Formalization and analysis of BPMN using graph grammars

Tim Kräuter[*], Harald König[†*], Adrian Rutle[*], Yngve Lamo[*]
[*]Western Norway University of Applied Sciences, Bergen, Norway
[†]University of Applied Sciences, FHDW, Hannover, Germany
tkra@hvl.no, harald.koenig@fhdw.de, aru@hvl.no, yla@hvl.no

The BPMN is a widely used standard notation for defining intra- and inter-organizational workflows to improve the communication between domain experts and software developers. However, the informal description of the BPMN execution semantics hinders BPMN from reaching its full potential since BPMN constructs are interpreted differently, and behavioral properties cannot be checked. Consequently, we propose formalizing the BPMN execution semantics based on a model transformation to graph grammars. Our formalization is one of the most advanced regarding supported BPMN constructs and enables model checking of BPMN-specific and custom properties. Moreover, we implemented our approach in an open-source web-based tool.

1 Introduction

The Business Process Modeling Notation (BPMN) is a widely used standard notation to define intra- and inter-organizational workflows to improve the communication between domain experts and software developers. However, the informal description of the BPMN execution semantics hinders its full adoption [1, 4]. Consequently, we propose a new approach to formalize the BPMN execution semantics based on graph grammars. Our formalization is one of the most advanced regarding supported BPMN constructs and enables model checking of BPMN-specific and custom properties.

The approach is depicted as a BPMN process model in figure 1. It is based on a model transformation from BPMN process models to graph grammars. Thus, our approach constructs a new graph grammar, i.e., graph transformation rules and a start graph for each BPMN process model. This is a significant difference compared to other approaches such as [1, 7], where only the BPMN process model is parsed, but the rewrite rules are fixed. Generating specific rules for each model leads to possibly more but simpler transformation rules that can be matched faster. Essentially, complexity is partly shifted from the transformation rules to their generation.

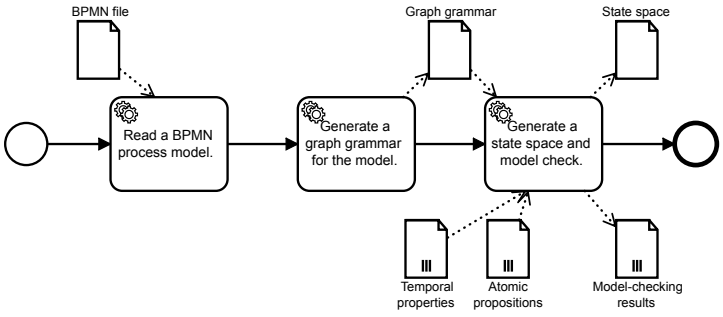


Figure 1: Overview of the proposed approach

Submitted to:
TERMGRAPH 2022

The remainder of this paper is structured as follows. First, we describe the semantics formalization using graph grammars (section 2) before explaining how this can be utilized for model checking BPMN-specific and custom properties (section 3). Then, we shortly present the implementation of our approach resulting in a web-based tool. Finally, we discuss related work regarding BPMN feature coverage in section 5 and conclude in section 6.

2 Semantics formalization

Since our approach is based on a model transformation from BPMN to graph grammars, it generates a *start graph* and *graph transformation rules* for a given BPMN process model (see figure 3a). However, we only support the BPMN constructs depicted in figure 2.

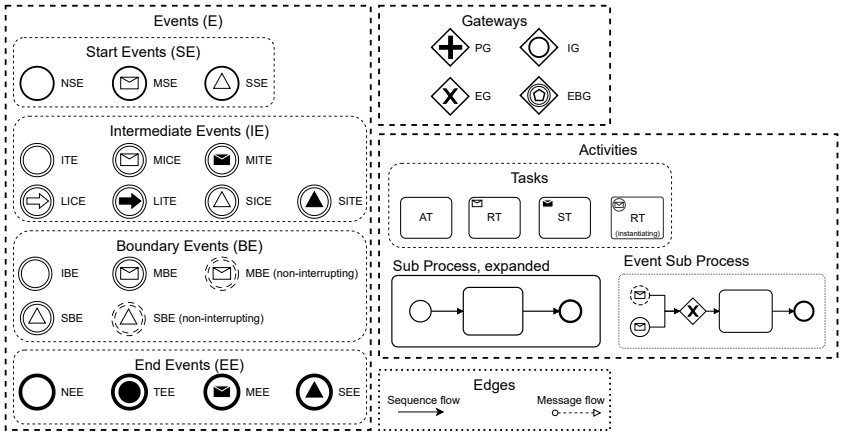
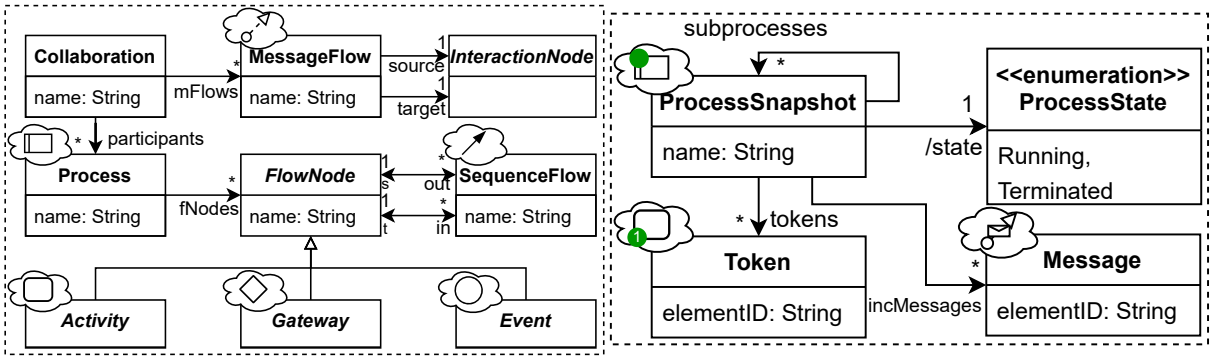


Figure 2: Overview of the supported BPMN constructs (structure adapted from [3])

Our graph-transformation semantics is token-based, as in the informal description of the BPMN specification [4]. Thus, to describe processes holding tokens during execution, we use the type graph shown in figure 3b. The type graph is depicted using a UML class diagram-like syntax.

We use ProcessSnapshot to denote a running BPMN process with a specific token distribution since it describes one state in the history of process states during the execution. Every ProcessSnapshot has a set of tokens, incoming messages, and subprocesses. A ProcessSnapshot has the state Terminated if it has no tokens or subprocesses. Otherwise, it has the state Running. A Token has an elementID, the id of either a BPMN activity or sequence flow it is located at. A Message has an elementID, pointing to a message flow. To depict graphs conforming to the type graph concisely, we introduce a concrete syntax in the clouds attached to the elements. Tokens are represented as colored circles and are drawn at their specified position in a model. Their color will match the color of the circle representing the process snapshot holding the token, which is located at the top left of the corresponding BPMN process. Using this type graph, we can now define how the start graph and graph-transformation rules for the different BPMN constructs are created.

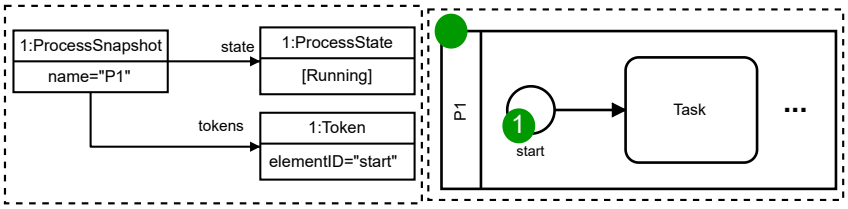
The generation of the start graph for a BPMN model is straightforward. For each process in the BPMN model, we generate a process snapshot if the process contains a none start event (NSE). Then, for each NSE, we add a token to the respective process snapshot. An example of a start graph is shown in figure 4 using abstract syntax (figure 4a) and concrete syntax (figure 4b). Furthermore, we consider allowing the user to define a start graph similar to how he can define atomic propositions for custom



(a) Simplified BPMN metamodel [4]

(b) BPMN execution type graph

Figure 3: BPMN models



(a) Abstract syntax

(b) Concrete syntax

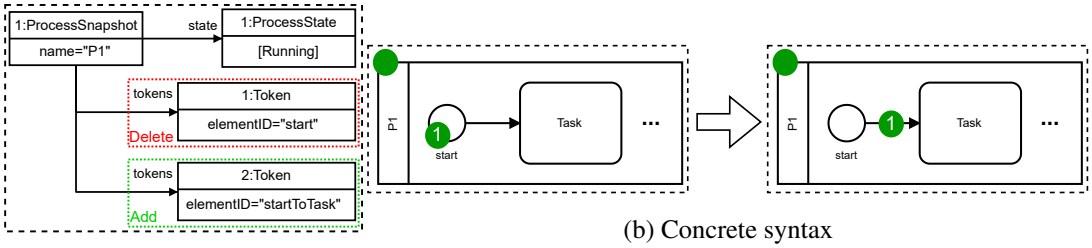
Figure 4: Example start graph

properties (see section 3.2).

We will now describe the rule generation for NSE's, tasks, and message events to give an overview of how our model transformation works. Figure 5 depicts an example graph transformation rule for a NSE in abstract and concrete syntax. The rule is straightforward and moves a token from the start event to its outgoing sequence flow.

The rule in figure 6 represents the start of the task, which will move one token from the incoming sequence flow to the task itself.

The left rule in figure 7 realizes a message throw event, and the right rule implements a message catch event. The message catch event rule is straightforward, consuming a token and a message and creating an outgoing token. The message throw event rule moves the token through the event and sends



(a) Abstract syntax

(b) Concrete syntax

Figure 5: Example graph transformation rule for a NSE

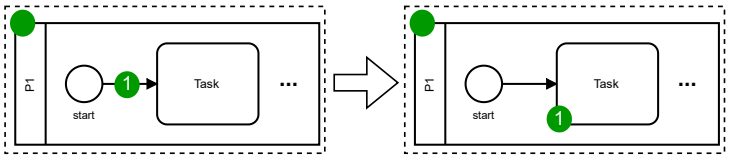


Figure 6: Example graph transformation rule to start a task

a message to a waiting process snapshot, which must have a token waiting at the corresponding message receive event. However, sending this message to a different process snapshot is optional. This can be implemented using nested quantification [5].

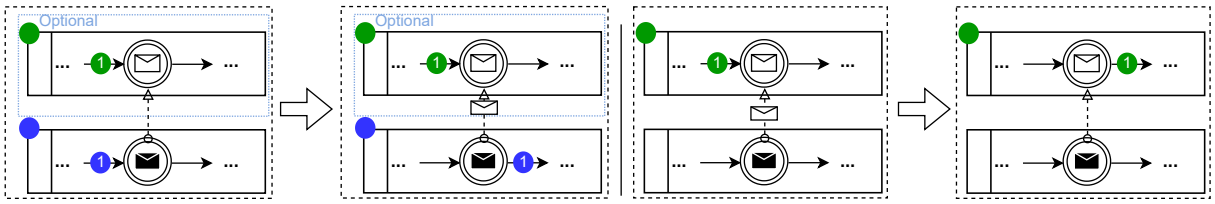


Figure 7: Rules for message intermediate throw/catch events.

End events consume but do not produce tokens. Thus, process termination can be implemented using a general rule applicable to all process snapshots. The rule is automatically generated once during the model transformation to graph grammars and is used to terminate processes, sub processes, and event sub processes. It uses negative-application conditions to forbid tokens and sub processes for a process snapshot and then changes its state from running to terminated[1].

3 Model checking BPMN

Model checking a BPMN process model is straightforward after a graph grammar has been generated. Besides a graph grammar, a set of temporal properties to be checked and the atomic propositions used in the properties must be supplied (see figure 1). An atomic proposition can be formalized as a graph and holds in a given state if it is a subgraph of the graph representing the state. This enables model checking of temporal properties using the defined atomic propositions.

Like other work, we differentiate between *BPMN-specific properties* defined generally for all BPMN process models and *custom properties* tailored towards a particular BPMN process model. We will now give an example of two predefined BPMN-specific properties and how they can be checked using our approach. Then, we describe how custom properties can be constructed and checked.

3.1 BPMN-specific properties

Two BPMN-specific behavioral properties, namely, Safeness and Soundness, are defined in [2]. Soundness is further decomposed into (i) *Option to complete*: any running process instance must eventually complete, (ii) *Proper completion*: at the moment of completion, each token of the process instance must be in a different end event, (iii) *No dead activities*: any activity can be executed in at least one process

[1] The terminate rule implemented in Groove is contained in the artifacts of this paper [6].

instance [2]. We do not consider structural properties since they can be checked using a standard process modeling tool without implementing execution semantics. As an example, we will now describe how to implement the *Safeness* and *Option to complete* properties.

A BPMN process model is *safe* if, during its execution, at most one token occurs along the same sequence flow [2]. *Safeness* is checked using the LTL property defined in (1). The atomic property Unsafe is true if two tokens of one process snapshot have the same position[2].

Option to complete is checked using the LTL property defined in (2). The atomic property AllTerminated is true if there exists no process snapshot in the state Running, i.e., all process snapshots are Terminated[2].

$$\Box(\neg \text{Unsafe}) \quad (1) \qquad \Diamond(\Box(\text{AllTerminated})) \quad (2)$$

Both properties can be checked using our implementation [6]. To fully check Soundness, the proper completion and no dead activities properties must be checked. However, the information to check these properties is contained in the state spaces of the generated graph grammars.

3.2 Custom properties

To make model checking user-friendly, we envision users defining atomic propositions in the BPMN syntax, i.e., the concrete syntax introduced earlier. Thus, to define an atomic proposition, we let the user attach tokens to his BPMN process model, which we can automatically convert to a graph condition.

For example, the token distribution shown in figure 8 defines two running process snapshots with a token in task A. Differently colored tokens define different process snapshots. Thus, a user must not be aware of the graph transformation semantics used for execution, which is a significant advantage compared to other approaches.

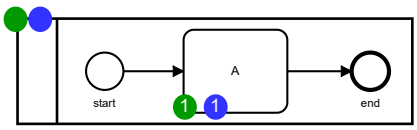


Figure 8: Token distribution defining an atomic proposition.

The token visualization was significantly inspired by the excellent bpmn-js-token-simulation[3].

4 Implementation

Figure 9 depicts a screenshot of the implemented tool. The tool is open-source, publicly available, and does not require any installation [6].

The first two steps of our approach, i.e., reading and transforming BPMN models to graph grammars, are implemented and usable through the web-based tool. Then one can use the graph-transformation tool Groove[4] for state space generation and model checking locally. We are currently working on extending our tool such that model-checking can be done without installing Groove locally. To evaluate the correctness of our implementation, we created a comprehensive test suite, which demonstrates correct rule generation for the implemented BPMN constructs [6].

[2]Groove rules for the atomic properties Unsafe and AllTerminated are contained in the artifacts of this paper [6].
[3]`https://github.com/bpmn-io/bpmn-js-token-simulation`
[4]`https://groove.ewi.utwente.nl/about`

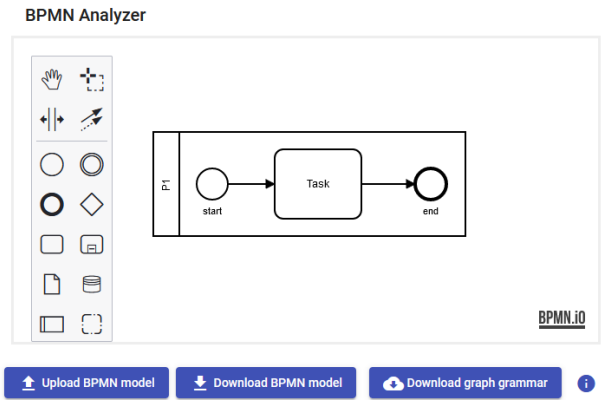


Figure 9: Screenshot of the tool

5 Related work

A BPMN formalization based on in-place graph transformation rules is given in [7]. The formalization covers a substantial part of the BPMN specification, including complex concepts such as inclusive gateway merge and compensation. In addition, graph transformation rules are visual and thus can easily be matched to the informal description of the execution semantics in the specification [4]. The graph transformation rules were implemented in a prototype using GrGen.NET. Unfortunately, the implementation is not publicly accessible anymore. Moreover, they do not support model checking since their goal is only formalization.

The tool BProVe[5] is based on formal BPMN semantics given in rewriting logic and implemented in the Maude system. Using these semantics, BProVe enables the verification of custom LTL properties and BPMN-specific properties, such as Safeness and Soundness. Furthermore, the tool is accessible online, not requiring any installation [1].

The verification framework fbpmn uses first-order logic to formalize and check BPMN process models [3]. This formalization is then realized in the TLA$^+$ formal language and can be model-checked using TLC. Their framework and related information is open source and freely available online[6]. Similar to BProVe, fbpmn allows checking BPMN-specific properties, such as Safeness and Soundness. However, they do not allow a user to define custom temporal properties.

We looked in detail at these three approaches since they support a significant subset of the BPMN constructs and have accessible and well-documented tools. Nevertheless, each approach supports a different subset of the BPMN constructs. The coverage of BPMN constructs greatly impacts how useful each approach is in practice. Table 1 depicts which BPMN constructs are supported by the different approaches compared to our approach.

Van Gorp et al. [7] cover a large part of the BPMN semantics. However, they do not support Event-based gateways and event subprocesses, while their support for boundary events is minimal. Especially, Event-based gateways are often used in practice. Corradini et al. [1] support message and terminate events. In addition to [1], Houhou et al. [3] support timer and the use of message and timer events as both interrupting and non-interrupting boundary events. However, many other event types exist and are used in practice.

[5] http://pros.unicam.it/bprove/
[6] https://github.com/pascalpoizat/fbpmn

Table 1: Constructs supported by different BPMN formalizations (overview based on [7]).

Feature	Van Gorp et al. [7]	Corradini et al. [1]	Houhou et al. [3]	This paper
Instantiation and termination				
Start event instantiation	X	X	X	X
Exclusive event-based gateway instantiation	X			X
Parallel event-based gateway instantiation				
Receive task instantiation				X
Normal process completion	X	X	X	X
Activities				
Activity	X	X	X	X
Subprocess	X	X	X	X
Ad-hoc subprocesses				
Loop activity	X			
Multiple instance activity				
Gateways				
Parallel gateway	X	X	X	X
Exclusive gateway	X	X	X	X
Inclusive gateway (split)	X	X	X	X
Inclusive gateway (merge)	X		X	X
Event-based gateway		X[1]	X	X
Complex gateway				
Events				
None Events	X	X	X	X
Message events	X	X	X	X
Timer Events			X	
Escalation Events				
Error Events (catch)	X			
Error Events (throw)	X			
Cancel Events	X			
Compensation Events	X			
Conditional Events				
Link Events	X			X
Signal Events	X			X
Multiple Events				
Terminate Events	X	X	X	X
Boundary Events	X[2]		X[3]	X
Event subprocess				X

[1] Does not support receive tasks after event-based gateways.
[2] Only supports interrupting boundary events on tasks.
[3] Only supports message and timer events.

Referring to table 1, we conclude that our formalization is comprehensive but still lacks support for some of the more advanced event types. An implementation of the missing event types is feasible, as shown in [7].

6 Conclusion & Future work

We presented a formalization of the BPMN execution semantics based on a model transformation to graph grammars. Thus, complexity shifts from the graph transformation rules to the model transformation when compared to fixed graph transformation rules. Our resulting BPMN formalization is comprehensive and supports model checking. In addition, we started implementing our approach in a web-based tool to make our ideas easily accessible to other researchers and potentially practitioners in the future.

We aim to improve our formalization and resulting tool in multiple ways in the future. First, we intend to extend our formalization to support even more BPMN constructs for example error, cancel and compensation events. Second, we plan to evaluate our approach on syntactical models and on models from open repositories such as the "BPM Academic Initiative Model Collection" [8] and "Camunda BPMN for Research"[7]. Third, we want to extend the features of our tool. One should be able to define atomic propositions for model checking in the tool directly, as described in section 3. Furthermore, counterexamples found during model checking should be visualized directly in the tool, like the implementation in [3], such that users must not understand the underlying implementation in Groove.

References

[1] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, Francesco Tiezzi & Andrea Vandin (2021): *A Formal Approach for the Analysis of BPMN Collaboration Models*. Journal of Systems and Software 180, p. 111007, doi:10.1016/j.jss.2021.111007.

[2] Flavio Corradini, Chiara Muzi, Barbara Re & Francesco Tiezzi (2018): *A Classification of BPMN Collaborations Based on Safeness and Soundness Notions*. Electronic Proceedings in Theoretical Computer Science 276, pp. 37–52, doi:10.4204/EPTCS.276.5.

[3] Sara Houhou, Souheib Baarir, Pascal Poizat, Philippe Quéinnec & Laïd Kahloul (2022): *A First-Order Logic Verification Framework for Communication-Parametric and Time-Aware BPMN Collaborations*. Information Systems 104, p. 101765, doi:10.1016/j.is.2021.101765.

[4] Object Management Group (2013): *Business Process Model and Notation (BPMN), Version 2.0.2*. https://www.omg.org/spec/BPMN/.

[5] Arend Rensink (2006): *Nested Quantification in Graph Transformation Rules*. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro & Grzegorz Rozenberg, editors: Graph Transformations, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 1–13, doi:10.1007/11841883_1.

[6] Tim Kräuter: *Artifacts - TERMGRAPH-2022*. https://github.com/timKraeuter/TERMGRAPH-2022.

[7] Pieter Van Gorp & Remco Dijkman (2013): *A Visual Token-Based Formalization of BPMN 2.0 Based on in-Place Transformations*. Information and Software Technology 55(2), pp. 365–394, doi:10.1016/j.infsof.2012.08.014.

[8] Mathias Weske, Gero Decker, Marlon Dumas, Marcello La Rosa, Jan Mendling & Hajo A. Reijers (2020): *Model Collection of the Business Process Management Academic Initiative*, doi:10.5281/zenodo.3758705.

[7] `https://github.com/camunda/bpmn-for-research`