Semantics and Analysis of Business Process Models in BPMN

Remco M. Dijkman[1], Marlon Dumas[2,3], and Chun Ouyang[3]

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

[2] Institute of Computer Science, University of Tartu
J Liivi 2, Tartu 50409, Estonia
marlon.dumas@ut.ee

[3] Faculty of Information Technology, Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
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 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 Modeling 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 modeling, including the XML Process Definition Language (XPDL) [21] and the Activity Diagrams component of the Unified Modeling 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 suggests that BPMN users sometimes produce models with semantic errors that could be detected using existing verification technology.[4]

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.

The rest of the paper is structured as follows. To make the paper self-contained, Sect. 2 provides an introductory overview of BPMN and Petri nets. Sect. 3 presents a mapping from BPMN to Petri nets. During this formalisation, some deficiencies were identified in the BPMN standard specification. These are discussed in Sect. 4. Sect. 5 reports the tool implementation and its application to static analysis. Finally, related work is discussed in Sect. 6 and conclusions are outlined in Sect. 7.

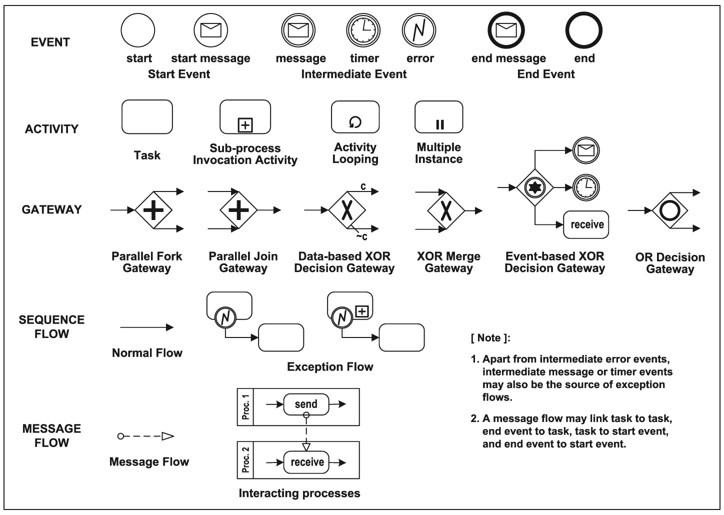
[4] See `www.brsilver.com/wordpress/2006/09/06/whats-wrong-with-this-picture/`.

2 Background

In this section, the Business Process Modelling Notation (BPMN) and the basic concepts of Petri nets are introduced in turn.

2.1 BPMN

A BPMN process is made up of BPMN elements. Figure 1 provides an overview of a set of BPMN elements related to control-flow specification. These include *objects*, *sequence flows*, and *message flows*. An object can be an *event*, *activity* or *gateway*. A sequence flow links two objects in a process diagram and denotes a control flow (i.e. ordering) relation. Message flows are used to capture the interaction between processes. The figure does not show BPMN elements that do not have a control-flow semantics such as lanes, artifacts, groups and associations.



[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. Overview of BPMN.

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 or receive a message. A *timer event* indicates that a given time instant has been reached, and an *error event* signals a fault or exception raised during the process. There are other types of events in BPMN, namely *link events*, *rule events*, *terminate events*, and *compensation events*. Link events are a notational convenience to spread a model into several "pages" and therefore they do not affect the semantics of a model. Rule events are similar to message events. They only differ in the way they are triggered: rule events are triggered by data updates while message events are triggered by arrival of messages. For the purposes of our work, we have found that we can treat them in very similar ways. Similarly, terminate events can be treated as a special type of error events. Finally, compensation events are outside the scope of this paper.

An activity can be a *task* or a *subprocess*. A task is an atomic activity, standing for work to be performed. There are 7 task types: *service*, *receive*, *send*, *user*, *script*, *manual*, and *reference*. A subprocess is a compound activity defined as a flow of other activities. It can be invoked via a *subprocess* *invocation activity*. There are *embedded* and *independent* subprocesses. An embedded subprocess is part of a process while an independent one can be called by different processes. Also, an activity may have attributes specifying its additional behaviour, such as *looping* and parallel *multiple instances*.

A gateway is defined as a routing construct. There are: *parallel fork gateways* (AND-split) for creating concurrent (sequence) flows, *parallel join gateways* (AND-join) for synchronising concurrent flows, *data/event-based XOR decision gateways* for selecting one out of a set of mutually exclusive alternative 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), *XOR merge gateways* (XOR-join) for joining a set of mutually exclusive alternative flows into one flow, and *inclusive OR decision gateways* (OR-split) for selecting any number of branches among all its outgoing flows. 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, which we call an "*exception 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 the exception occurs. Note that an error event on a normal sequence flow models "throwing" an error, while one 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.

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*, representing two participants (e.g., business entities or 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.

Finally, a *BPMN model* is composed of a set of BPMN processes which are related to each other via subprocess invocation activities or message flows.

2.2 Petri nets

Petri nets are a formal model of concurrent systems. Petri nets are particularly suited to model behavior of systems in terms of "flow", be it the flow of control or flow of objects or information. This feature makes Petri nets a good candidate for formally defining the semantics of BPMN models, since BPMN is also flow-oriented. In addition, Petri nets have been studied from a theoretical point of view for several decades, and this research had led to a number of tools that enable their automated analysis.

A Petri net is a directed graph composed of two types of nodes: *places* and *transitions*. This graphical syntax allows Petri nets to be intuitively visualized. Usually, places are represented as circles and transitions are represented as rectangles. Petri nets are bipartite graphs, meaning that an arc in the net may connect a place to a transition or vice-versa, but no arc may connect a place to another place or a transition to another transition. A transition has a number of immediately preceding places (called its *input places*) and a number of immediately succeeding places (called its *output places*).

Places are containers for *tokens*. Tokens represent the thing(s) that flow through the system. At a given point during the execution of a Petri net, each place may hold zero, one or multiple tokens. Thus, a state of a Petri net is represented as a function that assigns a number of tokens to each place in the net. Such a function is called a *marking*. For example, Figure 2(i) depicts a marking of a Petri net where there is one token in the leftmost place and no token in any other place. The state of a Petri net changes when one of its transitions fires. A transition may only fire if there is at least one token in each of its input places. In this case, we say that the transition is *enabled*. For example, in Figure 2(i), the transition labeled t_1 is enabled since this transition has only one input place and this input place has one token. When a transition fires, it removes one token from each of its input places and it adds one token to each of its output places. For example, Figure 2(ii) depicts the state obtained when transition t_1 fires starting from the marking in Figure 2(i). The token in the leftmost place has been removed, and a token has been added to each of the output places of transition t_1. In a given marking, there may be multiple enabled transitions simultaneously. In this situation, any of these enabled transitions may fire at any time. For example, in Figure 2(ii) there are two transitions enabled: "Send Invoice" and "Prepare Shipment". Any of these transitions may fire in the next execution step. Note that when the label attached to a transition is long (e.g. "Send Invoice") we place the label inside the rectangle representing this transition. Also, we will sometimes omit the label of a transition altogether. Transitions without labels correspond to "silent steps" which have no effect on the outside world, as opposed to a transition such as "Send Invoice".

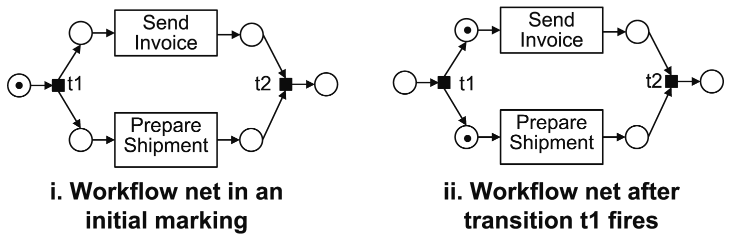

Send Invoice
Prepare Shipment
t1
t2
i. Workflow net in an initial marking
Send Invoice
Prepare Shipment
t1
t2
ii. Workflow net after transition t1 fires

Figure 2. Sample workflow net in two different states.

Constraints may be imposed on the structure of Petri nets depending on the intended purpose. In this paper, we aim at generating Petri nets that conform to the following restrictions (also known as *workflow nets*): there is a unique *source place* (i.e. a single place is not the target of any arc), a unique *sink place* (i.e. a single place that is not the source of any arc), and every other place and transition is on a directed path from the unique source place to the unique sink place. In other words, workflow nets have a distinguished start place and a distinguished end place. For example, the Petri net in Figure 2 is a workflow net. Intuitively, a workflow net models the execution of one instance of a business process, from its creation up to its completion. The initial marking of a workflow net contains a single token located in the source place, and in principle, at least one token should reach the end place. Completion policies for workflow nets will be discussed later in the paper.

3 Mapping BPMN onto Petri Nets

To facilitate the definition of the mapping, we first introduce a notion "well-formed BPMN process". Such a process has the following characteristics: (i) a start event or an exception event has just one outgoing (sequence) flow but no incoming flow; (ii) an end event has just one incoming flow but no outgoing flow; (iii) activities and intermediate events have exactly one incoming flow and one outgoing flow; (iv) fork or decision gateways have one incoming flow and more than one outgoing flows; and (v) join or merge gateways have one outgoing flow and more than one incoming flows. The first two of these conditions are syntactic restrictions inherent to the definition of start and end events in BPMN. The latter three conditions are not part of the syntactic restrictions of BPMN, but we have introduced them in order to simplify the presentation of the mapping. Importantly, these three latter conditions do not affect the generality of the proposed mapping. It is trivial to re-write any BPMN model that does not fulfill these conditions into one that does using the following transformation rules:[5]

– transform multiple incoming flows to an event or activity into one incoming flow, by preceding the corresponding object with an XOR-join gateway that has all the incoming flows of the object;
– transform multiple outgoing flows from an event or activity into one outgoing flow, by following the corresponding object with an AND-split gateway that has all the outgoing flows of the object;
– decompose an AND (or XOR) gateway with multiple incoming *and* multiple outgoing flows into an AND (or XOR) join gateway followed by an AND (or XOR) split gateways, where the join gateway has all the incoming flow and the split gateway has all the outgoing flows;
– transform a task with an input message flow *and* an output message flow into two related tasks; one with an input message flow and one with an output message flow;
– transform a process that does not have a start or an end event into a process that does, by preceding each task without incoming flows by a start event and succeeding each task without outgoing flows by an end event.

A BPMN model is well-formed if it consists of a set of well-formed BPMN processes. Below, we define a mapping of well-formed BPMN models to Petri nets. Both labelled and unlabelled Petri net transitions are used. The labelled transitions model tasks and events, and the unlabelled ones (also called "silent" transitions) capture internal actions that do not have directly visible effects.

3.1 Mapping Tasks, Events and Gateways

Figure 3 depicts the mapping from BPMN tasks, 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

[5] The BPMN2BPEL tool implementation automatically performs a pre-processing step to "normalize" the process model using these rules.

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.

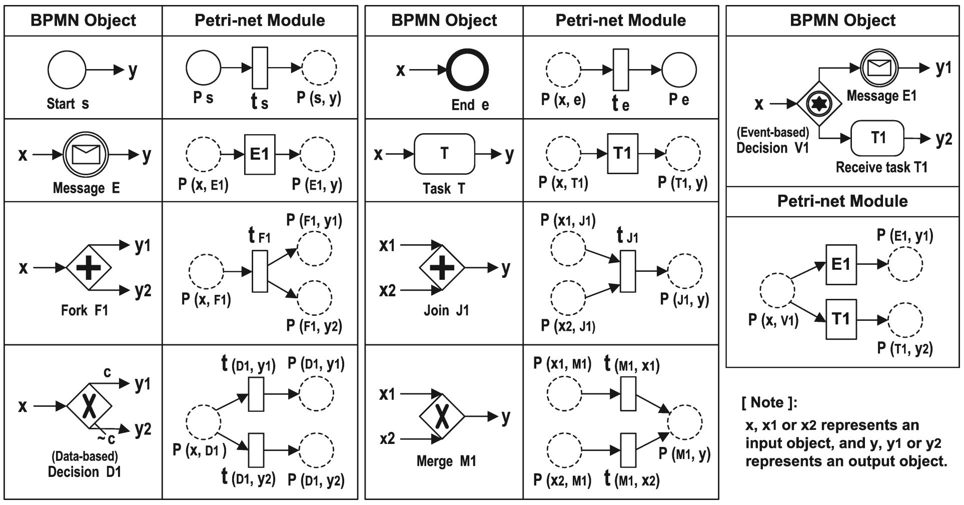


Figure 3. Mapping task, events, and gateways onto Petri-net modules.

Gateways, except event-based decision gateways and OR-split gateways, are mapped onto Petri-net modules with silent transitions capturing their routing behaviour. These mappings, as shown in Figure 3, are straightforward. For data-driven decision gateways, we model the boolean conditions in the outgoing flows as silent transitions that have a common place as input. Thus, these silent transitions will compete for a single token, and the choice as to which one will fire will be non-deterministic. In other words, we do not model the conditions themselves, but only the fact that one of the conditions will hold true when the gateway is reached. In the case of an event-based gateway, the race condition between events or receive tasks is captured by having the corresponding transitions compete for tokens in the place corresponding to the gateway's input flow (but without introducing silent transitions as we do for decision gateways). For an OR-split gateway, since its behaviour can be captured through a combination of AND-split and XOR-split gateways [5], the mapping, which is not shown in Figure 3, can be achieved accordingly.

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 thus are identified by both objects. Generally, any sequence flow is mapped onto a place except for event-based decision gateways.

3.2 Activity Looping and Multiple Instances

In BPMN, an activity may have attributes that specify special behaviour such as repetition (i.e. the activity is executed multiple times sequentially) and multiple instantiation (i.e. the activity is executed multiple times concurrently). There are two variants of sequential activity repetition: one corresponding to a "while" loop and the other corresponding to a "repeat-until" loop. From a control-flow perspective these repetition constructs can be seen as "macros", in the sense that they can be expanded in terms of decision and merge nodes as shown in Figure 4. Note that the value of attribute "TestTime" determines whether the repeated activity corresponds to a "while" loop or a "repeat-until" loop.

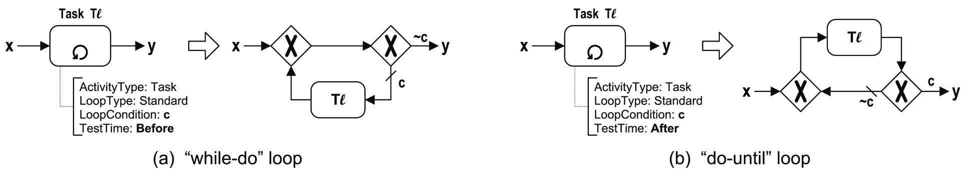


Figure 4. Macro expansions for repeated activities.

Activities with a "multiple instantiation" attribute, hitherto called *multi-instance activities*, are executed in multiple instances (i.e. copies) with each of these instances running concurrently and independently of the others. The number of instances (n) may be determined at design time or at runtime. If n is known at design time, the "multiple instantiation" construct can be regarded as a macro. Indeed, a multi-instance activity of this type can be replaced by n identical copies of the activity enclosed between an AND-split and an AND-join as shown in Figure 5. On the other hand, if n is only calculated at runtime, we need to synchronize an a priori unknown number of instances of the activity. This type of synchronization can be expressed using high-level Petri net features such as those found in Coloured Petri nets or YAWL [4]. Since we deliberately restrict the proposed mapping to produce plain Petri nets, we have chosen not to deal with multi-instance activities where n is only determined at runtime. Nonetheless, if the purpose of the mapping is to check for deadlocks in the process model, we can treat a multiple-instance activity as a single-instance one. Indeed, because the multiple instances (or copies) of the activity are executed independently, it is sufficient to check that one instance does not deadlock to ensure that the entire multi-instance activity does not deadlock. This is why our tool implementation offers the option of mapping multiple-instance activities with an a priori unknown n, with the assumption that $n = 1$.

3.3 Subprocess

A subprocess may be viewed as a standalone process. Figure 6 shows the mapping of a subprocess without exception handling and with a single start and end event. A BPMN process model may have multiple start or end events. The behaviour of such a process is however not clear in the BPMN specification

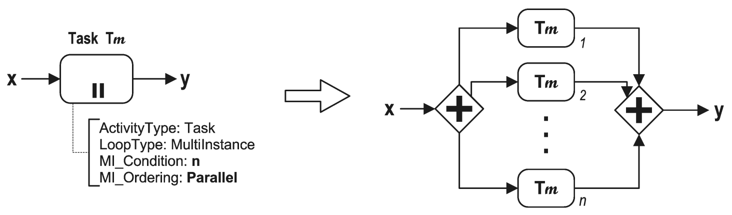


Figure 5. Macro expansion for a multi-instance activity where n is known at design time.

(see Sect. 4 for a detailed discussion). Hence, we have restricted the mapping to sub-processes with a single start event and a single end event only. This restriction could be lifted if a clear semantics for multiple (sub-)processes with multiple start and end events was given.

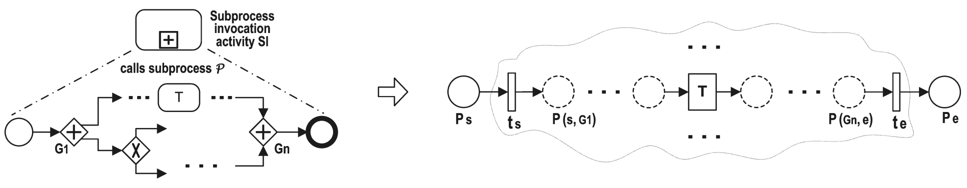


Figure 6. Mapping of a subprocess without exception handling.

Figure 7 depicts the mapping of calling a subprocess (P) via a subprocess invocation activity (SI). Two places drawn in dashed borders capture respectively the incoming and outgoing flows of activity SI. There are two new transitions: one identified as $t_{(SI,call)}$ modelling the invocation of subprocess P, the other $t_{(SI,return)}$ modelling the flow returns to the parent process after P completed.

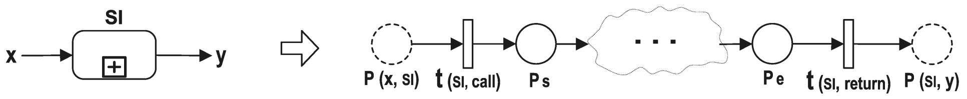


Figure 7. Calling a subprocess via a subprocess invocation activity.

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. Figure 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.

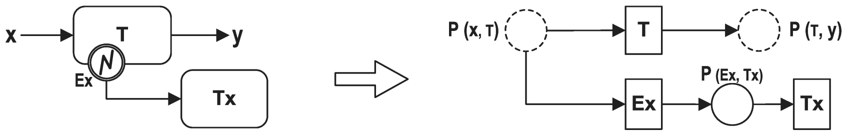


Figure 8. Mapping of a task with an exception flow.

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].

Hence, to model the cancellation of a subprocess, we adopt an approach based on the idea of “by-passing” tasks and events inside the subprocess upon occurrence of an 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. To this end each task and event has a SKIP counterpart. When the status flag is set to OK tasks and events can be executed and their SKIP counterparts cannot. When the status flag is set to NOK tasks and events cannot be executed, but their SKIP counterparts can. This way, when an exception occurs, 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 this bypassing phase finishes, the exception handling may start.

Following this approach, Figure 9 depicts the mapping of a subprocess P with an exception flow that originates from an error event Ex. Two places $p_{(P,ok)}$ and $p_{(P,nok)}$ model the OK and NOK values of the status flag attached to P, respectively. Once P starts, the flag is set to OK, and each task or event along the normal flow in P needs to check this value (via the bidirectional arc to $p_{(P,ok)}$) before it can be executed. If exception Ex occurs before subprocess P ends, the value of the flag will change from OK to NOK. As a result, any remaining task or event in P will be skipped (e.g. transition $t_{(T,skip)}$ models skipping task T). Finally, before reaching the end of P, the flow switches to the exception handling (which starts with task Tx) via transition $t_{(P,excp)}$. The occurrence of $t_{(P,excp)}$ also clears the NOK value of the status flag. Whereas, if no exception occurs, the flag will remain OK until the end of subprocess P.

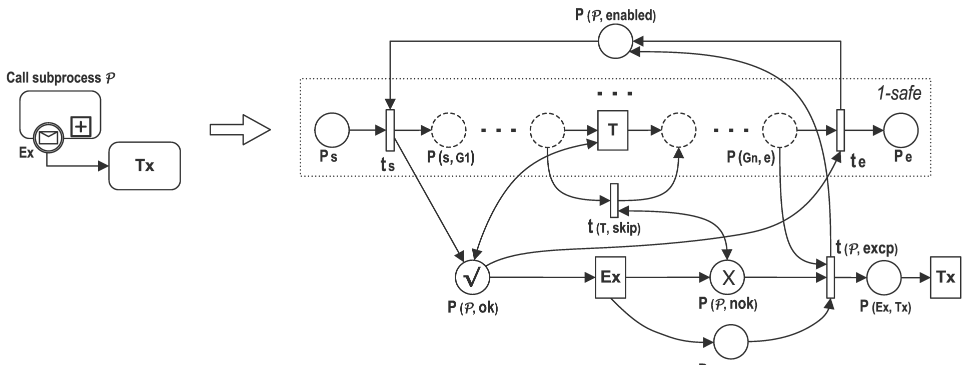


Figure 9. Mapping of a subprocess with an exception flow.

This mapping works both for exceptions thrown internally within the subprocess and for exceptions triggered from outside the subprocess (such as exceptions triggered by the arrival of a message or the expiry of a timer). In the first case, the error event at the boundary of the subprocess has a matching error event inside the subprocess, which corresponds to "throwing" the exception. In the second case, the error event at the boundary of the subprocess does not have a matching error event inside the subprocess. In terms of the mapping, this distinction makes little difference, except for the fact that in the second case, the transition E_x, corresponding to the error event, is enabled immediately when the subprocess is entered. In the latter case, this transition is only enabled when the subprocess reached event which corresponds to "throwing" the exception. Only then can the transition corresponding to the error event fire.

On the other hand, the above mapping does not work properly if there are one or more activities within the subprocess which may be active multiple times *concurrently*. Such an activity would need to be "bypassed" multiple times (as many times as it is active) and thus a counter would be required to record how many times the activity in question needs to be bypassed. Such counter can not be encoded in plain Petri nets if the maximum value of the counter is unknown [3]. To address this issue, we check that the contents of the subprocess can be mapped onto a *1-safe* Petri net. A Petri net is 1-safe if there is no marking reachable from the initial marking through a series of transition firings, in which there is more than one token in any given place. In other words, in a 1-safe net, there can not be more than one token in a given place at a time. In BPMN terms, this means that no activity will ever be enabled or running more than once concurrently. Indeed, if an activity was enabled multiple times concurrently, this would translate into the corresponding transition being enabled multiple times at once, which means that its input place would have two tokens simultaneously.

Thus, given a subprocess with an exception flow (e.g. subprocess P in Figure 9), we first translate the subprocess itself into a Petri net; then, we check that this Petri net is 1-safe, and only if it is 1-safe we may proceed to translating the associated exception flow into a Petri net module. Otherwise we do not proceed with the translation. This means that the net enclosed within the dotted box in Figure 9 must be 1-safe.

The fact that a subprocess is 1-safe ensures that, assuming the subprocess is not executed multiple times concurrently, none of its tasks or 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 executions of a given task/event is external to the subprocess. Rather than excluding these scenarios from the mapping, we propose to map them into a Petri net which would prevent a subprocess from being executed multiple times concurrently. This is achieved by introducing a "blocking mechanism" that withholds a new execution of the subprocess until the previous execution has completed. In terms of Figure 9 this would mean adding a 'status' place $p_{(P,enabled)}$, which, when holding a token, represents that the process can occur or, when not holding a token, represents that the process cannot occur. When an instance of sub-process P starts, it consumes the token, such that no other instances can start. When the instance completes, it puts the token back on the place. In this way ensuring that a new execution of P has to wait until the previous execution of P finishes. Intuitively, place $p_{(P,enabled)}$ can be viewed as holding a "resource" for execution of subprocess P. This resource is created when the top-level process starts and will be collected when the top-level process ends.

While studying the above issues, we found out that the semantics of exception flows attached to subprocesses that may be executed multiple times concurrently is unclear in the BPMN specification. This is further discussed in 4.

Finally, we note that if a subprocess P is nested within another subprocess P', the execution of P may be cancelled at due to the cancellation of P', regardless of the reason why P' is cancelled. Accordingly, each task or event in P needs to check the OK status of both P and P' to ensure that once P' is cancelled the execution of P stops as well.

3.5 Message Flow

A message flow describes the interaction between processes. It can be mapped to a place with an incoming arc from the transition modelling a send action and an outgoing arc to the transition modelling a receive action. A special case is the mapping of a message flow to a start event where the process is instantiated each time a message is received. In this case, the message flow is directly mapped to an arc linking the transition that models sending the message to the place that signals triggering the start event (e.g., place p_s in the mapping of start event s shown in Figure 3, which we refer to as the "trigger place" of start event s). Figure 10 shows four mapping rules, each capturing a case for 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.

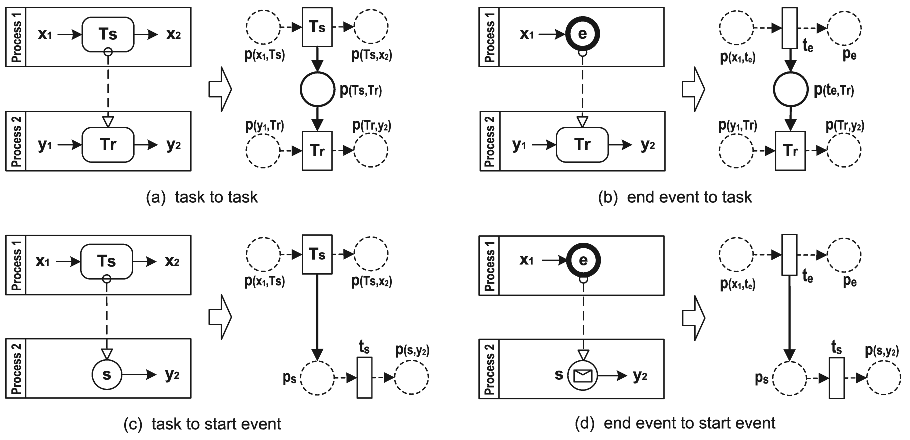


Figure 10. Mapping of message flows between BPMN processes.

The above mapping is restricted to tasks that either send or receive messages but not both (such as *user* task and *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 such as a *send* followed by a *receive*.

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. The basic idea for configuring the initial marking is to mark the trigger places for each of the start events that do not have any incoming message flows and that the processes they belong to are top-level processes. A message flow that has as a target the start event of a process, will create an instance of the process upon message delivery. So, the mapping should ensure that the trigger place of each start event with an incoming message flow does not contain a token in the initial marking, because the process can only be instantiated as a consequence of this event when a message has arrived. A special case is that each top-level process is instantiated by another process via an incoming message flow to its start event. Then, the model designer will need to determine which of the top-level process start events is triggered initially.

Figure 11 depicts the initial marking configuration for three typical examples of BPMN models that consist of two interacting 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 trigger place p_{s_1} for event s_1 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 instantiation dependency between P_1 and P_2, where neither event s_1 nor s_2 has an incoming message flow, and the interaction occur after both processes are initiated. In this case, the trigger places for both s_1 and s_2 (i.e. p_{s_1} and p_{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 trigger place (i.e. p_{s_1} or p_{s_2}) will be initially marked.

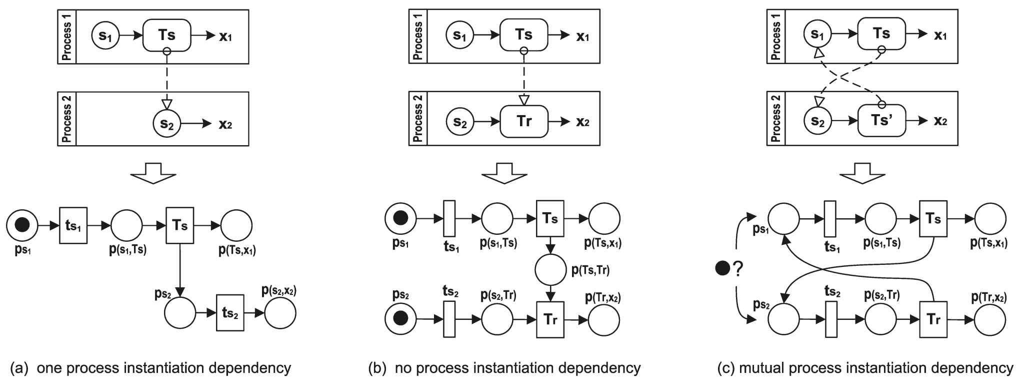

Process 1
Process 2
s1
Ts
x1
s2
Tr
Ts'
x2
ps1
ts1
p(s1,Ts)
p(Ts,x1)
ps2
ts2
p(s2,x2)
p(Ts,Tr)
p(s2,Tr)
p(Tr,x2)
●?
(a) one process instantiation dependency
(b) no process instantiation dependency
(c) mutual process instantiation dependency

Figure 11. Configuring initial markings of BPMN models with interacting processes.

4 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”. We suggest that the BPMN standard should clarify the allowed combinations of start events that may occur in the context of a lawful execution of a process model with multiple start events.

The mapping described in this paper can be applied to BPMN models with multiple start events. However, it would not yield a workflow net, but instead, a Petri net with multiple source places (i.e. places with no incoming arcs). Such Petri nets can still be analyzed from a control-flow perspective using Petri net techniques. However, one needs to establish the initial marking of the net from which the analysis should be performed.

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 there is no activity within the process instance that is enabled or active? 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 [13]. 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 standard specification should adopt this completion policy.

Again, the mapping proposed in this paper can be applied to BPMN models with multiple end events. However, it would not yield a workflow net. Instead, it would yield a Petri net with multiple *sink places* (i.e. places with no outgoing arcs). Analysis of such process models is possible using Petri net techniques provided that a completion policy is defined. In addition, if the BPMN model is 1-safe, the resulting Petri net with multiple sink places can be translated into a Petri net with a single sink place as shown in [13]. Our tool implementation does not implement this feature but instead would produce a Petri net with multiple end places.

Exception handling for multi-instance subprocesses The BPMN specification is also unclear regarding the semantics of an exception handler attached to a 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 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. 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.

5 Analysis of BPMN Models

The mapping from BPMN to Petri nets presented in Sect. 3 serves as a specification for a tool that transforms BPMN models into Petri nets. This section shows how we implemented such a tool (Sect. 5.1), how it can be used to semantically analyse the BPMN models (Sect. 5.2), and the result of testing the tool over a number of BPMN models available in practice (Sect. 5.3).

5.1 Tool Design and Implementation

Figure 12 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`.

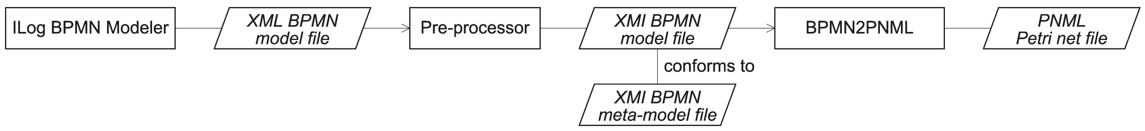


Figure 12. Structure of the BPMN to Petri net transformation tool.

The tool takes an XML Metadata Interchange (XMI) [15] file that contains the model as input. XMI is a standardised 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 standardised for BPMN yet.[6] Pending such a standard we defined our own meta-model, which is presented and motivated in details in a companion technical report [8]. This report also contains a mathematical specification of the mapping which has been used to implement the tool. Basically, the specification consists of a set of rules which specify how to transform one or a combination of BPMN element(s) into a Petri net module, i.e. a set of places, transitions and arcs whose identifiers are chosen according to the identifiers of the source BPMN element(s). By computing the union of the sets of places, transitions and arcs produced by each rule, we obtain the full Petri net. This specification was used to implement the core component of the tool, which takes as input an instance of the adopted BPMN meta-model and generates a Petri net represented as a set of places, transitions and arcs. Each individual rule is implemented as a Java method and another overarching method is used to fire the appropriate rule(s) for each element in the BPMN model and to compute the union of the obtained Petri net modules.

We use the ILog BPMN Modeller as a graphical editor to create BPMN models. Since the ILog tool does not generate standard XMI output, we implemented a simple pre-processor to transform the tool's output into XMI. The transformation tool subsequently loads this XMI representation of the BPMN model, applies the transformation and exports the resulting Petri net in the form of a PNML file. PNML [6] 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] A meta-model called BPDM has been proposed, but at the time of writing this paper this meta-model is not yet fully aligned with BPMN.

5.2 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 [9], 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.
- *Proper completion*, i.e. any process instance eventually reaches a completion state. As formulated in Sect. 4, 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*[7] 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).

Figure 13 shows three examples of BPMN models and the corresponding Petri nets, which violate the above properties. The first example (shown in Figure 13(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 (Figure 13(b)) 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 (Figure 13(c)) 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.

5.3 Empirical Evaluation

We tested BPMN2PNML on a set of models. These are: models collected from the BPMSWatch Web Log[8], models distributed with the ILOG BPMN Process Modeler, models collected from the BPMN Wikipedia entry[9], and models designed by the authors in their separate work [7, 18]. This set of models, together with the three examples shown in Figure 13, are included in the distribution of the BPMN2PNML tool.

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. Finally, the table shows the time (in

[7] A marking is *dead* if it does not enable any transition.

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

[9] `http://en.wikipedia.org/wiki/Business_Process_Modeling_Notation`

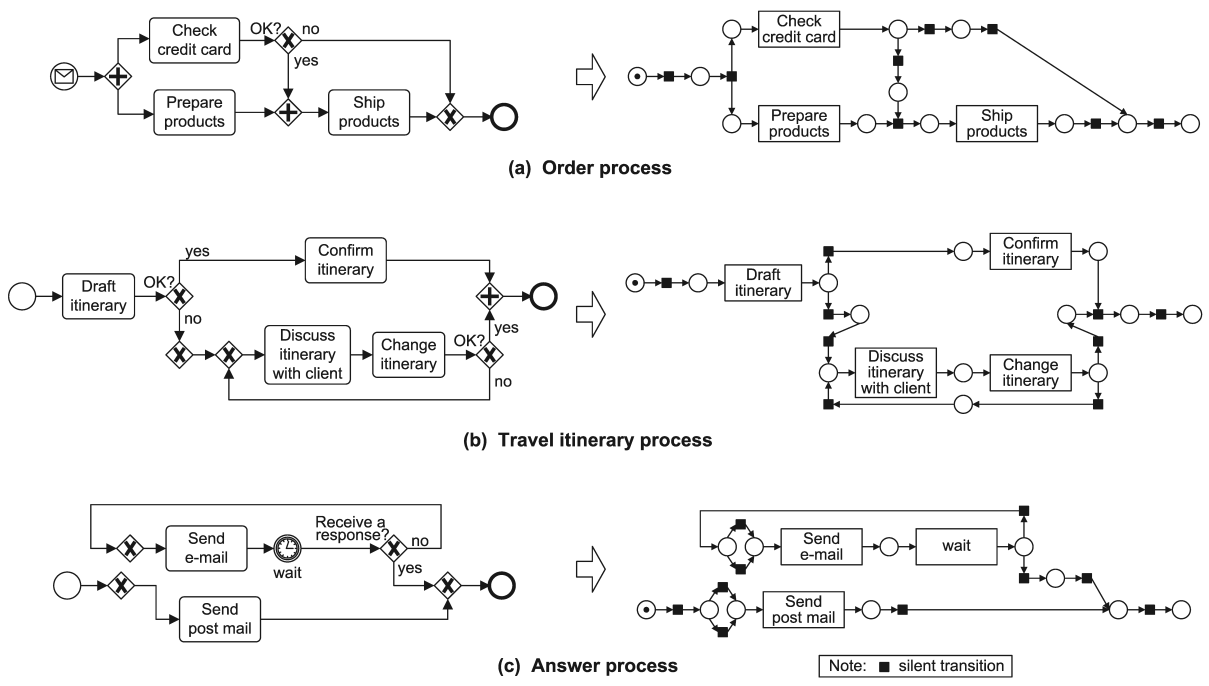


Figure 13. Examples of BPMN process models and transformations to Petri nets.

milliseconds) it took a regular desktop computer to transform the models. We distinguished between the time it took to perform the actual transformation and the total time. The total time includes the time it takes to initialize the model repository and load the BPMN model into the repository. We consider this time separately, because in practical modeling tools a repository will already have been initialized and the model will already have been loaded. Therefore, this overhead time may not be necessary in practical tools. The computation times show that the transformation can be performed within a reasonable time, even if it is not implemented efficiently. They also show that the computation time does not increase much then the model size increases, therewith providing evidence that the transformation scales.

We detected errors in models 5, 7 and 11. Model 5 contained dead tasks, model 7 contained incomplete process executions, and model 11 contained a livelock.

6 Related Work

To the best of our knowledge, the only other attempt to define a comprehensive formal semantics of BPMN is that of Wong & Gibbons [22], which use Communicating Sequential Processes (CSP) as the target formal model. Like our semantics can be checked by Petri net checking tools, their semantics can be checked by CSP checking tools such as FDR [10]. 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

Table 1. Evaluation results of BPMN2PNML.

Model	BPMN Model							Petri Net Model		Processing Times (ms)	
No.	tasks	events	XOR*	AND	subprocesses	messages	exceptions	places	transitions	total	transformation
1	11	2	9	2				31	34	16828	1234
2	7	4	4	4				23	21	13875	1297
3	9	8	3		2		2	35	39	14703	2031
4	4	2	2					10	10	13109	703
5	3	2	2	2				12	11	15375	734
6	4	8	4			4		24	20	13781	1218
7	5	12	4			5		31	25	13828	1265
8	2	2	4					11	12	13187	797
9	5	2	2					11	11	13000	641
10	5	2		2				11	9	15516	750
11	6	4	2	2				19	16	13891	1125
12	6	4	3		2		1	20	19	13625	1093
13	12	4	10	2	1		1	38	43	14657	2016

* *This includes both data-based and event-based gateways.*

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 [22] do not show how can the CSP semantics be used to detect various types of errors.

Puhlmann & Weske [20] present the foundations of a tool for static analysis of BPMN process models. This tool relies on a mapping from a subset of BPMN to π-calculus. However, this mapping only covers a small subset of BPMN. In particular, it does not take into account error handling, which is a key feature of BPMN. Puhlmann & Weske also show that the π-calculus expressions produced by this tool can be used to check the soundness of BPMN models using existing reasoning tools based on the π-calculus, in particular using the Mobility Workbench. Experiments show however that this approach does not scale beyond relatively small BPMN models (less than 10 nodes), whereas our approach can cope with models at least three times larger without being affected by efficiency issues.

Several 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 (a subclass of Petri 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. In task structures the intended termination semantics is that of *implicit termination* as defined in [13] – 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, [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 also it uses weighted arcs with potentially large weights. The 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 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.

In this paper we use workflow nets as a basis for verifying BPMN models, using standard verification techniques such as deadlock and livelock analysis [2]. Other behavior verification techniques for business process models exist [14] that allow a designer to specify pre- and postconditions of a business process. These verification techniques then check whether the specified pre- and postconditions hold for a certain process.

BPMN shares several features with the Business Process Execution Language (BPEL) [12]. A number of formal semantics of BPEL have been defined in terms of Petri nets and other models of concurrency [11, 19]. These formalizations have been used to develop verification tools for BPEL. However, the types of verification problems for BPEL are different from those in BPMN. In particular, livelocks and deadlocks that may arise in BPMN models do not arise in BPEL process definitions because of the block-structured nature of BPEL's control-flow constructs.

7 Conclusion

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 BPMN to Petri nets. The mapping has been implemented in a tool and its application to verifying the soundness of BPMN models has been tested using the ProM framework. In addition, this formalisation has permitted us to unveil a number of issues in the BPMN specification and to suggest solutions.

The proposed mapping does not fully deal with: (i) exception handling for subprocesses that may execute multiple times concurrently and (ii) OR-join gateways. These two missing features coincide with the limitations of Petri nets that motivated the design of the YAWL worflow definition language [4]. YAWL extends Petri nets with a concept of cancellation region, which allows an entire region of the net to be interrupted at once when an event occurs. In future work we plan to adapt the proposed mapping so that it generates YAWL nets in those cases where a translation to Petri nets is not feasible. The resulting YAWL nets can be analysed using techniques such as those described in [23]. The tradeoff is that verification of YAWL nets is computationally more complex than the corresponding verification problems on Petri nets. Studying this tradeoff, both analytically (in terms of worst-case complexity) and empirically, is a direction for future work.

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. Elsevier.
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. Elsevier.
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. J. Billington and et. al. The Petri Net Markup Language: Concepts, technology, and tools. In *Applications and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.
7. R.M. Dijkman. Choreography-Based Design of Business Collaborations. BETA Working Paper WP-181, Eindhoven University of Technology, 2006.
8. R.M. Dijkman, M. Dumas, and C. Ouyang. Formal semantics and analysis of BPMN process models. Technical Report Preprint 7115, Queensland University of Technology, 2007. URL: `https://eprints.qut.edu.au/archive/00007115`.
9. 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 *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005.
10. Formal Systems (Europe) Ltd. Failure-Divergences Refinement FDR2 User Manual, 1998.
11. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In *Proceedings of the International Conference on Business Process Management (BPM'2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2005.
12. D. Jordan and J. Evdemon (Editors). *Web Services Business Process Execution Language Version 2.0.* OASIS Standard, April, 2007.
13. 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.
14. S. Lu, A.J. Bernstein, and P.M. Lewis Automatic workflow verification and generation. *Theoretical Computer Science*, 353(1-3):71–92, 2006.
15. OMG. OMG XML Metadata Interchange (XMI) specification. Available Specification formal/02-01-01, Object Management Group, 2002.
16. OMG. *Unified Modeling Language: Superstructure.* UML Superstructure Specification v2.0, formal/05-07-04. Object Management Group, 2005.
17. OMG. *Business Process Modeling Notation (BPMN) Version 1.0.* OMG Final Adopted Specification. Object Management Group, 2006.
18. C. Ouyang, M. Dumas, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Pattern-based translation of BPMN process models to BPEL Web services. Accepted for publication in *International Journal of Web*

Services Research, Idea Group Publishing. Technical report version available via http://eprints.qut.edu.au/archive/00006810.

19. 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. *Science of Computer Programming*, 67(2-3):162–198, 2007. Elsevier.
20. F. Puhlmann and Matthias Weske. Investigations on Soundness Regarding Lazy Activities. In *Proceedings of the International Conference on Business Process Management (BPM'2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2006.
21. WFMC. Workflow Management Coalition Workflow Standard: Workflow Process Definition Interface – XML Process Definition Language (XPDL). Technical report, Workflow Management Coalition, 2002. Lighthouse Point, Florida, USA, 2002.
22. 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
23. 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.