Chapter 10
Process Implementation with Executable Models

You don't make progress by standing on the sidelines, whimpering and complaining. You make progress by implementing ideas.
Shirley Chisholm (1924–2005)

In the previous chapters, we have learned how to create *conceptual process models* and use them for documentation and analysis purposes. Because of their purpose, these models are intentionally abstract in nature, i.e., they do not provide technical implementation details. This means that conceptual process models must be systematically reworked into *executable process models* to be interpreted and automatically executed by a software system, such as a BPMS.

In this chapter, we propose a five-step method to incrementally transform a conceptual process model into an executable one, using the BPMN language. As part of this method, we also show how to make use of two other standards complementary to BPMN: the Case Management Model and Notation (CMMN) and the Decision Model and Notation (DMN). The steps are:

1. Identify the automation boundaries,
2. Review manual tasks,
3. Complete the process model,
4. Bring the process model to an adequate level of granularity, and
5. Specify execution properties.

Through these steps, the conceptual model will incrementally become less abstract and more IT-oriented. These steps should only be carried out on a process model that is syntactically correct. For example, if the model contains behavioral errors like deadlocks, then the BPMS may get stuck while executing an instance of this process model. This may have a negative impact on the operations of the organization (e.g., slowdowns or impediments in the fulfillment of purchase orders). We have already discussed verification in Section 5.4.1. In the following, we assume that the process model is sound.

M. Dumas et al., *Fundamentals of Business Process Management*,
https://doi.org/10.1007/978-3-662-56509-4_10

10.1 Identify the Automation Boundaries

A conceptual process model does not typically describe how each process task should be implemented. Depending on its nature, a task may not easily be implemented automatically or it may not be possible to implement it at all via a BPMS. Accordingly, the principle driving this first step is that *not all processes can be automated*. Based on this principle, we start by identifying which parts of our process can be coordinated by the BPMS and which parts cannot. To do so, we distinguish three types of tasks, in line with the BPMN language: *automated*, *manual*, and *user* tasks. Automated tasks are performed by the BPMS itself or by an external service. Manual tasks are performed by process participants without the aid of any software. User tasks sit between automated and manual tasks. A user task is a task that is performed by a participant with the assistance of the worklist handler of the BPMS or an external task list manager.

The differentiation between automated, manual, and user tasks is important: Automated and user tasks can easily be coordinated by a BPMS, while manual tasks cannot. Therefore, we first have to identify the type of each task. In the next step, we review the manual tasks and assess whether we can find a way to hook them up to the BPMS. If this is not possible, we will have to consider whether or not it is convenient to automate the rest of the process without these manual tasks.

Let us consider again the order-to-cash process model that we created in Chapter 3. It is shown in Figure 10.1 for convenience (for the moment, please discard the markers). Let us assume that we obtain this model from a process analyst. Our job is to automate it from the seller's viewpoint. As such, we need to focus on the process in the seller pool and discard the rest. The first task, "Check stock availability", belongs to the ERP lane. This means that it was already identified as an automated task at the conceptual level. ERP systems provide modules to manage inventories, which automatically check the stock levels of a product against a warehouse database. This task is highly repetitive since it is performed for each purchase order received. Performing it manually would be inefficient, because it would keep a process participant busy with a trivial yet time-consuming task. Similar observations can be made for "Check raw materials availability", which is also an automated task. Another example is the "Manufacture product" task. This is performed by the manufacturing plant, which exposes its functionality via an IT service interface. From the perspective of a BPMS, it is also an automated task.

Continuing with our example, there are other tasks, such as "Request raw materials from Supplier 1(2)" and "Get shipping address", that are devoted to sending and receiving messages. These are examples of automated tasks, too. They can be implemented via an automatic email exchange or a Web service invocation. Note that BPMSs typically provide these capabilities. So far, these tasks are not explicitly modeled inside a system lane. Recall that we are looking at a conceptual process model, where it may not be relevant to model all existing systems (in this case an email service or a Web service) via lanes.

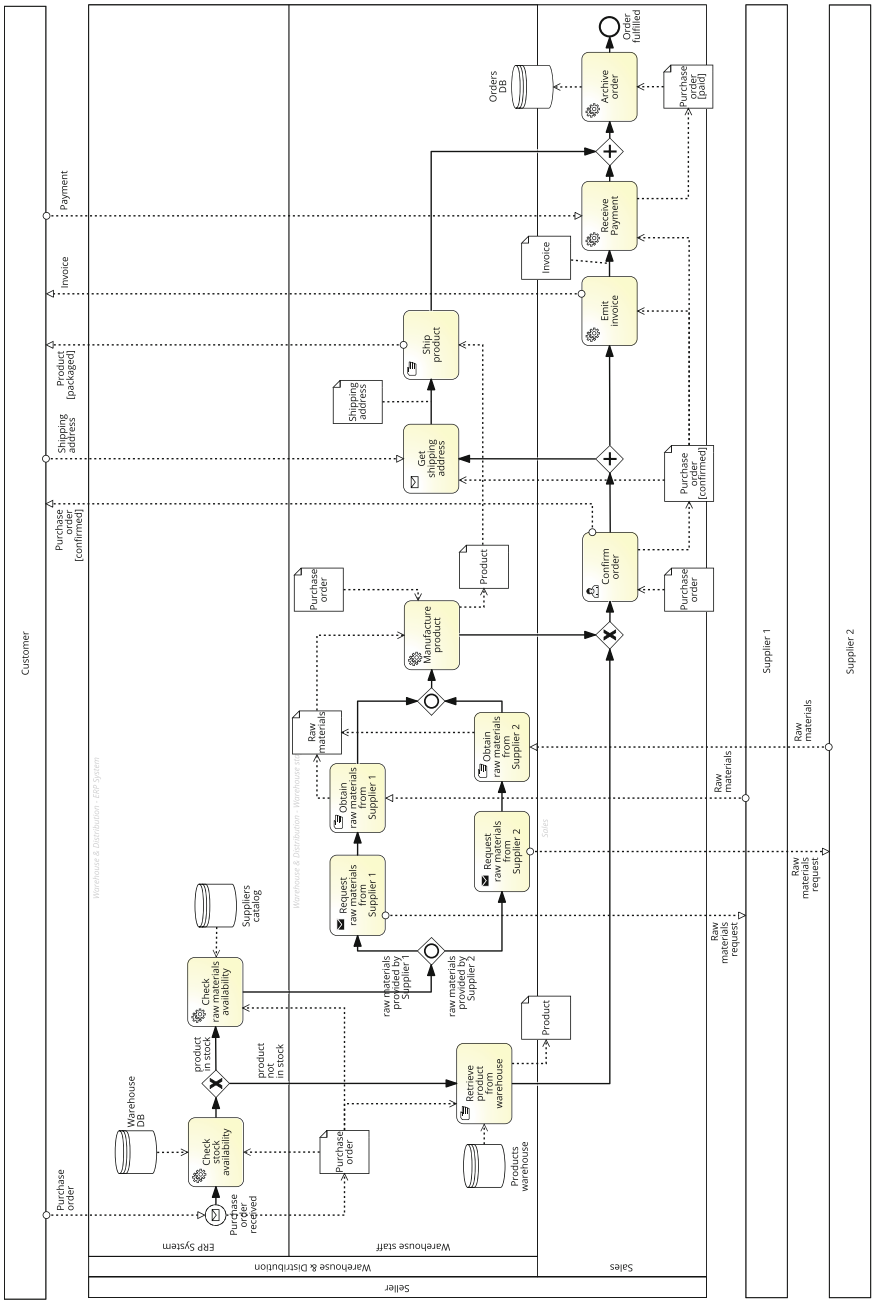

Fig. 10.1 The order-to-cash model that we want to automate

Other tasks like "Retrieve product from warehouse", "Obtain raw materials from Supplier 1(2)", and "Ship product" are manual. For example, "Retrieve product from warehouse" requires a warehouse worker to physically pick up the product from the shelf for shipping. In the presence of a manual task we have two options: (i) we isolate the task and focus on the automation of the process before and after it, or (ii) we find a way for the BPMS to be notified when the manual task has started or completed. We will get back to this point in the second step. For now, all we need to do is identify these manual tasks.

"Confirm order" is an example of a user task: it requires somebody in sales (e.g., an order clerk) to verify the purchase order and then confirm that the order is correct. User tasks are typically managed by the worklist handler of the BPMS. In our example, an electronic form of the purchase order will be rendered on screen for the order clerk, who will verify that the order is in good state, confirm the order, and submit the form back to the execution engine.

The distinction between automated, manual, and user tasks is captured in BPMN via specific markers on the top-left corner of the task box. Manual tasks are marked with a hand, while user tasks are marked with a user icon. Automated tasks are further classified into the following subtypes in BPMN:

- *Script* (script marker), if the task executes some code (the script) internally to the BPMS. This task can be used when the functionality is simple and does not require access to an external application, e.g., opening a file or selecting the best quote from a number of suppliers.
- *Service* (gears marker), if the task is executed by an external application, which exposes its functionality via a service interface, e.g., "Check stock availability" in our example.
- *Business rule* (table marker), if the task triggers a business rule to be executed by a rules engine external to the BPMS, e.g., the rule for approving a loan.
- *Send* (filled envelope marker), if the task sends a message to an external service, e.g., "Request raw materials from Supplier 1".
- *Receive* (empty envelope marker), if the task waits for a message from an external service, e.g., "Get shipping address".

These markers apply to tasks only. They cannot be used on sub-processes, since a sub-process may contain tasks of different types. The relevant markers for our example are shown in Figure 10.1.

Exercise 10.1 Assume you have to automate the loan assessment process model of Solution 3.8 (page 111) for the loan provider. Start by classifying the tasks of this process into manual, automated, and user tasks. Then, represent them with appropriate task markers.

10.2 Review Manual Tasks

Once we have identified the type of each task, in the second step of our method we need to check whether we can link the manual tasks with the BPMS. The principle driving this step is: *if the task cannot be seen by the BPMS, it does not exist*. So, we either find a way to support manual tasks via technology or, alternatively, we need to isolate these tasks and automate the rest of the process. There are two ways of linking a manual task to a BPMS: we implement it either via a user task or via an automated task.

Implement as User Task: If the participant involved in the manual task can notify the BPMS of the task completion using the worklist handler of the BPMS, then the manual task can be turned into a user task. For example, the warehouse worker performing task "Retrieve product from warehouse" could check out a work item from the worklist to indicate that the task is being worked on, manually retrieve the product from the shelf, and then check in the work item back into the BPMS engine. Alternatively, check-out and check-in can be combined in a single step, by which the worker notifies the worklist handler of the completion of the task.

Implement as Automated Task: In some cases, a process participant may use technology that is integrated with the BPMS to notify the engine of a work item completion. For example, the warehouse worker could use a device such as a barcode scanner to scan the barcode of the raw materials that are picked up. If the device is connected to the BPMS, scanning the barcode will automatically signal the completion of task "Obtain raw materials from Supplier 1(2)". In this case, the manual task can be implemented as a receive task, which will be awaiting the notification from the scanner, or as a user task handled by a worklist handler, which in turn is connected to the scanner. If we use a receive task, the BPMS will only be aware of the work item's completion: informing the warehouse worker that a new work item is available will be outside the scope of the BPMS. If we use a user task, the worker will be notified of the new work item by the BPMS and will use the scanner to signal the work item's completion to the BPMS engine. Similar considerations hold for task "Ship order". Since each manual task of our example can be linked with a BPMS, this process can be entirely automated.

Exercise 10.2 Consider the loan assessment model that you analyzed in Exercise 10.1. Review the manual tasks of this model in order to link them to a BPMS.

There are cases in which it is not convenient to link manual tasks to a BPMS.

Example 10.1 Let us consider the university admission process described in Exercise 1.1 (see page 5), with the improvements discussed in Solution 1.5 (page 29). The process can be automated until the point where the application is batched for the admission committee (shown in Figure 10.2a). Once all the applications have been batched, the committee will meet and examine all of them at once. However, this part of the process (shown in Figure 10.2b) is outside the scope of a BPMS.

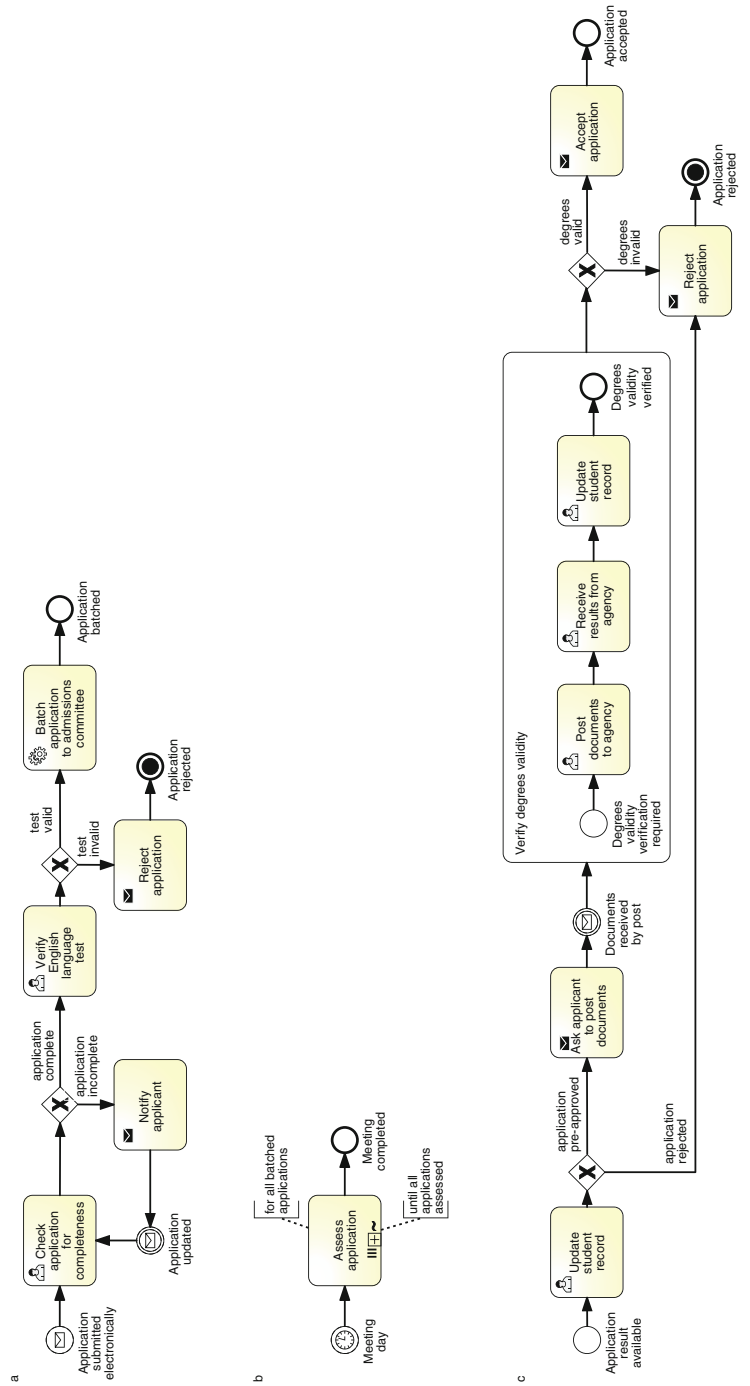

Fig. 10.2 Admission process: the initial (**a**) and final (**c**) assessments can be automated in a BPMS; the assessment by the committee (**b**) is a manual process outside the scope of the BPMS

The tasks required for assessing applications cannot be automated, because they involve various human participants who interact on an ad hoc basis. It would not be convenient to synchronize all these tasks with the BPMS. Eventually, the committee will decide on a list of accepted candidates and transfer it to the admissions office. Then, a clerk at the admissions office will update the various student records, at which time the rest of the process can proceed within the scope of the BPMS (shown in Figure 10.2c).

In this example, we cannot automate the whole process. So, we need to isolate the task "Assess application", an ad hoc task containing various manual tasks, and automate the process before and after this task. An option is to split the model into three fragments as shown in Figure 10.2 and only automate the first and the third fragment. Another option is to keep one model and simply remove the ad hoc task. Some BPMSs are tolerant to the presence of manual tasks and ad hoc tasks in executable models, and will discard them at deployment time (like comments in a programming language). If this is the case, we can keep these elements in.

Observe the use of the untyped event to start the third process model fragment in Figure 10.2. In BPMN, a process that starts with an untyped event indicates that instances of this process are explicitly started by a BPMS user, in our case a clerk at the admissions office. This process initiation is called *explicit instantiation*. *Implicit instantiation* refers to the situation where process instances are triggered automatically by the event type indicated in the start event, e.g., an incoming message or a timer. □

Exercise 10.3 Consider the final part of the prescription fulfillment process described in Exercise 1.6 (page 30):

> Once the prescription passes the insurance check, it is assigned to a technician who collects the drugs from the shelves and puts them in a bag with the prescription stapled to it. After the technician has filled a given prescription, the bag is passed to the pharmacist who double-checks that the prescription has been filled correctly. After this quality check, the pharmacist seals the bag and puts it in the pick-up area. When a customer arrives to pick up a prescription, a technician retrieves thise prescription and asks the customer for the co-payment or for the full payment in case the drugs in the prescription are not covered by the customer's insurance policy.

One way of modeling this fragment is by defining the following tasks: "Check insurance", "Collect drugs from shelves", "Check quality", "Collect payment" (triggered by the arrival of the customer), and finally "Retrieve prescription bag". Assume the pharmacy system automates the prescription fulfillment process. Identify the type of each task and if there are any manual tasks, specify how these can be linked to the pharmacy system.

There are other modeling elements besides manual tasks that are relevant at a conceptual level but cannot be interpreted by a BPMS. These are physical data objects and data stores, messages bearing physical objects, and text annotations. Pools and lanes are only meaningful at the conceptual level, too. In fact, as we have seen, pools and lanes are often used to capture coarse-grained resource assignments, e.g., task "Confirm order" is done within the sales department. When it comes

to execution, we need to define resource assignments for each task and capturing this information via dedicated lanes (potentially one for each task) would make the model too cluttered. Electronic data stores are also not directly interpreted by a BPMS, since the BPMS assumes the existence of dedicated services that can access these data stores, e.g., an inventory information service that can access the warehouse database. So, the BPMS will interface with these services rather than directly with the data stores. Also, the state of a data object indicated in the object's label, e.g., "Purchase order [confirmed]", cannot be interpreted as such by a BPMS. Later, we will show how to explicitly represent object states so that they can be interpreted by a BPMS.

Some BPMSs tolerate the presence of non-executable elements in their modeling tool. If this is the case, it is good practice to leave these elements in. Especially pools, lanes, message flows bearing electronic objects, electronic data stores, and annotations will guide us in the specification of some execution properties. For example, the Sales lane in the order-to-cash model indicates that the participant who is to be assigned the "Confirm order" task has to be from the sales department. Other BPMSs do not support these elements, so it is not possible to represent them in the process model.

Exercise 10.4 Consider the loan assessment model that you obtained in Exercise 10.2 (page 375). Identify the modeling elements that cannot be interpreted by a BPMS.

10.3 Complete the Process Model

Once we have established the automation boundaries of the process and reviewed the manual tasks, we need to check that our process model is *complete*. Two principles underlie this step: (i) *exceptions are the rule* and (ii) *no data implies no decisions and no task handoff*. Often, conceptual process models neglect certain information; because modelers deem it as irrelevant for the specific modeling purpose, they assume it is common knowledge, or they are simply not aware of it. Depending on the application scenario, it may be fine to neglect this information in a conceptual model. However, information that is not relevant in a conceptual model may be highly relevant for a process model to be executed.

A typical example is when the process model focuses on the "sunny-day" scenario and neglects all negative situations that may arise during the execution of the process, working under the assumption that everything will work well. As we saw in Chapter 4, there are various exceptions that can occur in the order-to-cash process. For example, this process may be aborted if the materials required to manufacture the product are not available at the suppliers or if the customer cancels the order. So, based on the first principle above, we need to make sure that all exceptions are handled using appropriate exception handlers. For example, if the order cancelation is received after the product has been shipped or after the payment

has been received, then we also have to compensate for these tasks by returning the product and reimbursing the customer. Another exception that is commonly neglected is the situation when a task cannot complete correctly. What happens if the customer's address is never received? Or if the ERP module for checking the stock availability does not respond? We cannot assume that the other party will always respond or that a system will always be functional. Similarly, we cannot assume that tasks always lead to a positive outcome. For example, an order may not always be confirmed.

You may be surprised about how rarely exceptions are captured in a conceptual process model in practice. Thus, in the majority of cases, such a model will require to be completed with these aspects before being executed.

Looking at the second principle, in this step we also need to specify all *electronic data objects* that are required as input and output by the tasks of our process. For instance, in Figure 10.1 (see page 373) there is no input data object to task "Request raw materials from Supplier 1(2)", though this task does need the list of raw materials to be ordered. Another example is task "Check stock availability". This task uses the purchase order as input (to obtain the code of the product to be looked up in the Warehouse DB), but does not produce any output data to store the results of the search. However, without this information, the subsequent XOR-split cannot determine which branch to take (we can now better grasp why this is called a *data-based XOR-split*). If you have not noticed the absence of these data objects so far, it is probably because you assumed their existence. This is fine in a conceptual model where only aspects relevant to the specific modeling purpose are documented, but not in an executable model, where a software engine has to run the model. So, make sure each task has the required input and output electronic data objects. The point is that every data object needed by the BPMS engine to pass control between tasks and to take decisions must be modeled.

The completed order-to-cash example, including exception handlers and data objects that are relevant for execution, is shown in Figure 10.3.[1]

Exercise 10.5 Take the loan assessment model that you obtained in Exercise 10.1 (page 374) after incorporating the revisions from Exercise 10.2 (page 375). Complete this model with control-flow and data-flow aspects relevant for automation. For the sake of simplicity, you may disregard the modeling elements that are not interpretable by a BPMS.

[1]The content of the sub-processes and some of the elements that cannot be interpreted by a BPMS have been omitted for simplicity.

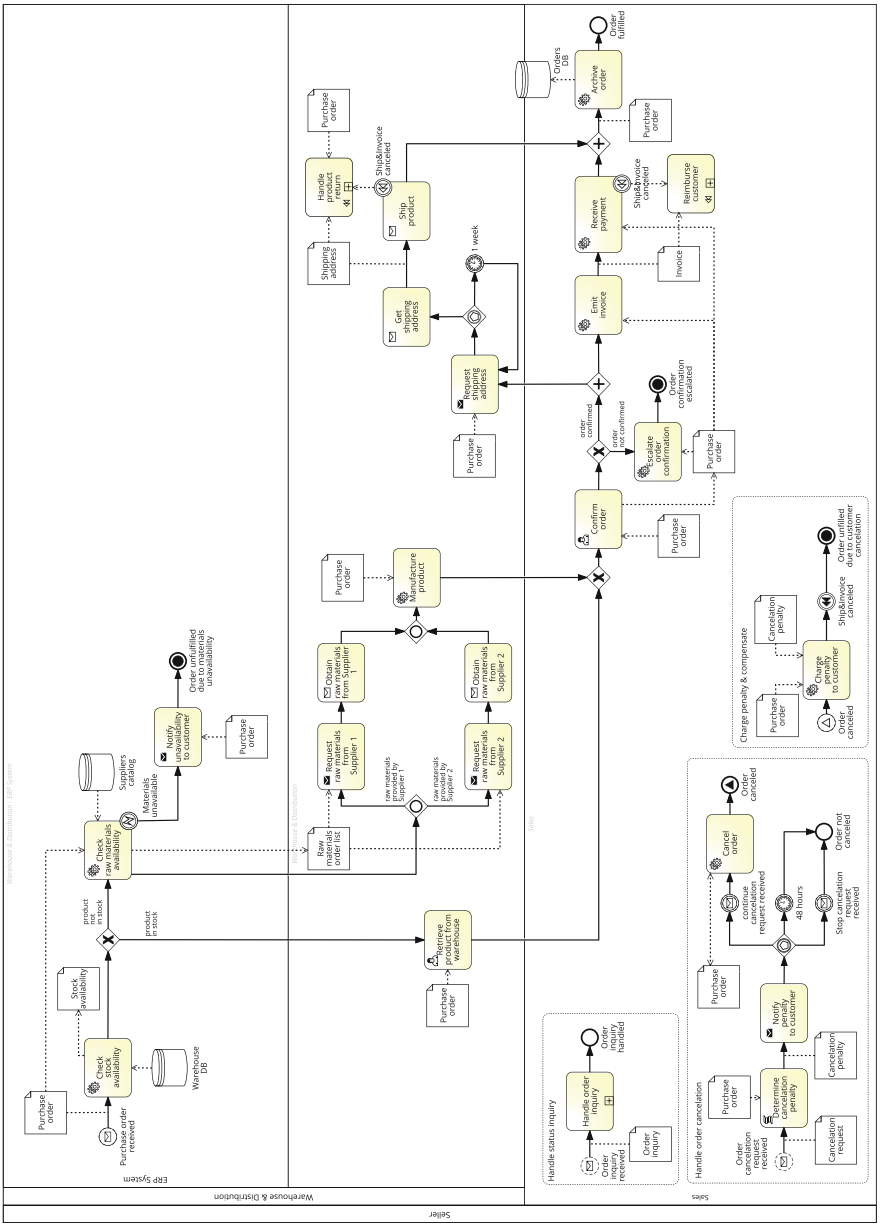

Fig. 10.3 The order-to-cash model of Figure 10.1, completed with control-flow and data-flow aspects relevant for automation

10.4 Bring the Process Model to an Adequate Granularity Level

Tasks in a conceptual model may not be at the right level of granularity for implementation. They may be either too abstract, in which case we need to decompose them, or too detailed, in which case they should be aggregated. For example, two consecutive tasks assigned to the same resource are candidates for *aggregation*. In a similar way, if a task requires more than one resource to be performed, then it is too *coarse-grained*. We should then decompose it into finer-grained tasks such that these can be assigned to different resources. The principle driving these examples is that *a BPMS adds value if it coordinates handoffs of work between resources*. Indeed, we should keep in mind that a BPMS is intended to coordinate and manage handoffs of work between multiple resources (human or non-human). If this were not the case, the BPMS would not add value between tasks.

A special case are ad hoc sub-processes, which are difficult to define in terms of the order of tasks within the sub-process. These sub-processes may be implemented using the Case Management Model and Notation (CMMN), a language complementary to BPMN.

10.4.1 Task Decomposition

If a task requires more than one resource to be performed, we should *decompose* it into more fine-grained tasks, such that these can be assigned to different resources. For example, a task "Enter and approve money transfer" is likely to be performed by two different participants even if they have the same role. In this case, we typically want to enforce a *separation of duties*: first a financial officer enters the order, then a different financial officer approves of it.

Exercise 10.6 Figure 10.4 shows the model for the sales process of a business-to-business (B2B) service provider. The process starts when an application is received

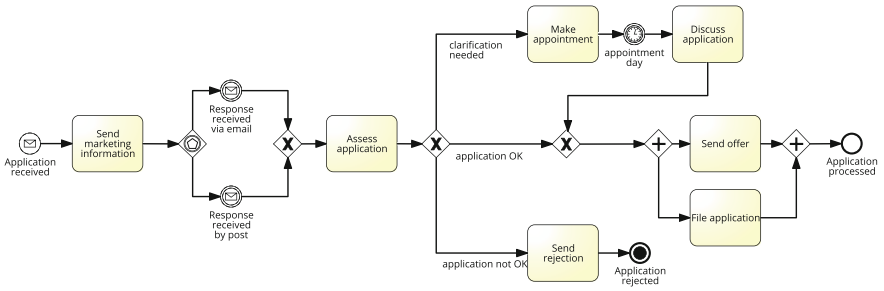

Fig. 10.4 The sales process of a B2B service provider

from a potential client. The client is then sent information about the available services. A response by the client is awaited to arrive via either email or postal mail. When the response is received, the next action is decided upon. Either an appointment can be made with the client to discuss the service options in person or the application is accepted. It could also be rejected right away. If the application is accepted, an offer is sent to the client and at the same time the application is filed. If it is rejected, the client is sent a thank-you note. If an appointment has to be made, this is done and, at the time of the appointment, the application is discussed with the client. Then, the process continues as if the application had been accepted right away.

1. Identify the type of each task and find ways of linking the manual tasks to a BPMS.
2. Remove elements that cannot be interpreted by a BPMS.
3. Complete the model by adding the control-flow and data aspects required for execution.
4. Bring the resulting model to a granularity level that is adequate for execution.

Acknowledgement: This exercise is adapted from a similar exercise developed by Remco Dijkman, Eindhoven University of Technology.

10.4.2 Decomposition of Ad Hoc Sub-Processes with CMMN

Unordered tasks, such as those within an ad hoc sub-process (see Section 4.1.2), are often difficult to implement using a BPMS based on the BPMN language. Take, for example, the model in Figure 4.6 (page 122). This model, which captures the order-to-cash process from the perspective of the customer, has three tasks within an ad hoc sub-process: “Check order status”, “Update details”, and “Cancel order”. These tasks can hardly be coordinated by a BPMS based on BPMN, since there is no strict order for their execution. Also, each of these may potentially be repeated multiple times. As a rule of thumb, a (sub-)process whose tasks are performed in an ad hoc manner, without any predictable order, is not suitable for automation via a BPMN-based BPMS. In this case, a case management system or an ad hoc workflow system is more appropriate.

Several BPMSs, such as Camunda, do not only support the BPMN language, but also the Case Management Model and Notation (CMMN) language. This is another standard by OMG, available in version 1.1. BPMN and CMMN differ in the way they describe processes. BPMN builds on the explicit specification of those execution sequences that are allowed. Thereby, it forbids any other order of processing. CMMN defines which tasks have to be executed, although potentially restricted by certain conditions. In this way, it remains underspecified how the case is to be handled, except for those tasks that are bound to conditions. Therefore, CMMN is often considered as a more flexible way to describe *what* has to be achieved in a process instead of *how* to achieve it. Often, CMMN will be used as a

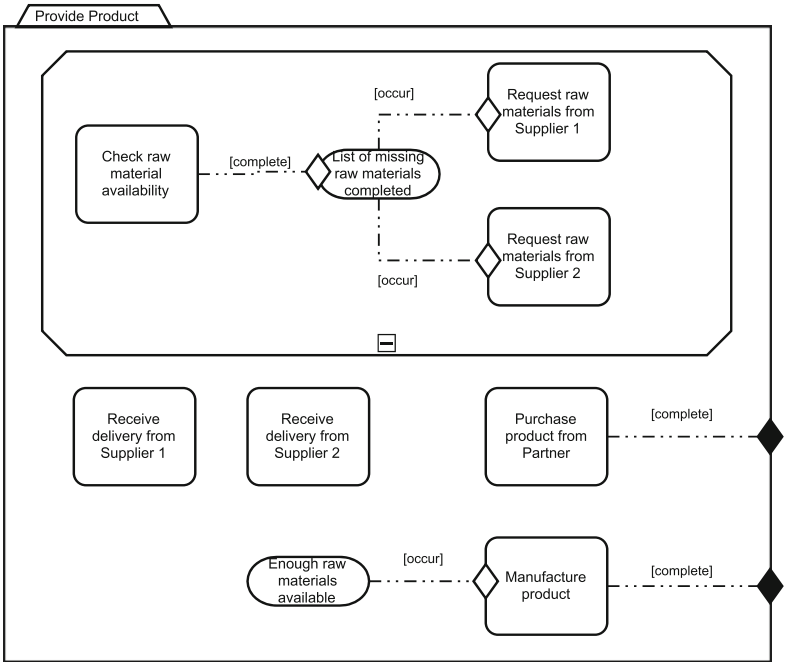

Fig. 10.5 Excerpt of an order-to-cash process model (from out-of-stock product to product provided) captured in CMMN

type of sub-process in a BPMN model, but also vice versa: tasks in a CMMN model can have BPMN sub-processes.

CMMN offers a set of elements for describing processes. This set includes tasks and events with the same graphical symbols as we know them from BPMN. Figure 10.5 shows a simple example including the most important elements. The model was created by a process analyst who felt that the the order-to-cash process within their company was difficult to model in BPMN from the point where the product is out-of-stock to the provisioning of the product. In a CMMN model, everything is organized in a *case*, depicted as a large box with a tab to make it look like a folder. For example, consider the "Provide product" case in our example. A case contains a stage, tasks, milestones, sentries, and connections. The *stage* is a large octagon, which can be used to group other elements. Here, it is used to describe how raw materials are checked for availability as a *task*, how this contributes to arriving at the *milestone* of compiling a list of missing raw materials, and how this milestone must occur before raw materials can be requested from Supplier 1 or 2. This condition is expressed using a *sentry*, which is a small, diamond-shaped symbol on the entry-side of elements. The reason why the process analyst decided to use CMMN is the fact that those requests do not exactly match deliveries. First, the suppliers deliver standard materials on a regular basis, even without an explicit request. Second, it often happens that one request is served by several separate

deliveries. In other to express the fact that deliveries can happen at any time, the two "Receive delivery" tasks do not have any sentries. There is another milestone in the lower part of the model, which is called "Enough raw materials available". If this is reached, the product can be manufactured. This leads to a successful completion of the case, as indicated by the black diamond element on the border of the case container. Our example model also includes another option to provide the product, i.e., by purchasing it from a partner.

10.4.3 Task Aggregation

Tasks on a conceptual level can also be too fine-grained. For example, a sequence of user tasks "Enter customer name", "Enter customer policy number", and "Enter damage details" should be aggregated into a single user task "Enter claim" if they are all supposed to be performed by the same claims handler. Otherwise, all the BPMS would do would be to interfere with the work of the claims handler. Accordingly, two or more consecutive tasks assigned to the same resource are candidates for *aggregation*.

There are some cases, though, where we may actually need to keep consecutive tasks separate, despite that they are performed by the same resource. For example, in Figure 10.2c we have three user tasks within the sub-process "Verify degrees validity": "Post documents to agency", "Receive results from agency", and "Update student record". While these may be performed by the same admin clerk, we want to keep track of when each task has been completed for the sake of monitoring the progress of the application and managing potential exceptions. For example, if the results are not received within a given timeframe, we can handle this delay by adding an exception handler to the "Receive results from agency" task.

Exercise 10.7 Are there tasks that can be aggregated in the model as obtained in Exercise 10.5 (page 379)?
Hint: candidate tasks for aggregation may not necessarily be consecutive due to a sub-optimal order of tasks in the conceptual model. In this case, you need to resequence the tasks first.

10.5 Specify Execution Properties

At the end of the fourth step, we obtain a *to-be-executed* process model, i.e., a process model that contains the right elements and is at the right level of granularity to be automated with a BPMS. However, this model is still technology-agnostic. That is to say, it is independent of the specific BPMS technology we will choose for automation. As such, software engineers may be supported by process analysts in the incremental transformation of a conceptual model into a to-be-executed model.

To make the model fully executable, we need to specify in the last step *how* each model element is effectively implemented by our BPMS of choice. For example, take the first service task of our revised order-to-cash example: "Check stock availability". Saying that this task requires the purchase order as input to contact the warehouse ERP system is not enough. We need to specify which specific service provided by the ERP system is to be used to check the stock levels, the location of its interface in the network, the format of its input object (the purchase order), and the format of its output object (the stock availability). These implementation details are called *execution properties*. They are required to obtain a fully-executable process model. More specifically, these properties are:

- Variables, messages, signals, errors, and their data types,
- Data mappings,
- Service details for service, send and receive tasks, and for message and signal events,
- Code snippets for script tasks,
- Participant assignment rules and user interface structure for user tasks,
- Task, event, and sequence flow expressions, and
- Other BPMS-specific properties.

The BPMN language provides the means to specify most of these properties. However, in practice, BPMS vendors often diverge from the standard way of specifying these properties and rather offer alternative, sometimes proprietary, mechanisms. This may be because of legacy reasons or to gain a competitive advantage. In the rest of this section, we will focus on how the above properties can be defined according to the standard BPMN specification and point out to some alternatives, where available.

Execution properties do not have a graphical representation in a BPMN model, but are stored in the BPMN interchange format. The BPMN interchange format is a textual representation of a BPMN model in XML format. It is intended to support the interchange of BPMN models between tools and also serves as input to a BPMN execution engine. BPMN modeling tools provide a visual interface to edit most of these non-graphical properties. So, most of the times you will not need to write XML directly. Still, you will need to at least understand standard Web technology and be familiar with the notion of Web service to be able to implement an executable process model. This section assumes that you have a basic knowledge of technologies such as XML and XML Schema (XSD). We provide pointers to further readings on these technologies at the end of this chapter.

Figure 10.6 shows the structure of the BPMN exchange format. It consists of a list of elements, where some are optional (those with a dashed border) and others are mandatory (those with solid borders). The process element is mandatory. It consists of data objects, events, tasks, and flows. The elements outside the process are reusable components needed by the various process elements, such as message definitions and service interfaces that are used by service, send, and receive tasks, and by message and signal events. With reference to this structure, let us now go through each of the execution properties listed above.

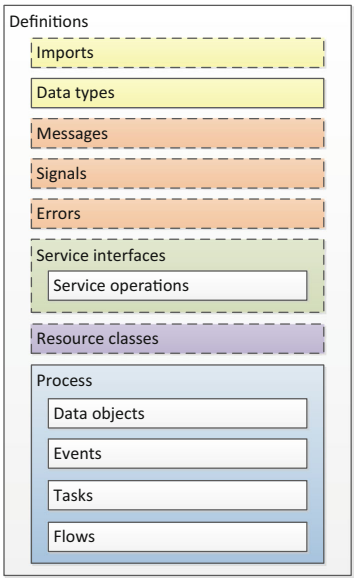

Fig. 10.6 Structure of the BPMN format

10.5.1 Variables, Messages, Signals, Errors, and Their Data Types

Process variables are managed by the BPMS engine to allow data exchange between process elements. Each electronic data object, e.g., the purchase order in the order-to-cash process, is represented by a process variable. Each of the variables we want to use in the process has to be explicitly defined by assigning a *data type* to it. In BPMN, the type of each variable is specified as an XSD type. However, some BPMSs may use different languages or proprietary definitions. With reference to XSD, the type of a variable can be *simple* or *complex*. The complex types can be either defined directly in the BPMN model or imported from an external document (see Figure 10.6). Simple types are strings, integers, doubles (numbers containing decimals), booleans, dates, times, etc.. These are already defined in the XSD specification. For example, the object "Stock availability" can be represented as a variable of type integer (capturing the number of available units of a product). Complex types are hierarchical compositions of other types. A complex type can be used to represent, for example, a business document, such as a purchase order or an invoice. Figure 10.7a shows the possible format of a purchase order, captured as a complex type called "purchaseOrderType". Figure 10.7b is the XML representation of a particular instance of this purchase order at runtime. From the type definition we can see that a purchase order contains a sequence of two elements:

- "order" to store the order information (order number, order date, status, currency, product code and quantity), and

a)

```
<complexType name="purchaseOrderType">
   <sequence>
      <element name="order">
         <complexType>
            <sequence>
               <element name="orderNumber" type="integer"/>
               <element name="orderDate" type="date"/>
               <element name="status" type="string"/>
               <element name="currency" type="string"/>
               <element name="productCode" type="string"/>
               <element name="quantity" type="integer"/>
            </sequence>
         </complexType>
      </element>
      <element name="customer">
         <complexType>
            <element name="name" type="string"/>
            <element name="surname" type="string"/>
            <element name="address">
               <complexType>
                  <sequence>
                     <element name="street" type="string"/>
                     <element name="city" type="string"/>
                     <element name="state" type="string"/>
                     <element name="postCode" type="string"/>
                     <element name="country" type="string"/>
                  </sequence>
               </complexType>
            </element>
            <element name="phone" type="string"/>
            <element name="fax" type="string"/>
         </complexType>
      </element>
   </sequence>
</complexType>
```

b)

```
<purchaseOrder>
   <order>
      <orderNumber>15664</orderNumber>
      <orderDate>2012-10-23</orderDate>
      <status>confirmed</status>
      <currency>EUR</currency>
      <productCode>345-EAR</productCode>
      <quantity>10</quantity>
   </order>
   <customer>
      <name>John</name>
      <surname>Brown</surname>
      <address>
         <street>8 George St</street>
         <city>Brisbane</city>
         <state>Queensland</state>
         <postCode>4000</postCode>
         <country>Australia</country>
      </address>
      <phone>+61 7 3240 0010</phone>
      <fax>+61 7 3221 0412</fax>
   </customer>
</purchaseOrder>
```

Fig. 10.7 The XSD describing the purchase order (**a**) and one of its instances (**b**)

- "customer" to store the customer information (name, surname, address, phone and fax).

The data fields order, customer, and address are complex types that can contain sub-elements. Also, observe the field "status" within order: It is used to capture the state of the purchase order, e.g., "confirmed".

Process variables are assigned a data type and defined to be used within the whole lifetime of a process instance. They are visible at the process level in which they are defined and in all the sub-processes within the process model. This means that a variable defined in a sub-process (i.e., a sub-process variable) is not visible in the parent process. The data objects that we discussed above, like purchase order, are typically defined as process variables.

Similarly to process variables, we also need to assign data types to each message, signal, and error used in the process model. For the *messages*, we can look at the existing message flows in the diagram and define one data type for each uniquely-labeled message flow. So, for example, if we have two message flows labeled

"purchase order", then they will obviously take the same type "purchaseOrderType". If message flows are not modeled, we can look at the send, receive, and service tasks, as well at the message events present in the model in order to understand what messages to define.

For *signals* and *errors* we have to look at the signal and error events that we have defined in the diagram. While for a signal the data type describes the content of the signal being broadcasted or listened to, for an error the data type defines what information is carried with the error. For example, if it is a system error, then we can use this to specify the error message returned by the system. In addition, we need to assign an *error code* to each error. This code uniquely identifies an error, so that a catching error event can be related to a throwing error event.

Besides the above data elements, we need to define the *internal variables* of each task, which are called *data inputs* and *data outputs* in BPMN. Data inputs and outputs act as interfaces between a task and the task's input and output data objects. They also need to refer to an XSD type that defines their structure, but, differently from process variables, they are only visible within the task (or sub-process) in which they are defined. Data inputs capture data that is required by the task to be executed. Data outputs capture data that is produced by the task upon completion. Data inputs are populated with the content of input data objects while data outputs are used to populate the content of output data objects. For example, we need a data input for task "Check stock availability" to store the content of the purchase order. Thus, the type of this data input must match that of the input object, i.e., "purchaseOrderType". Similarly, the data output must be of type integer to store the number of items in stock, so that this information can be copied into the output object stock availability upon task completion.

Similarly to tasks, events that transmit or receive data, i.e., message, signal and error events, also have internal variables. Specifically, the catching version of these events has one data output only, which is used to store the content of the event being caught (e.g., an incoming message). The throwing version has one data input only, which is there to store the content of the event being thrown (e.g., an error). Thus, we also need to assign to these data inputs and outputs a type that has to match that of the message, signal, or error associated with the event. For example, the start catching message event "Purchase order received" in the order-to-cash example uses a data output to store the purchase order message once this has been received. So, this data output must match the type of the incoming message, which is precisely "purchaseOrderType". In turn, the output object must have the same type as the output data to contain the purchase order.

10.5.2 Data Mappings

In BPMN, data is manipulated and processed inside tasks and events. The *mapping* between data objects and task (event) data inputs or outputs is defined via the task *data associations*. Data associations can also be used to define complex data

assignments beyond one-to-one mappings. For example, consider task "Manufacture product" in our order-to-cash example. The service invoked by this task only requires the order product code and quantity to start the manufacturing of the product. We can use a data association to extract the product code and quantity from the input purchase order and populate a data input containing two sub-elements of type string and integer, respectively. In most cases, the BPMS will automatically create all the tedious data mappings between data objects and tasks. For the case above, for example, all we need to do is to select the sub-elements of the purchase order we want to use as input to "Manufacture product". The BPMS will then create the required data inputs and their mappings for this task. BPMN relies on XPATH 1.0 as the default language for expressing data assignments like the one above. However, other languages can be used like Java Universal Expression Language (UEL) or Groovy. The choice depends on the BPMS adopted. For example, Activiti supports UEL, Bonita and Camunda support Groovy while Bizagi supports its own expression language.

10.5.3 Service Tasks

Once we have defined the types of all data elements and mapped task and event data inputs and outputs to these types, we have to specify how tasks and events have to be implemented. For service tasks, we need to specify how to communicate with the external application that will execute the task. Be it a complex system or a simple application, all that the BPMS needs to know is its *service interface* that the service task can use. A service interface contains one or more *service operations*, each of which describes a particular way of interacting with a given service. For example, a service for retrieving inventory information provides two operations: one to check the current stock levels and one to check the stock forecast for a given product (based on product code or name). An operation can be either *in-out* or *in-only*. In an in-out operation (also called *synchronous* operation), the service expects a request message, then replies with a response message once the operation has been completed or, optionally, provides an error message if something goes wrong. For example, the service invoked by task "Check raw materials availability" receives stock availability information as input message and replies with a list of raw materials to be ordered as output message. Alternatively, if the service experiences an exception (e.g., the suppliers catalog is unreachable), it replies with an error message. The message triggers the boundary error event of this task, such that the related exception handler can be executed.[2] Conversely, in an in-only operation (also called *asynchronous* operation), the service expects a request message but will not

[2]Note that there is no throwing end error event inside "Check raw materials availability" since the catching error event is triggered by the receipt of an error message by the service task. The ability to link error messages with error events is a common feature of BPMSs.

reply with a response message. For example, the task "Archive order" notifies an archival service for the purchase order; however, the process does not wait for an archival confirmation.

Each message of a service operation needs to reference a message in the BPMN model, so that it can be assigned a data type. For instance, the request and the response messages to interact with the inventory service have the data type "purchaseOrderType" and XSD integer, respectively. For each interface, we also need to specify how this is concretely implemented, i.e., what communication protocols are used by the service and where the service is located in the network. The BPMN specification recommends the use of Web service technology to implement service interfaces. It relies on WSDL 2.0 to specify this information. In practice, this corresponds to defining one or more external WSDL documents (which specify the interface of the service to be accessed) and importing them into our BPMN model. Once again, other implementations are possible, e.g., one could implement a service interface via a Java remote procedure call or plain XML over HTTP.

Contemporary BPMSs allow the use of Web services designed according to the Representational State Transfer (REST) architectural style, a style that is supported by WSDL 2.0. A *RESTful* service is viewed as a resource or as a source of specific information (data and functionality) and identified with a Uniform Resource Identifier (URI). A service task accesses a RESTful service using the URI and a fixed set of operations. The service returns a representation of the resource. The latter typically takes the form of an XML or JavaScript Object Notation (JSON) file, which is transmitted over HTTP. When HTTP is used as the transport protocol, the operations that can be performed are the creation, retrieval, updating, and deletion of resources. BPMSs such as Camunda or Bonita offer the possibility for service tasks to invoke RESTful services. Additionally, these BPMSs expose their own functionality (e.g., the ability to create new cases of a process or to check the status of a case) via a REST Application Programming Interface (API).

After defining the service interfaces for our process, we need to associate each service task with a service operation as defined in a service interface. Based on the type of the operation (in-out or in-only), we then need to define a single data input that must match the type of the request message in the referenced service operation and, optionally, a single data output that must match the type of the response message in the operation. The BPMS engine will correctly bind the data input of the task to the request message and send it out to the service. Once the response message has been received, it will bind the content of this message to the data output of the task.

10.5.4 Send and Receive Tasks, Message and Signal Events

Send and receive tasks work similarly to service tasks. A send task is a special case of the service task: it sends a message to an external service using its data input, without expecting a reply. An example is the "Notify unavailability to customer"

task. A receive task waits for an incoming message and uses its data output to store the message content. The "Get shipping address" task is an example of this. Both task types need to reference an in-only service operation where the message is defined. For the receive, the message being received is seen as a request coming from an external service requester. So, in this case, the process itself acts as the service provider.

A receive task can also be used to receive the response of an asynchronous service, which has previously been invoked with a send task. This is the case for the "Request shipping address" and "Get shipping address" tasks. The asynchronous service is provided by the customer. Accordingly, in the send task the seller's process acts as the service requester sending a request message to the customer. In the receive task, the roles are swapped: the seller acts as the service provider to receive the response message from the customer. This pattern is used for long-running interactions, where the response may arrive after a while. The drawback of using a synchronous service task instead of a pair of send-receive task is that the service task blocks the execution of the process (or the execution of the branch of the process where it is located) until a response is received. This is not the case in Figure 10.3 (page 380). Here, the send and receive tasks are in parallel to "Emit invoice", which may well be performed between these tasks.

Message and signal events work exactly as send and receive tasks. Signal events should be used when the service being consumed has publish-subscribe capabilities, e.g., a Web service for subscribing to RSS feeds. In all other cases, we should either use message events or send and receive tasks.

10.5.5 Script Tasks

For script tasks, we need to provide the snippet of code that will be executed by the BPMS. This code can be written in a scripting or programming language, such as JavaScript or Python. BPMN does not prescribe the use of a specific language, so the choice depends on the BPMS to be used and any organizational preferences. The task's data inputs store the parameters for invoking the script while its data outputs store the results of executing the script. For example, for the "Determine cancelation penalty" task we can define a script that extracts the order date and the cancelation request date from two data inputs. These are mapped to the input objects purchase order and cancelation request. The information is then used to compute a penalty of € 15 for each day past the order date, which is then copied to the data output.

10.5.6 User Tasks

For each user task, we need to specify the rules for allocating work items to process participants at runtime, the technology for communicating with process participants,

and the details of the user interface to use. Moreover, like for any other task, we need to define data inputs to pass information to the participant, as well as the data outputs to receive the results.

Process participants that can be assigned user tasks are called *potential owners* in BPMN. A potential owner is a member of a resource class. In the context of user tasks, a resource class identifies a static list of *participants* sharing certain characteristics, e.g., holding the same role or belonging to the same department or unit. An example of a resource class for the order-to-cash process is *order clerk*, which groups all participants holding this role within the sales department of the seller organization. Note that these resource classes are unrelated to pools and lanes, which are only notational elements in a conceptual process model. A resource class can be further characterized by one or more *resource parameters*, where a parameter has a name and a data type. For example, we can define two parameters *product* and *region* of type string to indicate the particular products an order clerk works with, and the region he or she works in.

Once we have defined all required resource classes and, optionally, their parameters, we can assign each user task to one or more resource classes based on an expression. For example, we can express that work items of the "Confirm order" task must be assigned to all participants of type "Order clerk" who deal with the particular product being ordered and work in the same region as the customer. For this, we can define an XPATH expression that selects all members of the order clerk resource class who are responsible for the country specified in the purchase order.

Finally, we need to specify the implementation technology used to offer the work item to the selected participant(s). This entails aspects such as how to reach the participant (e.g., via email or worklist notification), how to render the content of the task data inputs on screen (e.g., via one or more Web forms organized through a particular screenflow), and the strategy to assign the work item to a single participant out of those satisfying the assignment expression (e.g., assign it to the order clerk with the shortest queue). Different allocation strategies for assigning work items to process participants are discussed in the box "Task allocation strategies". The configuration of all these aspects, as well as the association of participants to resource classes is dependent on the specific BPMS being used.

TASK ALLOCATION STRATEGIES

There are different strategies that can be used to allocate tasks to process participants. Research on workflow resource patterns has gathered an extensive set of such strategies. They include patterns for determining the set of participants who are considered for a task (so-called *creation patterns*), including the following ones:

Direct allocation: The participant responsible for a task is defined at design time.

(continued)

Role-based allocation: A task is assigned to a specific role at design time. At runtime, work items are offered to all participants belonging to this role.

Deferred allocation: The participant who will work on a task is only determined at runtime.

Authorization: Work items are made available only to those participants who are authorized to work on them.

Separation of duties: Two given tasks must be executed by different participants.

Case handling: Work items of a case are all allocated to the same resource.

Retain familiar: Two given tasks must be executed by the same participant.

Capability-based allocation: Work items are made available to those participants who possess the right capabilities to work on them.

History-based allocation: Work items are allocated to those participants who have successfully conducted them in the past.

Organizational allocation: Work items are allocated to those participants who hold a specific position in the organizational hierarchy.

Once this set of participants is determined, the BPMS might deliberately choose one specific participant to work on a task. Among others, the following strategies might be used (so-called *push patterns*):

Allocation by offer: A new work item is offered to participants who can check them out, having the effect that these items are no longer available to others.

Random allocation: A new work item is allocated to a randomly selected participant who fulfills the allocation condition.

Round-robin allocation: A new work item is allocated, in a circular way, to the participant who has not received an item for the longest time.

Shortest queue: A new work item is given to that participant with the shortest work item queue.

Other task allocation mechanisms are possible, such as mechanisms where participants can select the work items they wish to work on. In practice, two other mechanisms are important. **Delegation** refers to mechanisms through which participants can hand off work items to other participants. Consider that John is about to close his work day before his annual leave. He delegates all work items that are still in his list to a colleague. **Escalation** refers to mechanisms to monitor progress and automatically trigger counter-measures. Assume John forgot to delegate one of his work items. In the first week of his holidays, the BPMS discovers that an item has been pending without progress for 3 days and escalates it to his boss Mary. Mary re-allocates the work item to a colleague of John.

10.5.7 Task, Event, and Sequence Flow Expressions

Finally, we need to write expressions for the various attributes of tasks and events, as well as for the sequence flows with conditions. For instance, in a loop task we need to write a boolean expression that implements the text annotation indicating the loop condition (e.g., "until response approved"). This boolean expression will determine if the loop task is repeated. This expression can be defined over data elements, e.g., it can be an XPATH expression that extracts the value of the boolean element "approved" from the response object. We can also use *instance attributes* inside these expressions. These are variables that vary by instance at runtime. An example is *loop count*, which counts the number of iterations for a loop task. For the timer event we need to specify an expression to capture the temporal event informally expressed by its label (e.g., "Friday afternoon"). Here, we have three options: we can either provide a temporal expression in the form of a precise date or time, a relative duration, or a repeating interval. Once again, these expressions can be linked to data elements and instance properties so as to be resolved dynamically at runtime. For example, we can set an order confirmation timeout based on the number of line items in an order.

Finally, we need to write a boolean expression to capture the condition attached to each sequence flow following an (X)OR-split. For example, the condition "product in stock" after the first XOR-split in the order-to-cash example can be implemented as an XPATH expression that checks whether the value of variable stock availability is at least equal to the product quantity contained in the purchase order. There is no need to assign an expression to a default sequence flow, since it is taken by the BPMS engine if the expressions assigned to all other arcs emanating from the same (X)OR-split evaluate to false.

10.5.8 Implementing Rules with DMN

Sometimes, the conditions that allow a process instance to be routed towards one or another path in the model can be quite complex. The reason is that the business rules underpinning these conditions are inherently complex, such as the rules for assessing the credit risk of a loan applicant. Business rules may also happen to change over time, for example when a request is approved or an application is accepted. It would be convenient if in such a case only the rules were to be changed, not the whole process model. For these reasons, OMG has developed the Decision Model and Notation (DMN) standard, which at the time of writing is available in version 1.1. Instead of defining complex expressions in the outgoing arcs of (X)OR-splits, DMN allows us to define business rules separately and link these with BPMN business rule tasks or conditional events. Specifically, the rule is defined at design time and evaluated at runtime by a DMN rules engine. This engine is invoked by a business rule task or conditional event whenever an instance of such elements is

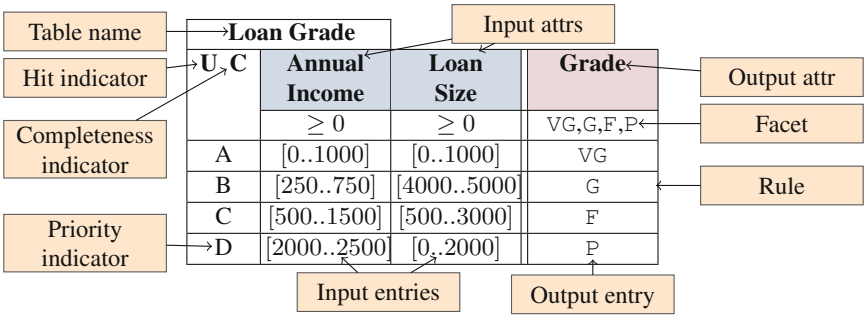

Fig. 10.8 Example of a decision table for loan applications

reached during process execution. Note, however, that the link between BPMN and DMN has not been standardized by OMG yet. Hence, each BPMS vendor offers a proprietary mechanism to link BPMN elements to DMN rules.

In essence, DMN provides three parts for the specification of business rules: the Decision Requirements Graph (DRG) that describes how data is propagated between different decisions, the Simple Expression Language (S-FEEL) to define how values are extracted from variables, and Decision Tables (DMN tables). Here, we briefly introduce DMN tables.

Figure 10.8 shows a DMN table that captures the rules for grading loan applications. The figure provides explanations of the various components of a DMN table. Each table has a name, indicators of hit, completeness and priority, input and output attributes with corresponding entries, facets, and, most importantly, rules. Each row represents a *rule* and the columns its respective *inputs* and *outputs*. Columns have a type (e.g. string, integer, or date) or specifically defined ranges called facets. In our example, a *facet* is defined with values VG (very good), G (good), F (fair), and P (poor). For a particular combination of input attributes, like $AnnualIncome = 500$ and $LoanSize = 4230$, we have to find the row that *matches* in order to identify the output value. The given values match row B and yield the $Grade = G$.

DMN tables have different indicators. The *hit indicator* specifies whether one or many rows are allowed to match a given input. The most prominent hit indicator is *U*, which indicates that any combination of input values should yield a unique output. If there is more than one match, the *priority indicator* defines which row to choose. In our example, A, B, C, D define an alphabetical order of priority. It is also possible to specify, for example, that the minimum or maximum value should be chosen. The *completeness indicator* specifies whether there must be at least one row for each possible input configuration, or if also no matches are allowed. The most prominent one is *C* indicating that the table should be complete, meaning that any combination of input values should map to an output. It is a good practice that DMN tables have unique hits and a complete set of rules.

Analysts can make mistakes when creating DMN tables. For example, it is a mistake to specify *overlapping rules* when the hit indicator does not allow it. In this case, there is an input configuration that matches several rows. It is also a mistake to have *missing rules* when the completeness indicator does not allow it. If a rule is missing, there is no row that matches a particular configuration.

Example 10.2 Let us check if the decision table of Figure 10.8 has overlapping and missing rules. Two rules are overlapping if their values on all input columns overlap. We observe the following overlaps for the first column of annual income: A and B, A and C, and B and C. For the column loan size, we observe: B and C, A and C, and A and D. This means that there is an overlap for the values where A and C overlap. This is an annual income between 500 and 1,000 together with a loan size between 500 and 1,000. This means that A and C are overlapping rules.

In order to find missing rules, we have to find values that are not covered by either column. We observe that there are missing rules for an annual income between 1,500 and 2,000, as well as for incomes greater than 2,500. There are also missing rules for a loan size between 3,000 and 4,000, and greater than 5,000. □

Exercise 10.8 Consider the DMN table in Figure 10.9. Identify overlapping and missing rules.

10.5.9 Other BPMS-Specific Properties

Strictly speaking, the only BPMS-specific properties that we have to configure in order to make a process model fully executable are those of user tasks. In practice, however, as we discussed, BPMSs may diverge from the standard way of specifying execution properties. Moreover, we likely need to connect our executable process model with one or more specific enterprise systems that are in use in our company. This is called *system binding*. BPMSs offer a range of predefined service task extensions, called *service adapters* (or *service connectors*), to implement common system binding functions in a convenient way. Examples of such binding functions include: performing a database lookup, sending an email notification, posting a message to Twitter, setting an event in Google Calendar, reading or writing a file, and adding a customer in a CRM system. Each adapter comes with a list

Fig. 10.9 Another decision table

U C	**Annual Income**	**Loan Size**	**Grade**
	≥ 0	≥ 0	VG,G,F,P
A	[0..2000]	[0..3000]	VG
B	[1000..3000]	[2000..5000]	G
C	[4000..6000]	[7000..9000]	F
D	[7000..8000]	[6000..6500]	P

of parameters that we need to configure. Many BPMSs provide *wizards* to auto-discover some of the parameter values. For instance, a database lookup requires the type of the database server (e.g., MySQL, Oracle DB) and the URL where the server can be reached, the schema to be accessed, the SQL query to run, and the credentials of the user authorized to run the query.

In our example, instead of implementing "Check stock availability" as a service task, we could implement it with a generic database lookup adapter, provided we know what to search for and where. Similarly, we could implement the tasks for communicating with the customer, such as "Notify unavailability to customer" and "Request shipping address", as email adapters. In this way, we do not need to implement a dedicated email service in our organization. The number and variety of adapters that a BPMS provides contribute to increasing user productivity with that particular BPMS.

Exercise 10.9 Consider the loan assessment process model that you obtained in Exercise 10.7 (page 384). The loan application contains these data fields:

- Applicant information:
  - Identify information (name, surname...)
  - Contact information (home phone, cell phone...)
  - Current address (street name and number, city...)
  - Previous address (as above plus duration of stay)
  - Financial information (job details, bank details)
- Reference information (identify, contact, address, relation to applicant)
- Property information (property type, address, purchasing price)
- Loan information (amount, number of years, start date, interest type: variable/fixed)
- Application identifier
- Submission date & time
- Revision date & time.
- Administration information (a section to be compiled by the loan provider):
  - Status (a string attribute with pre-defined values: "incomplete", "complete", "assessed", "rejected", "canceled", "approved")
  - Comments on status (optional, e.g., used to explain the reasons for rejection)
  - Eligibility (whether or not the applicant is eligible for a loan)
  - Loan officer identifier
  - Insurance quote required (a boolean to store whether or not a home insurance quote is sought).

The credit history report contains these data fields:

- Report identifier
- Financial officer identifier
- Reference to a loan application

- Applicant's credit information:
  - Loan applications made in the last five years (loan type: household/personal/domestic, amount, duration, interest rate)
  - Overdue credit accounts (credit type, default amount, duration, interest rate)
  - Current credit card information (provider: Visa, Mastercard…, start date, end date, interest rate)
  - Public record information (optional, if any):
    - Court judgements information
    - Bankruptcy information
- Credit assessment (a string with predefined values: AAA, AA, A, BBB, BB, B, unrated).

The risk assessment contains the following data fields:

- Assessment identifier
- Reference to a loan application
- Reference to a credit history report
- Risk weight (an integer from 0 to 100).

The property appraisal contains the following data fields:

- Appraisal identifier
- Reference to a loan application
- Property appraiser identifier
- Property information (property type, address)
- Value of three surrounding properties with similar characteristics
- Estimated property market value
- Comments on property (optional, to note serious flaws the property may have).

The agreement summary contains the following data fields:

- Reference to a loan application
- Conditions agreed (a boolean indicating if the applicant agreed with the loan conditions)
- Repayment agreed (a boolean indicating if the applicant agreed with the repayment schedule)
- Link to digitized copy of the repayment agreement.

The loan provider offers a website where applicants can submit and revise loan applications online, track the progress of their applications, and, if required, cancel applications in progress. This website implements an underlying Web service with which the loan assessment process interacts. In practice, this service acts as the applicant from the perspective of the loan assessment process. For example, if the applicant submits a new loan application through the website, then this service wraps this application into a message and sends it to the BPMS engine of the loan provider. In turn, this starts a new instance of the loan assessment process. If the loan

assessment process sends an application for review to this service, then the service presents this information to the applicant via the loan provider's website.

Further, the loan assessment process interacts with an internal service for assessing loan risks. This service determines a risk weight, which is proportional to the credit assessment contained in the credit history report, on the basis of the applicable risk rules. The risk assessment service returns a risk assessment containing an identifier (freshly generated), a reference to the loan application and one to the credit history report (both extracted from the credit history report), and the risk weight.

Based on the above information, we can specify the execution properties for the elements of this process model. It is neither required to define the actual XSD type of each data element, nor to specify the actual Groovy scripts or XPATH expressions. Instead, we identify what properties have to be specified, i.e., what data inputs and outputs, service interfaces, operations, messages, and errors are required, and determine their data type in relation to that of process variables. For example, a data input may map to a process variable or to a data field within this. For scripts, we define via task inputs and outputs what data is required by the script, what data is produced, and how the input data is transformed into the output one. For example, based on the value of a data field in a process variable, a script may write a particular value in the data field of another process variable. Similarly, for each user task, we identify what information is presented to the task performer and how the data output is obtained. Finally, we explain how each expression can be evaluated on the basis of data fields within process variables (e.g., to implement the condition of a sequence flow) or constant values like a date (e.g., to implement a timer event).

10.6 The Last Mile

Now that you have become familiar with what is required to turn a process model into an executable one, the last step for you is to take a process model and implement it using the BPMS of your choice (e.g., Activiti, Bonita, Bizagi, Camunda, IBM, Oracle, YAWL). The landscape of BPMSs and their specificities evolve continuously. We can identify three categories of BPMSs with respect to their support for BPMN:

1. **Pure BPMN:** These systems have been designed from the ground up to support BPMN natively. They follow the specification "to the letter" though they might not fully support it. Examples are Activiti and Camunda.
2. **Adapted BPMN:** These tools use a BPMN skin but rely on an internal representation to execute the process model. They can import and often also export BPMN. They typically predate BPMN and evolved from previous versions to support the specification. Examples are Bizagi and Bonita.

3. **Non BPMN:** There is, lastly, a general category of BPMSs that use their own proprietary language and semantics. These systems do not support BPMN. An example of such a system is YAWL.

Note that a BPMS may not fully cover all aspects of the BPMN specification that are relevant for execution. For example, some systems may not support compensation events or non-interrupting events. In this case, we need to give up on one or more of these elements depending of the BPMS that we adopt.

This section illustrated how to design executable BPMN models in a vendor-independent manner. The book's website[3] provides tutorial notes showing how to perform the last step of our method (the specification of execution properties) for various concrete BPMSs.

Exercise 10.10 Based on the execution properties that you specified in Exercise 10.9, implement the loan assessment process using a BPMS of your choice.

10.7 Recap

In this chapter, we presented a method for transforming conceptual process models into executable models, which can be interpreted by a BPMS. In this method, we first need to identify the type of each task (automated, manual, or user) and review manual tasks to find a way to link them to the BPMS whenever possible. Next, we have to complete the process model by specifying all control-flow and data aspects that are relevant for execution. As part of this step, we need to bridge the diverging level of granularity between conceptual and executable process models. Finally, we need to specify a number of execution properties for each model element. Some of these properties, such as those related to user tasks, are vendor-specific. They are supported in different ways by different BPMS vendors.

10.8 Solutions to Exercises

Solution 10.1 See Figure 10.10.

[3] http://fundamentals-of-bpm.org.

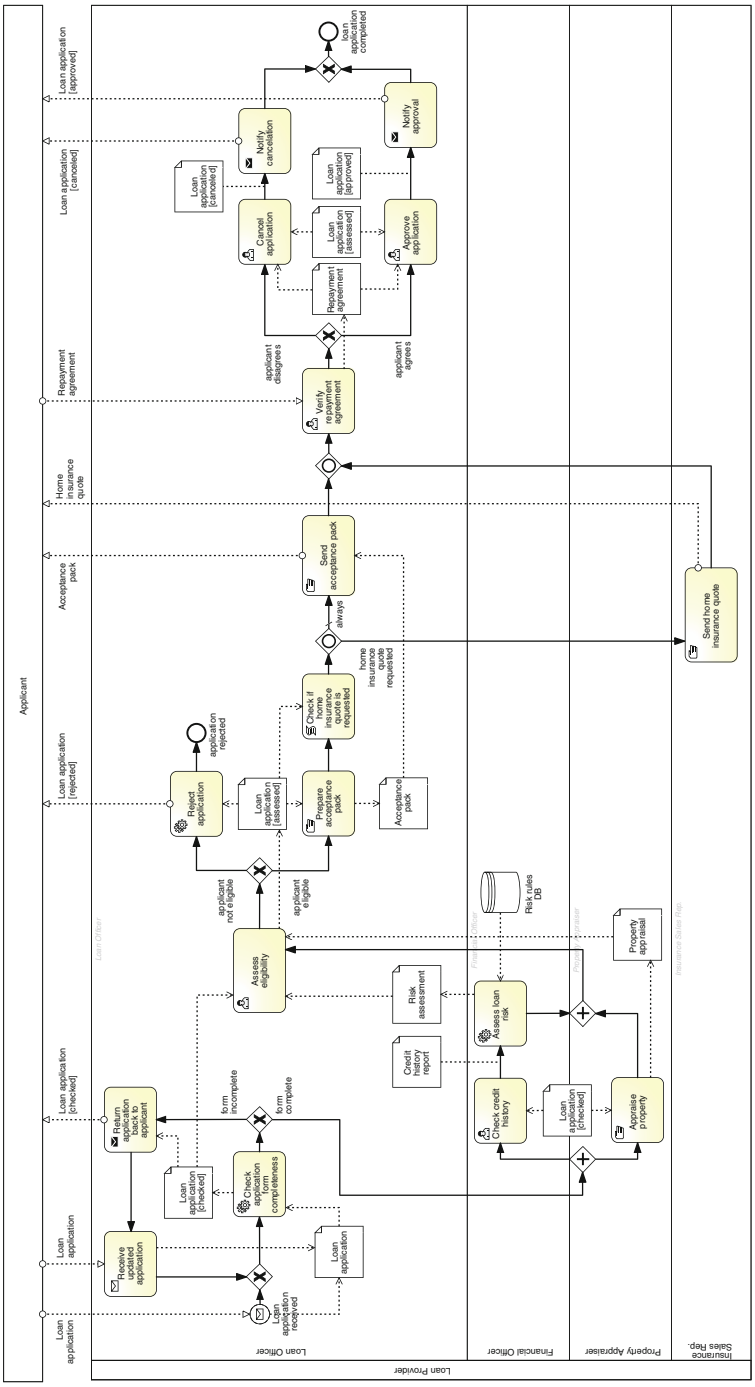

Fig. 10.10 Loan application process with task markers

Solution 10.2 All five manual tasks of this process, namely "Appraise property", "Prepare acceptance pack", "Send acceptance pack", "Send home insurance quote" and "Verify repayment agreement", can be implemented as user tasks. In "Appraise property", the property appraiser is notified through the worklist that a new property has to be appraised. The information on the property is carried by the work item of this task (e.g., property type and address). The property appraiser physically goes to the property address for an inspection and checks the value of surrounding properties. Once done, he or she prepares the appraisal on an electronic form and submits it to the BPMS engine via the worklist handler. "Prepare acceptance pack", "Send acceptance pack", "Send home insurance quote" can be implemented as user tasks in a similar way.

"Verify repayment agreement" appears in the loan officer's worklist as soon as the acceptance pack and, optionally, the insurance quote have been sent to the applicant. The officer checks out this work item once the repayment agreement is received from the applicant by post. He or she manually verifies the agreement, digitizes it and attaches it as a file to the agreement summary—an electronic form associated with this work item and pre-populated with information extracted from the loan application. If the applicant accepted all loan conditions and agreed with the repayment schedule, the officer ticks the respective checkboxes in the agreement summary and submits this to the BPMS engine.

Solution 10.3 Task "Check insurance" can be automated through a service that determines the amount of the co-payment based on the details of the prescription and on the customer's insurance policy.

Tasks "Collect drugs from shelves" and "Check quality" are manual tasks. These tasks can be implemented as user tasks in the automated process. To do so, the pharmacy technician who collects the drugs, and the pharmacist who quality-checks the prescription and seals the bag, should have a convenient mechanism to signal the completion of these tasks to the BPMS. This could be achieved by putting in place a system based on barcode scans to track prescriptions. For example, the technician would see a list of prescriptions to be filled from the worklist. He or she would then pick up one of the prescriptions and the system would associate the prescription to a new barcode which is printed on an adhesive label. The technician would then attach the label to a bag, collect the drugs and put them in a bag, and when done, he or she would scan the barcode from the label to record that the prescription has been fulfilled. This signals the completion of task "Collect drugs from shelves" to the pharmacy system. In turn, it generates a new work item of task "Check quality" in the pharmacist's worklist. The pharmacist can then quality-check the prescription and scan the barcode again.

Task "Collect payment" is also a manual task. This task could be implemented as a service task whereby the pharmacy system would push the task of collecting the payment for a prescription to a Point-of-Sale (POS) system and expect the POS system to indicate that the payment has been collected. The pharmacy technician would interact with the POS system once the customer arrives, but this interaction

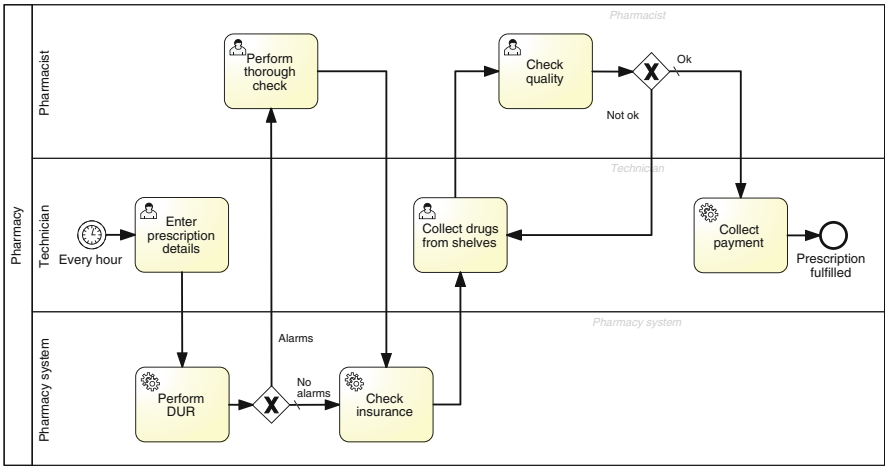

Fig. 10.11 The automated prescription fulfillment process

is outside the scope of the pharmacy system. The pharmacy system merely pushes work to the POS system and waits for completion.

The description of the process implicitly refers to a manual task whereby the pharmacist seals the bag and puts it into the pick-up area. However, this "Seal bag" task is not included in the executable process model. Instead, this task is integrated into the "Check quality" task. In other words, at the end of the quality check, the pharmacist is expected to seal the bag if the prescription is ready and drop the bag in the pick-up area. Task "Retrieve prescription bag" is also manual but there is no value in automating it in any way. So this task is left out of the executable process model, which completes once the payment has been made. The executable model of the entire prescription fulfillment process is illustrated in Figure 10.11.

Solution 10.4 It makes sense for tasks "Prepare acceptance pack" and "Send acceptance pack" to be performed by the same loan officer. However, task "Check if home insurance quote is requested" is meant to be executed between these two tasks. Since there is no temporal dependency between "Check if home insurance is requested" and the other two tasks, we can postpone the former to after "Send acceptance pack" or parallelize it with the other two tasks. This way we can aggregate the two consecutive tasks into "Prepare and send acceptance pack".

Solution 10.5

- Physical data objects: Acceptance pack (this is the loan offer on paper), Repayment agreement (this is signed by the applicant on paper and has been replaced by the Agreement summary, an electronic document containing a link to a digitized copy of the repayment agreement plus a reference to the loan application). We

assume all other communications between applicant and loan provider to occur via email
- Messages carrying physical objects: Acceptance pack, Repayment agreement, Home insurance quote (the quote is sent on paper)
- Data stores: Risk Rules DB
- States of data objects
- Pools and lanes.

Solution 10.6 A possible solution is given in Figure 10.12. Note that in this solution, a work item of task "Verify repayment agreement" automatically disappears from the loan officer's worklist if the officer does not start it within 2 weeks. This happens if the officer has not received the repayment agreement by post within that timeframe.

Solution 10.7 A possible solution is shown in Figure 10.13.

1. Task types: the manual task of this process is "Discuss application". This can be implemented as a user task that completes by producing a recommendation.
2. Non-executable elements: all elements can be interpreted by a BPMS. Note that the catching message event "Response received by post" assumes the existence of an internal service at the service provider that notifies the process when the response has been received by post.

3.1. Missing control-flow: task "Create electronic response" is needed to convert the response received by post into an electronic version, which can be consumed by a BPMS. Task "Assess response" may be interrupted by a request to cancel the application, for which the process is aborted. This request may also be received during the acceptance handling, in which case tasks "Send offer" and "File application" need to be compensated. A 1-week timeout is added to receive the response.

3.2. Missing data: all electronic data objects were missing in the conceptual model.

4. Granularity level: task "Make appointment" has been disaggregated to explicitly model the task of notifying the client of the result. Similarly, "Send offer" and "Send rejection" have been disaggregated to model the preparation of the offer and the rejection letter, respectively. Given that "Send offer" has been split into two tasks ("Make offer" and "Send offer") each needs to be compensated if a cancelation request is received.

Solution 10.8 Overlaps in the annual income column are A and B for values between 1,000 and 2,000. A and B also overlap in loan size of 2,000 and 3,000. This means A and B are overlapping rules. Missing rules exist for annual income between 3,000 and 4,000, 6,000 and 7,000 as well as greater than 8,000. There are also missing rules for loan size between 5,000 and 6,000, 6,500 and 7,000 as well as greater than 9,000.

Solution 10.9 We need two service interfaces to interact with the Web service behind the loan provider's website. One interface where the loan provider acts as the service provider and the other where the website service acts as the service

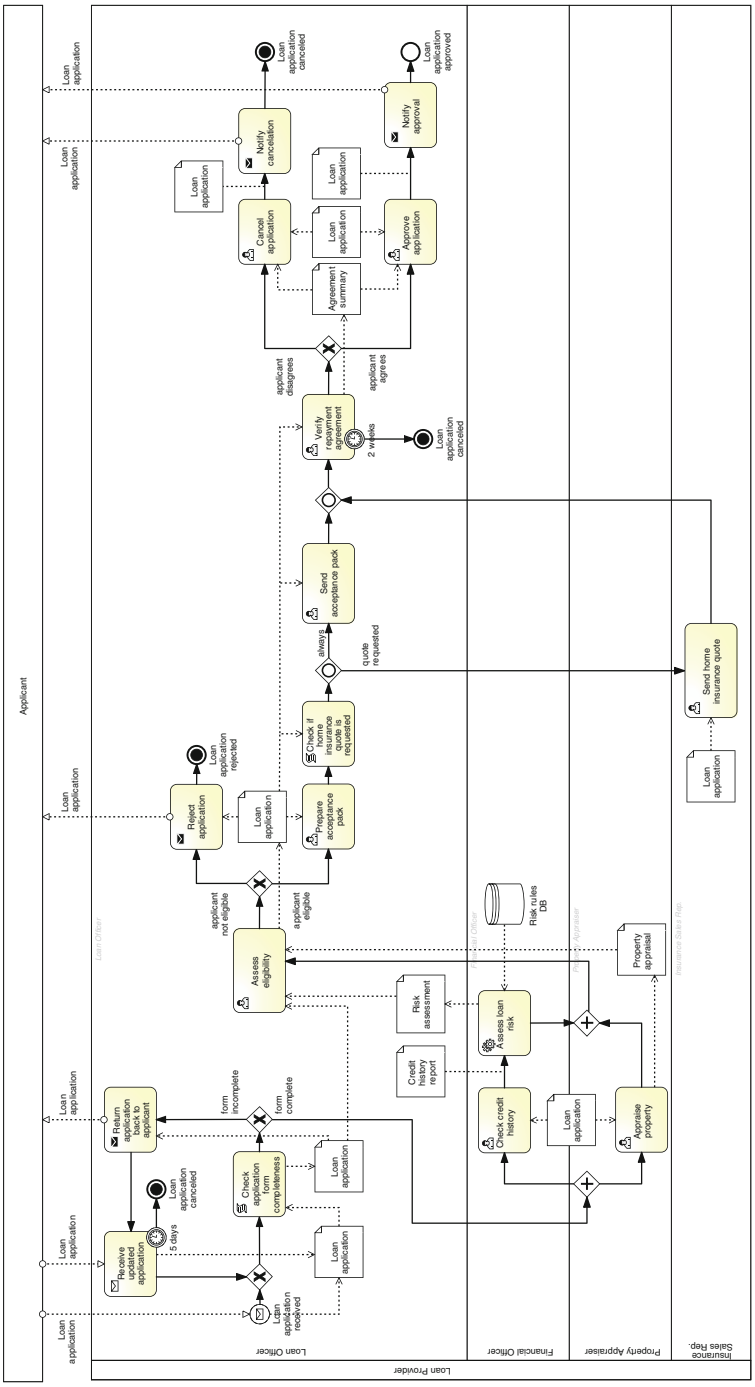

Fig. 10.12 Completed version of the loan application model

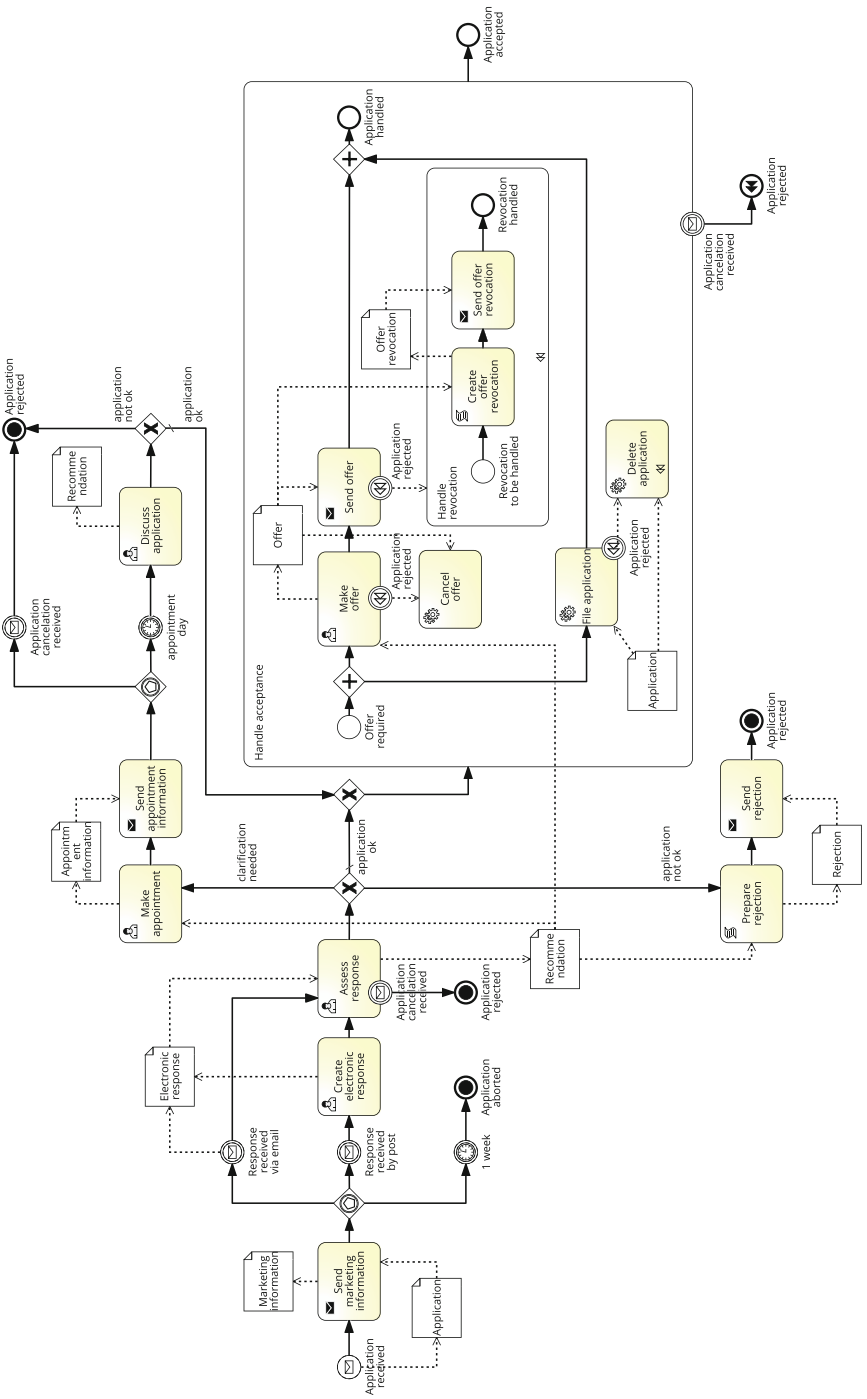

Fig. 10.13 The model for the sales process of a B2B service provider, completed with missing control-flow and data relevant for execution

provider. The former interface contains one in-only operation for the loan provider to receive the initial loan application. The latter interface contains the following four operations for the website service:

- an in-out operation to receive the assessed loan application (containing change requests), and to respond with the revised loan application (where changes have been made)
- an in-only operation to receive the rejected loan application
- an in-only operation to receive the approved or canceled loan application.

The four operations above require five messages in total, all of the same data type as the loan application's. These operations are assigned to the start message event, the four send tasks and the receive task of the process, which need to have suitable data inputs and outputs to contain the loan application. The mapping of these data inputs and outputs to data objects is straightforward, except for the send task "Reject application", which needs to modify the status of the loan application to "rejected" while copying this input data object into the task data input.

A third service interface is required to interact with the service for assessing loan risks in task "Assess loan risk". This interface has an in-out operation with two messages: an input message to contain the credit history report and an output message for the risk assessment.

The script for task "Check application form completeness" takes a loan application as input and checks that all required information is present. Depending on the outcome of the check it changes the application status to either "complete" or "incomplete", assigns a fresh application identifier to the application if empty, writes the submission or revision date and time and, if applicable, fills out the status comments section with pointers to incomplete data fields. The script task "Check if home insurance quote is requested" is actually not needed. While in a conceptual model it is important to explicitly capture each decision with a task as we have illustrated in Chapter 3, in an executable model this can be directly embedded in the conditions of the outgoing arcs of an (X)OR-split if the outcome of the decision can easily be verified. In fact, our example just needs to check the value of a boolean field in the application, which can be achieved with an XPATH expression directly on the arc labeled "quote requested".

All user tasks of this process are implemented via the worklist handler of the loan provider and offered to participants having the required role (e.g., task "Assess eligibility" is offered to a participant with role loan officer). This implementation depends on the BPMS adopted. The mapping between data objects and data inputs and outputs for these tasks is straightforward. In the case of task "Assess eligibility", at runtime the loan officer will see an electronic form for the loan application (editable), and two more forms for the risk assessment and for the property appraisal (non-editable). The officer is required to edit the loan application by entering the identifier, specifying whether or not the applicant is eligible for the loan and adding status comments in case of ineligibility. The other user tasks work similarly.

We have already discussed how to implement the condition of arc "quote requested". The conditions on the other sequence flows can be implemented with an

expression that extracts data from a data object in a similar way. The expression for the arc labeled "always" is simply "true" as this arc is always taken. The temporal expression for the two timer events is a relative duration (5 days and 2 weeks).

Solution 10.10 The book's companion website[4] provides tutorials showing how to automate the loan application process in several BPMSs, including Bizagi, Camunda, IBM BPM, and Oracle.

10.9 Further Exercises

Exercise 10.11 Identify the type of the tasks in Figure 4.13 (page 148), and represent them using appropriate BPMN markers.

Exercise 10.12 Consider the following business processes. Identify which of these models can be automated and justify your choice.

1. Recruiting a new soldier.
2. Organizing a court hearing.
3. Buying an item at an auction on eBay.
4. Managing inventory assets disposition.
5. Booking a trip online.
6. Handling an IT-system maintenance job.
7. Servicing a used car at a mechanic.
8. Making online trade customs declarations.
9. Processing employee payrolls.
10. Synchronizing data servers in a distributed environment.

Exercise 10.13 Figure 10.14 shows the process model that FixComp follows when a client files a complaint. Upon receipt of a new complaint from a client, the process starts by sending an automatic reply to the client in order to reassure it that FixComp is following up on its request. A complaints representative then takes the complaint for discussion with people in the department the complaint refers to. Next, the complaints representative sends a personal letter of apology to the client and proposes a solution. The client can either accept or reject the solution. If the client accepts the solution, the solution is executed by the relevant department. If the client rejects the solution, the client is called by phone to discuss possible alternatives by the complaints representative. If one of these alternatives is promising, it is discussed with the department and the process continues. If no agreement can be reached, the case is brought to court.

The company wants to automate this process to deal with complaints in a more efficient manner. Your task is to prepare this model for execution.

[4] http://fundamentals-of-bpm.org/supplementary-material.

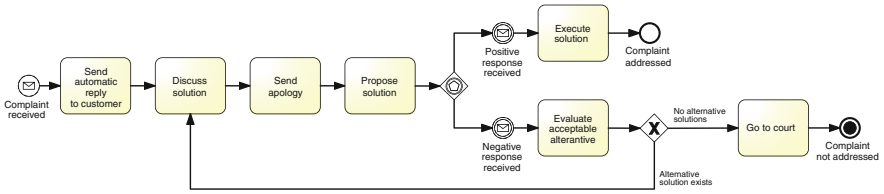

Fig. 10.14 FixComp's process model for handling complaints

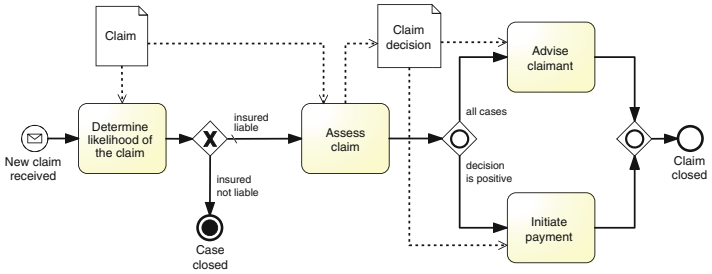

Fig. 10.15 Claims handling process model

Acknowledgement This exercise is adapted from a similar exercise developed by Remco Dijkman, Eindhoven University of Technology.

Exercise 10.14 Consider the claims handling process modeled in Figure 10.15. Implement this business process using a BPMS of your choice.

The process starts when a customer submits a new insurance claim. Each insurance claim goes through a two-stage evaluation process. First of all, the liability of the customer is determined. Secondly, the claim is assessed in order to determine if the insurance company has to cover this liability and to what extent. If the claim is accepted, payment is initiated and the customer is advised of the amount to be paid. All tasks except "Initiate Payment" are performed by claim handlers. There are three claim handlers. The task "Initiate Payment" is performed by a financial officer. There are two financial officers.

As shown in the model, there are two data objects involved in this process: Claim and Claim decision. A claim includes the following data fields:

- Name of claimant
- Policy number (a string with alphanumeric characters)
- Description of the claim
- Amount claimed.

A claim decision consists of the following data fields:

- Reference to a claim
- Decision (positive or negative)
- Explanation

- Amount to be reimbursed (greater than zero if the decision is positive).

You may add other data fields into the above objects if you deem that necessary.

Exercise 10.15 Consider the following equipment rental process, which is a variant of the one described in Example 1.1 (page 3). Implement this business process using a BPMS of your choice.

The rental process starts when a site engineer fills in an equipment rental request containing the following details:

- Name or identifier of the site engineer who initiates the request
- Requested start date & time of the equipment rental
- Expected end date & time of the equipment rental
- Project for which the equipment is to be rented
- Construction site where the equipment will be used
- Description of required equipment
- Expected rental cost per day (optional)
- Preferred supplier (optional)
- Supplier's equipment reference number (optional)
- Comments to the supplier (optional).

The rental request is taken over by one the clerks at the company's depot. The clerk consults the catalogs of the equipment suppliers and calls or sends emails to the supplier(s) in order to find the most cost-effective available equipment that complies with the request. Once the clerk has found a suitable piece of equipment available for rental, he or she recommends that it be rented. At this stage, the clerk must add the following data to the equipment rental request:

- Selected supplier
- Reference number of the selected equipment
- Cost per day.

Equipment rental requests have to be approved by a works engineer (who also works at the depot). In some cases, the works engineer rejects the equipment rental request, meaning that no equipment will be hired. Of course, before rejecting a request in this way, the works engineer should first discuss his or her decision with the site engineer and also add an explanatory note to the equipment rental request. In other cases, the works engineer rejects the recommended equipment (but not the entire request) and asks the clerk to find an alternative equipment. Again, in this case the works engineer should communicate the decision to the clerk and add an explanatory note.

Rental requests where the cost per day is below € 100 are automatically approved, without going through a works engineer.

Once a request is approved, a purchase order is automatically generated from the data contained in the approved rental request. The purchase order includes:

- Supplier's equipment identification
- Cost per day

- Construction site where the plant is to be delivered
- Delivery date & time
- Pick-up date & time
- Comments to the supplier (optional).

The supplier delivers the equipment to the construction site at the required date. The site engineer inspects the equipment. If everything is in order, he or she accepts the equipment, adds the date of delivery to the purchase order and optionally a note to indicate any issues found during the inspection. Similarly, when the equipment is picked up by the supplier at the end of the renting period, another inspection is performed, and the supplier marks the pick-up date in the purchase order (possibly with a pick-up note).

Sometimes, the site engineer asks for an extension of the rental period. In this case, the site engineer writes down the extended pick-up time into the purchase order, and the revised purchase order is automatically resent to the supplier. Prior to doing this change, the site engineer is expected to call the supplier in order to agree on the change of pick-up date.

A few days after the equipment is picked up, the supplier sends an invoice to the clerk by email. The clerk records the following details:

- Supplier's details
- Invoice number
- Purchase order number
- Equipment reference number
- Delivery date & time
- Pick-up date & time
- Total amount to be paid.

Having entered these invoice details, the clerk verifies the invoice details against the purchase order and marks the invoice as accepted or rejected. In case of rejection, the clerk adds an explanatory note (e.g., requesting the supplier to send a revised invoice). Eventually, the supplier may send a revised invoice if needed.

The accepted invoice is forwarded to the financial department for payment, but this part of the process is handled separately and is not part of this exercise.

Exercise 10.16 Define appropriate data types for the sales process shown in Figure 10.13 (page 406) and implement it using a BPMS of your choice.

10.10 Further Readings

A discussion on executable BPMN 2.0 is included in Silver's book [163]. He also wrote a book with Sayles about DMN [164]. A good coverage of the three modeling standards BPMN, CMMN, and DMN, and their usage in making processes executable, is provided in the book by Freund & Rücker [49]. An in-depth coverage of process automation using the YAWL language is given by ter

Hofstede et al. [67]. Weske also extensively discusses the implementation aspects of executable business processes in his book [193]. A classic book on how process execution works inside the process engine is Leymann & Roller [89].

A gentle introduction to XML, XML Schema, and XPath can be found in Møller & Schwartzbach's book [116]. Web services are covered in depth by Erl et al. [45]. This latter book also includes a discussion of WSDL 2.0, the default technology for implementing service interfaces in BPMN 2.0. There are also books on RESTful Web Services, including the one by Richardson & Ruby [141]. A good discussion of technical concerns, but from a technology-independent perspective, is included in the book on workflow patterns [155]. This book also discusses various strategies of allocating work to resources.