor proprietary extensions as well.

5.4.11 Non-automatable tasks

We have to mention that not all the elements of a BPMN model are taken into account in the implementation. For example, a manual task, one processed entirely without IT support, is an element that is not executable and has no defined execution semantics. According to the specification, the process engine can either "devise" its own behavior —which means to extend the execution semantics —or it can simply ignore those elements.

Non-automatable elements are mainly manual or abstract tasks, physical process objects, and other less-exciting attributes to be looked up in the specification. Figure 5.28 shows a process with manual tasks that can be ignored by the process engine, though one problem becomes apparent: the decision in the gateway is redundant in automation; it does not change anything in the executed process. Some modeling tool may be able to represent that, though we expect that the gateway will always need to be provided with a default flow to maintain the part of the process that is automatable in this example.

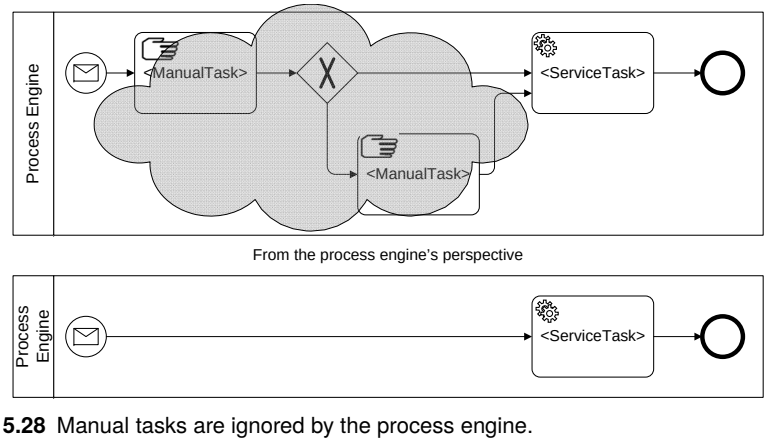


FIGURE 5.28 Manual tasks are ignored by the process engine.

" "
...