Business Processes Modelling

MPB (6 cfu, 295AA)

Roberto Bruni
http://www.di.unipi.it/~bruni

22 - Business process modelling notation
Object

We overview BPMN and their analysis based on Petri nets

Ch.4.7, 5.7 of Business Process Management: Concepts, Languages, Architectures
Ch.3, 4 of Fundamental of Business Process Management. M. Dumas et al.
BPMN

Main goal:

to define a **graphical notation** that is **readily understandable**:

by **business analysts** (initial drafts of processes)

by **technical developers** (process implementation)

by **business people** (process management)
Standardisation

In the context of graphical models for business processes the development of BMPN is an important step in:

- **reducing the fragmentation** that existed with myriad of process modelling tools and notations
- **exploiting experiences** with many divergent proposals to consolidate the best ideas
- supporting the adoption of **inter-operable** business process management systems
Short history

2000 - **Business Process Management Initiative** (BPML.org)
(independent organization, studying *open specifications* for the
management of *e-Business processes*)

2005 - BPMI and the Object Management Group™ (OMG™) merge their activities on BPM forming the
**Business Modeling & Integration Domain Task Force** (BMI -DTF)

2006 - **BPMN 1.0 approved**
2007 - BPMN 1.1 approved
2009 - BPMN 1.2 approved

2009 - BPMN 2.0 Beta 1 proposed
2010 - BPMN 2.0 Beta 2 proposed
2011 - **BPMN 2.0 Final delivered**
A joint effort!
Business process diagrams

BPMN defines a standard for Business Process Diagrams (BPD) based on flowcharting technique

Four categories of elements

<table>
<thead>
<tr>
<th>swimlanes</th>
<th>flow objects</th>
<th>connecting objects</th>
<th>artefacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lane</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- swimlanes
- flow objects
- connecting objects
- artefacts

Data Store

Text Annotation Allows a Modeler to provide additional Information
Wrong use of flows in/between Pool A Rule Timer Message General define message flows between Pools. Model the process in each Pool are incorrectly used to connect pools. When modelling Pools, sequence example: events are wrongly mo...
BPMN 2.0 vs 1.0

Updated (new markers):
Tasks/SubProcesses
Events
Gateways
Artefacts

Added:
Choreographies
Full metamodel
XML Serialization
Diagram Interchange
BPMN Execution Semantics (verbal)
What is BPMN?

BPMN is a graphical notation that depicts the steps (end to end flow) in a business process.

Specifically designed to coordinate the sequence of processes and the messages that flow between participants in a related set of activities.
Why is BPMN important?

The world of business processes has changed dramatically over the past few years. Processes can be coordinated from behind, within and over organizations boundaries. A business process now spans multiple participants and coordination can be complex.

Until BPMN, there has not been a standard modelling technique developed that addresses these issues. BPMN provides users with a royalty free notation.

This will benefit users in a similar manner in which UML standardised the world of software engineering. There will be training courses, books and a body of knowledge that users can access in order to better implement a business process.
Will there be a major rewrite?

Not for 2 or 3 years…

(good work! 10 years and still no revision is planned)
Strong points of BPMN

**Simplicity**: A small set of basic symbols

**Extensibility**: many decorations available (new ones can be added in the future)

  Graphical design: **intuitive**

**Generality**: **orchestration** + **choreography**

**Tool availability**: **exchange format**
Weaknesses of BPMN

over 100 graphical elements

verbose description (500 pages)

difficult to learn comprehensively:
different readings of the same diagram are possible

different BPMN vendors implement the execution of BPMN diagrams in different ways (and for different subsets)
1 - BPMN basics
Swimlanes
(pools, lanes)
Swimlanes

A **swimlane** is a mechanism to **organise activities into separate visual categories** to illustrate different capabilities or **responsibilities**

Present in many process modelling methodologies

BPMN supports two main swimlane objects:

- **Pool**
- **Lanes**
Pools

A pool represents a participant (or role) in a process (represented as a rectangle with a name)
Collapsed pools

Internal process is not exposed
Lanes

A lane is a hierarchical sub-partition within a pool that is used to organise and categorise activities (inner rectangle that extends to the entire length of the pool)
### Constraints

<table>
<thead>
<tr>
<th>Pool</th>
<th>Lane</th>
</tr>
</thead>
</table>

A Pool MUST contain 0 or 1 business process.

A Pool can contain 0 or more lanes.

Two pools can only be connected with message flows.
Naming conventions

Process models:
- a noun possibly preceded by an adjective
- the label is often obtained by "nominalizing" the verb that describe the main action in the process
  (e.g., claim handling, order fulfillment)

Avoid long labels
- Articles are often omitted
Flow Objects
(events, activities, gateways)
Flow objects

Rationale:

fix a small set of core elements
so that modellers must learn a small number of shapes:

- **events**
- **activities**
- **gateways**
Flow objects

Rationale:

fix a small set of core elements
so that modellers must learn a small number of shapes:

events

activities

gateways

use different border styles and internal markers
to add many more information
(this way the notation is extensible)
Flow objects: Events
Events

An event is something that “happens” during the course of a business process.

An event is represented as a circle. Different borders define the type of the event.

- **Start**
- **Intermediate**
- **End**
Naming conventions

Events:
the label should begin with a noun and end with a verb in past participle form to indicate something that just happened (e.g., Invoice emitted)

the noun can be preceded by an adjective (e.g., Urgent order sent)

Avoid long labels
Articles are often omitted
Flow objects: Activities
Activities

An activity is some “unit of work” (job) to be done during the course of a business process

An activity is represented as a rounded box

BPMN has two main types of activities

atomic (task) or compound (sub-process)
Sub-processes

Large process models are hard to parse: we improve readability by hiding certain parts within sub-processes

A **sub-process** is a self-contained, composite activity that can be broken into smaller units of work
Sub-processes

Large process models are hard to parse: we improve readability by hiding certain parts within sub-processes.

A **sub-process** is a self-contained, composite activity that can be broken into smaller units of work.

Collapsed Subprocess

Expanded Subprocess

implicit start / end
Naming conventions

Activities:
verb in the imperative form followed by a noun
(e.g., Approve order)

the noun can be preceded by an adjective
(e.g., Issue driver license)

the verb may be followed by a complement
(e.g., Renew driver license via offline agencies)

Avoid long labels
Articles are often omitted
Flow objects: Gateways
Gateways

A **gateway** is used to split/join the sequence flow (conditional, fork, wait)

A gateway is represented as a diamond shape
internal markers indicate the nature of behaviour control

**Data-based Exclusive Gateway**
When splitting, it routes the sequence flow to exactly one of the outgoing branches based on conditions. When merging, it waits one incoming branch to complete before triggering the outgoing flow.

**Parallel Gateway**
When used to split the sequence flow, all outgoing branches are activated simultaneously. When merging parallel branches it waits for all incoming branches to complete before triggering the outgoing flow.
Connecting objects
(sequence flow, message flow, association)
Connecting objects

The Flow objects are connected together in a diagram to create the basic skeletal structure of a business process.

Three connecting objects can be used:

<table>
<thead>
<tr>
<th>Sequence flow</th>
<th>Message flow</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>connected objects must reside in the same pool (but they can be in different lanes)</td>
<td>connected objects must reside in different pools to be discussed later</td>
<td>connects flow objects with artefacts to be discussed later</td>
</tr>
</tbody>
</table>
Sequence flow

A **sequence flow** is used to show the order in which activities are to be performed.

The term “control flow” is generally avoided in BPMN.

A sequence flow is represented by a solid line with a solid arrowhead.
### Constraints

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Each event:**
  - At most one incoming and at most one outgoing sequence flow

- **Each activity:**
  - Exactly one incoming and exactly one outgoing sequence flow

- **Each gateway:**
  - One-to-many, many-to-one, many-to-many
Multiple flows and implicit gateways

In principle each activity should have exactly:
one incoming arc, one outgoing arc

Be careful if this is not the case!

Multiple incoming flows are mutually exclusive
Multiple flows and implicit gateways

In principle each activity should have exactly:
   one incoming arc, one outgoing arc

Be careful if this is not the case!

Multiple outgoing flows are activated in parallel
(unless conditions are attached to them)
In your final projects

Please avoid
Artefacts:
(data-objects, groups, text annotations)
Artefacts

BPMN is designed to allow modellers and modelling tools some flexibility in extending the basic notation.

Any kind of artefacts can be added to a diagram as appropriate for the specific modelling domain.

BPMN includes three pre-defined types of artefacts:

- **data objects**
- **groups**
- **text annotation**

[Images of artefacts: data objects, groups, text annotation]

to be discussed later to be discussed later
An association is used to associate data, text, and other artefacts with flow objects.

An association is represented by a dotted line (with an optional line-arrowhead).

..........→

.......... used especially for text annotation
Any object can be associated with a text annotation to provide any additional information and documentation that can be needed.

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

Example 3.1

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

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

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

Fig. 3.4 An example of the use of XOR gateways forked with an XOR-split. An XOR gateway is indicated with an empty diamond or with a diamond marked with an “X”. From now on, we will always use the “X” marker.

Example 3.2 Invoice checking process. As soon as an invoice is received from a customer, it needs to be checked for mismatches. The check may result in either of these three options: i) there are no mismatches, in which case the invoice is posted; ii) there are mismatches but these can be corrected, in which case the invoice is re-sent to the customer; and iii) there are mismatches but these cannot be corrected, in which case the invoice is blocked. Once one of these three activities is performed the invoice is parked and the process completes.

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

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

It is useful to annotate branches with the conditions under which they are taken.
Annotated sequence flow

- **Sequence Flow** defines the execution order of activities.
- **Conditional Flow** has a condition assigned that defines whether or not the flow is used.
- **Default Flow** is the default branch to be chosen if all other conditions evaluate to false.

read as `otherwise`
Parallel activities: airport security check

Fig. 3.5 An example of the use of AND gateways

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

The AND-split splits the token coming from activity “Proceed to security check” into two tokens. Each of these tokens independently flows through one of the two branches. This means that when we reach an AND-split, we take all outgoing branches (note that an AND-split may have multiple outgoing arcs). As we said before, a token is used to indicate the state of a given instance. When multiple tokens of the same color are distributed across a process model, e.g. as a result of executing an AND-split, they collectively represent the state of an instance. For example, if a token is on the arc emitting from activity “Pass luggage screening” and another token of the same color is on the arc incident to activity “Pass personal security screening”, this indicates an instance of the security check process where a passenger has just passed the luggage screening.

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

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

The resulting model is shown in Fig. 3.6.
Multiple start events are often considered as a convenient notation (they capture mutually exclusive triggers to start a process instance)
Multiple end events: order fulfilment

Multiple end events are often considered as a convenient notation (they are mutually exclusive in the example).

BPMN adopts implicit termination semantics: a case ends only when each ``token” reaches the end.
Inclusive decisions: order distribution

Only XOR / AND gateways, but the diagram is convoluted!
What if we had three or more warehouses? (does not scale)
Inclusive decisions: order distribution

Only XOR / AND gateways, the diagram can ```scale”’, but is it correct? (also the case no-warehouse is now possible)
Inclusive decisions (one, many)

Inclusive Gateway
When splitting, one or more branches are activated based on branching conditions. When merging, it awaits all active incoming branches to complete.
Inclusive decisions: order distribution

Fig. 3.10
Modeling an inclusive decision with the OR gateway

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

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

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

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

Inclusive decisions:

using OR gateways, the diagram can ``scale'', but all the issues with unmatched OR-joins in EPC are still valid!

Order distribution

Use OR-gateways only when strictly necessary
XOR + AND + OR: order fulfilment

3.2 Branching and Merging

Fig. 3.12 The order fulfillment process diagram with product manufacturing

3.2.4 Rework and Repetition

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

Example 3.7

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

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

Better if gateways are balanced
XOR + AND + OR: order fulfillment

Better if gateways are balanced
XOR + AND + OR: order fulfillment

Fig. 3.12 The order fulfillment process diagram with product manufacturing

3.2 Branching and Merging

3.2.4 Rework and Repetition

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

Example 3.7

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

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

Better if gateways are balanced
Resources as lanes: order fulfillment
Resources as lanes: order fulfillment
Placing items in lanes

**events**: must be placed in the proper lane

**activities**: must be placed in the proper lane

**gateways**:

(X)OR-splits: same lane as preceding decision activity

**AND-split**: placement is irrelevant

(any kind of) **join**: placement is irrelevant

**data-objects**: placement is irrelevant
Rework and repetition: ministerial correspondence

A repetition block starts with a XOR-join and ends with a decision gateway (XOR-split)
Identify sub-processes

Fig. 4.1 Identifying sub-processes in the order fulfillment process of Fig. 3.12.

Thus these activities, and their connecting gateways, can be encapsulated in a sub-process. In other words, they can be seen as the internal steps of a macro-activity called "Acquire raw materials." Similarly, the two parallel branches for shipping and invoicing the order can be grouped under another sub-process activity called "Ship and invoice." Figure 4.1 illustrates the resulting model, where the above activities have been enclosed in two sub-process activities. We represent such activities with a larger rounded box which encloses their internal steps. As we observe from Fig. 4.1, we add a start event and an end event inside each sub-process activity, to explicitly indicate when the sub-process starts and completes.

Recall that our initial objective was to simplify a process model. Once we have identified the boundaries of the sub-processes, we can simplify the model by hiding the content of its sub-processes, as shown in Fig. 4.2. This is done by replacing the macro-activity representing the sub-process with a standard-size activity. We indicate that this activity hides a sub-process by marking it with a small square with a plus sign (+) inside (like we could expand the content of that activity by pressing the plus button). This operation is called collapsing a sub-process. By collapsing a sub-process we reduce the total number of activities (the order fulfillment process has only six activities now), thus improving the model readability. In BPMN, a sub-process which hides its internal steps is called a collapsed sub-process, as opposed to an expanded sub-process which shows its internal steps (as in Fig. 4.1).
Collapsed sub-processes

Exercise 4.1
Identify suitable sub-processes in the process for assessing loan applications modeled in Exercise 3.5.

Hint
Use the building blocks that you created throughout Exercises 3.1 – 3.4.

Collapsing a sub-process does not imply losing its content. The sub-process is still there, just defined at an abstraction level below. In fact, we can nest sub-processes in multiple levels, so as to decompose a process model hierarchically. An example is shown in Fig. 4.3, which models a business process for disbursing home loans. In the first level we identified two sub-processes: one for checking the applicant's liability, the other for signing the loan. In the second level, we factored out the scheduling of the loan disbursement within the process for signing loans into a separate sub-process.

As we go down the hierarchical decomposition of a process model, we can add more details. For example, we may establish a convention that at the top level we only model core business activities, at the second level we add decision points, and so on all the way down to modeling exceptions and details that are only relevant for process automation.

Question
When should we decompose a process model into sub-processes?

We should use sub-processes whenever a model becomes too large that is hard to understand. While it is hard to precisely define when a process model is “too large”, since understandability is subjective, it has been shown that using more than approximately 30 flow objects (i.e. activities, events, gateways) leads to an increased probability of making mistakes in a process model (e.g. introducing behavioral issues). Thus, we suggest to use as few elements as possible per each process model level, and in particular to decompose a process model if this has more than 30 flow objects.

Reducing the size of a process model, for example by collapsing its sub-processes, is one of the most effective ways of improving a process model’s readability. Other structural aspects that affect the readability include the density of the collapsed sub-processes.
2 - BPMN key features (with some examples)
Markers
(events, activities, gateways)
Activity types and markers

Internal markers indicate: the activity nature (task type) and the way it is executed (activity marker)

Some types:
- Send Task
- Receive Task
- User Task
- Manual Task
- Business Rule Task
- Service Task
- Script Task

Some markers:
- Sub-Process Marker
- Loop Marker
- Parallel MI Marker
- Sequential MI Marker
- Ad Hoc Marker
- Compensation Marker
Some activity markers

**Multiple Instances** of the same activity are started in parallel or sequentially, e.g. for each line item in an order.

**Loop Activity** is iterated if a loop condition is true. The condition is either tested before or after the activity execution.

**Ad-hoc Subprocesses** contain tasks only. Each task can be executed arbitrarily often until a completion condition is fulfilled.
Fig. 3.13: A process model for addressing ministerial correspondence.

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

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

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

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

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

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

COMPONENTS OF A MODELING LANGUAGE
A modeling language consists of three parts: syntax, semantics, and notation.

The syntax provides a set of modeling elements and a set of rules to govern
Loop marker: ministerial correspondence

![Diagram of ministerial correspondence process]

- The loop-symbol decoration marks the possible repetition of the sub-process activity.
- It is important to define exit conditions from loops!
Loop marker: ministerial correspondence

we can further simplify the inner process (implicit start / end)
Loop marker: ministerial correspondence

we can hide internal details
Catching and throwing

An event can catch a **trigger** or throw a **result**
Internal markers denote the trigger or result

**Catching**

- **Start Event**: Catching an event starts a new process instance.

- **Intermediate Event (catching)**: The process can only continue once an event has been caught.

**Throwing**

- **End Event**: An event is thrown when the end of the process is reached.

- **Intermediate Event (throwing)**: An event is thrown and the process continues.
## Some internal markers

<table>
<thead>
<tr>
<th></th>
<th>Start</th>
<th>Intermediate</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Catching</strong></td>
<td><img src="circle.png" alt="Plain" /></td>
<td><img src="envelope.png" alt="Message" /></td>
<td><img src="clock.png" alt="Timer" /></td>
</tr>
<tr>
<td><strong>Throwing</strong></td>
<td><img src="symbol.png" alt="Error" /></td>
<td><img src="circle.png" alt="Terminate" /></td>
<td>Untyped events, typically showing where the process starts or ends.</td>
</tr>
<tr>
<td><strong>Plain</strong></td>
<td><img src="circle.png" alt="Plain" /></td>
<td><img src="envelope.png" alt="Message" /></td>
<td><img src="clock.png" alt="Timer" /></td>
</tr>
<tr>
<td><strong>Message</strong></td>
<td><img src="symbol.png" alt="Error" /></td>
<td><img src="circle.png" alt="Terminate" /></td>
<td>Receiving and sending messages.</td>
</tr>
<tr>
<td><strong>Timer</strong></td>
<td><img src="symbol.png" alt="Error" /></td>
<td><img src="circle.png" alt="Terminate" /></td>
<td>Cyclic timer events, points in time, time spans or timeouts.</td>
</tr>
<tr>
<td><strong>Error</strong></td>
<td><img src="symbol.png" alt="Error" /></td>
<td><img src="circle.png" alt="Terminate" /></td>
<td>Catching or throwing named errors.</td>
</tr>
<tr>
<td><strong>Terminate</strong></td>
<td><img src="symbol.png" alt="Error" /></td>
<td><img src="circle.png" alt="Terminate" /></td>
<td>Triggering the immediate termination of a process.</td>
</tr>
</tbody>
</table>
4.4.2 Temporal Events

Besides the message event, there are other triggers that can be specified for a start event. One worth of notice is the timer event. This event type indicates that process instances start upon the occurrence of a specific temporal event, e.g., every Friday morning, every working day of the month, every morning at 7am.

At time a timer event may also be used as an intermediate event, to model a temporal interval that needs to elapse before the process instance can proceed. To indicate a timer event, we mark the event symbol with a light watch inside the circle. Timer events are catching events only since a timer is a trigger outside the control of the process.

Example 4.3

Let us consider the following process at a small claims tribunal.

In a small claims tribunal, callovers occur once a month, to set down the matter for the upcoming trials. The process for setting up a callover starts three weeks prior to the callover day, with the preparation of the callover list containing information such as contact details of the involved parties and estimated hearing date. One week prior to the callover, the involved parties are contacted to determine if they are all ready to go to trial. If this is the case, the callover is set, otherwise it is deferred to the next available slot. Finally, on the callover day, the callover material is prepared and the callover is held.

This process is driven by three temporal events: it starts three weeks prior to the callover date, continues one week prior to the callover date, and concludes on the day of the callover. To model these temporal events we need one start and two intermediate timer events, as shown in Fig. 4.12.

Let us see how this process works from a token semantics point of view. A token capturing a new instance is generated every time it is three weeks prior to the callover date (we assume this date has been scheduled by another process). Once the first activity

the clock annotation denotes a timer start event: an instance of the process is created when some temporal event happens

the clock annotation denotes a timer intermediate event: the process is blocked until a time-out expires
Process break (event waiting)

the envelope annotation denotes an intermediate message event: it signals the receipt of a message
Collaboration diagrams
(and message passing)
Message annotated events and activities

A start event can be annotated with a white-envelope: a process instance is created when a certain message is received

An end event can be annotated with a black-filled envelope: when the process ends a message is sent

Intermediate events and activities can be annotated with both kinds of envelope white = receipt of a message, black = the sending of a message
Events vs Activities

Should we use 📧 or 📥? 

No clear distinction is made, but typically 

events are instantaneous 

activities take time (have a duration)
Collaboration

A collaboration contains two or more pools, each representing a participant in the collaboration.

A pool may be collapsed or exhibit the process within.

Each possible communication corresponds to a message flow between pools (or between objects from different pools).
Message flow

A message flow represents communications (send/receive) between two separate participants (business entities or business roles)

A message flow is represented by a dashed line with a open arrowheads

○-----→
Message flow constraints

- Each event: at most one message flow
- Each activity: at most one message flow
- Each gateway: no message flow!
- Each pool: any number of message flows
From processes to collaborations

a stand-alone process
From processes to collaborations

a collaboration with a collapsed pool
From processes to collaborations

A collaboration between processes
From processes to collaborations

a more detailed collaboration
Example: Seller
Example: Seller & Customer
Example: Seller & Customer
Artefacts: message data objects

A **message data object** depicts the data that are communicated between two participants.

A message data object is represented as an envelope.
Example: Seller & Customer
Example: Seller, Customer & Suppliers
Deferred choice
(event based decisions)
Event-based decisions

Event-based Exclusive Gateway
Is always followed by catching events or receive tasks. Sequence flow is routed to the subsequent event/task which happens first.

Event-based (split) gateways must be used to model decisions that depends on some external event
A negotiation without choice
Some remarks

Lanes are often used to separate activities associated with a specific company function or role.

Sequence flow cannot cross the boundaries of a pool (it can cross lanes in the pool).

Message flow cannot connect flow objects in the same pool.
3 - more on BPMN (with some examples)
More artefacts
(data-objects, groups)
A **data object** represents information flowing through the process, such as documents, emails and letters.

A data object is often represented by the usual file icon.
Association, again

Attaching a data object with an Undirected Association to a sequence flow indicates hand-over of information between the activities involved.

A Directed Association indicates information flow. A data object can be read at the start of an activity or written upon completion.

A Bidirected Association indicates that the data object is modified, i.e. read and written during the execution of an activity.
More data objects

A Collection Data Object represents a collection of information, e.g., a list of order items.

A Data Store is a place where the process can read or write data, e.g., a database or a filing cabinet. It persists beyond the lifetime of the process instance.
An arbitrary set of objects can form a group (if they logically belong together) it has no behavioural effect (only documentation)

A group is represented by rounded corner rectangles with dashed lines.
80 3 Essential Process Modeling

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

Often the output from an activity coincides with the input to a subsequent activity. For example, once Raw materials have been obtained, these are used by activity "Manufacture product" to create a Product. The Product in turn is packaged and sent to the customer by activity "Ship product". Effectively, data objects allow us to model the information flow between process activities. Bear in mind, however, that data objects and their associations with activities cannot replace the sequence flow. In other words, even if an object is passed from an activity A to an activity B, we still need to model the sequence flow from A to B. A shorthand notation for passing an object from an activity to the other is by directly connecting the data object to the sequence flow between two consecutive activities via an undirected association. See for example the Shipment address being passed from activity "Get 108 Example: artefacts for convenience, the same data object can appear several times it can have different states"
80 3 Essential Process Modeling

Fig. 3.14 The order fulfillment example with artifacts from activity "Obtain raw materials from Supplier 1" to Raw materials, indicates that Raw materials is an output object for this activity. To avoid cluttering the diagram with data associations that cross sequence flows, we may repeat a data object multiple times within the same process model. However, all occurrences of a given object do conceptually refer to the same artifact. For example, in Fig. 3.14, Purchase order is repeated twice as input to "Check stock availability" and to "Confirm order" since these two activities are far away from each other in terms of model layout. Often the output from an activity coincides with the input to a subsequent activity. For example, once Raw materials have been obtained, these are used by activity "Manufacture product" to create a Product. The Product in turn is packaged and sent to the customer by activity "Ship product". Effectively, data objects allow us to model the information flow between process activities. Bear in mind, however, that data objects and their associations with activities cannot replace the sequence flow. In other words, even if an object is passed from an activity A to an activity B, we still need to model the sequence flow from A to B. A shorthand notation for passing an object from an activity to the other is by directly connecting the data object to the sequence flow between two consecutive activities via an undirected association. See for example the Shipment address being passed from activity "Get order" to "Ship product".

Example: artefacts

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

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

Example: artefacts

artifacts provide additional information, but can compromise diagram readability
Call activities
Advanced Process Modeling

Fig. 4.3

A process model for disbursing home loans, laid down over three hierarchical levels via process model connections, the number of parallel branches, the longest path from a start to an end event, as well as cosmetic aspects such as the layout, the labels style (e.g. always use a verb-noun style), the colors palette, the lines thickness, etc. More information on establishing process modeling guidelines can be found in Chap. 5.

We have shown that we can simplify a process model by first identifying the content of a sub-process, and then hiding this content by collapsing the sub-process activity. Sometimes, we may wish to proceed in the opposite direction, meaning that when modeling a process we already identify activities that can be broken down in smaller steps, but we intentionally under-specify their content. In other words, we do not link the sub-process activity to a process model at a lower level capturing its content (like if by pressing the plus button nothing would happen). The reason for doing this is to tell the reader that some activities are made up of sub-steps, but that disclosing the details of these is not relevant. This could be the case of activity “Ship product” in the order fulfillment example, for which modeling the distinction between its internal steps for packaging and for shipping is not relevant.

4.2 Process Reuse

By default a sub-process is embedded within its parent process model, and as such it can only be invoked from within that process model. Often, when modeling a business process we may need to reuse parts of other process models of the same organization. For example, a loan provider may reuse the sub-process for signing home loans.

Nesting sub-processes:
Global sub-processes: home / student loans

suppose the "Sign loan" process is defined as a separate model: it can be reused
Call activities

A Call Activity is a wrapper for a globally defined Sub-Process or Task that is reused in the current process.
4.2 Process Reuse 101

Fig. 4.4
The process model for disbursing student loans invokes the same model for signing loans used by the process for disbursing home loans, via a call activity.

In BPMN, we can define the content of a sub-process outside its parent process, by defining the sub-process as a global process model. A global process model is a process model that is not embedded within any other process model, and such a call can be invoked by other process models within the same process model collection. To indicate that the sub-process being invoked is a global process model, we use the collapsed sub-process activity with a thicker border (this activity type is called call activity in BPMN). Coming back to the loan disbursement example of Fig. 4.3, we can factor out the sub-process for signing loans and define it as a global process model, so that it can also be invoked by a process model for disbursing student loans (see Fig. 4.4).

Question: Embedded or global sub-process?

Our default choice should be to define sub-processes as global process models so as to maximize their reusability within our process model collection. Supporting processes such as payment, invoicing, HR, printing, are good candidates for being defined as global process models, since they are typically shared by various business processes within an organization. Besides reusability, another advantage of using global process models is that any change made to these models will be automatically propagated to all process models that invoke them. In some cases, however, we may want to keep the changes internal to a specific process. For example, an invoicing process used for corporate orders settlement would typically be different.

Call activities:

- Home / Student loans

thick borders denote call activities (to global sub-processes)
Global processes: advantages

Readability: processes tend to be smaller

Reusability: define once, use many time

Sharing: any change made to a global process is automatically propagated to all models that invoke it
Attached events

Attached Intermediate Event: The activity is aborted once an event is caught.
Recovery from faults: image manipulation

The lightning annotation denotes an error-catching event
Throwing and catching:
order fulfillment

The lightning annotation denotes a throwing event: it models an out-of-stock exception.
Throwing and catching: order fulfillment

The error event is depicted as an event with a lightning marker. Following the BPMN conventions for throwing and catching events, the lightning is empty for the catching intermediate event and full for the end throwing event.

An example of error events is shown in Fig. 4.19 in the context of our order fulfillment process. If there is an out of stock exception, the acquisition of raw materials is interrupted and the recovery procedure is triggered, which in this case simply consists of a task to notify the customer before aborting the process. In terms of token semantics, upon throwing an end error event, all tokens are removed from the enclosing sub-process (causing its interruption), and one token is sent through the exception flow emanating from the boundary error event. There is no restriction on the modeling elements we can put in the exception flow to model the recovery procedure. Typically, we would complete the exception flow with an end terminate event to abort the process, or wire this flow back to the normal sequence flow if the exception has been properly handled.

4.5.3 External Exceptions

An exception may also be caused by an external event occurring during an activity. For example, while checking the stock availability for the product in a purchase order, the Seller may receive an order cancellation from the customer. Upon this request, the Seller should interrupt the stock availability check and handle the order cancellation. Scenarios like the above are called unsolicited exceptions since they originate externally to the process. They can be captured by attaching a catching intermediate message event to an activity's boundary, as shown in Fig. 4.20. From token semantics, when the intermediate message event is triggered, the token is removed from the enclosing activity, consequently causing the activity interruption, and sent through the exception flow emanating from the boundary event, to perform the recovery procedure.

end terminate event: causes the immediate cessation of the current process instance (and of any sub-process, but not of the parent process if any)
Choreographies
Choreography

A **choreography** defines the sequence of interaction between participants.

A choreography does not exist in a pool and it is not executable.

It describes how the participants are supposed to behave.

A choreography can also use message data objects.
Choreography task

A choreography task is an activity in a choreography that consists of a set (one or more) communications.

A choreography task involves two or more participants that are displayed in different bands.

top/bottom band positioning is inessential.
Choreography flow

Ordinary sequence flow and gateways are used within choreographies to show the sequence of tasks involved.

Sequence Flow will define the order of Choreography elements.

The initiator of the second interaction must be involved in the previous one.
12.2 Data

A Choreography does not have a central control mechanism and, thus, there is no mechanism for maintaining any central Process (Choreography) data. Thus, any element in a Process that would normally depend on conditional or assignment expressions, would not have any central source for this data to be maintained and understood by all the Participants involved in the Choreography.

As mentioned above, neither Data Objects nor Repositories are used in Choreographies. Both of these elements are used exclusively in Processes and require the concept of a central locus of control. Data Objects are basically variables and there would be no central system to manage them.

Data can be used in expressions that are used in Exclusive Gateways, but only that data which has been sent through a Message in the Choreography.

12.3 Use of BPMN Common Elements

Some BPMN elements are common to both Process and Choreography diagrams, as well as Collaboration; they are used in these diagrams. The next few sections will describe the use of Messages, Message Flow, Participants, Sequence Flow, Artifacts, Correlations, Expressions, and Services in Choreography.

The key graphical elements of Gateways and Events are also common to both Choreography and Process. Since their usage has a large impact, they are described in major sections of this chapter (see page 369 for Events and page 375 for Gateways).
BPMN Semantics
BPMN execution semantics
The execution semantics are described informally (textually), and this is based on prior research involving the formalization of execution semantics using mathematical formalisms.

A Process is instantiated when one of its Start Events occurs.

A Process can also be started via an Event-Based Gateway or a Receive Task that has no incoming Sequence Flows.

Each Start Event that occurs creates a token on its outgoing Sequence Flows, which is followed as described by the semantics of the other Process elements.

A Process instance is completed, if and only if the following three conditions hold:

• If the instance was created through an instantiating Parallel Gateway, then all subsequent Events (of that Gateway) MUST have occurred.
• There is no token remaining within the Process instance.
• No Activity of the Process is still active.

For a Process instance to become completed, all tokens in that instance MUST reach an end node.

A token reaching an End Event triggers the behavior associated with the Event type. If a token reaches a Terminate End Event, the entire Process is abnormally terminated.
BPMN formal semantics?

Many attempts:
Abstract State Machines (ASM)
Term Rewriting Systems
Graph Rewrite Systems
Process Algebras
Temporal Logic

... Petri nets

(Usual difficulties with OR-join semantics)
Sound BPMN diagrams

We can exploit the formal semantics of nets to give unambiguous semantics to BPMN process diagrams BPMN collaboration diagrams

We transform BPMN process diagrams to wf nets BPMN collaboration diagrams to wf systems

A BPMN diagram is sound if its net is so

We can reuse the verification tools to check if the net is sound
Translation of BPMN to Petri nets
Abstract

The Business Process Modelling Notation (BPMN) is a standard for capturing business processes in the early phases of systems development. The mix of constructs found in BPMN makes it possible to create models with semantic errors. Such errors are especially serious, because errors in the early phases of systems development are among the most costly and hardest to correct. The ability to statically check the semantic correctness of models is thus a desirable feature for modelling tools based on BPMN. Accordingly, this paper proposes a mapping from BPMN to a formal language, namely Petri nets, for which efficient analysis techniques are available. The proposed mapping has been implemented as a tool that, in conjunction with existing Petri net-based tools, enables the static analysis of BPMN models. The formalisation also led to the identification of deficiencies in the BPMN standard specification.

Keywords: Business process modelling and analysis; BPMN; Petri nets

1. Introduction

The Business Process Modelling Notation (BPMN) [17] is a standard notation for capturing business processes, especially at the level of domain analysis and high-level systems design. The notation inherits and combines elements from a number of previously proposed notations for business process modelling, including the XML Process Definition Language (XPDL) [21] and the Activity Diagrams component of the Unified Modelling Notation (UML) [16]. BPMN process models are composed of: (i) activity nodes, denoting business events or items of work performed by humans or by software applications and (ii) control nodes capturing the flow of control between activities. Activity nodes and control nodes can be connected by means of a flow relation in almost arbitrary ways.

Languages that follow a similar paradigm, known as graph-oriented process definition languages, have been previously studied from a formal perspective (e.g., the work on task structures [2]). It is known that models defined in this family of languages may exhibit a range of semantic errors, including deadlocks and livelocks. Such errors are especially problematic at the levels of domain analysis and high-level systems design, because errors at these levels are among the hardest and most costly to correct. BPMN even increases the types of semantic errors with respect to traditional graph-oriented languages, because it combines graph-oriented features with other features, drawn from a range of sources including Workflow Patterns [5] and Business Process Execution Language (BPEL) [12], a standard for defining business processes at the implementation level. These features include the ability to define: (i) subprocesses that may be executed multiple times concurrently; (ii) subprocesses that may be interrupted as a result of exceptions; and (iii) message flows between processes. The interactions between these features are an additional source of semantic errors.

For these reasons the ability to statically analyse BPMN models is likely to become a desirable feature for tools supporting process modelling in BPMN. Anecdotal evidence
Simplified BPMN

a start / exception event has just one outgoing flow and no incoming flow

an end event has just one incoming flow and no outgoing flow

all activities and intermediate events have exactly one incoming flow and one outgoing flow

all gateways have either one incoming flow (and multiple outgoing) or one outgoing flow (and multiple incoming)
Simplified BPMN

The previous constraints are no real limitation:

- events or activities with multiple incoming flows: insert a preceding XOR-join gateway
- events or activities with multiple outgoing flows: insert a following AND-split gateway
- gateways with multiple incoming and outgoing flows: decompose in two gateways
- insert start / end events if needed
Pay attention to gateways

Stands for

Stands for

Stands for

Stands for
My suggestions

stands for

stands for

stands for

stands for
Simplified BPMN

Avoid OR-gateways
(all problems seen with EPC apply to BPMN as well)

Limited form of sub-processing

No transactions and compensations
The twist!

BPMN object

- Event
- Activity
- Sequence flow
- Message flow

Net fragment

- Transition
- Place
Roughly

A place for each arc

one transitions for each event

one transition for each activity

one or two transitions for each gateway

... with some exceptions!

(start event, end event, event-based gateways, loops, ...)

no dummy objects!
The strategy

From BPMN process diagrams to wf nets in three steps

<table>
<thead>
<tr>
<th>Travel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
</tr>
<tr>
<td>Travel request</td>
</tr>
<tr>
<td>Book flight</td>
</tr>
<tr>
<td>Book hotel</td>
</tr>
<tr>
<td>Travel planned</td>
</tr>
</tbody>
</table>

Step 1: convert sequence flow message flow

Step 2: convert flow objects

Step 3: enforce initial place final place
Step 1: convert flows

We insert a place for each sequence flow and message flow.
Step 2: convert flow objects

Then insert transitions

Step 1
sequence flow
message flow

Step 2
flow objects
Step 2: gateways

BPMN object

- AND split / join
- XOR split
- XOR join

net fragment

transition
Step 2: event-based

BPMN object

Step 1
sequence flow
message flow

Step 2
place fusion

net fragment
Step 3: add unique start
Step 3: add unique end

Steps 1+2 → Step 3

XOR end
(sometimes AND can be preferred)
Example:
Order process
Order process

Sound?
Order process: step 1

Step 1 sequence flow message flow
Order process: step 2

Step 2 flow objects
Order process: (desugar)
Order process: step 3

Step 3
enforce
initial place
final place
Soundness analysis

Not sound!
Soundness analysis
Soundness analysis

Not sound!
Example: Travel itinerary
Travel itinerary

Sound?
Travel itinerary: step 1

Step 1 sequence flow
message flow
Travel itinerary: step 2

Step 2 flow objects

Draft itinerary

Confirm itinerary

Discuss itinerary

Change itinerary

160
Travel itinerary: (desugar)
Travel itinerary: step 3

Step 3 enforce initial place final place
Soundness analysis

Not sound!
Soundness analysis
Example:
Always sushi
Always sushi

Sound?
Sushi lover

Sound?
Sushi lover: step 1

Step 1
sequence flow
message flow
Sushi lover: step 2

Step 2

flow objects
Sushi lover: (desugar)
Sushi lover: step 3

Step 3
enforce
initial place
final place
Soundness analysis

safe & sound
(s-net)
Sushi doomed

Sound?
Sushi doomed: step 1

Step 1 sequence flow message flow
Sushi doomed: step 2

Step 2
flow objects
Sushi doomed: (desugar)
Sushi doomed: step 3

Step 3
enforce
initial place
final place
Soundness analysis

safe & sound
(s-net)
Sushi system

Sound?

Sushi doomed
Sushi system: step 1
Sushi system: step 1+2+3
Soundness analysis

Sound!
Step 0: preprocessing BPMN diagrams
Overview

EVENT
- start
- start message
- message
- timer
- error
- end message
- end

ACTIVITY
- Task
- Sub-process Invocation Activity
- Activity Looping
- Multiple Instance

GATEWAY
- Parallel Fork Gateway
- Parallel Join Gateway
- Data-based XOR Decision Gateway
- XOR Merge Gateway
- Event-based XOR Decision Gateway
- OR Decision Gateway

SEQUENCE FLOW
- Normal Flow
- Exception Flow

MESSAGE FLOW
- Message Flow

[Note]:
1. Apart from intermediate error events, intermediate message or timer events may also be the source of exception flows.
2. A message flow may link task to task, end event to task, task to start event, and end event to start event.
Activity looping

(a) “while-do” loop

(b) “do-until” loop
Multiple instances (design-time bounded)
Sub-processes

3.4. Exception handling

In BPMN, exception handling is captured by exception flows. An exception flow originates from an error event attached to the boundary of an activity. For presentation purposes, it is convenient to distinguish the case where the activity is a single task, from the case where it is a subprocess.

Fig. 8 shows the mapping of an error event associated with a task. Given that the execution of task $T$ is atomic, the occurrence of exception $E_x$ may only interrupt $T$ when $T$ is enabled and has not yet completed. In Petri net terms, this means that the occurrence of exception $E_x$ can "steal" the input token that would normally be consumed by the transition corresponding to task $T$.

In the case of an exception flow associated to a subprocess, the occurrence of the exception (i.e., the error event) will cancel the execution of the subprocess assuming that this latter has started but has not yet completed. The mapping is complicated by the fact that it needs to capture the cancellation of the running subprocess at any point when the exception occurs. This means that when the transition corresponding to the error event fires, all the tokens left in the Petri net fragment corresponding to the subprocess need to be removed. However, due to the local nature of Petri net transitions, it is cumbersome to model a "vacuum cleaner" that would remove all tokens from a given fragment of a net [3].
Exception handling: single task

In BPMN, exception handling is captured by exception flows. An exception flow originates from an error event attached to the boundary of an activity. For presentation purposes, it is convenient to distinguish the case where the activity is a single task, from the case where it is a subprocess.

Fig. 8 shows the mapping of an error event associated with a task. Given that the execution of task $T$ is atomic, the occurrence of exception $E_x$ may only interrupt $T$ when $T$ is enabled and has not yet completed. In Petri net terms, this means that the occurrence of exception $E_x$ can ''steal'' the input token that would normally be consumed by the transition corresponding to task $T$.

In the case of an exception flow associated to a subprocess, the occurrence of the exception (i.e., the error event) will cancel the execution of the subprocess assuming that this latter has started but has not yet completed. The mapping is complicated by the fact that it needs to capture the cancellation of the running subprocess at any point when the exception occurs. This means that when the transition corresponding to the error event fires, all the tokens left in the Petri net fragment corresponding to the subprocess need to be removed. However, due to the local nature of Petri net transitions, it is cumbersome to model a ''vacuum cleaner'' that would remove all tokens from a given fragment of a net [3].

Fig. 4. Macro expansions for repeated activities.
Fig. 5. Macro expansion for a multi-instance activity where $n$ is known at design time.
Fig. 6. Mapping of a subprocess without exception handling.
Fig. 7. Calling a subprocess via a subprocess invocation activity.
Fig. 8. Mapping of a task with an exception flow.
Exception handling: sub-processes

accounts for separate execution of multiple instances
Exercises

Model the following fragments of business processes for assessing loan applications:
Exercise: loan application 1

Once a loan application has been approved by the loan provider, an acceptance pack is prepared and sent to the customer.

The acceptance pack includes a repayment schedule which the customer needs to agree upon by sending the signed documents back to the loan provider.

The latter then verifies the repayment agreement:
if the applicant disagreed with the repayment schedule, the loan provider cancels the application;
if the applicant agreed, the loan provider approves the application.
In either case, the process completes with the loan provider notifying the applicant of the application status.
A loan application is approved if it passes two checks: (i) the applicant’s loan risk assessment, which is done automatically by a system, and (ii) the appraisal of the property for which the loan has been asked, carried out by a property appraiser.

The risk assessment requires a credit history check on the applicant, which is performed by a financial officer.

Once both the loan risk assessment and the property appraisal have been performed, a loan officer can assess the applicant’s eligibility.

If the applicant is not eligible, the application is rejected, otherwise the acceptance pack is prepared and sent to the applicant.
A loan application may be coupled with a home insurance which is offered at discounted prices. The applicant may express their interest in a home insurance plan at the time of submitting their loan application to the loan provider.

Based on this information, if the loan application is approved, the loan provider may either only send an acceptance pack to the applicant, or also send a home insurance quote.

The process then continues with the verification of the repayment agreement.
Exercise: loan application 4

Once a loan application is received by the loan provider, and before proceeding with its assessment, the application itself needs to be checked for completeness.

If the application is incomplete, it is returned to the applicant, so that they can fill out the missing information and send it back to the loan provider.

This process is repeated until the application is complete.
Exercise: loan application 5

Put together the four fragments of the loan assessment process that you created in previous Exercises.

Then extend the resulting model by adding all the required artifacts.

Moreover, attach annotations to specify the business rules behind:
(i) checking an application completeness,
(ii) assessing an application eligibility, and
(iii) verifying a repayment agreement.
Exercise: loan application 6

Extend the business process for assessing loan applications that you created in previous exercises by considering the following resource aspects.

The process for assessing loan applications is executed by four roles within the loan provider:
- a **financial officer** takes care of checking the applicant’s credit history;
- a **property appraiser** is responsible for appraising the property;
- an **insurance sales representative** sends the home insurance quote to the applicant if this is required.
All other activities are performed by the **loan officer** who is the main point of contact with the applicant.
Exercises: loan application 7

Extend the loan application model by representing the interactions between the loan provider and the applicant.