Chapter 3
Essential Process Modeling

Essentially, all models are wrong, but some are useful.
George E.P. Box (1919–)

Business process models are important at various stages of the BPM lifecycle. Before starting to model a process, it is crucial to understand why we are modeling it. The models we produce will look quite differently depending on the reason for modeling them in the first place. There are many reasons for modeling a process. The first one is simply to understand the process and to share our understanding of the process with the people who are involved with the process on a daily basis. Indeed, process participants typically perform quite specialized activities in a process such that they are hardly confronted with the complexity of the whole process. Therefore, process modeling helps to better understand the process and to identify and prevent issues. This step towards a thorough understanding is the prerequisite to conduct process analysis, redesign or automation.

In this chapter we will become familiar with the basic ingredients of process modeling using the BPMN language. With these concepts, we will be able to produce business process models that capture simple temporal and logical relations between activities, data objects and resources. First, we will describe some essential concepts of process models, namely how process models relate to process instances. Then, we will explain the four main structural blocks of branching and merging in process models. These define exclusive decisions, parallel execution, inclusive decisions and repetition. Finally, we will cover information artifacts and resources involved in a process.

3.1 First Steps with BPMN

With over 100 symbols, BPMN is a fairly complex language. But as a learner, there is no reason to panic. A handful of those symbols will already allow you to cover many of your modeling needs. Once you have mastered this subset of BPMN, the remaining symbols will naturally come to you with practice. So instead of describing each and every BPMN symbol at length, we will learn BPMN by introducing its symbols and concepts gradually, by means of examples.

M. Dumas et al., *Fundamentals of Business Process Management*,
DOI 10.1007/978-3-642-33143-5_3, © Springer-Verlag Berlin Heidelberg 2013

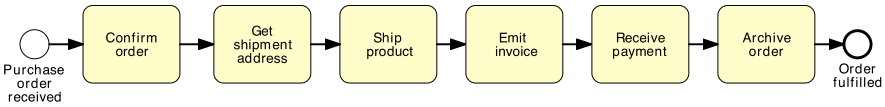


Fig. 3.1 The diagram of a simple order fulfillment process

In this chapter we will become familiar with the core set of symbols provided by BPMN. As stated earlier, a business process involves *events* and *activities*. Events represent things that happen instantaneously (e.g. an invoice has been received) whereas activities represent units of work that have a duration (e.g. an activity to pay an invoice). Also, we recall that in a process, events and activities are logically related. The most elementary form of relation is that of *sequence*, which implies that one event or activity A is followed by another event or activity B. Accordingly, the three most basic concepts of BPMN are event, activity, and arc. Events are represented by circles, activities by rounded rectangles, and arcs (called *sequence flows* in BPMN) are represented by arrows with a full arrow-head.

Example 3.1 Figure 3.1 shows a simple sequence of activities modeling an order fulfillment process in BPMN. This process starts whenever a purchase order has been received from a customer. The first activity that is carried out is confirming the order. Next, the shipment address is received so that the product can be shipped to the customer. Afterwards, the invoice is emitted and once the payment is received the order is archived, thus completing the process.

From the example above we notice that the two events are depicted with two slightly different symbols. We use circles with a thin border to capture start events and circles with a thick border to capture end events. Start and end events have an important role in a process model: the start event indicates when *instances* of the process start whereas the end event indicates when instances complete. For example, a new instance of the order fulfillment process is triggered whenever a purchase order is received, and completes when the order is fulfilled. Let us imagine that the order fulfillment process is carried out at a seller's organization. Every day this organization will run a number of instances of this process, each instance being independent of the others. Once a process instance has been spawned, we use the notion of *token* to identify the progress (or *state*) of that instance. Tokens are created in a start event, flow throughout the process model until they are destroyed in an end event. We depict tokens as colored dots on top of a process model. For example Fig. 3.2 shows the state of three instances of the order fulfillment process: one instance has just started (black token on the start event), another is shipping the product (red token on activity "Ship product"), and the third one has received the payment and is about to start archiving the order (green token in the sequence flow between "Receive payment" and "Archive order").

While it comes natural to give a name (also called *label*) to each activity, we should not forget to give labels to events as well. For example, giving a name to each start event allows us to communicate what triggers an instance of the process,

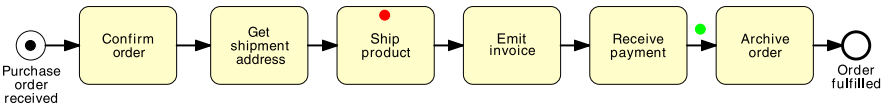


Fig. 3.2 Progress of three instances of the order fulfillment process

meaning, when should a new instance of the process be started. Similarly, giving a label to each end event allows us to communicate what conditions hold when an instance of the process completes, i.e. what the outcome of the process is.

We recommend the following naming conventions. For activities, the label should begin with a verb in the imperative form followed by a noun, typically referring to a business object, e.g. "Approve order". The noun may be preceded by an adjective, e.g. "Issue driver license", and the verb may be followed by a complement to explain how the action is being done, e.g. "Renew driver license via offline agencies". However, we will try to avoid long labels as this may hamper the readability of the model. As a rule of thumb, we will avoid labels with more than five words excluding prepositions and conjunctions. Articles are typically avoided to shorten labels. For events, the label should begin with a noun (again, this would typically be a business object) and end with a verb in past participle form, e.g. "Invoice emitted". The verb is a past participle to indicate something that has just happened. Similar to activity labels, the noun may be prefixed by an adjective, e.g. "Urgent order sent". We capitalize the first word of activity and event labels.

General verbs like "to make", "to do", "to perform" or "to conduct" should be replaced with meaningful verbs that capture the specifics of the activity being performed or the event occurring. Words like "process" or "order" are also ambiguous in terms of their part of speech. Both can be used as a verb ("to process", "to order") and as a noun ("a process", "an order"). We recommend to use such words consistently, only in one part of speech, e.g. "order" always as a noun.

To name a process model we should use a noun, potentially preceded by an adjective, e.g. "order fulfillment" or "claim handling" process. This label can be obtained by nominalizing the verb describing the main action of a business process, e.g. "fulfill order" (the main action) becomes "order fulfillment" (the process label). Nouns in hyphenated form like "order-to-cash" and "procure-to-pay" indicating the sequence of main actions in the process, are also possible.

We do not capitalize the first word of process names, e.g. the "order fulfillment" process. By following such naming conventions we will keep our models more consistent, make them easier to understand for communication purposes and increase their reusability.

The example in Fig. 3.1 represents one possible way of modeling the order fulfillment process. However, we could have produced a quite different process model. For example, we could have neglected certain activities or expanded on certain others, depending on the specific intent of our modeling. The box "A bit on modeling theory" reflects on the properties that underpin a model and relates these to the specific case of process models.

A BIT ON MODELING THEORY

A model is characterized by three properties: mapping, abstraction, and fit for purpose. First, a model implies a *mapping* of a real-world phenomenon—the modeling subject. For example, a residential building to be constructed could be modeled via a timber miniature. Second, a model only documents relevant aspects of the subject, i.e. it *abstracts* from certain details that are irrelevant. The timber model of the building clearly abstracts from the materials the building will be constructed from. Third, a model serves a particular *purpose*, which determines the aspects of reality to omit when creating a model. Without a specific purpose, we would have no indication on what to omit. Consider the timber model again. It serves the purpose of illustrating how the building will look like. Thus, it neglects aspects that are irrelevant for judging the appearance, like the electrical system of the building. So we can say that a model is a means to abstract from a given subject with the intent of capturing specific aspects of the subject.

Fig. 3.3 A building (**a**), its timber miniature (**b**) and its blueprint (**c**). ((**b**): © 2010, Bree Industries; (**c**): used by permission of planetclaire.org)

A way to determine the purpose of a model is to understand the *target audience* of the model. In the case of the timber model, the target audience could be a prospective buyer of the building. Thus, it is important to focus on the appearance of the building, rather than on the technicalities of the construction. On the other hand, the timber model would be of little use to an engineer who has to design the electrical system. In this case, a blueprint of the building would be more appropriate.

Thus, when modeling a business process, we need to keep in mind the specific purpose and target audience for which we are creating the model. There are two main purposes for process modeling: *organizational design* and *application system design*. Process models for organizational design are *business-oriented* . They are built by process analysts and mainly used for understanding and communication, but also for benchmarking and improvement. As such, they need to be intuitive enough to be comprehended by the various stakeholders, and will typically abstract from IT-related aspects. The target audience includes managers, process owners and business analysts. Process models for application system design are *IT-oriented* . They are built by

system engineers and developers, and used for automation. Thus, they must contain implementation details in order to be deployed to a BPMS, or used as blueprints for software development.

In this and in the next chapter we will focus on the business-oriented process models. In Chap. 9 we will learn how to turn these process models executable.

3.2 Branching and Merging

Activities and events may not necessarily be performed sequentially. For example, in the context of a claim handling process, the approval and the rejection of a claim are two activities which exclude each other. So these activities cannot be performed in sequence, since an instance of this process will perform either of these activities. When two or more activities are alternative to each other, we say they are *mutually exclusive*.

Let us consider another situation. In the claim handling process, once the claim has been approved, the claimant is notified and the disbursement is made. Notification and disbursement are two activities which are typically performed by two different business units, hence they are independent of each other and as such they do not need to be performed in sequence: they can be performed in parallel, i.e. at the same time. When two or more activities are not interdependent, they are *concurrent*.

To model these behaviors we need to introduce the notion of *gateway*. The term gateway implies that there is a gating mechanism that either allows or disallows passage of tokens through the gateway. As tokens arrive at a gateway, they can be merged together on input, or split apart on output depending on the gateway type. We depict gateways as diamonds and distinguish them between splits and joins. A *split* gateway represents a point where the process flow diverges while a *join* gateway represents a point where the process flow converges. Splits have one incoming sequence flow and multiple outgoing sequence flows (representing the branches that diverge), while joins have multiple incoming sequence flows (representing the branches to be merged) and one outgoing sequence flow.

Let us now see how examples like the above ones can be modeled with gateways.

3.2.1 Exclusive Decisions

To model the relation between two or more alternative activities, like in the case of the approval or rejection of a claim, we use an *exclusive* (*XOR*) *split*. We use an *XOR-join* to merge two or more alternative branches that may have previously been

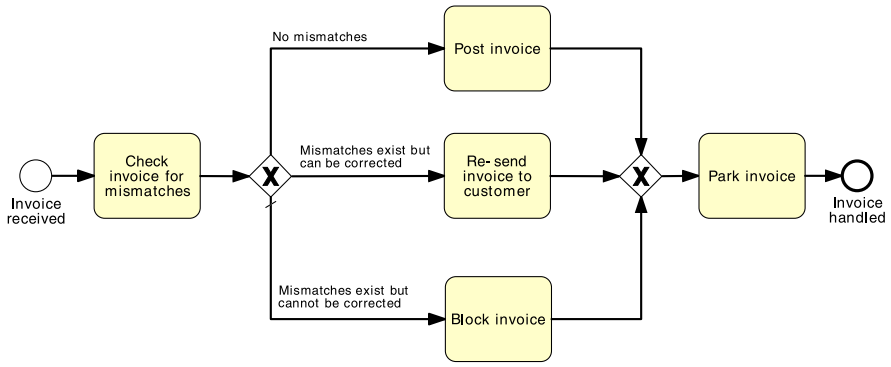


Fig. 3.4 An example of the use of XOR gateways

forked with an XOR-split. An XOR gateway is indicated with an empty diamond or with a diamond marked with an "X". From now on, we will always use the "X" marker.

Example 3.2 Invoice checking process.

> As soon as an invoice is received from a customer, it needs to be checked for mismatches. The check may result in either of these three options: i) there are no mismatches, in which case the invoice is posted; ii) there are mismatches but these can be corrected, in which case the invoice is re-sent to the customer; and iii) there are mismatches but these cannot be corrected, in which case the invoice is blocked. Once one of these three activities is performed the invoice is parked and the process completes.

To model this process we start with a decision activity, namely "Check invoice for mismatches" following a start event "Invoice received". A *decision activity* is an activity that leads to different outcomes. In our example, this activity results in three possible outcomes, which are mutually exclusive; so we need to use an XOR-split after this activity to fork the flow into three branches. Accordingly, three sequence flows will emanate from this gateway, one towards activity "Post invoice", performed if there are no mismatches, another one towards "Re-send invoice to customer", performed if mismatches exist but can be corrected, and a third flow towards "Block invoice", performed if mismatches exist which cannot be corrected (see Fig. 3.4). From a token perspective, an XOR-split routes the token coming from its incoming branch towards one of its outgoing branches, i.e. only one outgoing branch can be taken.

When using an XOR-split, make sure each outgoing sequence flow is annotated with a label capturing the condition upon which that specific branch is taken. Moreover, always use mutually exclusive conditions, i.e. only one of them can be true every time the XOR-split is reached by a token. This is the characteristic of the XOR-split gateway. In our example an invoice can either be correct, or contain mismatches that can be fixed, or mismatches that cannot be fixed: only one of these conditions is true per invoice received.

In Fig. 3.4 the flow labeled "mismatches exist but cannot be corrected" is marked with an oblique cut. This notation is optional and is used to indicate the *default flow*, i.e. the flow that will be taken by the token coming from the XOR-split in case the conditions attached to all the other outgoing flows evaluate to false. Since this arc has the meaning of *otherwise*, it can be left unlabeled. However, we highly recommend to still label this arc with a condition for readability purposes.

Once either of the three alternative activities has been executed, we merge the flow back in order to execute activity "Park invoice" which is common to all three cases. For this we use an XOR-join. This particular gateway acts as a *passthrough*, meaning that it waits for a token to arrive from one of its input arcs and as soon as it receives the token, it sends the token to the output arc. In other words, with an XOR-join we proceed whenever an incoming branch has completed.

Coming back to our example, we complete the process model with an end event "Invoice handled". Make sure to always complete a process model with an end event, even if it is obvious how the process would complete.

Exercise 3.1 Model the following fragment of a business process for assessing loan applications.

> Once a loan application has been approved by the loan provider, an acceptance pack is prepared and sent to the customer. The acceptance pack includes a repayment schedule which the customer needs to agree upon by sending the signed documents back to the loan provider. The latter then verifies the repayment agreement: if the applicant disagreed with the repayment schedule, the loan provider cancels the application; if the applicant agreed, the loan provider approves the application. In either case, the process completes with the loan provider notifying the applicant of the application status.

3.2.2 Parallel Execution

When two or more activities do not have any order dependencies on each other (i.e. one activity does not need to follow the other, nor it excludes the other) they can be executed concurrently, or *in parallel*. The *parallel* (*AND*) *gateway* is used to model this particular relation. Specifically, we use an *AND-split* to model the parallel execution of two or more branches, and an *AND-join* to synchronize the execution of two or more parallel branches. An AND gateway is depicted as a diamond with a "+" mark.

Example 3.3 Security check at the airport.

> Once the boarding pass has been received, passengers proceed to the security check. Here they need to pass the personal security screening and the luggage screening. Afterwards, they can proceed to the departure level.

This process consists of four activities. It starts with activity "Proceed to security check" and finishes with activity "Proceed to departure level". These two activities have a clear order dependency: a passenger can only go to the departure level after

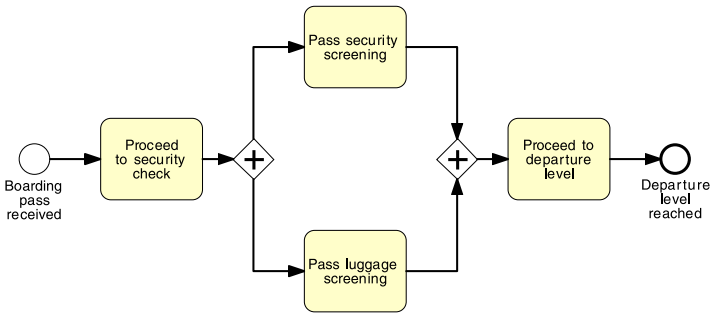


Fig. 3.5 An example of the use of AND gateways

undergoing the required security checks. After the first activity, and before the last one, we need to perform two activities which can be executed in any order, i.e. which do not depend on each other: "Pass personal security screening" and "Pass luggage screening". To model this situation we use an AND-split linking activity "Proceed to security check" with the two screening activities, and an AND-join linking the two screening activities with activity "Proceed to departure level" (see Fig. 3.5).

The AND-split *splits* the token coming from activity "Proceed to security check" into two tokens. Each of these tokens independently flows through one of the two branches. This means that when we reach an AND-split, we take all outgoing branches (note that an AND-split may have multiple outgoing arcs). As we said before, a token is used to indicate the state of a given instance. When multiple tokens of the same color are distributed across a process model, e.g. as a result of executing an AND-split, they collectively represent the state of an instance. For example, if a token is on the arc emitting from activity "Pass luggage screening" and another token of the same color is on the arc incident to activity "Pass personal security screening", this indicates an instance of the security check process where a passenger has just passed the luggage screening but not yet started the personal security screening.

The AND-join of our example waits for a token to arrive from each of the two incoming arcs, and once they are all available, it *merges* the tokens back into one. The single token is then sent to activity "Proceed to departure level". This means that we proceed when all incoming branches have completed (note again that an AND-join may have multiple incoming arcs). This behavior of waiting for a number of tokens to arrive and then merging the tokens into one is called *synchronization*.

Example 3.4 Let us extend the order fulfillment example of Fig. 3.1 by assuming that a purchase order is only confirmed if the product is in stock, otherwise the process completes by rejecting the order. Further, if the order is confirmed, the shipment address is received and the requested product is shipped *while* the invoice is emitted and the payment is received. Afterwards, the order is archived and the process completes.

The resulting model is shown in Fig. 3.6. Let us make a couple of remarks. First, this model has two activities that are mutually exclusive: "Confirm order" and "Re-

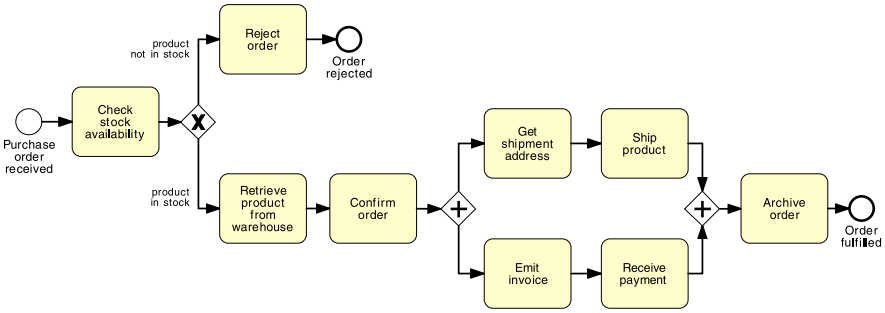


Fig. 3.6 A more elaborated version of the order fulfillment process diagram

ject order", thus we preceded them with an XOR-split (remember to put an activity before an XOR-split to allow the decision to be taken, such as a check like in this case, or an approval). Second, the two sequences "Get shipment address"–"Ship product" and "Emit invoice"–"Receive payment" can be performed independently of each other, so we put them in a block between an AND-split and an AND-join. In fact, these two sets of activities are typically handled by different resources within a seller's organization, like a sales clerk for the shipment and a financial officer for the invoice, and thus can be executed in parallel (note the word "meantime" in the process description, which indicates that two or more activities can be performed at the same time).

Let us compare this new version of the order fulfillment process with that in Fig. 3.1 in terms of events. The new version features two end events while the first version features one end event. In a BPMN model we can have multiple end events, each capturing a different outcome of the process (e.g. balance paid vs. arrears processed, order approved vs. order rejected). BPMN adopts the so-called *implicit termination* semantics, meaning that a process instance completes only when each token flowing in the model reaches an end event. Similarly, we can have multiple start events in a BPMN model, each event capturing a different trigger to start a process instance. For example, we may start our order fulfillment process either when a new purchase order is received or when a revised order is resubmitted. If a revised order is resubmitted, we first retrieve the order details from the orders database, and then continue with the rest of the process. This variant of the order fulfillment model is shown in Fig. 3.7. An instance of this process model is triggered by the first event that occurs (note the use of an XOR-join to merge the branches coming from the two start events).

Exercise 3.2 Model the following fragment of a business process for assessing loan applications.

A loan application is approved if it passes two checks: (i) the applicant's loan risk assessment, done automatically by a system, and (ii) the appraisal of the property for which the loan has been asked, carried out by a property appraiser. The risk assessment requires a

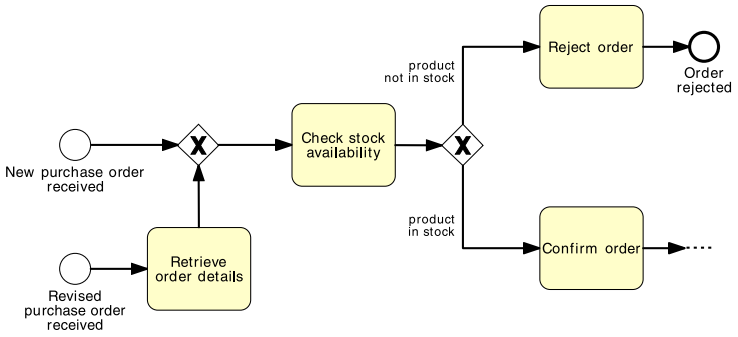


Fig. 3.7 A variant of the order fulfillment process with two different triggers

credit history check on the applicant, which is performed by a financial officer. Once both the loan risk assessment and the property appraisal have been performed, a loan officer can assess the applicant's eligibility. If the applicant is not eligible, the application is rejected, otherwise the acceptance pack is prepared and sent to the applicant.

There are two situations when a gateway can be omitted. An XOR-join can be omitted before an activity or event. In this case, the incoming arcs to the XOR-join are directly connected to the activity/event. An example of this shorthand notation is shown in Fig. 1.6, where there are two incident arcs to activity "Select suitable equipment". An AND-split can also be omitted when it follows an activity or event. In this case, the outgoing arcs of the AND-split emanate directly from the activity/event.

3.2.3 Inclusive Decisions

Sometimes we may need to take one *or more* branches after a decision activity. Consider the following business process.

Example 3.5 Order distribution process.

A company has two warehouses that store different products: Amsterdam and Hamburg. When an order is received, it is distributed across these warehouses: if some of the relevant products are maintained in Amsterdam, a sub-order is sent there; likewise, if some relevant products are maintained in Hamburg, a sub-order is sent there. Afterwards, the order is registered and the process completes.

Can we model the above scenario using a combination of AND and XOR gateways? The answer is yes. However, there are some problems. Figures 3.8 and 3.9 show two possible solutions. In the first one, we use an XOR-split with three alternative branches: one taken if the order only contains Amsterdam products (where the sub-order is forwarded to the Amsterdam warehouse), another taken if the order only contains Hamburg products (similarly, in this branch the sub-order is forwarded

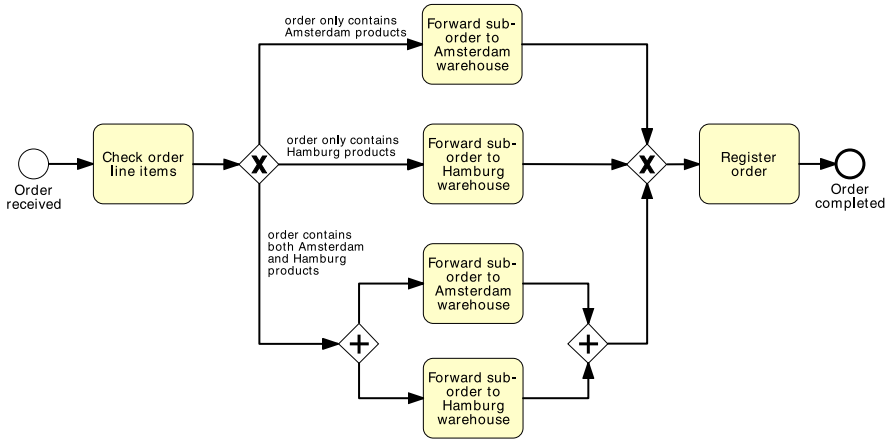


Fig. 3.8 Modeling an inclusive decision: first trial

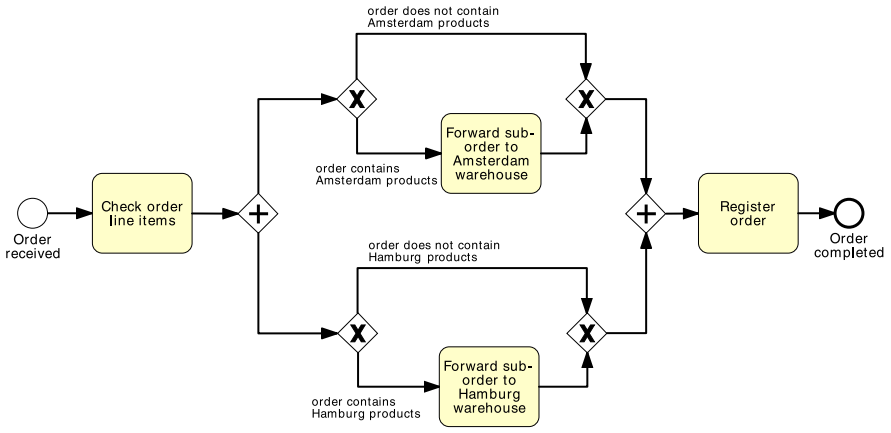


Fig. 3.9 Modeling an inclusive decision: second trial

to the Hamburg warehouse), and a third branch to be taken in case the order contains products from both warehouses (in which case sub-orders are forwarded to both warehouses). These three branches converge in an XOR-join which leads to the registration of the order.

While this model captures our scenario correctly, the resulting diagram is somewhat convoluted, since we need to duplicate the two activities that forward sub-orders to the respective warehouses twice. And if we had more than two warehouses, the number of duplicated activities would increase. For example, if we had three warehouses, we would need an XOR-split with seven outgoing branches, and each activity would need to be duplicated four times. Clearly this solution is not scalable.

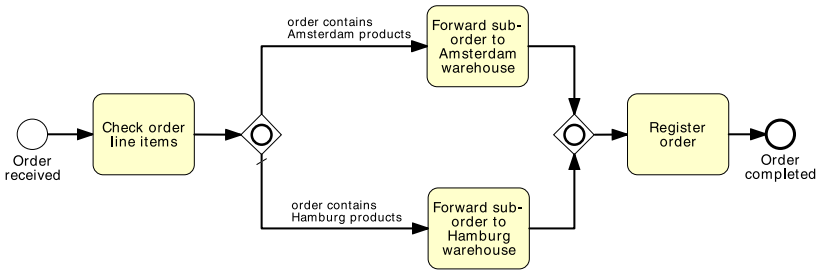


Fig. 3.10 Modeling an inclusive decision with the OR gateway

In the second solution we use an AND-split with two outgoing arcs, each of which leads to an XOR-split with two alternative branches. One is taken if the order contains Amsterdam (Hamburg) products, in which case an activity is performed to forward the sub-order to the respective warehouse; the other branch is taken if the order does not contain any Amsterdam (Hamburg) products, in which case nothing is done until the XOR-join, which merges the two branches back. Then an AND-join merges the two parallel branches coming out of the AND-split and the process completes by registering the order.

What is the problem with this second solution? The example scenario allows three cases: the products are in Amsterdam only, in Hamburg only, or in both warehouses, while this solution allows one more case, i.e. when the products are in neither of the warehouses. This case occurs when the two empty branches of the two XOR-splits are taken and results in doing nothing between activity "Check order line items" and activity "Register order". Thus this solution, despite being more compact than the first one, is wrong.

To model situations where a decision may lead to one or more options being taken at the same time, we need to use an *inclusive* (*OR*) *split gateway*. An *OR-split* is similar to the XOR-split, but the conditions on its outgoing branches do not need to be mutually exclusive, i.e. more than one of them can be true at the same time. When we encounter an OR-split, we thus take one or more branches depending on which conditions are true. In terms of token semantics, this means that the OR-split takes the input token and generates a number of tokens equivalent to the number of output conditions that are true, where this number can be at least one and at most as the total number of outgoing branches. Similar to the XOR-split gateway, an OR-split can also be equipped with a default flow, which is taken only when all other conditions evaluate to false.

Figure 3.10 shows the solution to our example using the OR gateway. After the sub-order has been forwarded to either of the two warehouses or to both, we use an *OR-join* to synchronize the flow and continue with the registration of the order. An OR-join proceeds when all *active* incoming branches have completed. Waiting for an active branch means waiting for an incoming branch that will ultimately de-

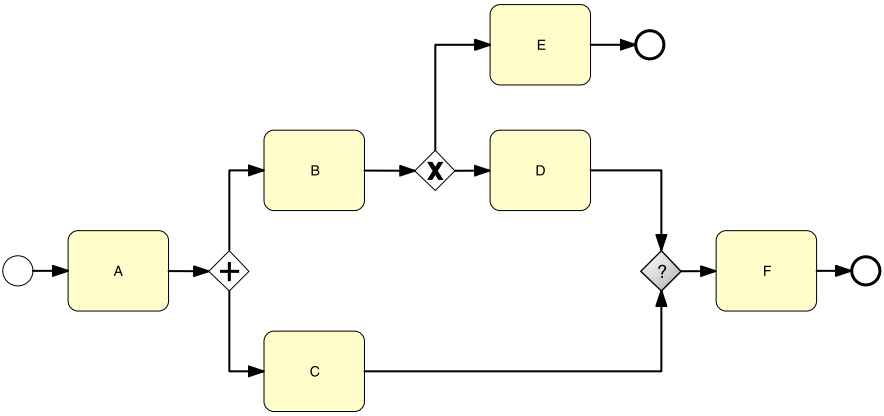


Fig. 3.11 What type should the join gateway have such that instances of this process can complete correctly?

liver a token to the OR-join. If the branch is active, the OR-join will wait for that token, otherwise it will not. Once all tokens of active branches have arrived, the OR-join synchronizes these tokens into one (similarly to what an AND-join does) and sends that token to its output arc. We call this behavior *synchronizing merge* as opposed to the simple merge of the XOR-join and the synchronization of the AND-join.

Let us delve into the concept of active branch. Consider the model in Fig. 3.11, which features a join gateway with undefined type (the one grayed out with a question mark). What type should we assign to this join? Let us try an AND-join to match the preceding AND-split. We recall that an AND-join waits for a token to arrive from each incoming branch. While the token from the branch with activity "C" will always arrive, the token from the branch with activities "B" and "D" may not arrive if this is routed to "E" by the XOR-split. So if activity "D" is not executed, the AND-join will wait indefinitely for that token, with the consequence that the process instance will not be able to progress any further. This behavioral anomaly is called *deadlock* and should be avoided.

Let us try an XOR-join. We recall that the XOR-join works as a passthrough by forwarding to its output branch each token that arrives through one of its input branches. In our example this means that we may execute activity "F" once or twice, depending whether the preceding XOR-split routes the token to "E" (in this case "F" is executed once) or to "D" ("F" is executed twice). While this solution may work, we have the problem that we do not know whether activity "F" will be executed once or twice, and we may actually not want to execute it twice. Moreover, if this is the case, we would signal that the process has completed twice, since the end event following "F" will receive two tokens. And this, again, is something we want to avoid.

The only join type left to try is the OR-join. An OR-join will wait for all incoming active branches to complete. If the XOR-split routes control to "E", the OR-join will not wait for a token from the branch bearing activity "D", since this will never arrive. Thus, it will proceed once the token from activity "C" arrives. On the other hand, if the XOR-split routes control to "D", the OR-join will wait for a token to also arrive from this branch, and once both tokens have arrived, it will merge them into one and send this token out, so that "F" can be executed once and the process can complete normally.

Question When should we use an OR-join?

Since the OR-join semantics is not simple, the presence of this element in a model may confuse the reader. Thus, we suggest to use it only when it is strictly required. Clearly, it is easy to see that an OR-join must be used whenever we need to synchronize control from a preceding OR-split. Similarly, we should use an AND-join to synchronize control from a preceding AND-split and an XOR-join to merge a set of branches that are mutually exclusive. In other cases the model will not have a lean structure like the examples in Fig. 3.8 or 3.10, where the model is made up of nested blocks each delimited by a split and a join of the same type. The model may rather look like that in Fig. 3.11, where there can be entry points into, or exist points from a block-structure. In these cases play the token game to understand the correct join type. Start with an XOR-join; next try an AND-join and if both gateways lead to incorrect models use the OR-join which will work for sure.

Now that we have learned the three core gateways, let us use them to extend the order fulfillment process. Assume that if the product is not in stock, it can be manufactured. In this way, an order can never be rejected.

Example 3.6

If the product requested is not in stock, it needs to be manufactured before the order handling can continue. To manufacture a product, the required raw materials have to be ordered. Two preferred suppliers provide different types of raw material. Depending on the product to be manufactured, raw materials may be ordered from either Supplier 1 or Supplier 2, or from both. Once the raw materials are available, the product can be manufactured and the order can be confirmed. On the other hand, if the product is in stock, it is retrieved from the warehouse before confirming the order. Then the process continues normally.

The model for this extended order fulfillment process is shown in Fig. 3.12.

Exercise 3.3 Model the following fragment of a business process for assessing loan applications.

A loan application may be coupled with a home insurance which is offered at discounted prices. The applicant may express their interest in a home insurance plan at the time of submitting their loan application to the loan provider. Based on this information, if the loan application is approved, the loan provider may either only send an acceptance pack to the applicant, or also send a home insurance quote. The process then continues with the verification of the repayment agreement.

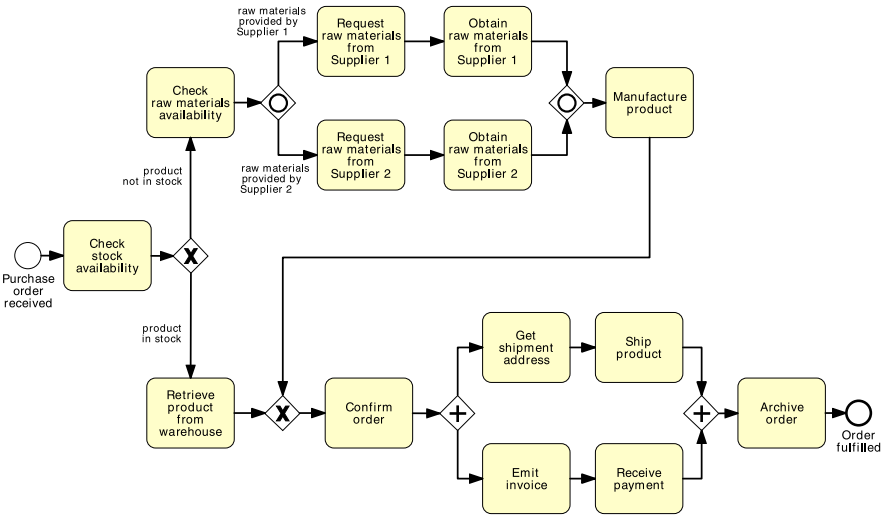


Fig. 3.12 The order fulfillment process diagram with product manufacturing

3.2.4 Rework and Repetition

So far we have seen structures that are linear, i.e. each activity is performed at most once. However, sometimes we may require to repeat one or several activities, for instance because of a failed check.

Example 3.7

> In the treasury minister's office, once a ministerial inquiry has been received, it is first registered into the system. Then the inquiry is investigated so that a ministerial response can be prepared. The finalization of a response includes the preparation of the response itself by the cabinet officer and the review of the response by the principal registrar. If the registrar does not approve the response, the latter needs to be prepared again by the cabinet officer for review. The process finishes only once the response has been approved.

To model rework or repetition we first need to identify the activities, or more in general the fragment of the process, that can be repeated. In our example this consists of the sequence of activities "Prepare ministerial response" and "Review ministerial response". Let us call this our *repetition block*. The property of a repetition block is that the last of its activities must be a decision activity. In fact, this will allow us to decide whether to go back before the repetition block starts, so that this can be repeated, or to continue with the rest of the process. As such, this decision activity should have two outcomes. In our example the decision activity is "Review ministerial response" and its outcomes are: "response approved" (in this case we continue with the process) and "response not approved" (we go back). To model these two outcomes, we use an XOR-split with two outgoing branches: one which allows us to continue with the rest of the process (in our example, this is simply

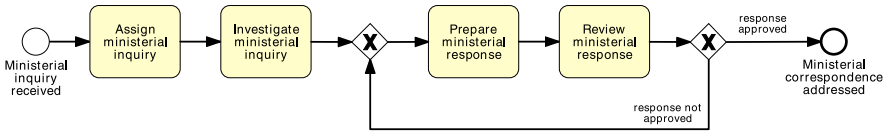


Fig. 3.13 A process model for addressing ministerial correspondence

the end event "Ministerial correspondence addressed"), the other which goes back to before activity "Prepare ministerial response". We use an XOR-join to reconnect this branch to the point of the process model just before the repetition block. The model for our example is illustrated in Fig. 3.13.

Question Why do we need to merge the loopback branch of a repetition block with an XOR-join?

The reason for using an XOR-join is that this gateway has a very simple semantics: it moves any token it receives in its input arc to its output arc, which is what we need in this case. In fact, if we merged the loopback branch with the rest of the model using an AND-join we would deadlock since this gateway would try to synchronize the two incoming branches when we know that only one of them can be active at a time: if we were looping we would receive the token from the loopback branch; otherwise we would receive it from the other branch indicating that we are entering the repetition block for the first time. An OR-join would work but is an overkill since we know that only one branch will be active at a time.

Exercise 3.4 Model the following fragment of a business process for assessing loan applications.

> Once a loan application is received by the loan provider, and before proceeding with its assessment, the application itself needs to be checked for completeness. If the application is incomplete, it is returned to the applicant, so that they can fill out the missing information and send it back to the loan provider. This process is repeated until the application is found complete.

We have learned how to combine activities, events, and gateways to model basic business processes. For each such element we have showed its graphical representation, the rules for combining it with other modeling elements and explained its behavior in terms of token rules. All these aspects fall under the term *components of a modeling language*. If you want to know more about this topic, you can read the box "Components of a modeling language".

COMPONENTS OF A MODELING LANGUAGE

A *modeling language* consists of three parts: syntax, semantics, and notation. The *syntax* provides a set of modeling elements and a set of rules to govern

how these elements can be combined. The *semantics* bind the syntactical elements and their textual descriptions to a precise meaning. The *notation* defines a set of graphical symbols for the visualization of the elements.

For example, the BPMN syntax includes activities, events, gateways, and sequence flows. An example of syntactical rule is that start events only have outgoing sequence flows whereas end events only have incoming sequence flows. The BPMN semantics describes which kind of behavior is represented by the various elements. In essence, this relates to the question how the elements can be executed in terms of token flow. For example, an AND-join has to wait for all incoming branches to complete before it can pass control to its outgoing branch. An example of BPMN notation is the use of labeled rounded boxes to depict activities.

3.3 Information Artifacts

As shown in Chap. 2, a business process entails different organizational aspects such as functions, business artifacts, humans, and software systems. These aspects are captured by different process modeling perspectives. So far we have seen the *functional perspective*, which indicates what activities should happen in the process, and the *control-flow perspective*, which indicates when activities and events should occur. Another important perspective that we ought to consider when modeling business processes is the *data perspective*. The data perspective indicates which information artifacts (e.g. business documents, files) are required to perform an activity and which ones are produced as a result of performing an activity.

Let us enrich the order fulfillment process of Example 3.6 with artifacts. Let us start by identifying the artifacts that each activity requires in order to be executed, and those that each activity creates as a result of its execution. For example, the first activity of the order fulfillment process is "Check stock availability". This requires a Purchase order as input in order to check whether or not the ordered product is in stock. This artifact is also required by activity "Check raw materials availability" should the product be manufactured. Artifacts like Purchase order are called *data objects* in BPMN. Data objects represent information flowing in and out of activities; they can be physical artifacts such as an invoice or a letter on a piece of paper, or electronic artifacts such as an e-mail or a file. We depict them as a document with the upper-right corner folded over, and link them to activities with a dotted arrow with an open arrowhead (called *data association* in BPMN). Figure 3.14 shows the data objects involved in the order fulfillment process model.

We use the direction of the data association to establish whether a data object is an input or output for a given activity. An incoming association, like the one used from Purchase order to activity "Check stock availability", indicates that Purchase order is an input object for this activity; an outgoing association, like the one used

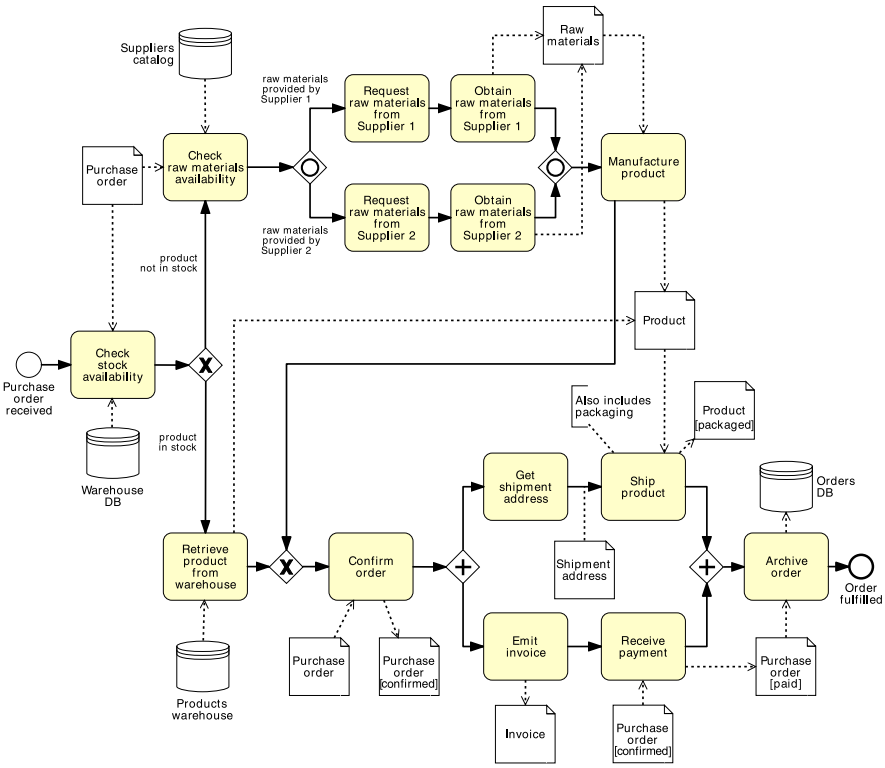


Fig. 3.14 The order fulfillment example with artifacts

from activity "Obtain raw materials from Supplier 1" to Raw materials, indicates that Raw materials is an output object for this activity. To avoid cluttering the diagram with data associations that cross sequence flows, we may repeat a data object multiple times within the same process model. However, all occurrences of a given object do conceptually refer to the same artifact. For example, in Fig. 3.14 Purchase order is repeated twice as input to "Check stock availability" and to "Confirm order" since these two activities are far away from each other in terms of model layout.

Often the output from an activity coincides with the input to a subsequent activity. For example, once Raw materials have been obtained, these are used by activity "Manufacture product" to create a Product. The Product in turn is packaged and sent to the customer by activity "Ship product". Effectively, data objects allow us to model the information flow between process activities. Bear in mind, however, that data objects and their associations with activities cannot replace the sequence flow. In other words, even if an object is passed from an activity A to an activity B, we still need to model the sequence flow from A to B. A shorthand notation for passing an object from an activity to the other is by directly connecting the data object to the sequence flow between two consecutive activities via an undirected association. See for example the Shipment address being passed from activity "Get

shipment address" to activity "Ship product", which is a shorthand for indicating that Shipment address is an output of "Get shipment address" and an input to "Ship product".

Sometimes we may need to represent the *state* of a data object. For instance, activity "Confirm order" takes a Purchase order as input, and returns a "confirmed" Purchase order as output: input and output objects are the same, but the object's state has changed to "confirmed". Similarly, activity "Receive payment" takes as input a "confirmed" Purchase order and transforms it into a "paid" Purchase order. An object can go through a number of states, e.g. an invoice is first "opened", then "approved" or "rejected" and finally "archived". Indicating data objects' states is optional: we can do so by appending the name of the state between square brackets to a data object's label, e.g. "Purchase Order [confirmed]", "Product [packaged]".

A *data store* is a place containing data objects that need to be persisted beyond the duration of a process instance, e.g. a database for electronic artifacts or a filing cabinet for physical ones. Process activities can read/write data objects from/to data stores. For example, activity "Check stock availability" retrieves the Stock levels for the ordered product from the Warehouse database, which contains Stock level information for the various Products. Similarly, activity "Check raw materials availability" consults the Suppliers catalog to check which Supplier to contact. The Warehouse database or the Supplier catalog are examples of data stores used as input to activities. An example of data store employed as output is the Orders database, which is used by activity "Archive order" to store the confirmed Purchase order. In this way, the order just archived will be available for other business processes within the same organization, e.g. for a business process that handles requests for product returns. Data stores are represented as an empty cylinder (the typical database symbol) with a triple upper border. Similar to data objects, they are connected to activities via data associations.

Question Do data objects affect the token flow?

Input data objects are required for an activity to be executed. Even if a token is available on the incoming arc of that activity, the latter cannot be executed until all input data objects are also available. A data object is available if it has been created as a result of completing a preceding activity (whose output was the data object itself), or because it is an input to the whole process (like Purchase order). Output data objects only affect the token flow indirectly, i.e. when they are used by subsequent activities.

Question Do we always need to model data objects?

Data objects help the reader understand the flow of business data from one activity to the other. However, the price to pay is an increased complexity of the diagram. Thus, we suggest to use them only when they are needed for a specific purpose, e.g. to highlight potential issues in the process under analysis (cf. Chaps. 6 and 7) or for automation (cf. Chap. 9).

Sometimes we may need to provide additional information to the process model reader, for the sake of improving the understanding of the model. For example, in the order fulfillment process we may want to specify that activity "Ship product" includes the packaging of the product. Also, we may want to clarify what business rule is followed behind the choice of raw materials from Suppliers. Such additional information can be provided via *text annotations*. An annotation is depicted as an open-ended rectangle encapsulating the text of the annotation, and is linked to a process modeling element via a dotted line (called *association*)—see Fig. 3.14 for an example. Text annotations do not bear any semantics, thus they do not affect the flow of tokens through the process model.

Exercise 3.5 Put together the four fragments of the loan assessment process that you created in Exercises 3.1–3.4.

Hint Look at the labels of the start/end events to understand the order dependencies among the various fragments. Then extend the resulting model by adding all the required artifacts. Moreover, attach annotations to specify the business rules behind (i) checking an application completeness, (ii) assessing an application eligibility, and (iii) verifying a repayment agreement.

3.4 Resources

A further aspect we need to consider when modeling business processes is the *resource perspective*. This perspective, also called the *organizational perspective*, indicates who or what performs which activity. *Resource* is a generic term to refer to anyone or anything involved in the performance of a process activity. A resource can be:

- A *process participant*, i.e. an individual person like the employee John Smith.
- A *software system*, for example a server or a software application.
- An *equipment*, such as a printer or a manufacturing plant.

We distinguish between *active resources*, i.e. resources that can autonomously perform an activity, and *passive resources*, i.e. resources that are merely involved in the performance of an activity. For example, a photocopier is used by a participant to make a copy of a document, but it is the participant who performs the photocopying activity. So, the photocopier is a passive resource while the participant is an active resource. A bulldozer is another example of a passive resource since it is the driver who performs the activity in which the bulldozer is used.

The resource perspective of a process is interested in active resources, so from now on with the term "resource" we refer to an "active resource".

Frequently, in a process model we do not explicitly refer to one resource at a time, like for example an employee John Smith, but instead we refer to a group of resources that are interchangeable in the sense that any member of the group can

perform a given activity. Such groups are called *resource classes*. Examples are a whole organization, an organizational unit or a role.[1]

Let us examine the resources involved in our order fulfillment example.

Example 3.8

> The order fulfillment process is carried out by a seller's organization which includes two departments: the sales department and the warehouse & distribution department. The purchase order received by warehouse & distribution is checked against the stock. This operation is carried out automatically by the ERP system of warehouse & distribution, which queries the warehouse database. If the product is in stock, it is retrieved from the warehouse before sales confirm the order. Next sales emit an invoice and wait for the payment, while the product is shipped from within warehouse & distribution. The process completes with the order archival in the sales department. If the product is not in stock, the ERP system within warehouse & distribution checks the raw materials availability by accessing the suppliers catalog. Once the raw materials have been obtained the warehouse & distribution department takes care of manufacturing the product. The process completes with the purchase order being confirmed and archived by the sales department.

BPMN provides two constructs to model resource aspects: *pools* and *lanes*. Pools are generally used to model resource classes, lanes are used to partition a pool into sub-classes or single resources. There are no constraints as to what specific resource type a pool or a lane should model. We would typically use a pool to model a *business party* like a whole organization such as the seller in our example, and a lane to model a department, unit, team or software system/equipment within that organization. In our example, we partition the Seller pool into two lanes: one for the warehouse & distribution department, the other for the sales department.

Lanes can be nested within each other in multiple levels. For example, if we need to model both a department and the roles within that department, we can use one outer lane for the department, and one inner lane for each role. In the order fulfillment example we nest a lane within Warehouse & Distribution to represent the ERP System within that department.

Pools and lanes are depicted as rectangles within which we can place activities, events, gateways, and data objects. Typically, we model these rectangles horizontally, though modeling them vertically is also possible. The name of the pool/lane is shown vertically on the left-hand side of a horizontal rectangle (or horizontally if the pool/lane is vertical); for pools, and for lanes containing nested lanes , the name is enclosed in a band. Figure 3.15 shows the revised order fulfillment example with resource aspects.

It is important to place an activity within the right lane. For example, we placed activity "Check stock availability" under the ERP System lane of Warehouse & Distribution to indicate that this activity is carried out automatically by the ERP system of that department. It is also important to place events properly within lanes. In our example we put event "Purchase order received" under the ERP system lane to indicate that the process starts within the ERP system of Warehouse & Distribution,

[1] In BPMN the term "participant" is used in a broad sense as a synonym of resource class, though in this book we do not adopt this definition.

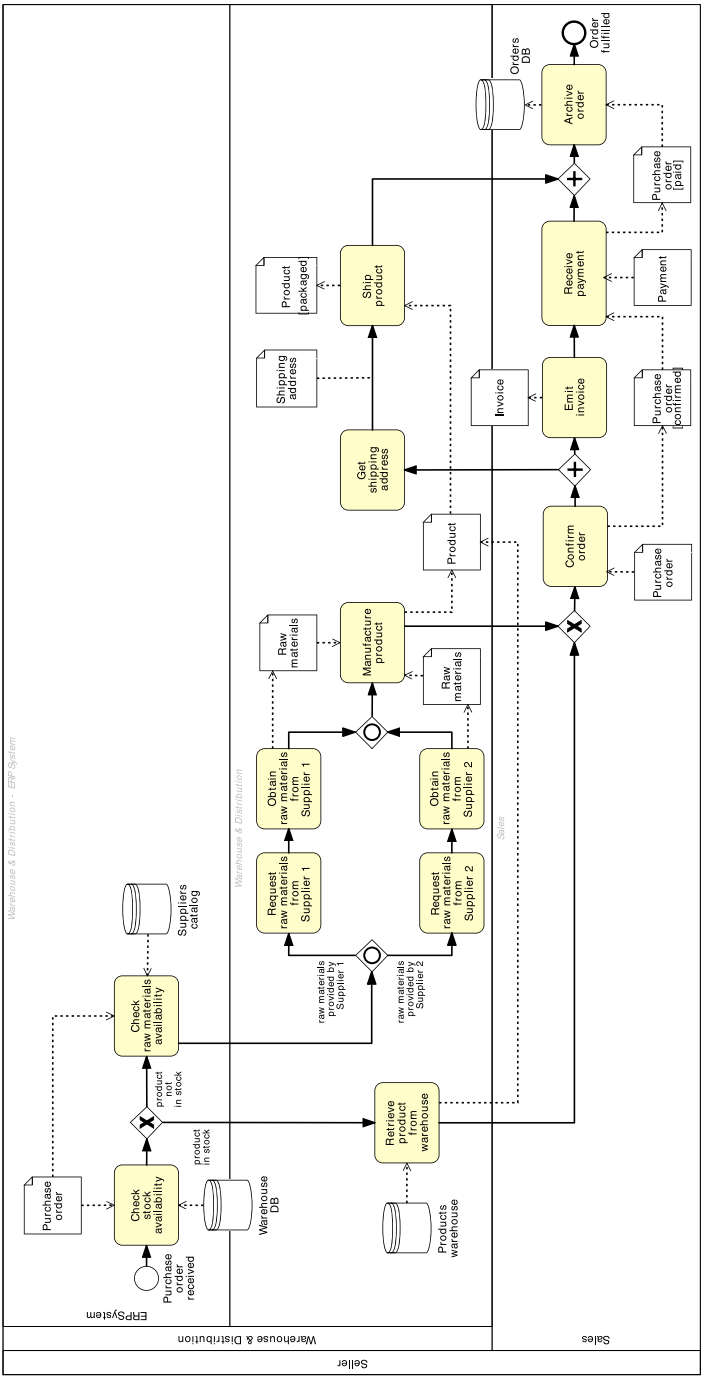


Fig. 3.15 The order fulfillment example with resource information

while we put event "Order fulfilled" under the Sales pool to indicate that the process completes in the sales department. It is not relevant where data objects are put, as they depend on the activities they are linked to. As per gateways, we need to place those modeling (X)OR-splits under the same lane as the preceding decision activity has been put in. On the other hand, it is irrelevant where we place an AND-split and all join gateways, since these elements are passive in the sense that they behave according to their context.

We may organize lanes within a pool in a matrix when we need to model complex organizational structures. For example, if we have an organization where roles span different departments, we may use horizontal lanes to model the various departments, and vertical lanes to model the roles within these departments. Bear in mind, however, that in BPMN each activity can be performed by one resource only. Thus, if an activity sits in the intersection of a horizontal lane with a vertical lane, it will be performed by the resource that fulfills the characteristics of both lanes, e.g. a resource that has that role and belongs to that department.

Exercise 3.6 Extend the business process for assessing loan applications that you created in Exercise 3.5 by considering the following resource aspects.

> The process for assessing loan applications is executed by four roles within the loan provider: a financial officer takes care of checking the applicant's credit history; a property appraiser is responsible for appraising the property; an insurance sales representative sends the home insurance quote to the applicant if this is required. All other activities are performed by the loan officer who is the main point of contact with the applicant.

Often there is more than one business party participating in the same business process. For example, in the order fulfillment process there are four parties: the seller, the customer and the two suppliers.

Each party can be modeled by a pool. In our example we can thus use one pool for the customer, one for the seller and one for each supplier. Each of these pools will contain the activities, events, gateways, and data objects that model the specific portion of the business process occurring at that organization. Or to put it differently, each pool will model the same business process from the perspective of a specific organization. For example, event "Purchase order received" which sits in the Sales pool, will have a corresponding activity "Submit purchase order" occurring in the Customer pool. Similarly, activity "Ship product" from Sales will have a counterpart activity "Receive product" in the Customer pool. So, how can we model the interactions among the pools of two collaborating organizations? We cannot use the sequence flow to connect activities that belong to different pools since the sequence flow cannot cross the boundary of a pool. For this, we need to use a specific element called *message flow*.

A message flow represents the flow of information between two separate resource classes (pools). It is depicted as a dashed line which starts with an empty circle and ends with an empty arrowhead, and bears a label indicating the content of the message, e.g. a fax, a purchase order, but also a letter or a phone call. That is, the

message flow models any type of communication between two organizations, no matter if this is electronic like sending a purchase order via e-mail or transmitting a fax, or manual like making a phone call or handing over a letter on paper.

Figure 3.16 shows the complete order fulfillment process model including the pools for the customer and the two suppliers. Here we can see that message flows are labeled with the piece of information they carry, e.g. "Raw materials" or "Shipment address". An incoming message flow may lead to the creation of a data object by the activity that receives the message. For example, the message flow "Raw materials" is received by activity "Obtain raw materials from Supplier 1" which then creates the data object "Raw materials". This is also the case of the purchase order, which is generated by the start event "Purchase order received" from the content of the incoming message flow. We do not need to create a data object for each incoming message flow, only when the information carried by the message is needed elsewhere in the process. In our case, "Raw materials" is consumed by activity "Manufacture product" so we need to represent it as a data object. Similarly, we do not need to explicitly represent the data object that goes into an outgoing message if this data object is not needed elsewhere in the process. For example, activity "Emit invoice" generates an invoice which is sent to the customer, but there is no data object "Invoice" since this is not consumed by any activity in the Seller pool.

A BPMN diagram that features two or more pools is called *collaboration diagram*. Figure 3.16 shows different uses of a pool in a collaboration diagram. A pool like that for the seller is called *private process*, or *white box* pool, since it shows how effectively the seller organization participates in the order fulfillment process in terms of activities, events, gateways, and data objects. On the contrary, a pool like that for the customer and the two suppliers is called *public process*, or *black box* pool, since it hides how these organizations actually participate in the order fulfillment process. In order to save space, we can represent a black box with a *collapsed pool*, which is an empty rectangle bearing the name of the pool in the middle.

Question Black box or white box?

Modeling a pool as a white box or as a black box is a matter of relevance. When working on a collaboration diagram, an organization may decide whether or not to expose their internal behavior depending on the requirements of the project at hand. For example, if we are modeling the order fulfillment process from the seller's perspective, it may be relevant to expose the business process of the seller only, but not that of the customer and the suppliers. That is, the internal behavior of the customer and that of the suppliers are not relevant for the sake of understanding how the seller should fulfill purchase orders, and as such they can be hidden. On the other hand, if we need to improve the way the seller fulfills purchase orders, we may also want to know what it takes for a supplier to provide raw materials, as a delay in the provision of raw materials will slow down the product manufacturing

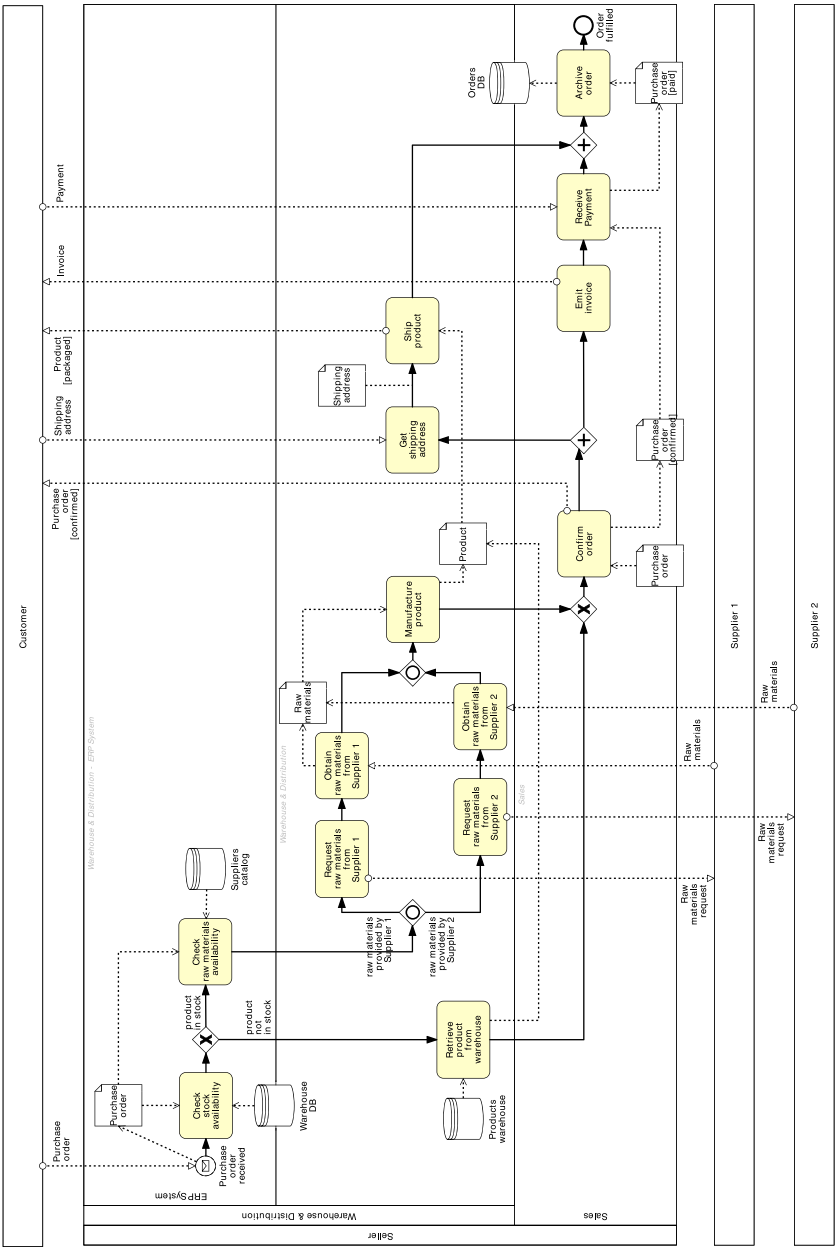


Fig. 3.16 Collaboration diagram between a seller, a customer and two suppliers

at the seller's side. In this case, we should also represent the suppliers using white box pools.

The type of pool affects the way we use the message flow to connect to the pool. Accordingly, a message flow may cross the boundary of a white box pool and connect directly to an activity or event within that pool, like the Purchase order message which is incident to the start event in the Seller pool. On the other hand, since a black box pool is empty, message flows must stop at the boundary or emanate from the boundary of a black box pool. Bear in mind that a message flow is only used to connect two pools and never to connect two activities within the same pool. For that, we use a sequence flow.

An activity that is the source of a message—such as "Emit invoice" in the Seller pool—is called a *send activity*. The message is sent upon completion of the activity's execution. On the other hand, an activity that receives a message—such as "Get shipping address"—is a *receive activity*.[2] The execution of such an activity will not start until the incoming message is available. An activity can act as both a receive and a send activity when it has both an incoming and outgoing message flow, e.g. "Make payment". The execution of this activity will start when both the control-flow token and the incoming message are available. Upon completion of the activity, a control-flow token will be put on the output arc and the outgoing message will be sent out. Finally, when a message flow is incident to a start event like "Purchase order received", we need to mark this event with a light envelope (see Fig. 3.16). This event type is called *message event*. A message event can be linked to an output data object in order to store the content of the incoming message. We will learn more about events in the next chapter.

Exercise 3.7 Extend the model of Exercise 3.6 by representing the interactions between the loan provider and the applicant.

In the order fulfillment example we used pools to represent business parties and lanes to represent the departments and systems within the sales organization. This is because we wanted to focus on the interactions between the seller, the customer and the two suppliers. As mentioned before, this is the typical use for pools and lanes. However, since BPMN does not prescribe what specific resource types should be associated with pools and lanes, we may use these elements differently. For example, if the focus is on the interactions between the departments of an organization, we can model each department with a pool, and use lanes to partition the departments, e.g. in units or roles. In any case, we should avoid to use pools and lanes to capture participants by their names since individuals tend to change frequently within an organization; rather, we should use the participant's role, e.g. financial officer. On the other hand, we can use pools and lanes to represent specific software systems or equipments, e.g. an ERP system, since these change less frequently in an organization.

[2]More specifically, "Emit invoice" is a *send task* and "Get shipping address" is a *receive task*. The distinction between activity and task will be discussed in Chap. 4.

3.5 Recap

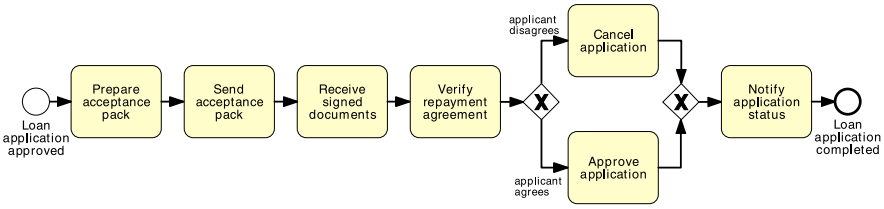
At the end of this chapter, we should be able to understand and produce basic process models in BPMN. A basic BPMN model includes simple activities, events, gateways, data objects, pools, and lanes. Activities capture units of work within a process. Events define the start and end of a process, and signal something that happens during the execution of it. Gateways model exclusive and inclusive decisions, merges, parallelism and synchronization, and repetition. We studied the difference between process model and process instance. A process model depicts all the possible ways a given business process can be executed, while a process instance captures one specific process execution out of all possible ones. The progress, or state, of a process instance is represented by tokens. Using tokens we can define the behavior of gateways.

We also learned how to use data objects to model the information flow between activities and events. A data object captures a physical or an electronic artifact required to execute an activity or trigger an event, or that results from the execution of an activity or an event occurrence. Data objects can be stored in a data store like a database or file cabinet such that they can be persisted beyond the process instance where they are created. Furthermore, we saw how pools and lanes can be used to model both human and non-human resources that perform process activities. Pools generally model resource classes while lanes are used to partition pools. The interaction between pools is captured by message flows. Message flows can be directly attached to the boundary of a pool, should the details of the interaction not be relevant.

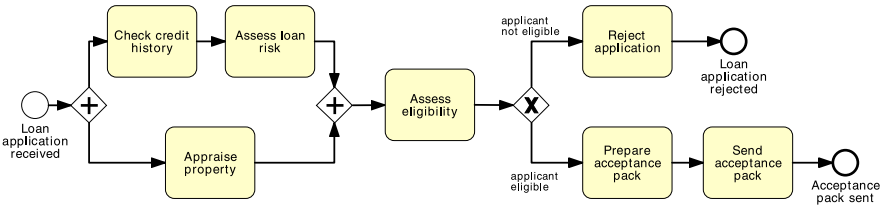
Activities, events, gateways, artifacts, and resources belong to the main modeling perspectives of a business process. The functional perspective captures the activities that are performed in a business process while the control-flow perspective combines these activities and related events in a given order. The data perspective covers the artifacts manipulated in the process while the resource perspective covers the resources that perform the various activities. In the next chapter, we will learn how to model complex business processes by delving into the various extensions of the core BPMN elements that we presented here.

3.6 Solutions to Exercises

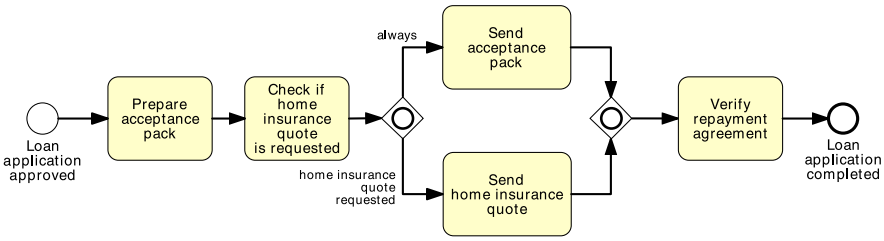
Solution 3.1



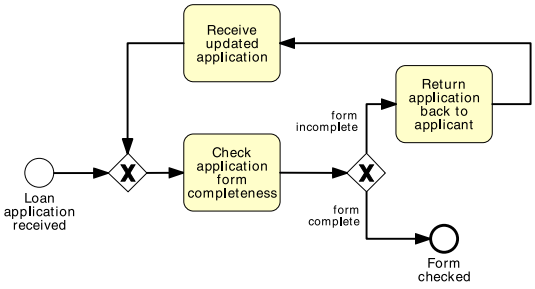
Solution 3.2



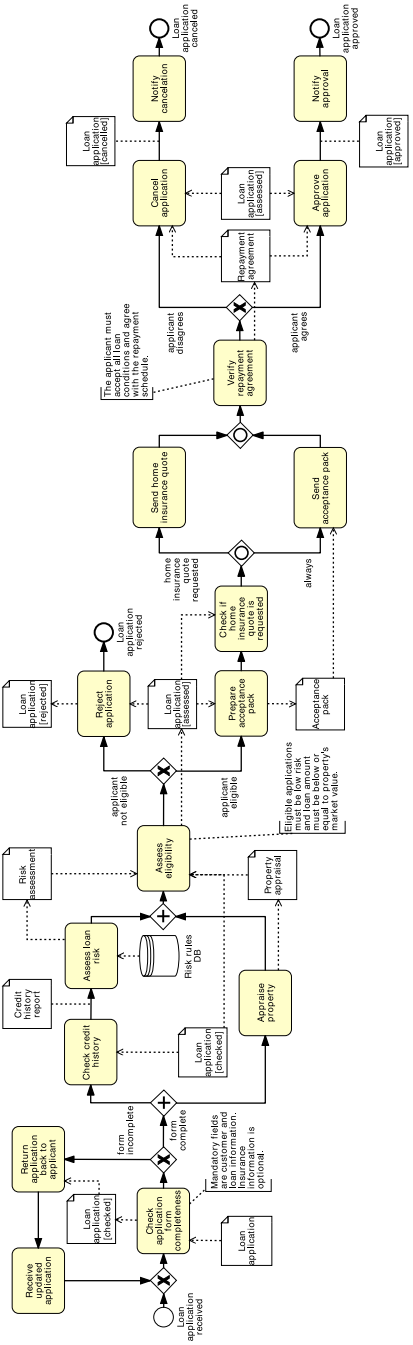
Solution 3.3



Solution 3.4

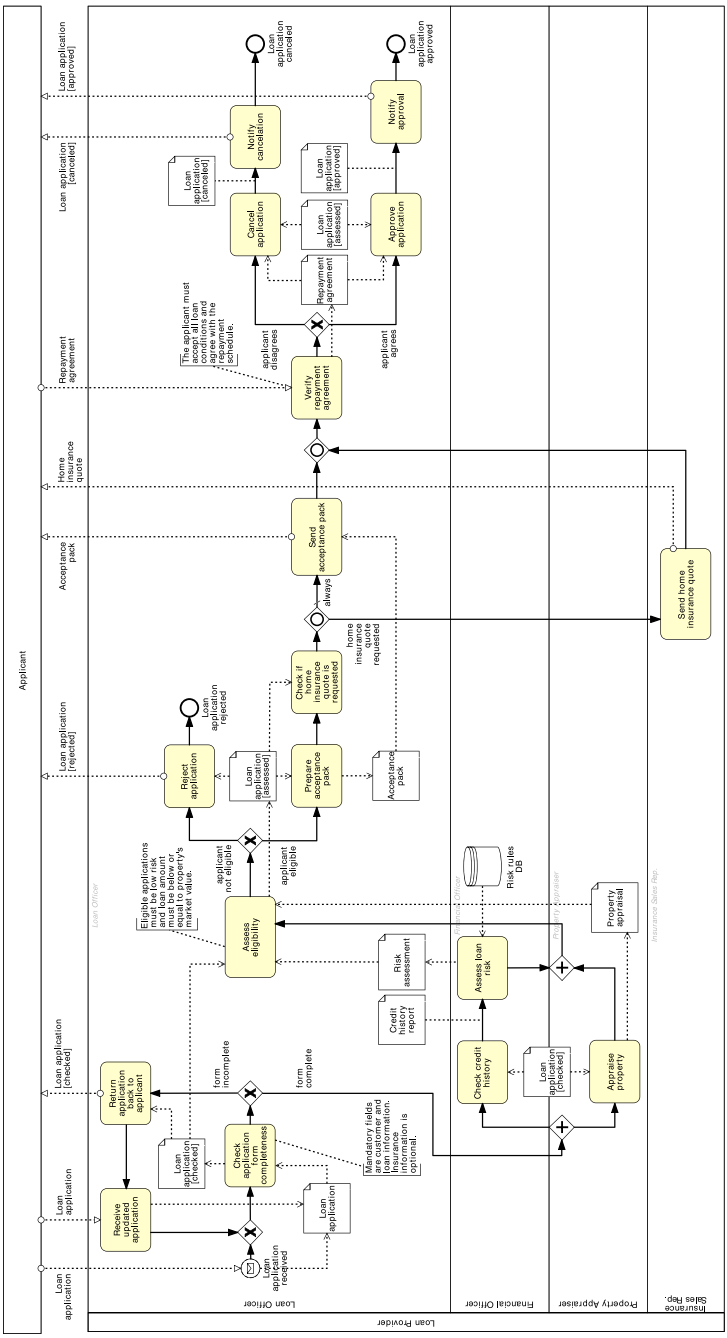


Solution 3.5


Loan application received
Receive updated application
Return application to back to applicant
Check application form completeness
Loan application [checked]
Loan application
Mandatory fields are customer and loan information. Insurance information is optional.
form incomplete
form complete
Check credit history
Credit history report
Loan application [checked]
Assess loan risk
Risk rules DB
Appraise property
Risk assessment
Assess eligibility
Property appraisal
Eligible applications must be low risk and the amount must be below or equal to property's market value.
applicant not eligible
applicant eligible
Reject application
Loan application [rejected]
Loan application [assessed]
Loan application rejected
Prepare acceptance pack
Acceptance pack
Check if home insurance quote is requested
home insurance quote requested
always
Send home insurance quote
Send acceptance pack
Verify repayment agreement
The applicant must accept all loan conditions and agree with the repayment schedule.
applicant disagrees
applicant agrees
Repayment agreement
Cancel application
Loan application [cancelled]
Loan application [assessed]
Approve application
Loan application [approved]
Notify cancelation
Notify approval
Loan application canceled
Loan application approved

Solution 3.6 See the Loan Provider pool in the model of Solution 3.7.

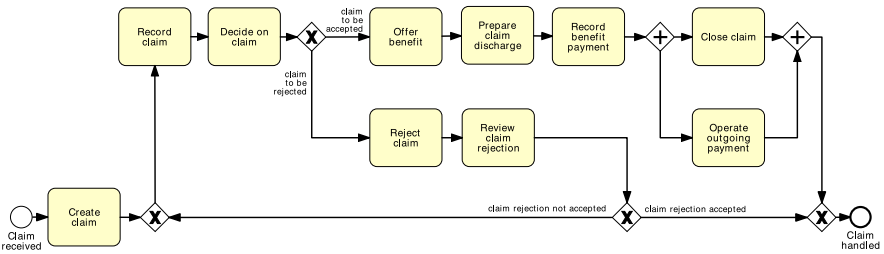
Solution 3.7



3.7 Further Exercises

Exercise 3.8 What types of splits and joins can we model in a process? Make an example for each of them using the security check at an airport as a scenario.

Exercise 3.9 Describe the following process model.



Exercise 3.10 Model the following business process for handling downpayments.

The process for handling downpayments starts when a request for payment has been approved. It involves entering the downpayment request into the system, the automatic subsequent payment, emission of the direct invoice and the clearance of the vendor line items. The clearing of the vendor line items can result in a debit or credit balance. In case of debit balance, the arrears are processed, otherwise the remaining balance is paid.

Exercise 3.11 Model the following business process for assessing credit risks.

When a new credit request is received, the risk is assessed. If the risk is above a threshold, an advanced risk assessment needs to be carried out, *otherwise* a simple risk assessment will suffice. Once the assessment has been completed, the customer is notified with the result of the assessment and *meantime* the disbursement is organized. For simplicity, assume that the result of an assessment is always positive.

Exercise 3.12 Model the following fragment of a business process for insurance claims.

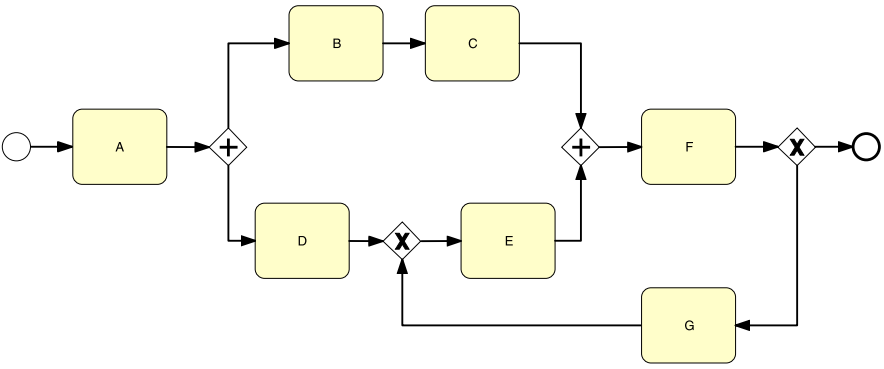
After a claim is registered, it is examined by a claims officer who then writes a settlement recommendation. This recommendation is then checked by a senior claims officer who may mark the claim as “OK” or “Not OK”. If the claim is marked as “Not OK”, it is sent back to the claims officer and the recommendation is repeated. If the claim is “OK”, the claim handling process proceeds.

Exercise 3.13 Model the control flow of the following business process for damage compensation.

If a tenant is evicted because of damages to the premises, a process needs to be started by the tribunal in order to hold a hearing to assess the amount of compensation the tenant owes the owner of the premises. This process starts when a cashier of the tribunal receives a request for compensation from the owner. The cashier then retrieves the file for those particular premises and checks that both the request is acceptable for filing, and compliant with the

description of the premises on file. Setting a hearing date incurs fees to the owner. It may be that the owner has already paid the fees with the request, in which case the cashier allocates a hearing date and the process completes. It may be that additional fees are required, but the owner has already paid also those fees. In this case the cashier generates a receipt for the additional fees and proceeds with allocating the hearing date. Finally, if the owner has not paid the required fees, the cashier produces a fees notice and waits for the owner to pay the fees before reassessing the document compliance.

Exercise 3.14 Can the process model below execute correctly? If not, how can it be fixed without affecting the cycle, i.e. such that "F", "G", and "E" all remain in the cycle?



Exercise 3.15 Write a BPMN model for the process described in Exercise 1.1. Make sure to include artifacts and annotations where appropriate.

Exercise 3.16 Extend the model of Exercise 3.13 by adding the artifacts that are manipulated.

Exercise 3.17 Extend the model of Exercise 3.16 by adding the involved resources. Is there any non-human resource?

Exercise 3.18 Model the following business process. Use gateways and data objects where needed.

In a court each morning the files that have yet to be processed are checked to make sure they are in order for the court hearing that day. If some files are missing a search is initiated, otherwise the files can be physically tracked to the intended location. Once all the files are ready, these are handed to the Associate; meantime the judge's lawlist is distributed to the relevant people. Afterwards, the directions hearings are conducted.

Exercise 3.19 Model the following business process. Use pools/lanes where needed.

The motor claim handling process starts when a customer submits a claim with the relevant documentation. The notification department at the car insurer checks the documents upon

completeness and registers the claim. Next, the Handling department picks up the claim and checks the insurance. Then, an assessment is performed. If the assessment is positive, a Garage is phoned to authorize the repairs and the payment is scheduled (in this order). Otherwise, the claim is rejected. In any case (whether the outcome is positive or negative), a letter is sent to the customer and the process is considered to be complete.

Exercise 3.20 Model the following business process. Use pools/lanes where needed.

When a claim is received, a claims officer first checks if the claimant is insured. If not, the claimant is informed that the claim must be rejected by sending an automatic notification via an SAP system. Otherwise, a senior claims officer evaluates the severity of the claim. Based on the outcome (simple or complex claims), the relevant forms are sent to the claimant, again using the SAP system. Once the forms are returned, they are checked for completeness by the claims officer. If the forms provide all relevant details, the claim is registered in the claims management system, and the process ends. Otherwise, the claimant is informed to update the forms via the SAP system. Upon reception of the updated forms, they are checked again by the claims officer to see if the details have been provided, and so on.

3.8 Further Reading

In this chapter we presented the basics of process modeling through the BPMN language. Other mainstream languages that can be used to model business processes are UML Activity Diagrams (UML ADs), Event-driven Process Chains (EPCs) and Web Services Business Process Execution Language (WS-BPEL). UML ADs are another OMG standard [60]. They are mainly employed in software engineering where they can be used to describe software behavior and linked to other UML diagram types, e.g. class diagrams, to generate software code. UML ADs offer a subset of the modeling elements present in BPMN. For example, constructs like the OR-join are not supported. A good overview of this language and its application to business process modeling is provided in [16].

EPCs were initially developed for the design of the SAP R/3 reference process model [9]. They obtained a widespread adoption by various organizations when they became the core modeling language of the ARIS toolset [12, 82]. Later, they were used by other vendors for the design of SAP-independent reference models such as ITIL and SCOR. The EPC language includes modeling elements corresponding to BPMN activities, AND, XOR and OR gateways, untyped events and data objects. An introduction to EPCs is provided in [50].

WS-BPEL (BPEL for short) version 2.0 [3] is a standard of the Organization for the Advancement of Structured Information Standards (OASIS). A good overview of BPEL is provided in [65]. BPEL is a language for process execution which relies on Web service technology to achieve inter-process communication. A mapping from BPMN to BPEL constructs is available in the BPMN specification [61]. However, this mapping is not complete since BPEL offers a restricted set of constructs compared to BPMN, and is essentially a *block-oriented* language, while BPMN is *graph-oriented*. BPEL is structured in blocks which need to be properly nested and

cannot overlap. A block is made up of a single entry node and a single exit node which matches the type of the entry node and collects all the outgoing branches from the entry node. For example, if the entry node is an AND-split, the exit node must be an AND-join. Moreover, BPEL does not feature a standard notation, since this was deemed to be out of scope by OASIS, though various vendors provide proprietary notations for this language. While BPMN 1.2 aimed to be the conceptual counterpart of BPEL, and mappings were thus available to move from the former to the latter language, BPMN 2.0 can also be used to specify executable processes (see Chap. 9). Thus BPMN 2.0 aims to replace BPEL in this respect.

Other process modeling languages originate from research efforts. Two of them are Workflow nets and Yet Another Workflow Language (YAWL). Workflow nets are an extension of Petri nets to model business processes. Their syntax is purposefully simple and revolves around two elements: places and transitions. The former roughly correspond to BPMN events, while the latter to BPMN activities. A good presentation of Workflow nets is provided in [95].

YAWL is a successor of Workflow nets in that it adds specific constructs to capture the OR-join behavior, multi-instance activities, sub-processes and cancellation regions. YAWL retains the simplicity and intuitiveness of Workflow nets, though it provides a much more expressive language. YAWL and its supporting environment are presented in detail in [92].

A comparison of the above languages in terms of their expressiveness along the control-flow, data and resource perspectives can be found in the Workflow Patterns Initiative website [108]. Over time this initiative has collected a repository of workflow patterns, i.e. recurring process behavior as it has been observed from a thorough analysis of various process modeling languages and supporting tools. Various languages and tools have been compared based on their support for such patterns.