Formalization and analysis of BPMN using graph transformation systems

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. However, the informal description of the BPMN execution semantics leads to different interpretations of BPMN constructs and difficulties in checking behavioral properties. In this paper, we propose a formalization of the execution semantics of BPMN that, compared to existing approaches, covers most of the BPMN constructs. Our approach is based on a higher-order transformation from BPMN models to graph transformation systems. As proof of concept, we have implemented our approach in an open-source web-based tool.

1 Introduction

Business Process Modeling Notation (BPMN) [6] is a widely used standard notation to define intra- and inter-organizational workflows. However, the informal description of the BPMN execution semantics leads to different interpretations of BPMN constructs and difficulties in checking behavioral properties [1]. To this end, we propose a formalization that, compared to other approaches, covers most of the BPMN constructs.

Our approach is based on a higher-order transformation from BPMN process models to graph transformation systems using rule generation templates. Thus, our approach constructs a new graph transformation system, 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, 10], 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. The generated rules are tailored to a given process model and thus are simpler than the general rules in [10]. Furthermore, our approach is easily accessible since it is implemented in an open-source web-based tool.

The remainder of this paper is structured as follows. First, we describe the BPMN semantics formalization using graph transformation systems (section 3) before explaining how this can be utilized for model checking BPMN-specific and custom properties (section 4). Then, we shortly present the web-based tool implementing our approach. Finally, we discuss related work regarding BPMN construct coverage in section 6 and conclude in section 7.

2 Preliminaries

In this paper, we apply graph transformations to formalize the execution semantics of BPMN. Thus, in this section, we will briefly introduce BPMN and its execution semantics. Please refer to [4] or the

Submitted to:
TERMGRAPH 2022

© Kräuter *et al.*
This work is licensed under the
Creative Commons Attribution License.

BPMN specification [6] for further information about BPMN. Furthermore, we describe the theoretical background behind our application of graph transformations.

2.1 BPMN

BPMN is a widely used standard notation to define intra- and inter-organizational workflows. Figure 1 depicts the structure of BPMN process models and the corresponding BPMN symbols contained in clouds.

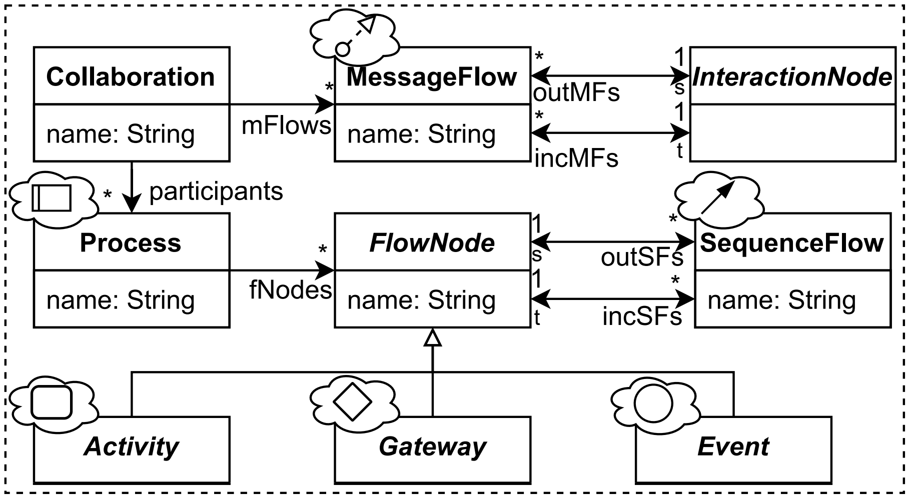


Figure 1: Simplified excerpt of the BPMN metamodel [6]

A BPMN process model is represented by a Collaboration that has participants and MessageFlows between InteractionNodes (see figure 1). Each participant is a Process containing FlowNodes connected by SequenceFlows. A FlowNode is either an Activity, Gateway, or Event. Many types of Activities, Gateways, and Events exist. Activities represent certain tasks to be carried out during a process, while events may happen during the execution of these tasks. Furthermore, gateways model conditions, parallelizations, and synchronizations [4].

The BPMN execution semantics are described using the concept of *tokens* [6]. BPMN process models are executed by triggering one or more of their start events, leading to the creation of a token at each triggered start event. Tokens are transported between flow nodes by sequence flows (arrows). Activities can start when at least one token is located on an incoming sequence flow. The start of an activity will move the incoming token to the activity. When an activity finishes, it will add one token at each outgoing sequence flow. Different gateway types exist, for example, for parallelization, synchronization, XOR, and OR distribution of tokens. Events delete and add tokens similar to activities but have additional semantics depending on their type. For example, message events will add or delete messages.

2.2 Theoretical background

We use typed attributed graphs for the formalization of the BPMN execution semantics. Each state, i.e., token distribution during the execution of a BPMN model, is represented as an attributed graph typed by the BPMN execution type graph, which is introduced in section 3.

Regarding graph transformation, we utilize the single-pushout (SPO) approach with negative application conditions (NAC) [3]. In addition, we utilize *nested rules* with quantification to make parts of a rule applied repeatedly or optionally [7, 8]. Moreover, we utilize NACs to implement more intricate parts in the BPMN execution semantics, such as the termination of processes. In SPO rewriting, a graph transformation rule is defined as a partial graph morphism $L \to R$, where in our case, L and R are typed attributed graphs.

3 BPMN semantics formalization

Since our approach is based on a higher-order transformation from BPMN to graph transformation systems, we generate a *start graph* and *graph transformation rules* for a given BPMN process model. The approach supports the BPMN constructs depicted in figure 2. BPMN constructs are divided into Events, Gateways, Activities, and Edges. Events and Activities are further divided into subgroups.

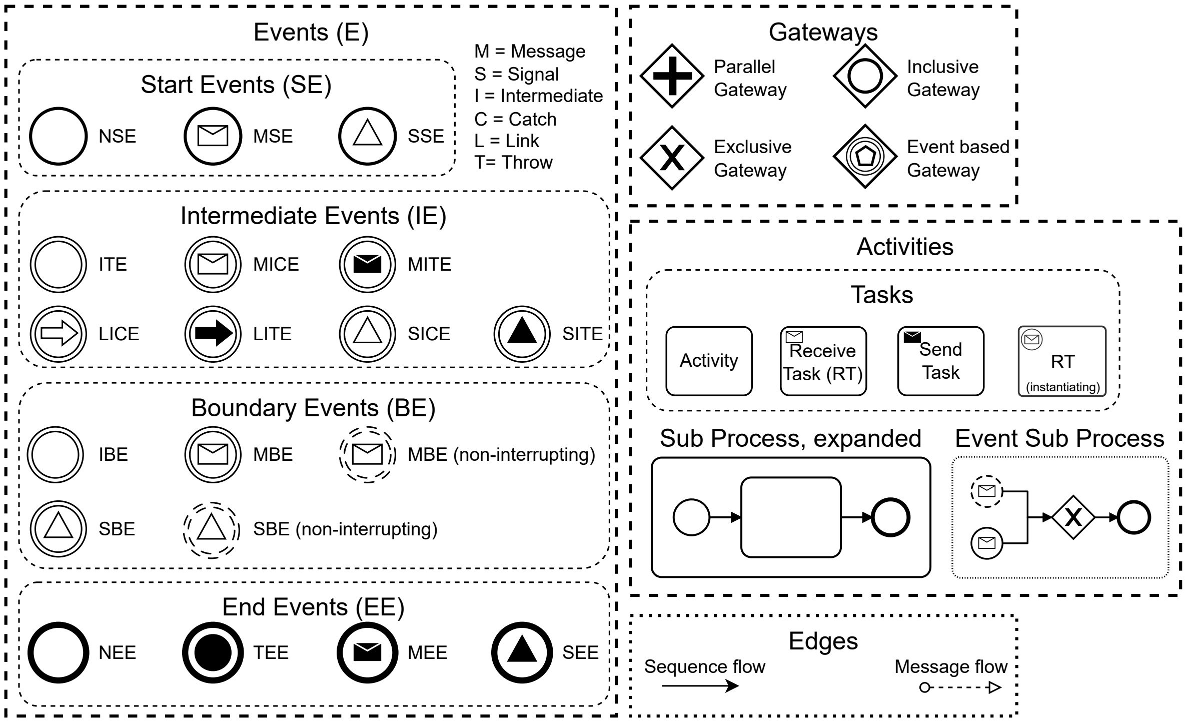


Figure 2: Overview of the supported BPMN constructs (structure adapted from [5])

Our formalization of BPMN is token-based, as in the informal description of the BPMN specification [6]. Thus, to describe processes holding tokens during execution, we use the type graph shown in figure 3. 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 that 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, which points to the BPMN Activity or the SequenceFlow at which it is located. A Message has an elementID pointing to a MessageFlow. To concisely depict graphs conforming to this type graph, we introduce a

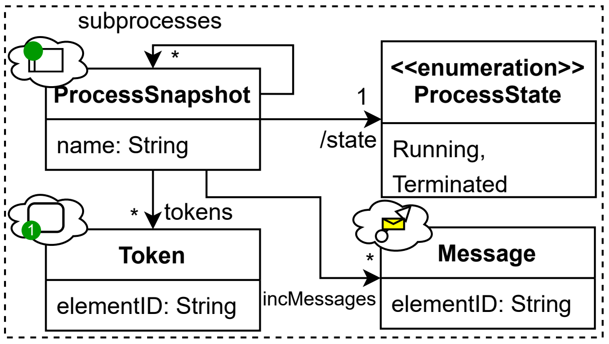


Figure 3: BPMN execution type graph

concrete syntax in the clouds attached to the elements. Our concrete syntax extends the BPMN syntax by adding process snapshots, tokens, and messages. Tokens are represented as colored circles drawn at their specified positions in a model. In addition, we use colored circles at the top left of the bounding box, representing instances of the BPMN Process; these circles represent process snapshots. The token's color must match the color of the process snapshot holding the token. The concrete syntax was inspired by the bpmn-js-token-simulation[1]. 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. First, 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 and concrete syntax. Furthermore, we consider allowing the user to define a start graph similar to how he can define atomic propositions for custom properties (see section 4.2).

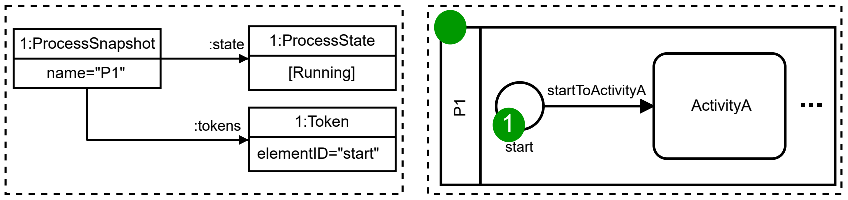


Figure 4: Example start graph in abstract (left) and concrete syntax (right)

The higher-order transformation generates one or more graph transformation rules for each FlowNode in a BPMN model. To give an intuition about the transformation, we will first describe example results, i.e., generated rules for a NSE, a task, a *message intermediate catch event* (MICE), and a *message intermediate throw event* (MITE) (see figure 2). Afterward, we will explain how these—and other—rules are created by our higher-order transformation.

Figure 5 depicts an example graph transformation rule ($L \to R$) for a NSE in abstract syntax. The rule is straightforward and moves a token from the start event to its outgoing sequence flow. For the rest of the paper, we will depict all rules in the concrete syntax introduced earlier. The rule from figure 5 depicted in concrete syntax is shown in figure 6.

The rule in figure 7 represents the start of a task, which will move one token from the incoming sequence flow to the task itself.

[1]https://github.com/bpmn-io/bpmn-js-token-simulation

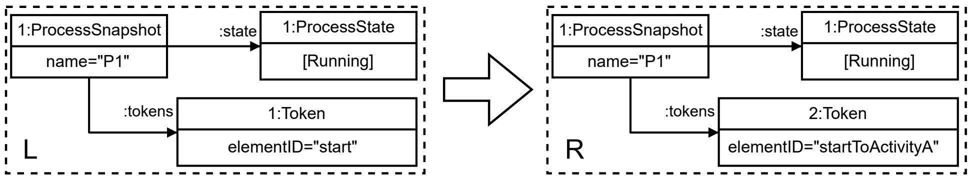


Figure 5: Example graph transformation rule for a NSE (abstract syntax)

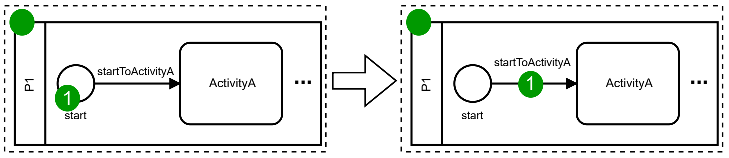


Figure 6: Example graph transformation rule for a NSE (concrete syntax)

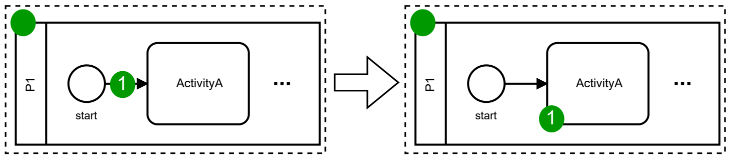


Figure 7: Example graph transformation rule to start a task.

The left rule in figure 8 realizes a MITE, and the right rule implements a MICE. The MICE rule deletes a token and a message and adds an outgoing token. The MITE rule moves the token through the event and may send a message to a waiting process snapshot if it has a token waiting at the corresponding MICE. An optional existential quantifier is used to send a message only if the receiving process is ready to receive it.

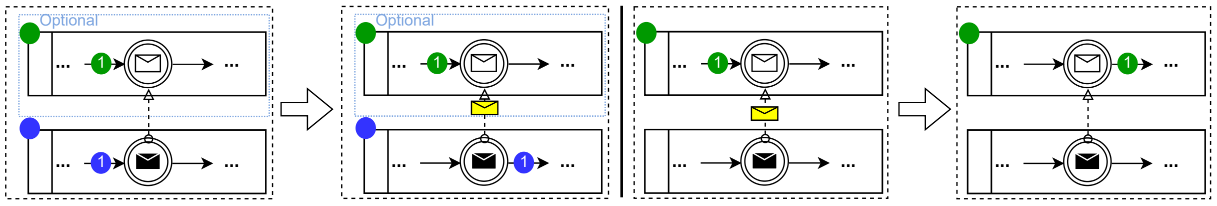


Figure 8: Rules for MITE (left) and MICE (right)

To summarize, we described four example rules and introduced a concrete syntax to depict them concisely and understandably. In the following subsections, we use this concrete syntax to describe how these rules and rules for other flow nodes are generated by our higher-order transformation. Elements of the higher-order transformation are depicted using rule generation templates that describe how specific rules are created for various flow nodes. We only explain rule generation for (i) process instantiation and termination, (ii) activities and subprocesses, (iii) gateways, as well as (iv) message events due to space constraints. However, our implementation covers all concepts shown in figure 2.

3.1 Process instantiation and termination

Figure 9 depicts the rule generation templates for start and end events (NSE and NEE in figure 2). All rule generation templates show a BPMN structure defined in the left column and the applicable rule generation template in the right column. The structures in the left column show instances of the BPMN metamodel (figure 1), and the ones in the right column show the rules typed by the BPMN execution type graph (see figure 3). All the rule generation templates can be found in [9].

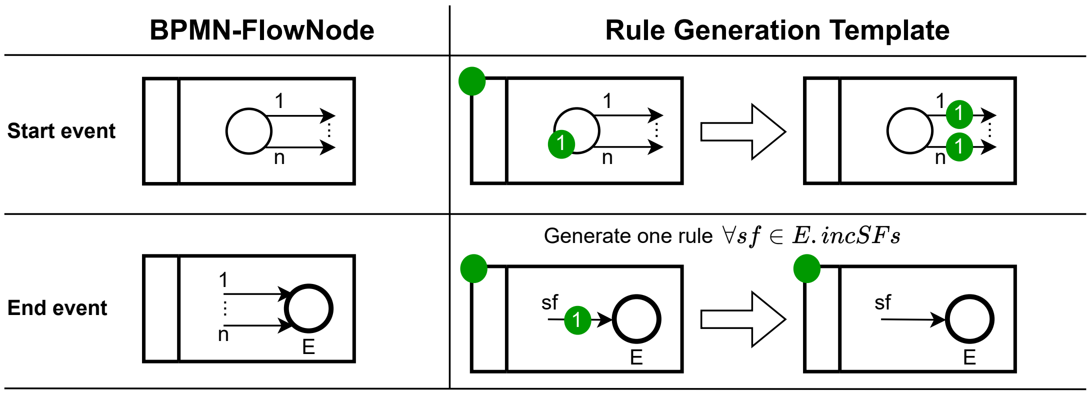


Figure 9: Rule generation templates for start and end events

The start event rule template generates the start event rule in figure 6. The tokens located on the start events are deleted by start event rules, while one token for each outgoing sequence flow is added. If a start event has more than one outgoing sequence flow, it functions as an *implicit parallel gateway*, forking the control flow by creating one token for each of the sequence flows (see parallel gateway in figure 12). Initially, the tokens on the start events are given by the start graph of the graph transformation system (see, e.g., figure 4).

The generated end event rules delete tokens one by one for each incoming sequence flow. However, they do not terminate processes. Process termination is implemented with a generic rule—independent of the input BPMN model—which is applicable to all process snapshots. The termination rule (see figure 10) is automatically generated once during the higher-order transformation. It is used to terminate processes and subprocesses if they do not have any tokens. The rule uses negative-application conditions to only change the state of the process snapshot from running to terminated if it has no tokens and subprocesses. The termination rule in Groove is given as an artifact [9].

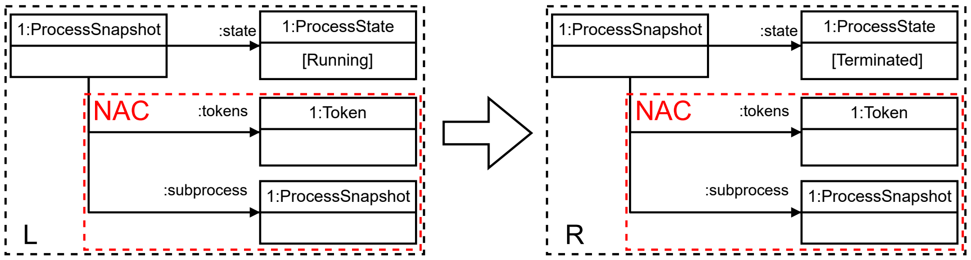


Figure 10: Termination rule

3.2 Activities & Subprocesses

Figure 11 depicts the rule generation templates for activities and subprocesses (see figure 2). Activity execution is divided into two steps implemented by two rule templates. The upper template generates one rule for each incoming sequence flow to start the activity. An activity can be started using a token positioned at any of its incoming sequence flows. Thus, multiple incoming sequence flows represent an *implicit exclusive gateway* (see exclusive gateway in figure 12). This rule template generates the sample rule in figure 7.

The bottom rule template generates one rule that ends the activity. It deletes a token at the activity and adds one token at each outgoing sequence flow. Like start events, this implicitly encodes the same forking behavior as a parallel gateway (see figure 12).

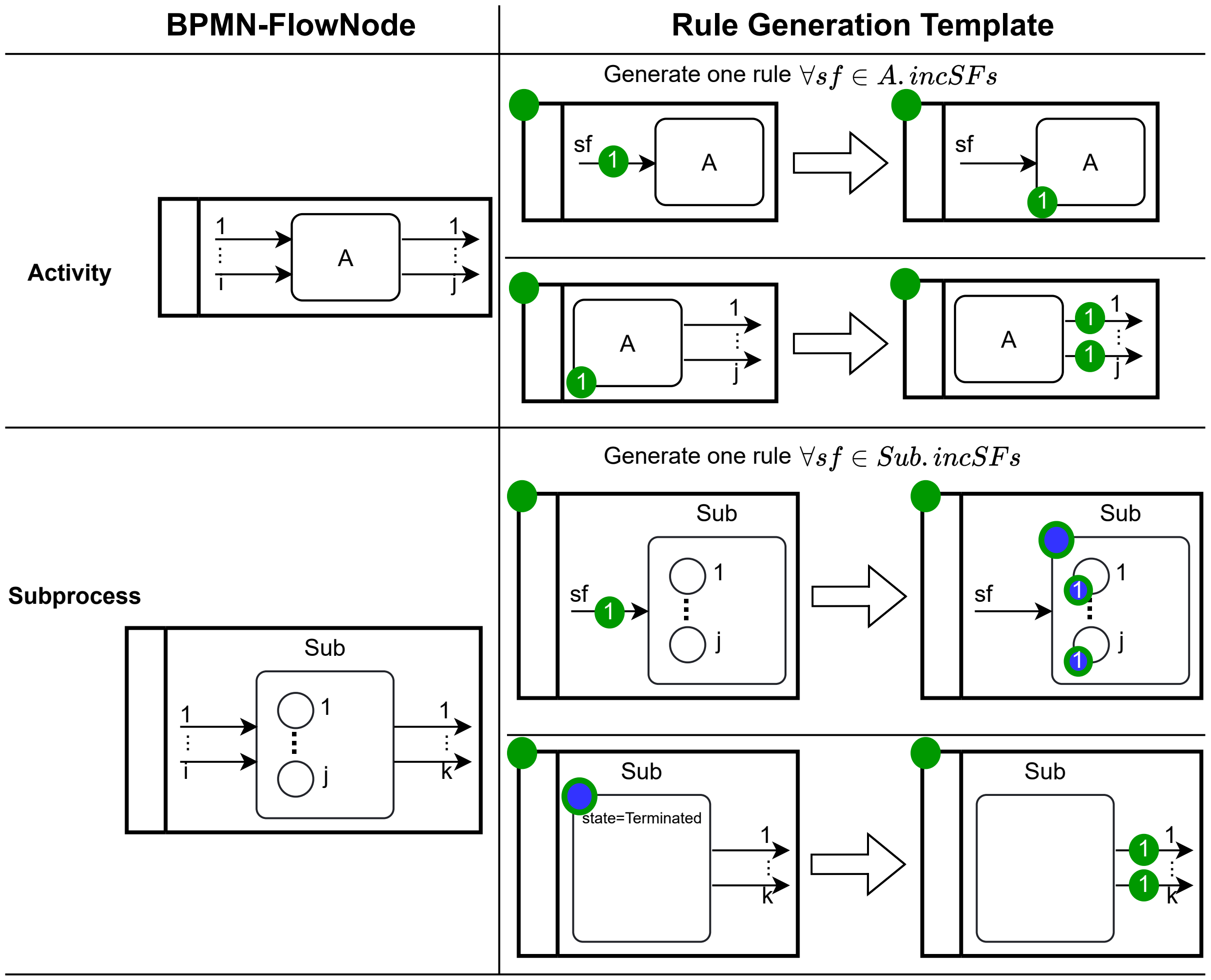

BPMN-FlowNode
Rule Generation Template
Generate one rule $\forall sf \in A.incSFs$
Activity
Generate one rule $\forall sf \in Sub.incSFs$
Subprocess
Sub
sf
A
state=Terminated

Figure 11: Rule generation template for activities and subprocesses

Subprocess execution is similar to activity execution. The upper template generates one rule for each incoming sequence flow. The rule deletes an incoming token and adds a process snapshot representing a subprocess. The addition of this process snapshot is represented with a colored circle on the top left corner of the subprocess with a token at each of its start events. To depict the subprocesses relation in

Figure 3, we chose the same color for the outer ring of the subprocess snapshot and its tokens. If the subprocess has no start events, a token for every activity and gateway without incoming sequence flows is added.

The bottom rule template generates one rule to delete a terminated process snapshot and adds tokens at each outgoing sequence flow. Subprocesses are terminated by a generic rule (see section 3.1) if they neither have tokens nor subprocesses.

3.3 Gateways

Figure 12 depicts the rule generation templates for parallel and exclusive gateways (see figure 2). A parallel gateway can synchronize incoming sequence flows and fork outgoing sequence flows simultaneously. Thus, one rule is generated that deletes one token from each incoming sequence flow and adds one token to each outgoing sequence flow.

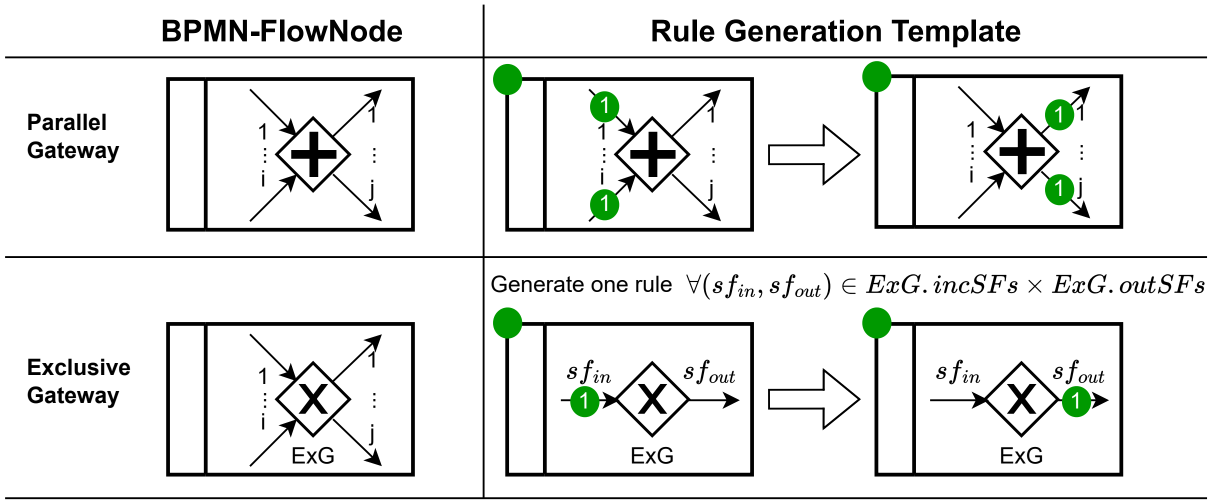


Figure 12: Rule generation template for gateways

Exclusive Gateways are triggered by exactly one incoming sequence flow, and exactly one outgoing sequence flow is triggered. Thus, one rule must be generated for every combination of incoming and outgoing sequence flows. However, the resulting rule is simple since it only deletes a token from an incoming sequence flow and adds a token to an outgoing sequence flow.

3.4 Message Events

Figure 13 depicts the rule generation templates for *message intermediate throw events* (MITE in figure 2). The upper rule describes how MITEs interact with *message intermediate catch events* (MICE in figure 2). A MITE deletes an incoming token and adds a token at each outgoing sequence flow. In addition, it sends one message to each waiting process by adding it to the incoming messages of the process. However, sending each message is optional, meaning that if a process is not ready to consume a message immediately, the message is not added. We implement optional message sending using a nested rule with quantification. Concretely, we use an optional existential quantifier to send a message only if the receiving process runs and is ready to receive it [7].

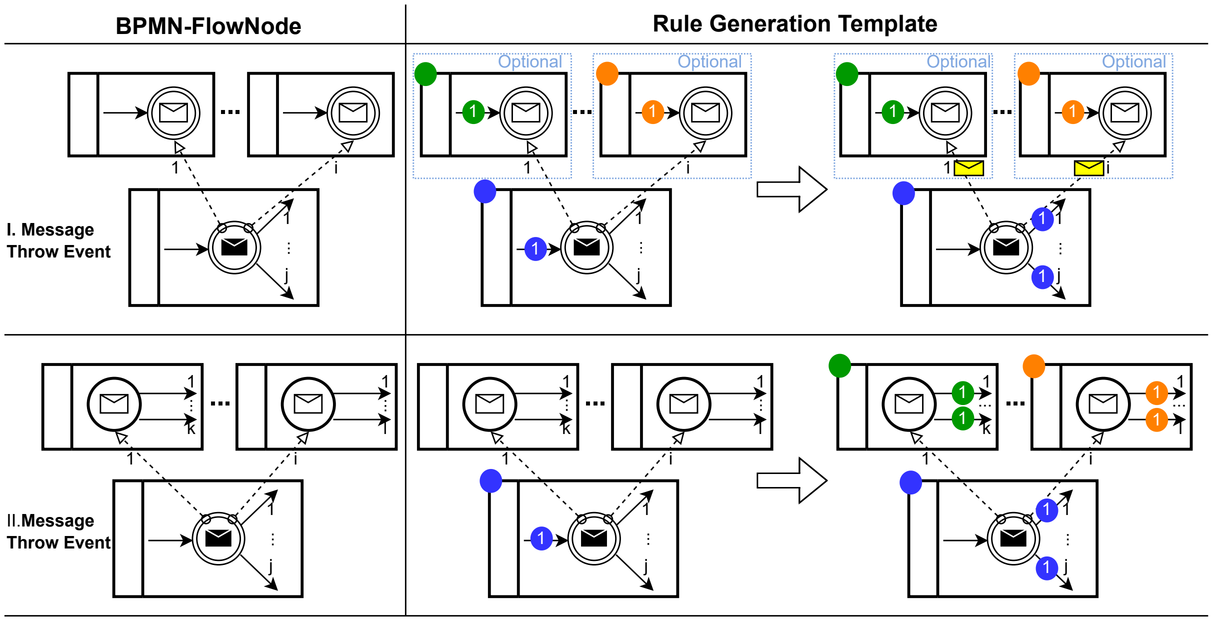


Figure 13: Rule generation template for message throw events

The bottom rule shows how MITEs trigger new process instances when interacting with *message start events*. For each receiving *message start event*, a new process snapshot with one token at each outgoing sequence flow is added. It is worth noting that a MITE might interact with MICEs and *message start events simultaneously*. Thus, the rule generation templates in figure 13 can be mixed, i.e., messages can be sent and processes instantiated by one MITE. We only separated message throw behavior into two rule templates for presentation purposes. This template generates the MITE rule in figure 8. Furthermore, *message end events* and *send tasks* behave similarly regarding message creation and process instantiation.

Figure 14 depicts the rule generation templates for MICEs and *receive tasks*. To trigger a MICE or a *receive task*, only one message at an incoming *message flow* is needed. Thus, one rule is generated for each incoming *message flow*. The upper rule template shows that MICEs delete one message and one token, and add a token at each outgoing sequence flow. This template generates the MICE rule in figure 8. In the bottom rule template, one can see that *receive task* rules are similar to the *activity* rules in figure 11, but they also delete a message when a *receive task* is started.

4 Model checking BPMN

Model checking a BPMN process model is possible using the generated graph transformation system. Besides a graph transformation system, a set of temporal properties to be checked and the atomic propositions used in these properties must be supplied. An atomic proposition is formalized as a graph and holds in a given state if a match exists from the underlying graph of the proposition to the graph representing the state. This enables model checking of temporal properties, for example, LTL properties, using the defined atomic propositions.

We differentiate between *BPMN-specific properties* defined for all BPMN process models and *custom properties* tailored towards a particular BPMN process model. We do not consider structural properties (like conformance to the syntax of PBMN) since they can be checked using a standard process modeling

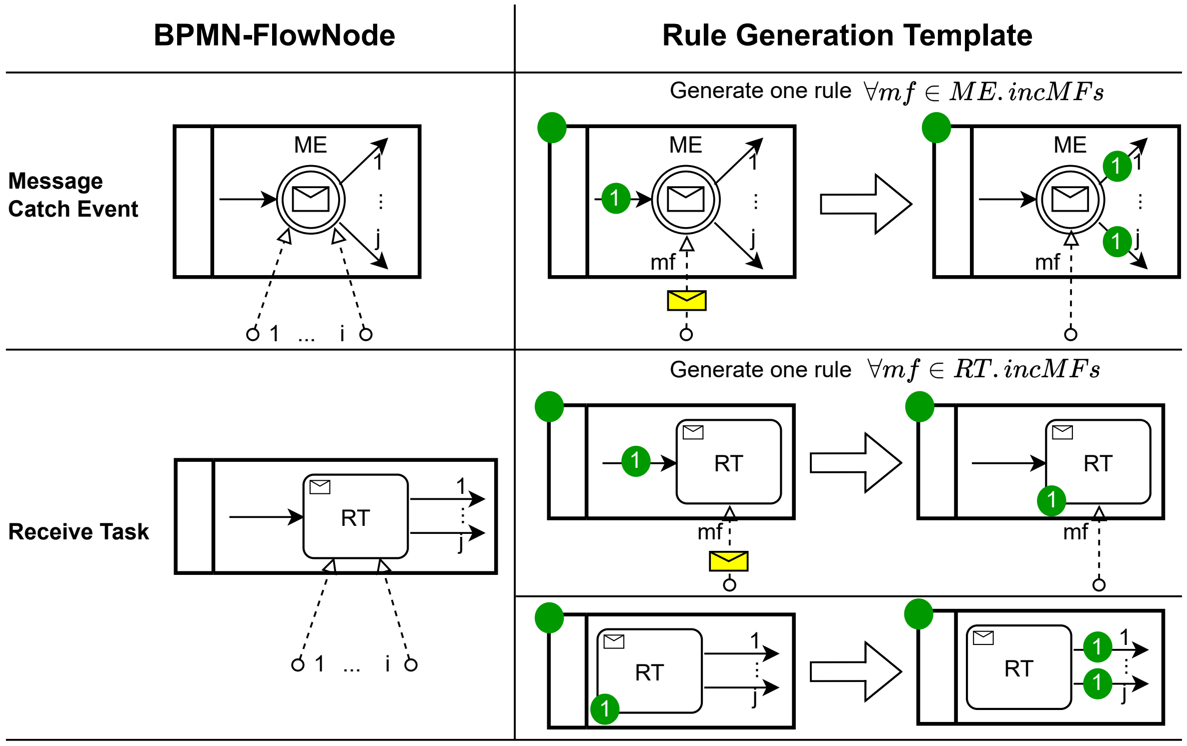

BPMN-FlowNode
Rule Generation Template
Message Catch Event
Receive Task
Generate one rule $\forall mf \in ME.incMFs$
Generate one rule $\forall mf \in RT.incMFs$
ME
RT
mf
1
j
1 ... i

Figure 14: Rule generation template for message catch events and receive tasks

tool without implementing execution semantics. We will now give an example of two predefined BPMN-specific properties and show how they can be checked using our approach. Then, we describe how custom properties can be constructed and checked.

4.1 BPMN-specific properties

Safeness and *Soundness* properties are defined for BPMN in [2]. A BPMN process model is *safe* if, during its execution, at most one token occurs along the same sequence flow [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, as well as (iii) *No dead activities*: any activity can be executed in at least one process instance [2]. For example, we will describe how to implement the *Safeness* and *Option to complete* properties.

Safeness is specified using the LTL property defined in (1). The atomic property Unsafe is true if two tokens of one process snapshot point to the same sequence flow. This atomic proposition *Unsafe* is depicted in figure 15; Groove rules for all the atomic propositions are included in [9].

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

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

Both properties can be checked using our implementation [9]. To fully check Soundness, we need to check *Proper Completion* and *No Dead Activities*. The information needed to check these properties is in the generated state space.

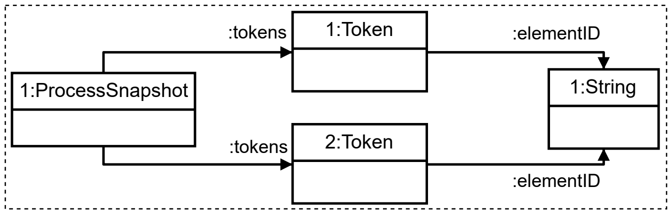


Figure 15: Atomic proposition Unsafe

4.2 Custom properties

To make model checking user-friendly, we envision users defining atomic propositions in the extended BPMN syntax, i.e., the concrete syntax introduced in figure 3. Thus, to define an atomic proposition, a user adds process snapshots and tokens for a BPMN process model, which we can automatically convert to a graph representing an atomic proposition.

For example, the token distribution shown in figure 16 defines two running process snapshots with a token in task A. Differently colored tokens define different process snapshots. A user could use this property, for example, to check if, eventually, two processes are executing task A simultaneously. Thus, a user does not need to be aware of the graph transformation semantics used for execution, which is a significant advantage compared to other approaches.

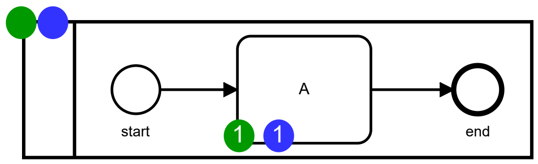


Figure 16: Token distribution defining an atomic proposition.

5 Implementation

Our overall approach is summarized in figure 17 as a BPMN process model. We use our higher-order transformation to generate a graph transformation system for a given BPMN model. Finally, we use the graph transformation system to check temporal properties, possibly leading to property violations.

The first step of the approach is implemented in our web-based tool. The tool is open-source, publicly available, and does not require any installation [9]. Figure 18 depicts a screenshot of the implemented tool. However, model checking must be run locally in the graph-transformation tool Groove[2]. We are currently working on extending our tool such that model-checking can be done without installing Groove locally. Groove implements SPO with NACs and thus has all the features we need. To evaluate the correctness of our implementation, we created a comprehensive test suite, which demonstrates correct rule generation for the implemented BPMN constructs [9].

[2]`https://groove.ewi.utwente.nl/about`

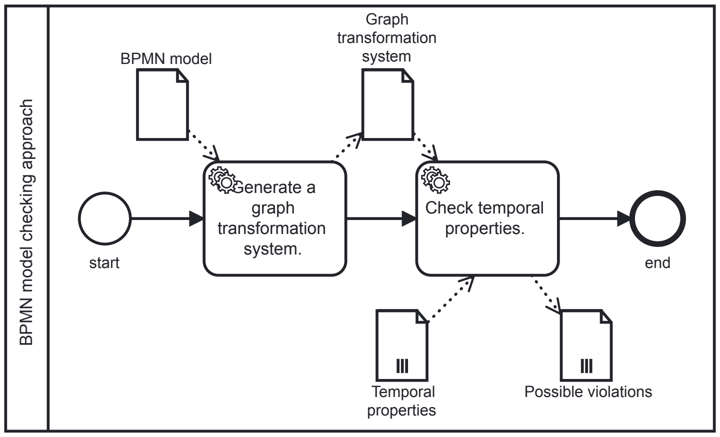
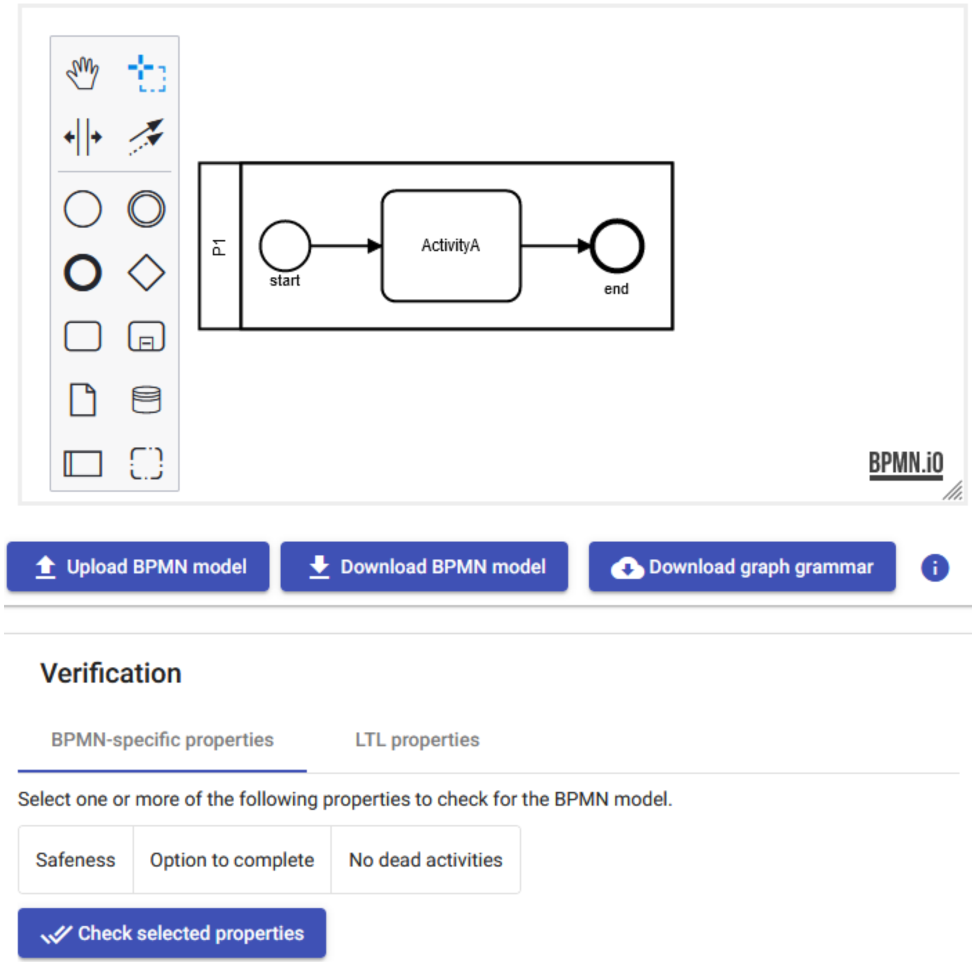

BPMN model checking approach
BPMN model
Graph transformation system
start
Generate a graph transformation system.
Check temporal properties.
end
Temporal properties
Possible violations

Figure 17: Overview of the proposed approach



Verification

BPMN-specific properties | LTL properties

Select one or more of the following properties to check for the BPMN model.

Safeness | Option to complete | No dead activities

Check selected properties

Figure 18: Screenshot of the tool

6 Related work

A BPMN formalization based on in-place graph transformation rules is given in [10]. The formalization covers a substantial part of the BPMN specification, including complex concepts such as inclusive gateway merge and compensation. In addition, the graph transformation rules are visual and thus can easily be aligned with the informal description of the execution semantics of BPMN. A key difference to our approach is that the rules in [10] are general and can be applied to any BPMN model, while we generate specific rules for every BPMN model based on rule generation templates. Thus, our approach can be seen as a program specialization compared to [10] since we process a concrete BPMN model before its execution. The graph transformation rules are 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 approach by Corradini et al. is based on formal BPMN semantics given in rewriting logic and implemented in the Maude system [1]. Using this formal semantics, the tool BProVe[3] can verify custom LTL properties and BPMN-specific properties, such as Safeness and Soundness. Furthermore, the tool is accessible online, not requiring any installation.

The verification framework fbpmn uses first-order logic to formalize and check BPMN process models [5]. 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[4]. Like BProVe, fbpmn allows checking BPMN-specific properties, such as Safeness and Soundness. However, they do not allow a user to define custom temporal properties.

Table 1 shows which BPMN constructs are supported by the approaches mentioned above compared to ours. We investigated these three approaches since they support a significant subset of the BPMN constructs and have accessible and well-documented tools. However, each approach supports a different subset of the BPMN constructs. The coverage of BPMN constructs greatly impacts how useful each approach is in practice.

Van Gorp et al. [10] 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. This is a major deficiency, especially since Event-based gateways are often used in practice. Corradini et al. [1] support message and terminate events. However, they do not support BPMN subprocesses. In addition to [1], Houhou et al. [5] support timer and the use of message and timer events as both interrupting and non-interrupting boundary events. However, they do not support signal and link events. Referring to table 1, we conclude that our formalization is comprehensive but still lacks support for some of the more advanced event types. Implementing the missing event types is feasible, as shown in [10].

7 Conclusion & future work

The approach presented in this paper utilizes a higher-order transformation from BPMN models to graph transformation systems. For each BPMN model, we automatically generate a start graph and a set of graph transformation rules. In this way, the graph transformation rules get simpler while the complexity is shifted to the higher-order transformation. Our resulting BPMN formalization is comprehensive and supports model checking. In addition, we provide a prototype implementation in a web-based tool to make our ideas easily accessible to other researchers and potential practitioners.

[3]`http://pros.unicam.it/bprove/`

[4]`https://github.com/pascalpoizat/fbpmn`

Table 1: Constructs supported by different BPMN formalizations (overview based on [10]).

Feature	Van Gorp et al. [10]	Corradini et al. [1]	Houhou et al. [5]	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
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.

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 models from open repositories such as the "BPM Academic Initiative Model Collection" [11] and "Camunda BPMN for Research"[5]. Third, we intend to extend the features of our tool so that the user can define atomic propositions for model checking in the tool directly, as described in section 4. Finally, counterexamples found during model checking should be visualized directly in the tool, like the implementation in [5], such that users are not forced to 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] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner & A. Corradini (1997): *ALGEBRAIC APPROACHES TO GRAPH TRANSFORMATION – PART II: SINGLE PUSHOUT APPROACH AND COMPARISON WITH DOUBLE PUSHOUT APPROACH*, pp. 247–312. WORLD SCIENTIFIC, doi:10.1142/9789812384720_0004.

[4] Jakob Freund & Bernd Rücker (2019): *Real-Life BPMN: Using BPMN and DMN to Analyze, Improve, and Automate Processes in Your Company*, 4th edition edition. Camunda, Berlin.

[5] 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.

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

[7] 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.

[8] Arend Rensink (2017): *How Much Are Your Geraniums? Taking Graph Conditions Beyond First Order.* In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: ModelEd, TestEd, TrustEd, 10500, Springer International Publishing, Cham, pp. 191–213, doi:10.1007/978-3-319-68270-9_10.

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

[10] 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.

[11] 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.

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