Formal semantics and analysis of BPMN process models

Marlon Dumas

Formal Semantics and Analysis of BPMN Process Models using Petri Nets*

Remco M. Dijkman[1], Marlon Dumas[2], and Chun Ouyang[2]

[1] Department of Technology Management, Eindhoven University of Technology,
GPO Box 513, NL-5600 MB, The Netherlands
{r.m.dijkman}@tm.tue.nl

[2] Faculty of Information Technology, Queensland University of Technology,
GPO Box 2434, Brisbane QLD 4001, Australia
{m.dumas,c.ouyang}@qut.edu.au

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 obtain models with a range of semantic errors. The ability to statically check the semantic correctness of models is thus a desirable feature for modelling tools based on BPMN. However, the static analysis of BPMN models is hindered by ambiguities in the standard specification and the complexity of the language. The fact that BPMN integrates constructs from graph-oriented process definition languages with features for concurrent execution of multiple instances of a subprocess and exception handling, makes it challenging to provide a formal semantics of BPMN. Even more challenging is to define a semantics that can be used to analyse BPMN models. This paper proposes a formal semantics of BPMN defined in terms of a mapping to Petri nets, for which efficient analysis techniques exist. The proposed mapping has been implemented as a tool that generates code in the Petri Net Markup Language. This formalisation exercise has also led to the identification of a number of deficiencies in the BPMN standard specification.

Keywords: Business process modelling, BPMN, Petri nets, System verification using nets.

1 Introduction

The Business Process Modeling Notation (BPMN) [12] has emerged as 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 modeling, including the XML Process Definition Language (XPDL) [17] and the Activity Diagrams component of the Unified Modeling Notation (UML) [11].

* This work is partially supported by the Australian Research Council under Discovery Grant DP0451092. The second author was also funded by a Smart State Fellowship co-sponsored by Queensland Government and SAP.

Like these predecessors, a key idea of BPMN is that 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 can exhibit a range of semantic errors, including deadlocks and livelocks. Furthermore, BPMN brings additional features not traditionally associated with graph-oriented languages, drawn from a range of sources including Workflow Patterns [5] and Business Process Execution Language (BPEL) [6], 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 interaction between these features and the more traditional features found in graph-oriented languages, increases the types of semantic errors that can be found in BPMN models. Hence, the ability to statically analyse BPMN models is likely to become a desirable feature for tools supporting process modelling in BPMN. Anecdotal evidence suggests that BPMN users are producing models with semantic errors that could be detected using existing verification technology.[3]

The semantic analysis of BPMN models is hindered by the heterogeneity of its constructs and the lack of an unambiguous definition of the notation. While syntactic rules are comprehensively documented in tables throughout the BPMN standard specification, the actual semantics is only described in narrative form using sometimes inconsistent terminology. This paper takes on the challenge of defining a formal semantics for a large subset of BPMN. The semantics is defined as a mapping between BPMN models and Petri nets. The choice of using plain Petri nets as a target for the mapping is motivated by the availability of efficient static analysis techniques. Thus, the proposed mapping not only serves the purpose of disambiguating the core constructs of BPMN, but it also provides a foundation to statically check the semantic correctness of BPMN models. To support this claim, we have implemented a tool that translates between the XML serialization of BPMN models supported by an existing BPMN tool, and the Petri Net Markup Language (PNML). The paper reports experiences in importing the resulting BPMN models into a Petri net analysis toolset for the purpose of performing semantic analysis.

The paper focuses on the control-flow perspective of BPMN, that is, the subset of the notation that deals with the order in which activities and events are allowed to occur. It does not deal with its non-functional features (i.e. artifacts, groups and associations) and organisational modeling features (i.e. lanes and pools). Also, the proposed mapping is specifically designed to produce Petri nets that are suitable for static analysis.

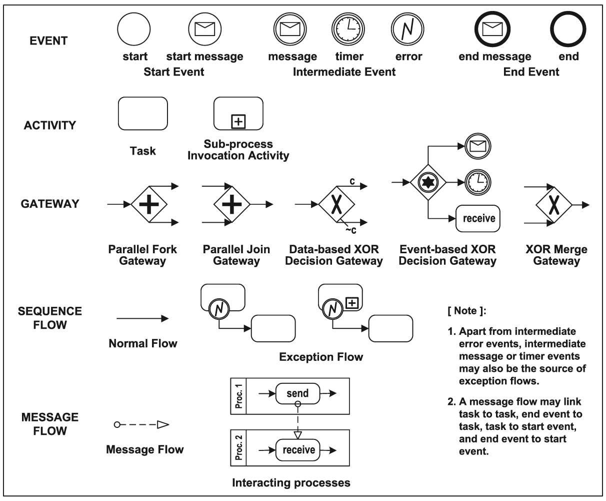
[3] See `www.brsilver.com/wordpress/2006/09/06/whats-wrong-with-this-picture/`.

The rest of the paper is structured as follows. Section 2 provides an overview of BPMN and introduces an abstract syntax capturing the essence of the notation. Section 3 presents a mapping from BPMN to Petri nets, and a mathematical definition of the mapping is given in Section 4. Section 5 addresses a number of deficiencies identified during the formalisation. Section 6 reports the corresponding tool implementation and its application to static analysis of BPMN models as well as the tool evaluation. Finally, related work is discussed in Section 7 while conclusions and future work are outlined in Section 8. A meta-model of BPMN defined during the tool implementation is included in an appendix.

2 Business Process Modeling Notation (BPMN)

2.1 Overview

BPMN essentially provides a graphical notation for business process modelling, with an emphasis on control-flow. It defines a *Business Process Diagram* (BPD), a kind of flowchart incorporating constructs tailored to business process modelling, such as AND-split, AND-join, XOR-split, XOR-join, and deferred (event-based) choice. A BPD is made up of BPMN elements. We consider a core subset of BPMN elements shown in Figure 1. There are *objects* and *sequence flows*. An object can be an *event*, an *activity* or a *gateway*. A sequence flow links two objects in a BPD and shows the control flow relation (i.e. execution order).



[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.

Figure 1. A core subset of BPMN elements.

An event may signal the start of a process (*start event*), the end of a process (*end event*), and may also occur during the process (*intermediate event*). A *message event* is used to send (not for *start event*) or receive (not for *end event*) a message. A *timer event* indicates a specific time-date being reached, and an *error event* signals an error being detected during a process.

An activity can be a *task* or a *subprocess*. A task is an atomic activity and stands for work to be performed within a process. There are seven task types: *service*, *receive*, *send*, *user*, *script*, *manual*, and *reference*. For example, a receive task is used when the process waits for a message to arrive from an external partner. A subprocess is a compound activity in that it is defined as a flow of other activities. There are *embedded* subprocesses and *independent* subprocesses. The difference is that an embedded subprocess is part of the process while an independent subprocess can be called by different processes. A subprocess can be invoked via a *subprocess invocation activity*.[4]

A gateway is a routing construct used to control the divergence and convergence of sequence flow. There are: *parallel fork gateways* (AND-split) for creating concurrent sequence flows, *parallel join gateways* (AND-join) for synchronizing concurrent sequence flows, *data/event-based XOR decision gateways* for selecting one out of a set of mutually exclusive alternative sequence flows where the choice is based on either the process data (data-based, i.e. XOR-split) or external event (event-based, i.e. deferred choice), and *XOR merge gateways* (XOR-join) for joining a set of mutually exclusive alternative sequence flows into one sequence flow. In particular, an event-based XOR decision gateway must be followed by either receive tasks or intermediate events to capture race conditions based on timing or external triggers (e.g. the receipt of a message from an external partner).

An intermediate message, timer, or error event attached to the boundary of an activity signals an exception. We use the term "*exception event*" to refer to such an event. The occurrence of the activity will be interrupted upon the occurrence of the exception, and the process execution along the normal sequence flow will switch to the exception flow at the point when exception occurs. Note that an intermediate error event on a normal sequence flow models "throwing" an error, while an error event attached on the boundary of the activity models "catching" an error. This is similar to the strictly hierarchical throw-catch mechanism used in most programming languages (e.g. Java).

Finally, a message flow is used to show transmission of messages between two interacting processes via communication actions such as send/receive task or message event. The two processes are located respectively within two separate *pools* (rectangles labelled with process names), representing two participants (e.g., business entities or business roles). In graphical representation, a message flow is drawn as a dashed line with an open arrowhead connected to the target process and a circle connected to the source process, and a pool is drawn as a rectangle labelled with the process name.

2.2 Abstract Syntax of BPMN

A BPMN process, which is described by a BPD using the core subset of BPMN elements shown in Figure 1, is referred to as a core BPMN process. First, we define the syntax of a core BPMN process.

[4] This is called a *collapsed subprocess* in the current BPMN specification [12].

Definition 1 (Core BPMN Process). *A core BPMN process is a tuple* $\mathcal{P} = (\mathcal{O}, \mathcal{A}, \mathcal{E}, \mathcal{G}, \mathcal{T}, \mathcal{S}, \mathcal{T}^R, \{e^S\}, \mathcal{E}^I, \{e^E\}, \mathcal{E}^{I_M}, \mathcal{E}^{I_T}, \mathcal{E}^{I_R}, \mathcal{G}^F, \mathcal{G}^J, \mathcal{G}^X, \mathcal{G}^M, \mathcal{G}^V, \mathcal{F}, \mathsf{Cond}, \mathsf{Excp})$ *where:*

- $\mathcal{O}$ *is a set of objects which can be partitioned into disjoint sets of activities* $\mathcal{A}$*, events* $\mathcal{E}$*, and gateways* $\mathcal{G}$*,*
- $\mathcal{A}$ *can be partitioned into disjoint sets of tasks* $\mathcal{T}$ *and subprocess invocation activities* $\mathcal{S}$*,*
- $\mathcal{T}^R \subseteq \mathcal{T}$ *is a set of receive tasks,*
- $\mathcal{E}$ *can be partitioned into disjoint sets of start event* $\{e^S\}$*, intermediate events* $\mathcal{E}^I$*, and end event* $\{e^E\}$*,*[5]
- $\mathcal{E}^I$ *can be partitioned into disjoint sets of intermediate message events* $\mathcal{E}^{I_M}$*, intermediate timer events* $\mathcal{E}^{I_T}$*, and intermediate error events* $\mathcal{E}^{I_R}$*,*
- $\mathcal{G}$ *can be partitioned into disjoint sets of parallel fork gateways* $\mathcal{G}^F$*, parallel join gateways* $\mathcal{G}^J$*, data-based XOR decision gateways* $\mathcal{G}^X$*, event-based XOR decision gateways* $\mathcal{G}^V$*, and XOR merge gateways* $\mathcal{G}^M$*,*
- $\mathcal{F} \subseteq \mathcal{O} \times \mathcal{O}$ *is the control flow relation, i.e. a set of sequence flows connecting objects,*
- Cond*:* $\mathcal{F} \cap (\mathcal{G}^X \times \mathcal{O}) \to \mathcal{C}$ *is a function which maps sequence flows emanating from data-based XOR gateways to conditions,*[6]*, and*
- Excp*:* $\mathcal{E}^I \nrightarrow \mathcal{A}$ *is a function assigning an intermediate event to an activity such that the occurrence of the event signals an exception and thus interrupts the performance of the activity.*

A core BPMN process is a directed graph with nodes (objects) $\mathcal{O}$ and arcs (sequence flows) $\mathcal{F}$. For any node $x \in \mathcal{O}$, input nodes of x are given by $\mathsf{in}(x) = \{y \in \mathcal{O} \mid y\mathcal{F}x\}$ and output nodes of x are given by $\mathsf{out}(x) = \{y \in \mathcal{O} \mid x\mathcal{F}y\}$. Also, for a core BPMN process $\mathcal{P}$, if ambiguity is possible, we use $\mathcal{P}$ as subscripts to each element defined in the tuple $\mathcal{P}$. For example, $\mathcal{S}_\mathcal{P}$ refers to the set of subprocess invocation activities in $\mathcal{P}$. Next, we define the syntax of a core BPMN model which may comprise a set of core BPMN processes.

Definition 2 (Core BPMN Model). *A core BPMN model is a tuple* $\mathcal{M} = (\mathcal{Q}, \mathcal{S}^\diamond, \mathsf{map}, \mathsf{HR}, \mathcal{F}^M)$ *where:*

- $\mathcal{Q}$ *is a set of core BPMN processes,*
- $\mathcal{S}^\diamond = \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{S}_\mathcal{P}$ *is the set of all subprocess invocation activities,*
- map*:* $\mathcal{S}^\diamond \to \mathcal{Q}$ *is an injective function which maps each subprocess invocation activity to a core BPMN process, and*
- $\mathsf{HR} = \{(\mathcal{P}_1, \mathcal{P}_2) \in \mathcal{Q} \times \mathcal{Q} \mid \exists_{s\in\mathcal{S}_{\mathcal{P}_1}} \mathsf{map}(s) = \mathcal{P}_2\}$ *is a connected graph,*

[5] Since the behaviour of a BPMN process with multiple start/end events is not clear in the current BPMN specification (see further discussion in Section 5), we have to restrict to a process with a single start event and a single end event only.

[6] $\mathcal{C}$ is the set of all possible conditions. A condition is a boolean function operating over a set of propositional variables. Note that we abstract from these variables in the control flow definition. We simply assume that a condition evaluates to true or false, which determines whether or not the associated sequence flow is taken during the process execution.

- $\mathcal{F}^M \subseteq (\cup_{\mathcal{P}\in\mathcal{Q}}(\mathcal{T}_\mathcal{P} \cup \mathcal{E}^E_\mathcal{P} \cup \mathcal{E}^{I_M}_\mathcal{P}) \times \cup_{\mathcal{P}\in\mathcal{Q}}(\mathcal{T}_\mathcal{P} \cup \mathcal{E}^S_\mathcal{P} \cup \mathcal{E}^{I_M}_\mathcal{P})) \setminus \cup_{\mathcal{P}\in\mathcal{Q}}(\mathcal{O}_\mathcal{P} \times \mathcal{O}_\mathcal{P})$ *is the set of message flows between processes.*

Remark 1. The function map is defined as an injective function capturing directly the concept of embedded subprocesses. For independent subprocesses, it is possible to call the same subprocess via different subprocess invocation activities in a BPMN model. In this case, we can generate multiple copies of the subprocess, assign different names to them, and then add them into the set $\mathcal{Q}$. This way the function map also captures the concept of independent subprocesses. Also note that BPMN does not forbid recursive calls, e.g., a process $\mathcal{P}_1$ invoking another process $\mathcal{P}_2$, and $\mathcal{P}_2$ invoking $\mathcal{P}_1$. Thus, the relationship between a process and its subprocesses is an arbitrarily connected graph, as opposed to a tree or a directed acyclic graph (DAG).

Definition 1 allows for graphs which are unconnected, not having start or end events, containing objects without any input and output, etc. Therefore we need to restrict the definition to *well-formed core BPMN processes* and *models*. Note that these restrictions are without loss of generality and are to facilitate the definition of the mapping. It is possible to re-write, for example, an activity with multiple incoming arcs into an activity with only one incoming arc and preceded by an XOR merge gateway.

Definition 3 (Well-formed core BPMN Process). *A core BPMN process $\mathcal{P}$ in Definition 1 is well formed if relation $\mathcal{F}$ satisfies the following requirements:*

- $\forall\ s \in \mathcal{E}^S \cup dom(\mathsf{Excp}),\ \mathsf{in}(s) = \varnothing \wedge \mid \mathsf{out}(s) \mid = 1$, *i.e. start events and exception events have an indegree of zero and an outdegree of one,*
- $\forall\ e \in \mathcal{E}^E$, $\mathsf{out}(e) = \varnothing \wedge \mid \mathsf{in}(e) \mid = 1$, *i.e., end events have an outdegree of zero and an indegree of one,*
- $\forall\ x \in \mathcal{A} \cup (\mathcal{E}^I \setminus dom(\mathsf{Excp}))$, $\mid \mathsf{in}(x) \mid = 1$ *and* $\mid \mathsf{out}(x) \mid = 1$, *i.e. activities and non-exception intermediate events have an indegree of one and an outdegree of one,*
- $\forall\ g \in \mathcal{G}^F \cup \mathcal{G}^X \cup \mathcal{G}^V$: $\mid \mathsf{in}(g) \mid = 1 \wedge \mid \mathsf{out}(g) \mid > 1$, *i.e. fork or decision gateways have an indegree of one and an outdegree of more than one,*
- $\forall\ g \in \mathcal{G}^J \cup \mathcal{G}^M$, $\mid \mathsf{out}(g) \mid = 1 \wedge \mid \mathsf{in}(g) \mid > 1$, *i.e. join or merge gateways have an outdegree of one and an indegree of more than one,*
- $\forall\ g \in \mathcal{G}^V$, $\mathsf{out}(g) \subseteq \mathcal{E}^{I_M} \cup \mathcal{E}^{I_T} \cup \mathcal{T}^R$, *i.e. event-based XOR decision gateways must be followed by intermediate message or timer events or receive tasks,*
- $\forall\ g \in \mathcal{G}^X$, $\exists$ *an order* $<_g$ *which is a strict total order over the set of flows* $\{g\} \times \mathsf{out}(g)$, *and* $\exists\ x \in \mathsf{out}(g)$ *such that* $\neg\ \exists_{f\in\{g\}\times\mathsf{out}(g)}((g,x)<_g f)$, *i.e.* (g,x) *is the default flow among all the outgoing flows from* g,
- $\forall\ x \in \mathcal{O}$, $\exists\ s \in \mathcal{E}^S \cup dom(\mathsf{Excp})$, $\exists\ e \in \mathcal{E}^E$, $s\mathcal{F}^* x \wedge x\mathcal{F}^* e$,[7] *i.e. every object is on a path from a start event or an execption event to an end event.*

Definition 4 (Well-formed core BPMN Model). *A core BPMN model $\mathcal{M}$ given in Definition 2 is well-formed, iff $\mathcal{Q}$ is a set of well-formed core BPMN processes and the relation HR is a DAG.*

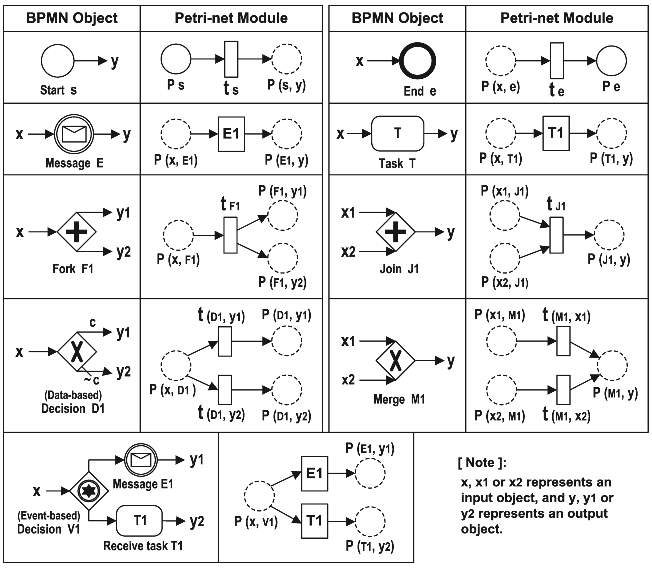
[7] $\mathcal{F}^*$ is a reflexive transitive closure of $\mathcal{F}$, i.e. $x\mathcal{F}^*y$ if there is a path from x to y and by definition $x\mathcal{F}^*x$.

3 Mapping BPMN onto Petri Nets

We establish a mapping of well-formed core BPMN models to Petri nets. We allow the usage of both labelled and non-labelled transitions. The labelled transitions model tasks and events. The transitions without a label ("silent" transitions) capture internal actions that cannot be observed by external users.

3.1 Task, Events, and Gateways

Figure 2 depicts the mapping from a set of BPMN task, events, and gateways to Petri-net modules. A task or an intermediate event is mapped onto a transition with one input place and one output place. The transition, being labelled with the name of that task or event, models the execution of the task or event. A start or end event is mapped onto a similar module except that a silent transition is used to signal when the process starts or ends.



[Note]:
x, x1 or x2 represents an input object, and y, y1 or y2 represents an output object.

Figure 2. Mapping task, events, and gateways onto Petri-net modules.

Gateways, except event-based decision gateways, are mapped onto small Petri-net modules with silent transitions capturing their routing behaviour. These mappings, as shown in Figure 2, are straightforward. For an event-based gateway, the race condition between events or receive tasks is captured in a way that all the corresponding event/task transitions compete for the token available in the output place from the mapping of the gateway's input object. Finally, places, which are drawn in dashed borders, indicate that their usage is not unique to one module. They are used to link the Petri net modules of two connecting BPMN objects and therefore are identified by both objects. Generally, any sequence flow is mapped onto a place except for event-based decision gateways.

3.2 Subprocess

A subprocess may be viewed as an independent BPMN process. Without considering exception handling, the mapping of a subprocess is straightforward, as shown in Figure 3. Figure 4 depicts the mapping of calling a subprocess ($\mathcal{P}$) via a subprocess invocation activity (SI). The two places drawn in dashed borders capture respectively the incoming and the outgoing flows of activity SI. There are also two new transitions: one with an identifier $t_{(SI,call)}$ modelling the invocation of the subprocess $\mathcal{P}$, the other with $t_{(SI,return)}$ modelling the return of the flow to the parent process after $\mathcal{P}$ has completed.

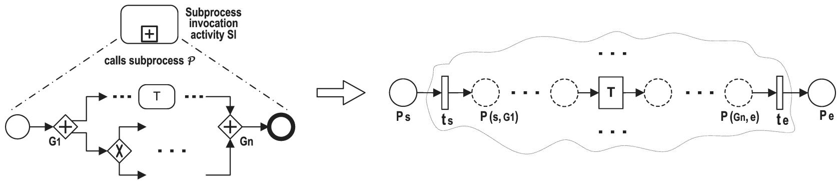

Subprocess invocation activity SI
calls subprocess $\mathcal{P}$
T
G1
Gn
Ps
ts
P (s, G1)
T
P (Gn, e)
te
Pe

Figure 3. Mapping of a subprocess $\mathcal{P}$ without considering exception handling.

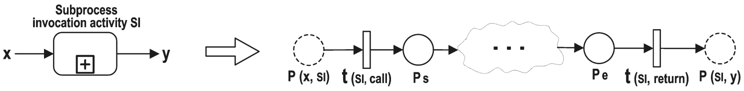


Figure 4. Calling a subprocess $\mathcal{P}$ via a subprocess invocation activity SI.

3.3 Exception Handling

In BPMN, exception handling is captured by exception flow. An exception flow originates from an exception event attached to the boundary of an activity, which is either a task or a subprocess. Figure 5 shows the mapping of an exception associated with a task. Given that a task is an atomic activity, the occurrence of the exception may only interrupt the normal flow at the point when it is ready to execute the task. Hence, this can be viewed as that there is a race condition between the execution of the task and the occurrence of the exception.

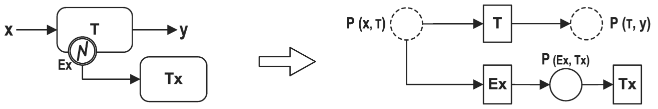


Figure 5. Mapping of a task T with an exception flow originating from event Ex.

For exception handling associated with a subprocess, the occurrence of the exception event will interrupt the execution of the normal flow within the subprocess when it is active (running). The mapping is then 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, in the mapping of this subprocess, when the transition modelling the exception event fires, all the tokens left in the net need to be removed from various places without knowing where the tokens reside. Due to the local nature of Petri net transitions, it is cumbersome to model a so-called "vacuum cleaner" removing tokens from selected parts of the net [3]. Hence, to model the cancellation of a subprocess, we adopt another approach of bypassing tasks and events in the subprocess upon occurrence of the exception. The basic idea is to attach a status flag to the subprocess, which may have a value of OK or NOK. If the flag is set to OK, it allows the normal flow to continue; otherwise (the flag is set to NOK), it signals the occurrence of the exception, and thereby stops the normal flow but enables to bypass the remaining tasks and events until the end of the subprocess. This way we direct all the tokens left in the various places in the net to flow to the end of the subprocess, without executing any remaining tasks or events on the way. After the above bypassing finishes, the normal flow will be withdrawn before the exception handling starts.

However, the above mapping of exception handling does not work properly in the case where an activity within the subprocess is enabled multiple times *concurrently*. In such a case, the activity would need to be bypassed multiple times (as many times as it is enabled) and thus a counter would be required to record how many times the activity needs to be bypassed. Hence, we have to impose two restrictions when mapping subprocesses with exception handling.

First, we will be able to map a subprocess with exception handling, only if this subprocess (without considering the exception handling) can be mapped onto a *1-safe* Petri net. For a subprocess that does not satisfy this condition, we cannot ensure in the mapping that *all* enabled tasks/events in the subprocess are correctly bypassed (i.e. cancelled) before the exception handling starts. Thus, given a subprocess with exception handling, we will first translate it into a Petri net; then, we will check whether this Petri net is 1-safe, and only if it is 1-safe, we would be able to translate the associated exception handling.

Second, the fact that a subprocess on itself is 1-safe ensures that, assuming the subprocess is not executed multiple times concurrently, none of its tasks/events will ever be executed multiple times concurrently. However, we still need to ensure that, once the subprocess has been invoked, it is not invoked again until the first invocation has completed. This condition may be violated as a result of the "unsafeness" coming from "upstream" in the process model. That is, we need to ensure that the fragment of the model that precedes the subprocess invocation is also 1-safe. This scenario is different from the previous one in that the cause for multiple concurrent enablements/executions of a given task/event is external to the subprocess. Rather than excluding these scenarios from the mapping, we propose to map such scenarios into a Petri net which would prevent a subprocess from being executed multiple times concurrently. This is achieved by introducing a "blocking mechanism" which effectively withholds a new execution of the subprocess until its previous execution has completed. This mechanism may be extended in such a way that a subprocess is allowed to be executed a bounded number of times concurrently. However, we will not be able to discuss such extension in this paper, as further effort on formalising this extended mechanism revealed that the semantics of the exception handling for concurrent executions of a subprocess is not clear in the BPMN specification (see detailed discussion in Section 5).

Figure 6 depicts the mapping of an exception flow associated with a subprocess $\mathcal{P}$ via a non-error exception event *Ex*. The two places, $p_{(\mathcal{P},ok)}$ and $p_{(\mathcal{P},nok)}$, model the OK and NOK values of the status flag attached to $\mathcal{P}$, respectively. As soon as $\mathcal{P}$ starts, the flag is set to OK, and each task or event along the normal flow in $\mathcal{P}$ needs to check this value (via the bidirectional arc to $p_{(\mathcal{P},ok)}$) before it can be executed. If the exception *Ex* occurs before subprocess $\mathcal{P}$ ends, the value of the flag will change from OK to NOK. As a result, any remaining task or event in $\mathcal{P}$ will be skipped (e.g. transition $t_{(T,skip)}$ models the skipping of task T). Finally, just before reaching the end of $\mathcal{P}$, the flow switches to the exception handling (which starts with task *Tx*) via transition $t_{(\mathcal{P},excp)}$. The occurrence of $t_{(\mathcal{P},excp)}$ also clears the NOK value of the status flag. Whereas, if no exception occurs, the flag will remain OK until the end of subprocess $\mathcal{P}$. Note that there is another place named $p_{(\mathcal{P},excp)}$ of which the details will be discussed shortly.

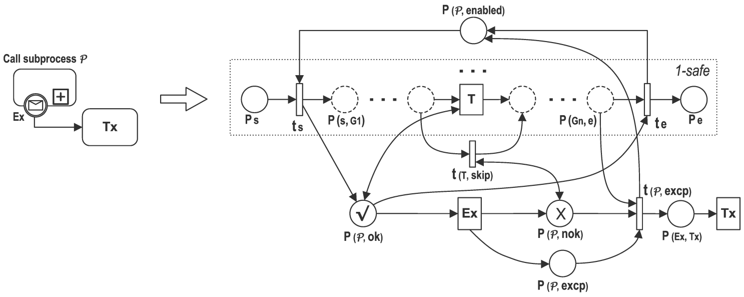


Figure 6. Mapping of a subprocess $\mathcal{P}$ associated with an exception flow via a non-error exception event *Ex*.

In Figure 6, the net enclosed within the dotted box, which is the mapping of subprocess $\mathcal{P}$ without considering the exception handling, must be a 1-safe net. Also, we restrict ourselves to the case where the subprocess is only allowed to be executed at most once at a time. According to this, a place named $p_{(\mathcal{P},enabled)}$ and its surrounding arcs are added to ensure that a new execution of $\mathcal{P}$ will have to wait until the previous execution of $\mathcal{P}$ finishes before it can start. Intuitively, the place $p_{(\mathcal{P},enabled)}$ can be viewed as holding a "resource" for execution of the subprocess $\mathcal{P}$. This resource is created when the top level process starts and will be collected when the top level process ends.

Note that a special case of exception handling is the "throw-catch" error exceptions. Figure 7 shows the corresponding mapping as an invariant of the mapping shown in Figure 6. As mentioned before, an error event (*Ex*) attached to the boundary of a subprocess models "catching" an error exception, and it must have a matching error event (*Et*) on the normal flow of the subprocess which models "throwing" that error exception. Therefore, both *Et* and *Ex* signal the occurrence of one error exception, and can be viewed as one atomic action. They are modelled by one transition named *TEx* in Figure 7.

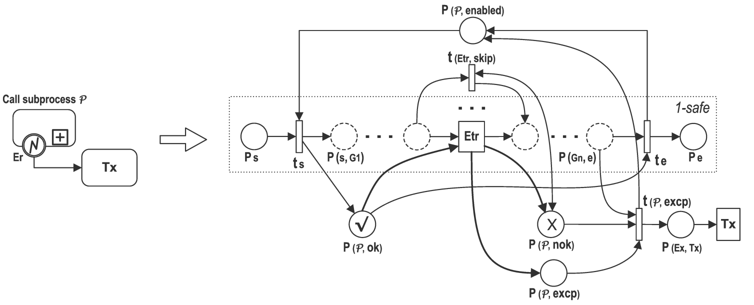


Figure 7. Mapping of a subprocess $\mathcal{P}$ associated with an exception flow via "throw-catch" error exception.

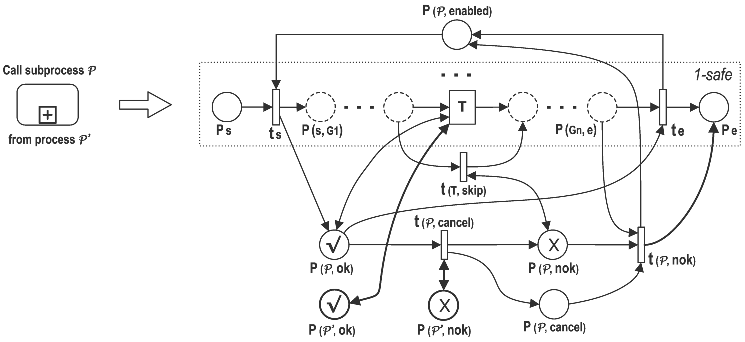


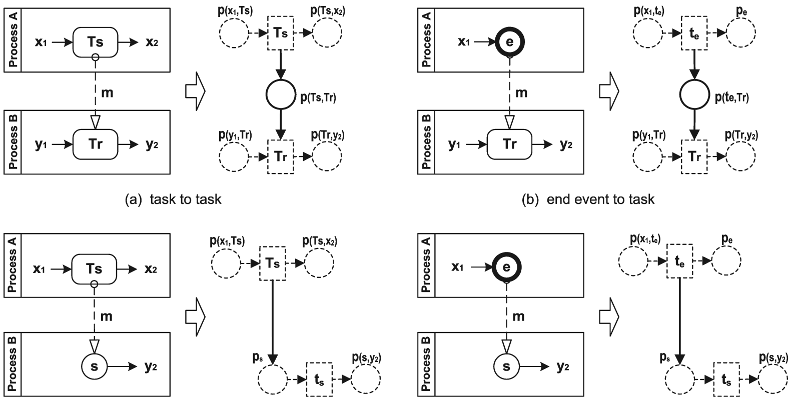
Figure 8. Mapping of a subprocess $\mathcal{P}$ that may be cancelled due to the cancellation of its parent subprocess $\mathcal{P}'$.

Finally, taking into account the exception handling, we may need to extend the mapping of a subprocess with cancellation. Assume that a subprocess $\mathcal{P}$ is nested within another subprocess $\mathcal{P}'$ (i.e. $\mathcal{P}'$ is the "parent" of $\mathcal{P}$). The execution of $\mathcal{P}$ may be cancelled at any point due to the cancellation of $\mathcal{P}'$, despite whether or not there is an exception associated with $\mathcal{P}$. Figure 8 shows the corresponding mapping. The transition $t_{(\mathcal{P},cancel)}$ is used to capture the trigger for cancelling the execution of $\mathcal{P}$, i.e. the cancellation of $\mathcal{P}$'s parent subprocess $\mathcal{P}'$. Note that each task or event in $\mathcal{P}$ also needs to check the OK status of $\mathcal{P}'$ (via the bidirectional arc to $p_{(\mathcal{P}',ok)}$). This is to ensure that once $\mathcal{P}'$ is cancelled the execution of $\mathcal{P}$ stops immediately. In Figure 8, when the bypassing has finished, the flow still continues along the normal flow (via transition $t_{(\mathcal{P},nok)}$), as opposed to the mapping of exception handling in Figure 6 where the flow switches to the exception flow (via transition $t_{(\mathcal{P},excp)}$) at that point. Also, it is necessary to add two places $p_{(\mathcal{P},excp)}$ and $p_{(\mathcal{P},cancel)}$ to ensure correct execution of transition $t_{(\mathcal{P},excp)}$ or $t_{(\mathcal{P},nok)}$. More generally, the execution of subprocess $\mathcal{P}$ may be cancelled due to the cancellation of one of its "ancester" subprocess $\mathcal{P}''$ (e.g. the "parent" of subprocess $\mathcal{P}'$). In this case, the cancellation can be viewed as propogated from, e.g., $\mathcal{P}''$ to $\mathcal{P}'$ and then $\mathcal{P}'$ to $\mathcal{P}$. The mapping is given in the next subsection.

3.4 Message Flow

A message flow is used to show the transmission of messages between two participants via communication actions such as send task, receive task, or message event. In an abstract way, it can be mapped to a place with an incoming arc from the transition modeling the send action and an outgoing arc to the transition modeling the receive action. A special case for the mapping is the mapping of a message flow to a start event. Such a message flow represents that the process is instantiated each time the a message is received. In this case the transition that represents sending the message has a flow from it to the place that represents the start event, such that the process can be started each time a message is received. The mapping should ensure that places that represent message start events do not contain a token in the initial marking (see detailed discussions in Section 3.6), because the process can only be instantiated as a consequence of this event when a message has arrived.

Figure 9 shows the four mapping rules. Each rule distinguishes a case for mapping a message sent by a task or an end event and received by a task or a start event. Note that a task may be replaced by an intermediate message event without changing the rule.



(a) task to task

(b) end event to task

(c) task to start event

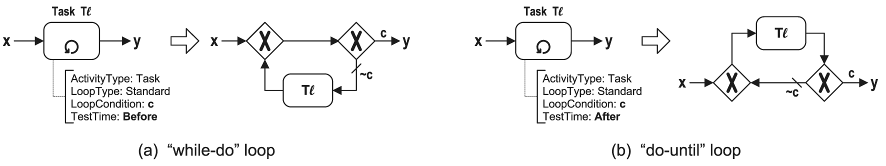
(d) end event to start event

Figure 9. Mapping of message flows between two interacting BPMN processes.

The above mapping is restricted to tasks that either send messages or receive messages but not both, although tasks that can do both do exist (the User task and the Service task). This restriction does not limit the expressive power of BPMN, because successively sending and receiving a message can be represented by two tasks (one send and one receive task) as well as one send/receive task. The mapping is also restricted to the instantiation of a process by a start event, although BPMN allows for the instantiation of a process by a task that is directly preceded by a start event. This restriction does not limit the expressive power of BPMN, because the instantiation of the process can be represented by the start event that precedes the task.

3.5 Macro and Advanced Constructs

In BPMN, an activity may have attributes specifying its additional behaviour, such as looping and parallel multiple instances. Activity looping constructs capture both "while-do" and "do-until" loops. These are "macros" for structured looping of the activity, and thus can be replaced by the normal activity construct connected via sequence flow looping. Figure 10 shows the two corresponding examples where the value of "TestTime" attribute determines whether it is a "while-do" or a "do-unitl" loop.



(a) "while-do" loop

(b) "do-until" loop

Figure 10. Macros for structured activity looping.

Parallel multi-instance activity constructs reflect the programming construct "for-each". In this case, the number of multiple instances (n) running in parallel is always known as a priori, either at design time or at runtime. If n is known at design time, the constructs are "macros" for concurrent executions of n copies of the activity, and thus can be replaced by n normal activity constructs enclosed within a pair of AND-split and AND-join gateways as shown in Figure 11. Whereas if n is only known at run time, to capture the computation of n (at run time) it is necessary to apply some high-level net constructs (e.g. arc expressions in Coloured Petri nets), which is beyond the scope of this paper.

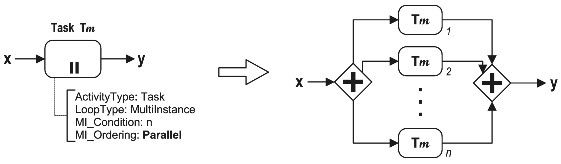


Figure 11. Macro for multi-instance activity of which the number of multiple instances is known at design time.

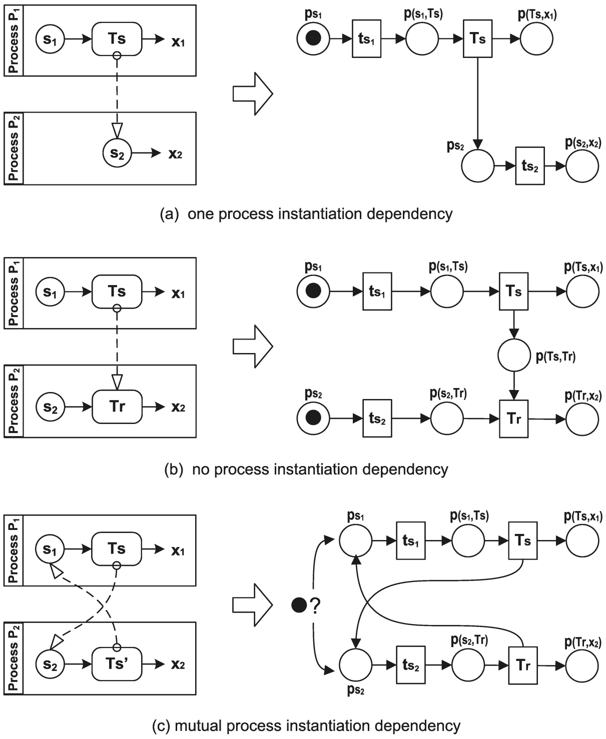
An OR-split gateway is known as an *inclusive OR decision gateway* in BPMN. It is used for selecting any number of branches among all its outgoing flows. Since the behaviour of an OR-split gateway can be captured through a combination of AND-split and XOR-split gateways [5], the mapping can be achieved accordingly.

3.6 Initial Marking Configuration

The initial state of a BPMN model can be specified by the initial marking of the corresponding Petri net model. As afore-mentioned, a start event signals the start of a BPMN process. We hereafter use term "start place" to refer to place p_s in the Petri net module of a start event s shown in Figure 2.

The basic idea for configuring the initial marking is to mark the start places for each of those start events that do not have any incoming message flows and that the processes they belong to are top-level processes. A top-level process is a process which is not invoked by any other process via a subprocess invocation activity. A message flow that has as a target the start event of a process, will create an instance of the process upon message delivery. Let $\mathcal{M} = (\mathcal{Q}, \mathcal{S}^{\diamond}$, map, HR, $\mathcal{F}^{M})$ be a well-formed core BPMN model as given in Definition 4, $\mathcal{Q}_{top} = \{\mathcal{P} \mid \mathcal{P} \in \mathcal{Q} \wedge \neg \exists \mathcal{P}' \in \mathcal{Q} \Rightarrow (\mathcal{P}', \mathcal{P}) \in \mathsf{HR}\}$ is the set of top-level processes in $\mathcal{M}$, and $\mathcal{E}^{S}_{mark} = \{e^{S}_{\mathcal{P}} \mid \mathcal{P} \in \mathcal{Q}_{top} \wedge \neg \exists\, x \in \cup_{\mathcal{P} \in \mathcal{Q}} \mathcal{O}_{\mathcal{P}} \Rightarrow (x, e^{S}_{\mathcal{P}}) \in \mathcal{F}^{M}\}$ is the set of top-level process start events that do not have any incoming message flows in $\mathcal{M}$. However, $\mathcal{E}^{S}_{mark}$ may be an empty set, which means that each top-level process $\mathcal{P}_{top}$ is instantiated by another process via an incoming message flow to the start event of $\mathcal{P}_{top}$. If this is the case, the model designer will need to determine which of the top-level process start events is triggered initially.

Figure 12 depicts the initial marking configuration for three typical examples of BPMN models that consist of two communicating (top-level) processes, namely P_1 and P_2. P_1 has a start event s_1, and P_2 has a start event s_2. The first example in (a) exhibits one process instantiation dependency, where process P_2 is initiated upon a message sent from process P_1. As a result, the start place for



(a) one process instantiation dependency

(b) no process instantiation dependency

(c) mutual process instantiation dependency

Figure 12. Initial marking configuration for three typical examples of BPMN models consisting of two communicating processes.

event s_2 is the only place being marked (with a token drawn as a big black dot) in the initial marking of the corresponding Petri net model. The second example in (b) shows no process intantiation dependency between P_1 and P_2, where neither event s_1 nor s_2 has an incoming message flow, and the communication occur after both processes are initiated. In this case, the start places for both s_1 and s_2 have to be marked initially. Finally, the example in (c) exhibits mutual process instantiation between P_1 and P_2, where event s_1 has an incoming message flow from process P_2 and vice versa. In this case, it is up to the model designer to determine which start place will be initially marked.

4 Formal Definition of the Mapping

We formally define the mapping of BPMN to Petri nets (using set notations). To facilitate the definition, we introduce two auxiliary functions par and anc. Let $\mathcal{M} = (\mathcal{Q}, \mathcal{S}^\diamond, \mathsf{map}, \mathsf{HR}, \mathcal{F}^M)$ be a well-formed core BPMN model. For any subprocess $\mathcal{P} \in \mathcal{Q}$, $\mathsf{par}(\mathcal{P})$ is the parent process of $\mathcal{P}$ (i.e. $(\mathsf{par}(\mathcal{P}), \mathcal{P}) \in \mathsf{HR}$), and $\mathsf{anc}(\mathcal{P}) = \{\mathcal{P}' \mid \mathcal{P}'\mathsf{HR}^+\mathcal{P}\}$[8] is the set of ancester processes of $\mathcal{P}$. Also, we define two predicates HasEH and AncHasEH, both operating over a set of subprocesses. For any subprocess $\mathcal{P} \in \mathcal{Q}$, $\textsc{HasEH}(\mathcal{P})$ is used to indicate if there is an exception associated with $\mathcal{P}$. It returns `true` iff $\mathsf{map}^{\text{-1}}(\mathcal{P}) \in \mathit{range}(\mathsf{Excp}_{\mathsf{par}(\mathcal{P})})$. Similarly, $\textsc{AncHasEH}(\mathcal{P})$ is used to indicate if there is an ancester process of $\mathcal{P}$ which has an exception. It returns `true` iff $\exists\, \mathcal{P}' \in \mathsf{anc}(\mathcal{P})$ such that $\textsc{HasEH}(\mathcal{P}')$ is true.

Definition 5 (Petri net semantics of well-formed core BPMN models). *Let $\mathcal{M} = (\mathcal{Q}, \mathcal{S}^\diamond, \mathsf{map}, \mathsf{HR}, \mathcal{F}^M)$ be a well-formed core BPMN model. Without considering exception handling and communication between interacting processes, $\mathcal{M}$ can be mapped to a preliminary Petri net $PN' = (P', T', F')$ where:*

$$\begin{array}{lll}
P' = \bigcup_{\mathcal{P}\in\mathcal{Q}}(& \{p_s \mid s \in \mathcal{E}^S_\mathcal{P}\} \cup & \textit{– start event} \\
 & \{p_e \mid e \in \mathcal{E}^E_\mathcal{P}\} \cup & \textit{– end event} \\
 & \{p_{(x,y)} \mid (x, y) \in \mathcal{F}_\mathcal{P} \wedge x \notin \mathcal{G}^V_\mathcal{P}\}) & \textit{– sequence flow} \\
T' = \bigcup_{\mathcal{P}\in\mathcal{Q}}(& \{x \mid x \in \mathcal{T}_\mathcal{P} \cup \mathcal{E}^I_\mathcal{P} \wedge \mathsf{in}(x) \neq \varnothing\} \cup & \textit{– task/interm.-event} \\
 & \{t_x \mid x \in \mathcal{G}^F_\mathcal{P} \cup \mathcal{G}^J_\mathcal{P}\} \cup & \textit{– fork/join} \\
 & \{t_{(x,y)} \mid x \in \mathcal{G}^X_\mathcal{P} \wedge y \in \mathsf{out}(x)\} \cup & \textit{– data decision} \\
 & \{t_{(x,y)} \mid x \in \mathcal{G}^M_\mathcal{P} \wedge y \in \mathsf{in}(x)\} \cup & \textit{– merge} \\
 & \{t_s \mid s \in \mathcal{E}^S_\mathcal{P}\} \cup & \textit{– to start a process} \\
 & \{t_e \mid e \in \mathcal{E}^E_\mathcal{P}\} \cup & \textit{– to end a process} \\
 & \{t_{(x,call)} \mid x \in \mathcal{S}_\mathcal{P}\} \cup & \textit{– subprocess call} \\
 & \{t_{(x,return)} \mid x \in \mathcal{S}_\mathcal{P}\}) & \textit{– subprocess return} \\
F' = \bigcup_{\mathcal{P}\in\mathcal{Q}}(& \{(p_{(x,y)}, y) \mid y \in \mathcal{T}_\mathcal{P} \cup \mathcal{E}^I_\mathcal{P} \wedge x \in \mathsf{in}(y)\backslash\mathcal{G}^V_\mathcal{P}\} \cup & \\
 & \{(y, p_{(y,z)})\} \mid y \in \mathcal{T}_\mathcal{P} \cup \mathcal{E}^I_\mathcal{P} \wedge \mathsf{in}(y) \neq \varnothing \wedge z \in \mathsf{out}(y)\} \cup & \textit{– task/interm.-event} \\
 & \{(p_{(x,y)}, t_y)\} \mid y \in \mathcal{G}^F_\mathcal{P} \cup \mathcal{G}^J_\mathcal{P} \wedge x \in \mathsf{in}(y)\} \cup & \\
 & \{(t_y, p_{(y,z)})\} \mid y \in \mathcal{G}^F_\mathcal{P} \cup \mathcal{G}^J_\mathcal{P} \wedge z \in \mathsf{out}(y)\} \cup & \textit{– fork/join} \\
 & \{(p_{(x,y)}, t_{(y,z)}) \mid y \in \mathcal{G}^X_\mathcal{P} \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} \cup & \\
 & \{(t_{(y,z)}, p_{(y,z)}) \mid y \in \mathcal{G}^X_\mathcal{P} \wedge z \in \mathsf{out}(y)\} \cup & \textit{– data decision} \\
 & \{(p_{(x,y)}, t_{(y,x)}) \mid y \in \mathcal{G}^M_\mathcal{P} \wedge x \in \mathsf{in}(y)\} \cup & \\
 & \{(t_{(y,x)}, p_{(y,z)}) \mid y \in \mathcal{G}^M_\mathcal{P} \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} \cup & \textit{– merge} \\
 & \{(p_{(x,y)}, z) \mid y \in \mathcal{G}^V_\mathcal{P} \wedge x \in \mathsf{in}(y) \wedge z \in \mathsf{out}(y)\} \cup & \textit{– event decision}
\end{array}$$

[8] HR^+ is a (non-reflexive) transitive closure, i.e. $x\mathsf{HR}^+y \Rightarrow x \neq y$.

$$
\begin{array}{ll}
\{(p_s, t_s) \mid s \in \mathcal{E}^S_{\mathcal{P}}\} \ \cup \{(t_s, p_{(s,y)}) \mid s \in \mathcal{E}^S_{\mathcal{P}} \wedge y \in \mathsf{out}(s)\} \cup & \text{– \textit{process start}} \\
\{(p_{(x,e)}, t_e) \mid e \in \mathcal{E}^E_{\mathcal{P}} \wedge x \in \mathsf{in}(e)\} \ \cup \{(t_e, p_e) \mid s \in \mathcal{E}^E_{\mathcal{P}}\} \cup & \text{– \textit{process end}} \\
\{(p_{(x,y)}, t_{(y,call)}) \mid y \in \mathcal{S}_{\mathcal{P}} \wedge x \in \mathsf{in}(y)\} \cup & \\
\{(t_{(y,call)}, p_s) \mid y \in \mathcal{S}_{\mathcal{P}} \wedge s \in \mathcal{E}^S_{\mathsf{map}(y)}\} \cup & \text{– \textit{subprocess call}} \\
\{(p_e, t_{(y,return)}) \mid y \in \mathcal{S}_{\mathcal{P}} \wedge e \in \mathcal{E}^E_{\mathsf{map}(y)}\} \cup & \\
\{(t_{(y,return)}, p_{(y,z)}) \mid y \in \mathcal{S}_{\mathcal{P}} \wedge z \in \mathsf{out}(y)\}) & \text{– \textit{subprocess return}}
\end{array}
$$

Then, by taking into account exception handling, $\mathcal{M}$ is mapped onto a Petri net $PN'' = (P'', T'', F'')$ where:

$$
\begin{array}{lll}
P'' = P' & \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge (\textsc{HasEH}(\mathcal{P}) \vee \textsc{AncHasEH}(\mathcal{P}))} \{p_{(\mathcal{P},ok)}, p_{(\mathcal{P},nok)}\} & \text{– \textit{status flag}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge (\textsc{HasEH}(\mathcal{P}) \vee \textsc{AncHasEH}(\mathcal{P}))} \{p_{(\mathcal{P},enabled)}\} & \text{– \textit{resource place}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge \textsc{HasEH}(\mathcal{P})} \{p_{(\mathcal{P},excp)}\} & \text{– \textit{exception}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge \textsc{AncHasEH}(\mathcal{P})} \{p_{(\mathcal{P},cancel)}\} & \text{– \textit{cancellation}}
\end{array}
$$

$$
\begin{array}{lll}
T'' = T' & \cup \bigcup_{\mathcal{P} \in \mathcal{Q}} \{x \mid x \in (\mathcal{E}^I_{\mathcal{P}} \backslash \mathcal{E}^{I_R}_{\mathcal{P}}) \cap dom(\mathsf{Excp}_{\mathcal{P}})\} & \text{– \textit{non-error exception}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge (\textsc{HasEH}(\mathcal{P}) \vee \textsc{AncHasEH}(\mathcal{P}))} & \\
& \quad \{t_{(x,skip)} \mid x \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \wedge \mathsf{in}(x) \neq \varnothing\} & \text{– \textit{skip task/interm.-event}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge \textsc{HasEH}(\mathcal{P})} \{t_{(\mathcal{P},excp)}\} & \text{– \textit{exception}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge \textsc{AncHasEH}(\mathcal{P})} \{t_{(\mathcal{P},cancel)}, t_{(\mathcal{P},nok)}\} & \text{– \textit{cancellation}}
\end{array}
$$

$$
\begin{array}{lll}
F'' = F' & \cup \bigcup_{\mathcal{P} \in \mathcal{Q}} \{(p_{(x,y)}, z) \mid y \in \mathcal{T}_{\mathcal{P}} \wedge x \in \mathsf{in}(y) \wedge z = \mathsf{Excp}^{\text{-}1}_{\mathcal{P}}(y)\} & \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q}} \{(z, p_{(z,x)}) \mid (\exists_{y \in \mathcal{T}_{\mathcal{P}}} \ z = \mathsf{Excp}^{\text{-}1}_{\mathcal{P}}(y)) \wedge x \in \mathsf{out}(z)\} & \text{– \textit{exception@task}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge (\textsc{HasEH}(\mathcal{P}) \vee \textsc{AncHasEH}(\mathcal{P}))} & \text{– $\downarrow$\textit{exception@subprocess}} \\
& \quad (\{(t_s, p_{(\mathcal{P},ok)}) \mid s \in \mathcal{E}^S_{\mathcal{P}}\} \cup \{(p_{(\mathcal{P},ok)}, t_e) \mid e \in \mathcal{E}^E_{\mathcal{P}}\} \cup & \text{– \textsc{ok} \textit{status}} \\
& \quad\ \{(p_{(x,y)}, t_{(y,skip)}) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \wedge x \in \mathsf{in}(y) \backslash \mathcal{G}^V_{\mathcal{P}}\} \cup & \\
& \quad\ \{(p_{(z,x)}, t_{(y,skip)}) \mid x \in \mathcal{G}^V_{\mathcal{P}} \wedge y \in \mathsf{out}(x) \wedge z \in \mathsf{in}(x)\} \cup & \\
& \quad\ \{(t_{(y,skip)}, p_{(y,z)}) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \wedge \mathsf{in}(y) \neq \varnothing \wedge z \in \mathsf{out}(y)\} \cup & \text{– \textit{skip task/interm.-event}} \\
& \quad\ \{(p_{(\mathcal{P},ok)}, y) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \backslash \mathcal{E}^{I_R}_{\mathcal{P}} \wedge \mathsf{in}(y) \neq \varnothing\} \cup & \\
& \quad\ \{(y, p_{(\mathcal{P},ok)}) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \backslash \mathcal{E}^{I_R}_{\mathcal{P}} \wedge \mathsf{in}(y) \neq \varnothing\} \cup & \text{– \textit{check} \textsc{ok} \textit{status}} \\
& \quad\ \{(p_{(\mathcal{P},nok)}, t_{(y,skip)}) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \wedge \mathsf{in}(y) \neq \varnothing\} \cup & \\
& \quad\ \{(t_{(y,skip)}, p_{(\mathcal{P},nok)}) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \wedge \mathsf{in}(y) \neq \varnothing\} \cup & \text{– \textit{check} \textsc{nok} \textit{status}} \\
& \quad\ \{(t_{e^S_{\mathcal{P}}}, p_{(\mathcal{P},enabled)}), (p_{(\mathcal{P},enabled)}, t_{e^E_{\mathcal{P}}})\} \cup & \text{– \textit{create/collect resource}} \\
& \quad\ \{(p_{(\mathcal{P},enabled)}, t_{e^S_{\mathcal{P}}}), (t_{e^E_{\mathcal{P}}}, p_{(\mathcal{P},enabled)}) \mid e \in \mathcal{E}^E_{\mathcal{P}}\}) \cup & \text{– \textit{consume/release resource}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge \textsc{HasEH}(\mathcal{P})} & \\
& \quad (\{(p_{(\mathcal{P},ok)}, x) \mid (\mathsf{Excp}_{\mathsf{par}(\mathcal{P})}(x) = \mathsf{map}^{\text{-}1}(\mathcal{P}) \wedge x \notin \mathcal{E}^{I_R}_{\mathsf{par}(\mathcal{P})}) & \\
& \qquad\qquad\qquad \vee x \in \mathcal{E}^{I_R}_{\mathcal{P}} \backslash dom(\mathsf{Excp}_{\mathcal{P}})\} \cup & \\
& \quad\ \{(x, p_{(\mathcal{P},nok)}) \mid (\mathsf{Excp}_{\mathsf{par}(\mathcal{P})}(x) = \mathsf{map}^{\text{-}1}(\mathcal{P}) \wedge x \notin \mathcal{E}^{I_R}_{\mathsf{par}(\mathcal{P})}) & \\
& \qquad\qquad\qquad \vee x \in \mathcal{E}^{I_R}_{\mathcal{P}} \backslash dom(\mathsf{Excp}_{\mathcal{P}})\} \cup & \text{– \textit{status-change@exception}} \\
& \quad\ \{(x, p_{(\mathcal{P},excp)}) \mid (\mathsf{Excp}_{\mathsf{par}(\mathcal{P})}(x) = \mathsf{map}^{\text{-}1}(\mathcal{P}) \wedge x \notin \mathcal{E}^{I_R}_{\mathsf{par}(\mathcal{P})}) & \\
& \qquad\qquad\qquad \vee x \in \mathcal{E}^{I_R}_{\mathcal{P}} \backslash dom(\mathsf{Excp}_{\mathcal{P}})\} \cup & \\
& \quad\ \{(p_{(\mathcal{P},excp)}, t_{(\mathcal{P},excp)})\} \cup & \text{– \textit{mark/unmark exception}} \\
& \quad\ \{(p_{(\mathcal{P},nok)}, t_{(\mathcal{P},excp)})\} \cup & \\
& \quad\ \{(p_{(x,e)}, t_{(\mathcal{P},excp)}) \mid e \in \mathcal{E}^E_{\mathcal{P}} \wedge x \in \mathsf{in}(e)\} \cup & \text{– \textit{clear-}\textsc{nok}\textit{@exception}} \\
& \quad\ \{(t_{(\mathcal{P},excp)}, p_{(x,y)}) \mid \mathsf{Excp}_{\mathsf{par}(\mathcal{P})}(x) = \mathsf{map}^{\text{-}1}(\mathcal{P}) \wedge y \in \mathsf{out}(x)\} \cup & \text{– \textit{switch to exception flow}} \\
& \quad\ \{(t_{(\mathcal{P},excp)}, p_{(\mathcal{P},enabled)})\}) & \text{– \textit{release-resource@exception}} \\
& \cup \bigcup_{\mathcal{P} \in \mathcal{Q} \wedge \textsc{AncHasEH}(\mathcal{P})} & \\
& \quad (\{(p_{(\mathcal{P},ok)}, t_{(\mathcal{P},cancel)}), (t_{(\mathcal{P},cancel)}, p_{(\mathcal{P},nok)})\} \cup & \text{– \textit{status-change@cancellation}} \\
& \quad\ \{(t_{(\mathcal{P},cancel)}, p_{(\mathcal{P},cancel)}), (p_{(\mathcal{P},cancel)}, t_{(\mathcal{P},nok)})\} \cup & \text{– \textit{mark/unmark cancellation}} \\
& \quad\ \{(p_{(\mathcal{P},nok)}, t_{(\mathcal{P},nok)})\} \cup & \\
& \quad\ \{(p_{(x,e)}, t_{(\mathcal{P},nok)}) \mid e \in \mathcal{E}^E_{\mathcal{P}} \wedge x \in \mathsf{in}(e)\} \cup & \text{– \textit{clear-}\textsc{nok}\textit{@cancellation}} \\
& \quad\ \{(t_{(\mathcal{P},nok)}, p_e) \mid e \in \mathcal{E}^E_{\mathcal{P}}\} \cup & \text{– \textit{continue@normal-flow}} \\
& \quad\ \{(t_{(\mathcal{P},nok)}, p_{(\mathcal{P},enabled)})\} \cup & \text{– \textit{release-resource@cancellation}} \\
& \quad\ \bigcup_{\mathcal{P}' \in \mathsf{anc}(\mathcal{P}) \wedge (\textsc{HasEH}(\mathcal{P}') \vee \textsc{AncHasEH}(\mathcal{P}'))} & \\
& \qquad (\{(p_{(\mathcal{P}',ok)}, y) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \wedge \mathsf{in}(y) \neq \varnothing\} \cup & \\
& \qquad\ \{y, (p_{(\mathcal{P}',ok)}) \mid y \in \mathcal{T}_{\mathcal{P}} \cup \mathcal{E}^I_{\mathcal{P}} \wedge \mathsf{in}(y) \neq \varnothing\}) \cup & \text{– \textit{check} \textsc{ok} \textit{of ancesters}} \\
& \quad\ \{(p_{(\mathsf{par}(\mathcal{P}),nok)}, t_{(\mathcal{P},cancel)}), (t_{(\mathcal{P},cancel)}, p_{(\mathsf{par}(\mathcal{P}),nok)})\}) & \text{– \textit{trigger-cancellation@parent}}
\end{array}
$$

Finally, by taking into account the communication between interacting processes, $\mathcal{M}$ is fully mapped onto a Petri net $PN_{\mathcal{M}} = (P_{\mathcal{M}}, T_{\mathcal{M}}, F_{\mathcal{M}})$ where:

$$\begin{array}{lll}
P_{\mathcal{M}} & = P'' \cup \{p_{(x,y)} \mid (x,y) \in \mathcal{F}^M \wedge y \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^S_{\mathcal{P}}\} & \textit{– connecting place} \\
T_{\mathcal{M}} & = T'' & \textit{– no newly added trans.} \\
F_{\mathcal{M}} & = F'' \cup \{(x, p_{(x,y)}) \mid (x,y) \in \mathcal{F}^M \wedge x \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^E_{\mathcal{P}} \wedge y \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^S_{\mathcal{P}}\} \cup & \\
 & \quad \{(p_{(x,y)}, y) \mid (x,y) \in \mathcal{F}^M \wedge x \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^E_{\mathcal{P}} \wedge y \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^S_{\mathcal{P}}\} \cup & \textit{– between tasks/interm.-events} \\
 & \quad \{(t_x, p_{(x,y)}) \mid (x,y) \in \mathcal{F}^M \wedge x \in \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^E_{\mathcal{P}} \wedge y \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^S_{\mathcal{P}}\} \cup & \\
 & \quad \{(p_{(x,y)}, y) \mid (x,y) \in \mathcal{F}^M \wedge x \in \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^E_{\mathcal{P}} \wedge y \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^S_{\mathcal{P}}\} \cup & \textit{– end event to task/interm.-event} \\
 & \quad \{(x, p_y) \mid (x,y) \in \mathcal{F}^M \wedge x \notin \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^E_{\mathcal{P}} \wedge y \in \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^S_{\mathcal{P}}\} \cup & \textit{– task/interm.-event to start event} \\
 & \quad \{(t_x, p_y) \mid (x,y) \in \mathcal{F}^M \wedge x \in \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^E_{\mathcal{P}} \wedge y \in \cup_{\mathcal{P}\in\mathcal{Q}}\mathcal{E}^S_{\mathcal{P}}\} & \textit{– end event to start event}
\end{array}$$

5 Issues in the BPMN Specification

During the formalisation, we identified a number of deficiencies in the BPMN specification. Below, we outline the most salient ones and discuss options for resolving them.

Process models with multiple start events A BPMN process model may have multiple start events but the meaning of BPMN process models with multiple start events is underspecified. The BPMN specification states that "each Start Event is an independent event. That is, a Process Instance SHALL be generated when the Start Event is triggered". Though ambiguous, this statement suggests that it is enough for one of the start events to occur for a process instance to be generated. However, once a process instance is generated by the occurrence of a start event, it is unclear whether the other start events may, must or may not occur as part of the execution of that process instance. To add to the confusion, the specification states that: "If there is a dependency for more than one Event to happen before a Process can start [...] then the Start Events must flow to the same activity within that Process. The attributes of the activity would specify when the activity could begin. If the attributes specify that the activity must wait for all inputs, then all Start Events will have to be triggered before the Process begins." This statement suggests that once a process instance has been created by the occurrence of one of its Start Events, it may be necessary to wait for the other Start Events to be triggered as well. However, closer examination of the attributes associated to activities shows that none of them allow one to model that an activity must "wait for all inputs". An activity with multiple incoming flows only needs to wait for a token from one of them. Furthermore, if we assume that once a process instance has been created by the occurrence of one of its start events, the other start events can still occur as part of that process instance, it means that the occurrence of a start event sometimes generates a new process instance, while other times it is routed to an existing process instance. Accordingly, a correlation mechanism is required to link the occurrence of a start event to an appropriate process instance. The definition of such correlation mechanism is an open issue.

Process instance completion The BPMN specification does not clearly state when should an execution of a process model be considered to be "completed". This is particularly problematic for process models with multiple end events since many options are possible in this case, e.g. is it enough that one end event occurs (or is reached) for the process instance to be completed, or should we wait for all end events to be reached, or should we wait until no end event can be reached any more? We could only find in the specification one statement regarding this issue: "the process MUST NOT end until all parallel paths have completed". However, the notion of "parallel path" is not defined, nor is the notion of "completion of a path". Completion policies for process models with multiple end tasks have been studied in [14]. This paper formalizes the notion that "an instance of a process model is completed when at least one of its end tasks has been executed at least once, and there is no other enabled task for that process instance". We suggest that the BPMN specification should adopt this completion policy. Interestingly, the adoption of this completion policy in BPMN would make it impossible to translate unsafe BPMN model into Workflow nets (see [14]). Such models would need to be translated into nets that may have multiple "sink places" (i.e. the place with no outgoing arcs).

Exception handling for concurrent subprocess instances The BPMN specification is also unclear regarding the semantics of an exception handler attached to a parallel multi-instance activity that invokes a subprocess. It is not clear from the BPMN specification whether an exception thrown by an instance of such a subprocess and caught by an exception handler attached to the parallel multi-instance activity, should interrupt: (i) only the subprocess instance in question; or (ii) all instances of that subprocess. If the first of these two options was adopted, another ambiguity would need to be resolved, namely: should the interrupted instance of the subprocess be considered as "completed" for the purpose of determining the completion of the multi-instance activity in question? Indeed, a multi-instance activity that invokes a subprocess is completed, by default, if all the subprocess instances it spawns have completed.

Also, a subprocess may be executed multiple times as a result of unsafeness in the parent process model. If a process is not 1-safe, it may happen that one of its activities invokes a subprocess once, and while the subprocess instance spawned by this invocation is still executing, the same activity is executed again and thus invokes the subprocess a second time, thus leading to two subprocess instances that execute concurrently. Again, if an exception is thrown by one of these instances and is caught by an exception handler attached to the invocation activity, it is unclear whether this exception would only affect that subprocess instance, or all subprocess instances spawned by the invocation activity.

OR-join gateway The BPMN specification states that an OR-join (i.e. inclusive merge) gateway "will wait for (synchronize) all Tokens that have been produced upstream" and that the "Process flow SHALL continue when the signals (Tokens) arrive from all of the incoming Sequence Flow that are expecting a signal based on the upstream structure of the Process". However, the notion of "upstream" is unclear, especially when the OR-join is part of a cycle in the process model, in which case the OR-join is "upstream" with respect to itself. Thus, situations may occur in which the firing of a given OR-join depends on whether or not this same OR-join may eventually fire, leading to a vicious cycle. The semantics of OR-join gateways has been extensively studied for other process modelling languages, most notably YAWL [19]. It is perhaps best for the BPMN specification to adopt an existing semantics with a formal foundation rather than attempting to define a new one.

6 Analysis of BPMN Models

6.1 Tool Implementation

The mapping from BPMN to Petri nets presented in Section 3 serves as a specification for a tool that transforms BPMN models into Petri nets. Figure 13 shows the structure of the tool that we implemented. The tool uses standard file formats to keep it as open as possible. It is freely available at `http://is.tm.tue.nl/staff/rdijkman/cbd.html#transformer`.

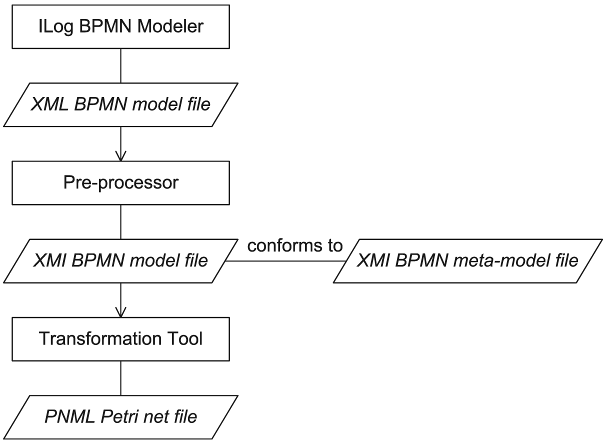


Figure 13. Structure of the BPMN to Petri net transformation tool.

The tool takes an XML Metadata Interchange (XMI) [10] file that contains the model as input. XMI is a standardized file format for storing models, such that if there is agreement on the meta-model, the XMI is tool independent. In that way all tools that conform to the XMI standard and a meta-model can seamlessly exchange models (which are instances of that meta-model). Seamless exchange of models would also be possible between our transformation tool and graphical modelling tools. However, to the best of our knowledge, no meta-model has been standardized for BPMN yet. Pending such a standard we defined our own meta-model, which follows straightforwardly from the BPMN specification. This meta-model is given and illustrated in the Appendix.

We use the ILog BPMN Modeller[9] as a graphical editor to create BPMN models. Since this tool does not generate standard XMI output, we need a simple pre-processor to transform the tools output into XMI.

The transformation tool subsequently transforms the BPMN model into a Petri net and exports the Petri net as a PNML file. PNML [8] is a standardized file format for storing Petri net models. PNML files can be read by a number of Petri net modelling and analysis tools.

6.2 Static Analysis

The PNML file for the mapping of a BPMN model can serve as input to a Petri net-based verification tool, e.g., ProM [16], for static analysis of the model. We can use ProM to check for the following properties:

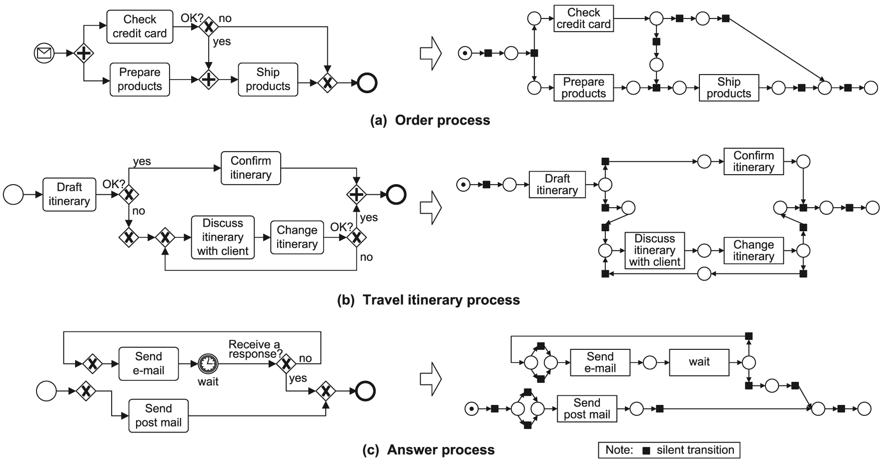
- *Absence of dead tasks*, i.e. there are no tasks that can never be performed within a model. It can be checked through the absence of dead transitions within the corresponding net.
- *Absence of improper completion*, i.e. any process instance eventually reaches proper completion. As formulated in Sect. 5, a process instance is *completed* if it has reached the end event and there are no enabled tasks. In Petri net terms, this corresponds to a *dead marking*[10] in which only the sink place is marked. Additionally, there are two types of undesirable dead markings: deadlocks (i.e. a dead marking where the sink place is unmarked) and markings with trash tokens (i.e. a dead marking where in addition to the sink place, other places are marked).

Fig. 14 shows three examples of BPMN models and the corresponding Petri nets, which violate the above properties.

- The first example – Fig. 14(a) – is an order process that may not complete properly. If the credit card check fails, the process will complete but a token is left in-between task “preparation of products” and “ship products”. Pragmatically, this means that the products are packed but not shipped because of payment issues. The process would need to be corrected to properly withdraw this remaining token and to undo any product preparations that may have been performed.
- The second example is a travel itinerary process that does not complete at all (i.e. it deadlocks). The reason is that initially there is a choice to *either* confirm the itinerary or to discuss it with the client, while the process only completes if *both* these tasks are executed.
- The third example is an answer process that contains dead tasks: the e-mail is never sent. This might indicate a design error, e.g., the designer forgot to draw a flow between the two data-based decision gateways at the start of the process.

[9] `http://www.ilog.com/products/jviews/diagrammer/bpmnmodeler/`

[10] A dead marking is one where there are no transitions enabled.



(a) Order process

(b) Travel itinerary process

(c) Answer process

Figure 14. Examples of BPMN process models and transformations to Petri nets.

6.3 Empirical Evaluation

We tested BPMN2PNML on a set of models[11], collected from the BPMSWatch Web log[12], models distributed with the ILOG BPMN Process Modeler, and models designed by one of the authors in separate work [9]. Table 1 shows the size of each tested BPMN model in terms of number of tasks, events, XOR-gateways, AND-gateways, subprocesses, message flows and exception flows. It also shows the size of the resulting Petri-nets in terms of number of places and transitions. BPMN2PNML was able to deal with all the models, although preprocessing was needed to transform some of them into well-formed BPMN models:

- transform multiple inflows (i.e. incoming flows) to an event, gateway or activity into one inflow, by preceding the activity with an XOR-join gateway that has all the inflows of the activity;
- transform a task with an input *and* an output message into two related tasks; one with an input and one with an output message;
- transform an activity looping construct into a normal activity construct that is placed in a structured loop using an XOR-split and an XOR-join gateway;
- transform a process that does not have a start or an end event into a process that does, by preceding each task without inflows by a start event and succeeding each task without outflows by an end event.

We detected errors in models 5 and 7 which came from the BPMSWatch web log. Model 5 contained dead tasks and model 7 contained incomplete process executions. Models 1, 2, 3, 4 and 6 did not contain any errors.

[11] This set of models, together with the three examples shown in Fig. 14, are included in the distribution of the BPMN2PNML tool.

[12] `http://www.brsilver.com/wordpress/about/`

Table 1. Results of applying the transformation tool to existing models.

Model	BPMN Model							Petri Net Model	
No.	tasks	events	XOR	AND	subprocesses	messages	exceptions	places	transitions
1	11	2	9	2				31	34
2	7	4	4	4				23	21
3	9	8	3		2		2	35	39
4	4	2	2					10	10
5	3	2	2	2				12	11
6	4	8	4			4		24	20
7	5	12	4			5		31	25

7 Related Work

To the best of our knowledge, the only other attempt to define a formal semantics for a subset of BPMN is that of Wong & Gibbons [18], which use Communicating Sequential Processes (CSP) as the target formal model. In their work, a BPMN model is mapped to a set of *CSP processes* and *events*. Each task object is mapped to a CSP process while the flow relations between task objects are captured through CSP events. The conditions for initiation of a task are encoded as possible combinations of CSP events that need to occur for the task to be enabled. When a task completes, it generates event occurrences that may then combine with other event occurrences to initiate other tasks. The CSP models produced in this way may be large and complex, and they do not preserve the structure of the BPMN model. For example, a simple sequence of BPMN activities is not translated as a sequence of processes. Also, Wong & Gibbons [18] do not show how can the CSP semantics be used to detect various types of errors.

On the other hand, formal semantics have been defined for other informal languages that share common features with BPMN. For example, [2] defines a mapping from a language called *workflow task structures* into Workflow nets while [1] provides a similar mapping for Event-driven Process Chains (EPCs). Task structures are composed of tasks, AND split and join gateways, and XOR split and join gateways. Task structures support subprocess invocation, but not exception handling or multiple instances of subprocesses as in BPMN, thus making their mapping to Petri nets easier. A task structure can have multiple sink tasks, like BPMN can have multiple end events. It can also have multiple start tasks, but the semantics is clearer than in BPMN: all start tasks must be performed for each instance of the process model. In task structures the intended termination semantics is that of *implicit termination* as defined in [14] – that is, an instance of the process model is considered to be completed when one of the sink tasks has been performed and no other task is active or enabled. To map this feature into Petri nets, the mapping defined in [2] uses so-called "shadow places" that keep track of the number of active parallel streams. Termination is detected once the number of streams goes back to zero. However, this solution only works when the resulting net is bounded and in addition, it uses weighted arcs with potentially large weights. This idea can be used to extend the proposed

BPMN mapping to deal with implicit termination, but it has an adverse effect on the complexity of analysis algorithms due to the use of weighted arcs.

Next, the mapping of EPCs in terms of Workflow nets (a subclass of Petri nets) provided in [1] is similar to the one for task structures discussed above. The main difference is that EPCs have OR-join connectors, which is present as OR-join gateways in BPMN. The possible semantics of such an OR-join has been discussed extensively in the literature and there is still a lack of consensus [19]. In addition, it seems clear that it would be difficult and perhaps impossible to map such a connector into plain Petri nets in the context of BPMN. This question is left as a direction for future work.

BPMN also shares many common features with Business Process Execution Language [7] for which a number of formal semantics in terms of Petri nets and other formal models of concurrency have been defined [13, 15]. In particular, BPEL provides an equivalent to XOR and AND gateways, although in a block-structured manner, i.e., every AND-split (XOR decision) gateway has a corresponding AND-join (XOR merge gateway) with which it forms a block that has a single entry and a single exit point. BPEL also supports error handling using a throw-catch style. The Petri net mapping for exception handling in BPMN given in this paper is partially based on the one developed for BPEL in [15].

8 Conclusions

The BPMN standard specification is relatively detailed when it comes to specifying syntactic constraints on BPMN models, but it is unsystematic and sometimes inconsistent when it comes to defining their semantics. The lack of formal semantics of BPMN hinders on the development of tool support for checking the correctness of BPMN models from a semantic perspective. This paper has taken a first step to address this gap by providing a mapping from a comprehensive subset of BPMN to Petri nets. The mapping has been implemented in a tool and its application to verifying the soundness and liveness of BPMN models has been tested using the ProM framework. In addition, this formalisation exercise has permitted us to unveil a number of issues in the BPMN specification and to suggest possible solutions.

The proposed mapping does not fully deal with: (i) parallel multi-instance activities; (ii) exception handling in the context of subprocesses that are executed multiple times concurrently; and (iii) OR-join gateways. These three missing features coincide with the limitations of Petri nets that motivated the design of YAWL [4]: a workflow definition language that extends Petri nets with a number of high-level features. In future work we plan to adapt the proposed mapping so that it can generate YAWL nets, especially in those cases where a translation to Petri nets is not feasible. The resulting YAWL nets can then be analysed using techniques such as those described in [19]. Of course, the tradeoff is that verification of YAWL nets is computationally more complex than the corresponding verification problems on Petri nets.

References

1. W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based Approach. *Information Systems*, 25(1):43–69, 2000.
3. W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: On the expressive power of (Petri-net-based) workflow languages (invited talk). In *Proceedings of 4th Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560 of *DAIMI*, pages 1–20. University of Aarhus, Demark, 2002.
4. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2004.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
6. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services Version 1.1.* BEA Systems, IBM Corporation, Microsoft Corporation, SAP AG, Siebel Systems, May 2003.
7. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language Version 2.0.* Committee Draft. WS-BPEL TC OASIS, 2005.
8. J. Billington and et. al. The Petri Net Markup Language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Applications and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.
9. R.M. Dijkman. Choreography-Based Design of Business Collaborations. BETA Working Paper WP-181, Eindhoven University of Technology, 2006.
10. Object Management Group. OMG XML Metadata Interchange (XMI) specification. Available Specification formal/02-01-01, Object Management Group, 2002.
11. Object Management Group. *Unified Modeling Language: Superstructure.* UML Superstructure Specification v2.0, formal/05-07-04. Object Management Group, 2005.
12. Object Management Group. *Business Process Modeling Notation (BPMN) Version 1.0.* OMG Final Adopted Specification. Object Management Group, 2006.
13. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the International Conference on Business Process Management (BPM2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235, Nancy, France, September 2005. Springer-Verlag.
14. B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.
15. C. Ouyang, H.M.W. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. Accepted for publication in *Science of Computer Programming* by Elsevier Science. A technical report version is available via `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-15.pdf`.
16. B. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A new era in process mining tool support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005.

17. WFMC. Workflow Management Coalition Workflow Standard: Workflow Process Definition Interface – XML Process Definition Language (XPDL) (WFMC-TC-1025). Technical report, Workflow Management Coalition, Lighthouse Point, Florida, USA, 2002.
18. P.Y.H. Wong and J. Gibbons. A Process Semantics for BPMN. Preprint, Oxford University Computing Laboratory, 2007. URL: `http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmn_extended.pdf`
19. M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets. In *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443. Springer-Verlag, 2005.

Appendix: A MOF M2 Model for BPMN

8.1 Models

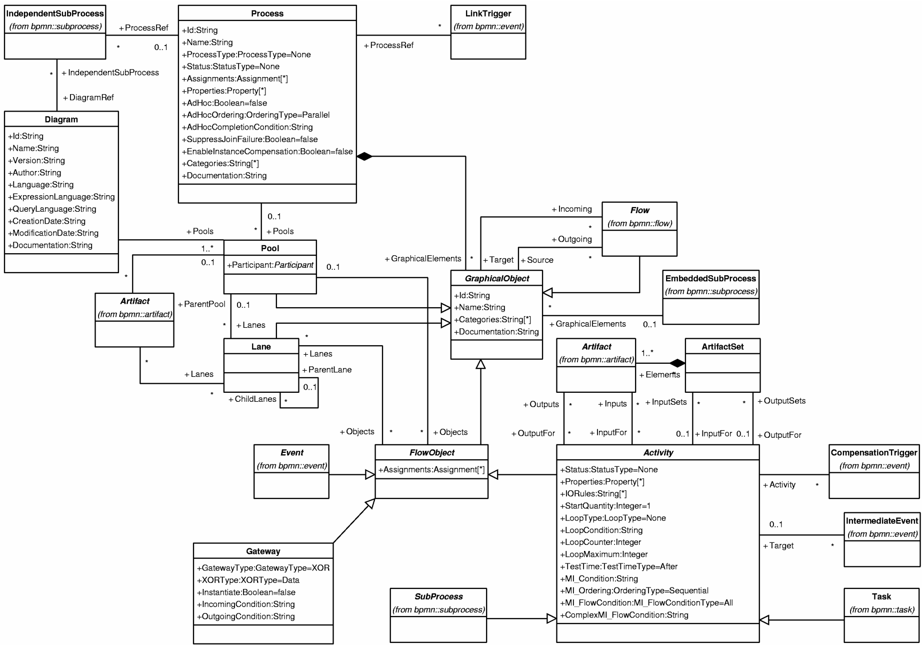


Figure 15. MOF M2 Model for the bpmn package.

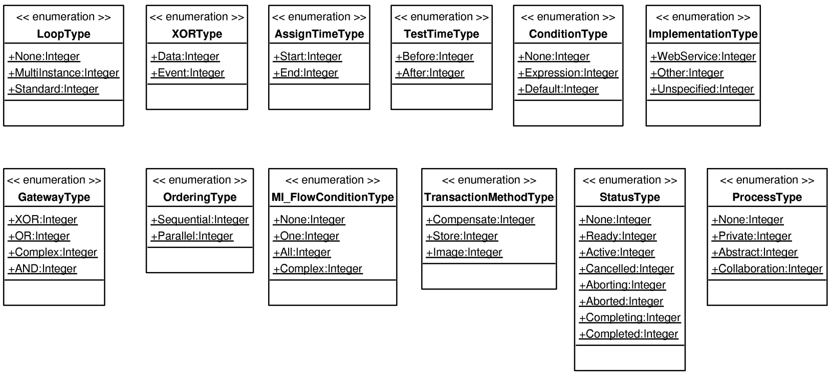


Figure 16. MOF M2 Model of Enumerations Used in the Specification.

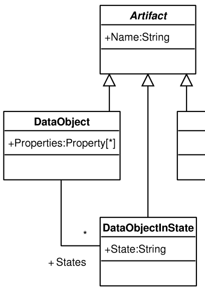


Figure 17. MOF M2 Model for the artifact package.

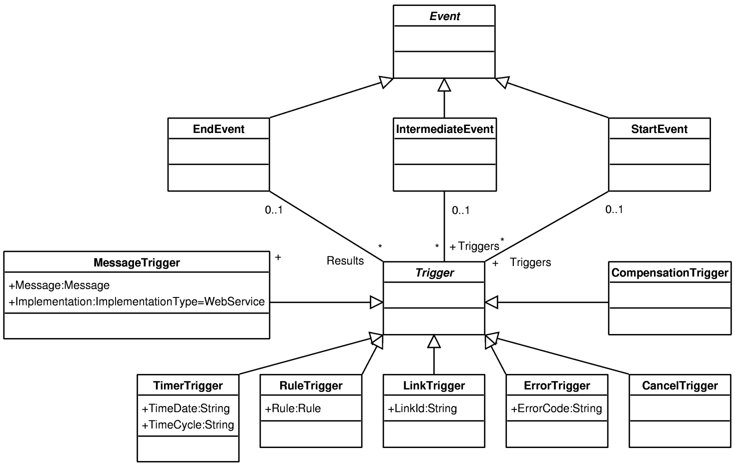


Figure 18. MOF M2 Model for the event package.

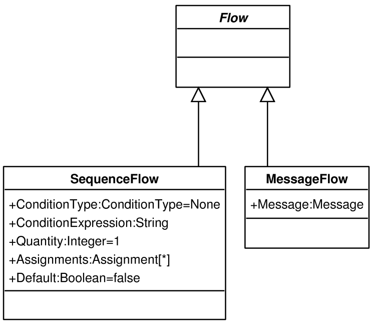

Flow
SequenceFlow
+ConditionType:ConditionType=None
+ConditionExpression:String
+Quantity:Integer=1
+Assignments:Assignment[*]
+Default:Boolean=false
MessageFlow
+Message:Message

Figure 19. MOF M2 Model for the flow package.

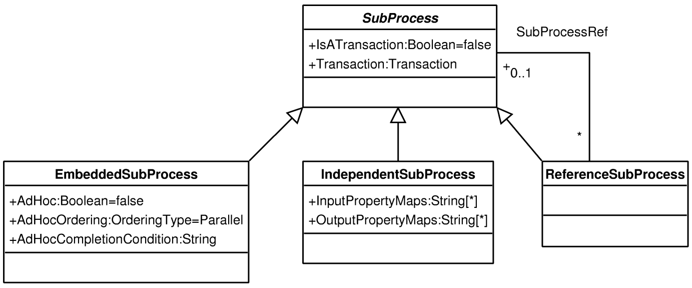

SubProcess
+IsATransaction:Boolean=false
+Transaction:Transaction
SubProcessRef
+0..1
*
EmbeddedSubProcess
+AdHoc:Boolean=false
+AdHocOrdering:OrderingType=Parallel
+AdHocCompletionCondition:String
IndependentSubProcess
+InputPropertyMaps:String[*]
+OutputPropertyMaps:String[*]
ReferenceSubProcess

Figure 20. MOF M2 Model for the subprocess package.

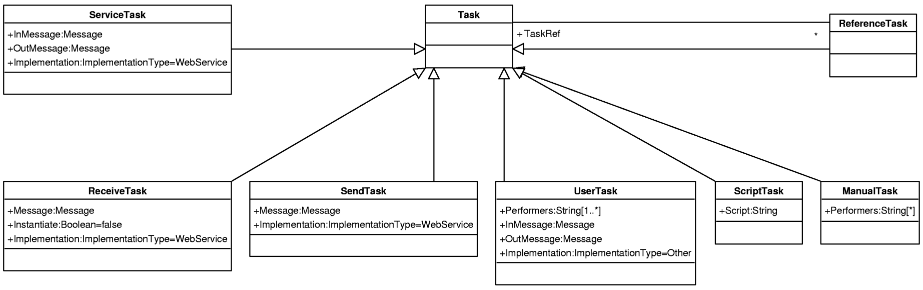


Figure 21. MOF M2 Model for the task package.

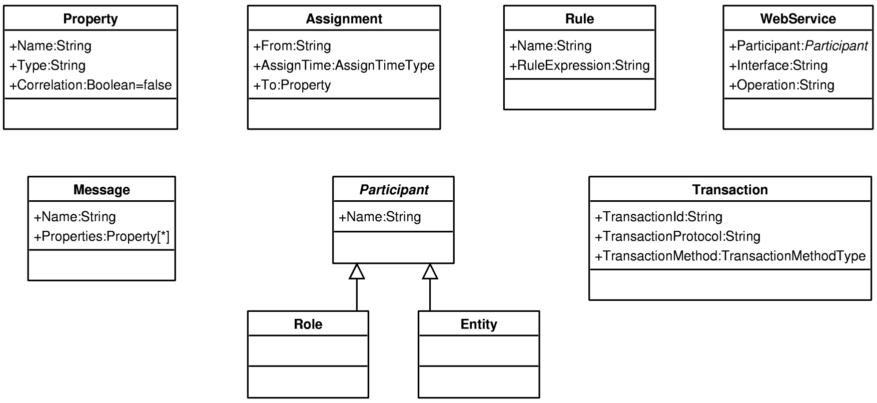


Figure 22. MOF M2 Model for the support package.

8.2 Justification of the Models

We constructed the MOF M2 model by transforming each concept from the BPMN specification into a MOF Class and each attribute of such a concept to a property of the corresponding MOF Class. If the attribute refers to another concept from the main specification, we transformed it into an association with that concept instead of a property. When making the transformation to relations or properties, we took the multiplicity constraints stated in the specification into account. We transformed attributes with a possible set of values associated into properties of an enumeration type. In this way the relation between the MOF M2 model and the specification is straightforward. We did make some exceptions to these transformation rules.

We transformed references to the Object and Expression supporting types into references to String, because Object and Expression both concepts only contain one attribute of type String and have no other distinguishing features. Hence, converting them to type String does not limit expressive power.

We did not transform a multiplicity constraint of “0..1” on an attribute. Instead, we propose that the property into which the attribute is transformed can either have a value or no value (the “null” value). This has the same effect.

The MOF does not define a Date data type. Therefore, we transformed references to Date into references to String.

We did not transform restrictions on when specific attributes must or must not be given a value (e.g., we did not express that property AdHocOrdering of the Process Class must only be given a value if property AdHoc has value True). Such restrictions can be transformed into OCL constraints.

We transformed gave the GraphicalObject Class a Name property, because each of its subclasses has a Name property. Therefore, they can inherit this property rather than defining it themselves.

In some cases where we considered this appropriate, we transformed an attribute with a possible set of values associated into a set of subclasses of the owning concept. For example, we transformed the range of values of the TaskType attribute into subclasses of the Task Class. This has the benefit that we can associate attributes that belong to a certain task type with the appropriate subclass. In case the range of values does not include a “None” value, we make the superclass abstract, such that no instances can be created for it. For example, the SubProcess Class is abstract (but the Task Class is not).

According to the BPMN specification a ReferenceSubProcess has an attribute SubProcessRef, with which it references a sub process. However, in the specification the attribute is of type Task. We believe this is a mistake and that it should be of type SubProcess.

According to the BPMN specification a CompensationTrigger references an Activity, by referencing that Activitys Id. However, the MOF allows us to reference instances of Classes directly. Therefore, we do that instead. Similarly, we let a RuleTrigger reference a Rule directly, instead of by its name.

The BPMN specification does not define Trigger as an explicit concept. However, Events do refer to such a concept. Therefore, we created a Trigger Class.

An Event can then be associated with instances of the Trigger Class. We did not define a "Multiple" Trigger nor a "None" Trigger, because these Trigger types are implicit when an Event is associated with multiple or with no Triggers.

We defined the State attribute of a DataObject in a separate Class that is related to the DataObject Class. In this way we only have to define the properties of a DataObject once, instead of once for each State of the DataObject. For the same reason we consider the RequiredForStart and ProducedAtCompletion attributes as implicitly defined by the relation between a DataObject and the Task or Process in which it is used. If we consider those attributes as explicit attributes of the DataObject, we would have to define one artifact for each Task or Process in which it is used.

We did not add graphics related concepts and attributes (i.e. the TextAnnotation and Association Class and the BoundaryVisible attribute), because we consider those outside the scope of this M2 model.

We did not add the Gate concept. Instead we added attributes to mark a SequenceFlow (rather than a Gate) as default and to add Assignments to a SequenceFlow (rather than to a Gate). Although this leads to a less clean model, in the sense that it allows for more non well-formed models (e.g. it allows a SequenceFlow to have the status default, even if it is not associated with a Gate), we believe it also leads to simpler models.

We did not consider the SwimLane class explicitly, because it does not have any distinguishing features of its own. It merely is the abstract superclass of the Pool and Lane Classes.

We used more flexible multiplicity constraints for associations with Lanes and Pools. In our model a GraphicalObject is only associated with the Lane or Pool in which it is directly contained, while in the BPMN specification a GraphicalObject is always contained in a Pool. Also, in our model a Pool does not have to contain at least one Lane, while in the BPMN specification it does (even though the specification acknowledges that there can be cases in which this is undesirable and specifies an exception for these cases). We believe this leads to a simpler model.

We separated the resulting model into a main package called "bpmn" and supporting packages contained in the main package. Figure 15 shows the diagram for the main package. Figure 16 represents the enumerations that are used as types for attributes with a predefined set of values. Figure 17 - Figure 21 represent all other packages and Figure 22 represent the support types that are defined in the specification and used as attributes types throughout. Role names on associations are not shown in the models if the Role name is the same as the name of the Class that it refers to. Multiplicity constraints of "1" are also not shown.