Formalization and analysis of BPMN using graph transformation systems

Tim Kräuter[1], Adrian Rutle[1], Harald König[2,1], and Yngve Lamo[1]

[1] Western Norway University of Applied Sciences, Bergen, Norway tkra@hvl.no, aru@hvl.no, yla@hvl.no
[2] University of Applied Sciences, FHDW, Hanover, Germany harald.koenig@fhdw.de

Abstract. The Business Process Modeling Notation (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 features 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 more BPMN features while allowing property checking. 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.

Keywords: BPMN · Model transformation · Graph transformation · Model checking · Formalization

1 Introduction

Business Process Modeling Notation (BPMN) [13] 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 features and difficulties in checking behavioral properties [1]. One can detect errors and optimization potential in process models already during their creation by checking behavioral properties. Thus, the cost of business process automation using BPMN can be reduced. Consequently, we propose a formalization that covers most of the BPMN features used in practice and supports checking behavioral properties.

Generally, we think about two fundamental concepts when formalizing the execution semantics of a behavioral language. First, *state structure*, i.e., how can one represent models during execution. The state structure corresponds to the typegraph in Graph Transformation (GT) systems. Second, *state-changing elements*, i.e., which elements in a model encode potential state changes. Those elements must then be implemented using GT rules. Our approach creates GT rules based on a Higher-Order model Transformation (HOT) and is summarized in Figure 1.

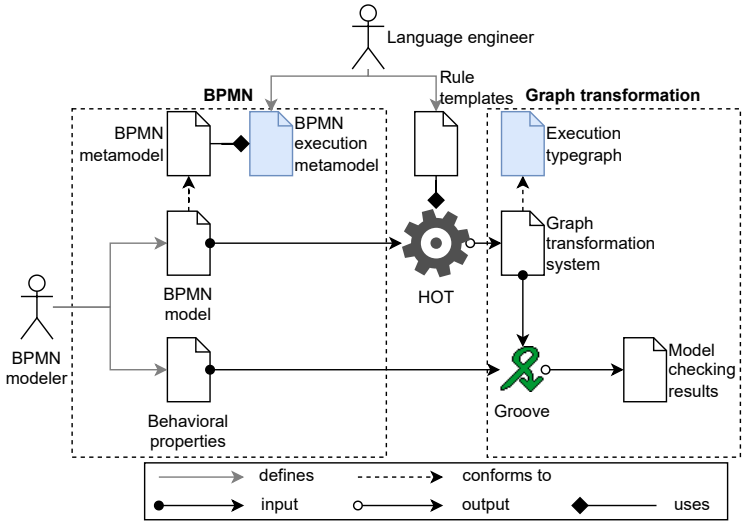

Fig. 1. Overview of the approach

First, a modeler defines a BPMN model and behavioral properties to check. The BPMN model conforms to the BPMN metamodel defined in the BPMN specification [13]. Using the BPMN metamodel, we define the state structure for BPMN in a so-called BPMN execution metamodel, fulfilling the role of a language engineer. Usually, an execution metamodel is defined by extending the structural metamodel.

Furthermore, we define a HOT from BPMN models to GT systems. We call the transformation *higher-order* since the resulting graph-transformation systems represent model-transformations themselves [18]. The HOT creates a GT system, i.e., GT rules and a start graph for a given BPMN model. It is defined using so-called rule generation templates, which describe how GT rules should be generated for each state-changing element in BPMN (see section 3). The obtained GT system conforms to the execution type graph representing the BPMN execution metamodel interpreted as a type graph. We colored both artifacts in blue to visualize that they contain the same information. Finally, we check the previously defined behavioral properties using Groove to run the GT system.

We implemented our approach in an open-source web-based tool such that it is easily accessible without needing any local installation. Furthermore, our approach is general since it can be used to formalize other behavioral languages. To formalize the execution semantics of a different behavioral language, one only needs to define a new execution metamodel and HOT (see language engineer in Figure 1)

The remainder of this paper is structured as follows. First, in section 2, we introduce BPMN and point out the theoretical background of this contribu-

tion. Second, we describe the BPMN semantics formalization using the HOT (section 3) before explaining how this can be utilized for model checking general BPMN and custom properties (section 4). Then, in section 5, we present the web-based tool implementing our approach. Finally, we discuss related work regarding BPMN feature coverage in section 6 and conclude in section 7.

2 Preliminaries

In this paper, we apply GTs to formalize the execution semantics of BPMN. Thus, in this section, we will briefly introduce BPMN and its execution semantics. Please refer to [7] or the BPMN specification [13] for further information about BPMN. Furthermore, we outline the theoretical background behind our application of GTs.

2.1 BPMN

Figure 2 depicts the structure of BPMN models with the corresponding concrete syntax BPMN symbols contained in clouds.

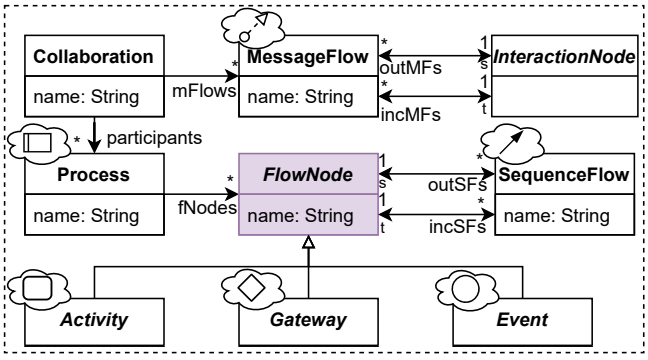

Fig. 2. Excerpt of the BPMN metamodel [13]

A BPMN model is represented by a Collaboration that has participants and MessageFlows between InteractionNodes (see Figure 2). 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 [7].

The BPMN execution semantics are described using the concept of *tokens* [13], which can be located at sequence flows and specific flow nodes. Tokens are consumed and created by flow nodes according to the connected sequence

flows. Thus, the flow nodes are colored purple in Figure 2 since they are the *state-changing elements* of BPMN and are crucial when formalizing the BPMN execution semantics in section 3. A BPMN process is triggered by one of its start events, leading to the creation of a token at the triggered start event. 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 deletes its token and adds one at each outgoing sequence flow. Furthermore, different gateway types exist, such as 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 GT, we utilize the single-pushout (SPO) approach with negative application conditions (NAC) [5], as implemented in Groove [15]. In addition, we utilize *nested rules* with quantification to make parts of a rule applied repeatedly or optionally [16,17]. Moreover, we utilize NACs to implement more intricate parts in the BPMN execution semantics, such as the termination of processes. In SPO rewriting, a GT 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 HOT from BPMN to GT systems, we generate a *start graph* and *GT rules* for a given BPMN model. The approach supports the BPMN features depicted in Figure 3. BPMN features are divided into Events, Gateways, Activities, and Edges. Events and Activities are further divided into subgroups. Due to space limitations, we only explain how the features marked with a green background are realized in this paper. However, the other features are also implemented and tested [11].

Before we explain our implementation of the BPMN features, we define the BPMN execution metamodel.

3.1 BPMN execution metamodel

Our formalization of BPMN is token-based, as in the informal description of the BPMN specification [13]. Thus, to describe processes holding tokens during execution, we use the type graph shown in Figure 4. The type graph is depicted as a UML class diagram without operations, which can be seen as an attributed type graph with inheritance [8].

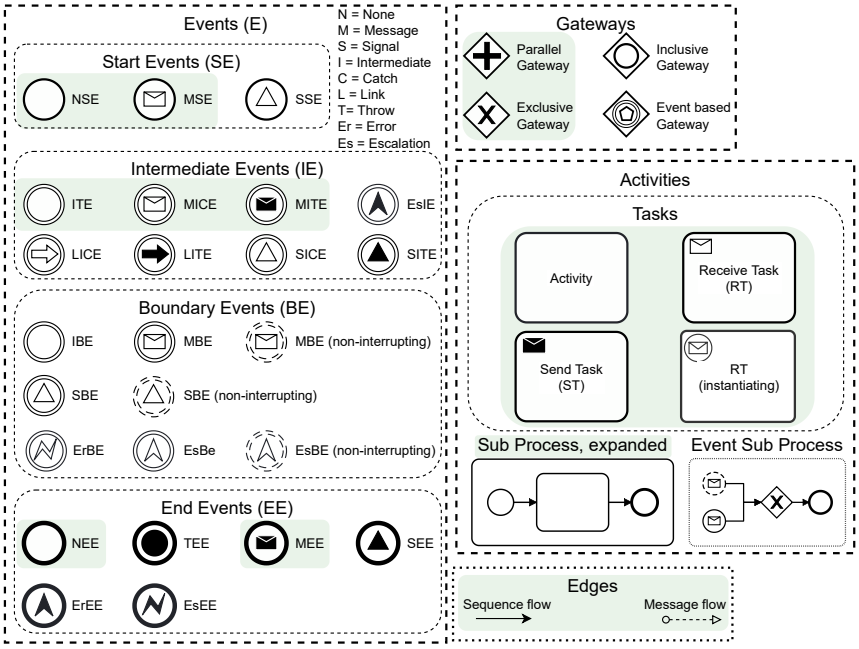

Fig. 3. Overview of the supported BPMN features (structure adapted from [9])

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 concrete syntax in the clouds attached to the elements. Our concrete syntax extends the BPMN syntax by adding process snapshots, subprocess relations, 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[3]. Using this type graph, we can now define how the start graph and graph-transformation rules for the different BPMN features 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). A NSE describes a start event

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

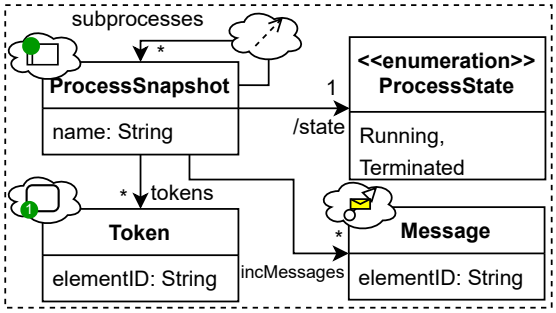

Fig. 4. BPMN execution metamodel

without a trigger (none). Popular start event triggers are message and signal (see MSE and SSE in Figure 3). Then, for each NSE, we add a token to the respective process snapshot. An example of a start graph is shown in Figure 5 using abstract and concrete syntax. Furthermore, we consider allowing the modeler to define a start graph similar to how he can define atomic propositions for custom properties (see subsection 4.2).

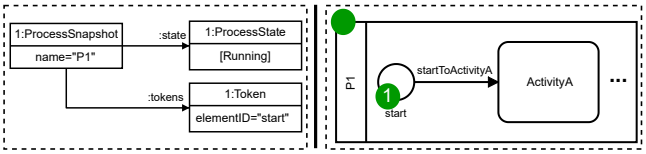

Fig. 5. Example start graph in abstract (left) and concrete syntax (right)

The HOT generates one or more GT 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 an NSE, a task, a *message intermediate catch event* (MICE), and a *message intermediate throw event* (MITE) (see Figure 3). Afterward, we will explain how these—and other—rules are created by our HOT.

Figure 6 depicts an example GT rule ($L \to R$) for an 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 6 depicted in concrete syntax is shown in Figure 7 on the left.

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

The left rule in Figure 8 realizes a MITE, and the right rule implements a MICE. The MITE rule moves the token through the event and may send a mes-

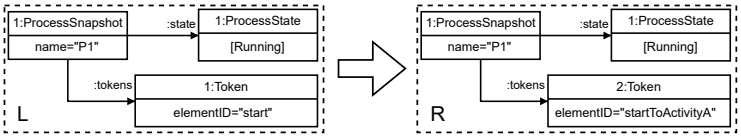

Fig. 6. Example GT rule for an NSE (abstract syntax)

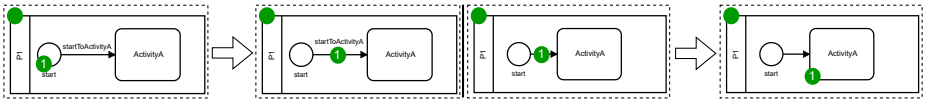

Fig. 7. Example GT rule for an NSE (left) and to start a task (right) in concrete syntax

sage to a waiting process snapshot if it has a token waiting at the corresponding MICE. An optional existential quantifier [16] is used to send a message only if the receiving process is ready to receive it. The MICE rule deletes a token and a message and adds an outgoing token.

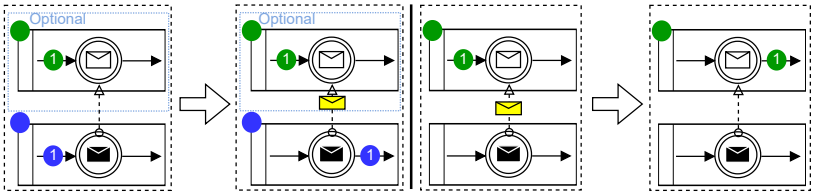

Fig. 8. Rules for MITE (left) and MICE (right)

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 HOT. Elements of the HOT 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 3.

3.2 Process instantiation and termination

Figure 9 depicts the rule generation templates for start and end events (NSE and NEE in Figure 3). 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 2), and the ones in the right column show the generated rules typed by the BPMN execution type graph (see Figure 4). If more than one rule is generated from a BPMN structure there is an expression defining how each rule is generated. For example, the expression ∀sf ∈ E.incSFs for the rule generation template of end events (see Figure 9) generates one rule for each incoming sequence flow *sf* of the end event *E*. We use "." in expressions to navigate along the associations of the BPMN metamodel Figure 2. In the example, E.incSFs means following all incSFs links for a FlowNode object, resulting in a set of SequenceFlow objects.

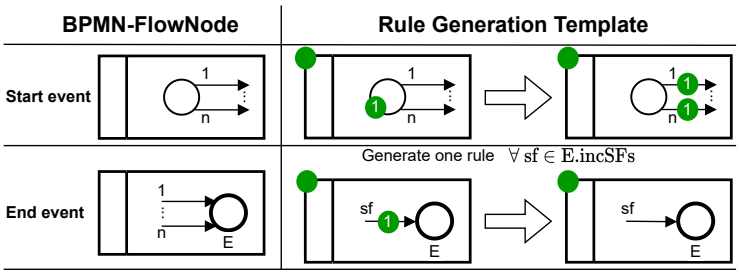

Fig. 9. Rule generation templates for start and end events

The start event rule template generates the start event rule in Figure 7. 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. Initially, the tokens on the start events are given by the start graph of the GT system (see, e.g., Figure 5).

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 in Figure 10 is automatically generated once during the HOT. The rule changes the state of the process snapshot from running to terminated if it has neither tokens nor subprocesses.

3.3 Activities & Subprocesses

Figure 11 depicts the rule generation templates for activities and subprocesses (see Figure 3). 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 se-

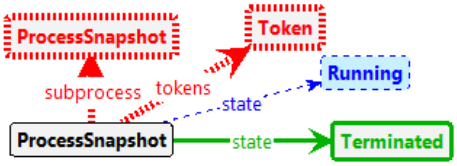

Fig. 10. Termination rule in Groove

quence flows represent an *implicit exclusive gateway* (see exclusive gateway in Figure 12). This rule template generates the sample rule in Figure 7 on the right.

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

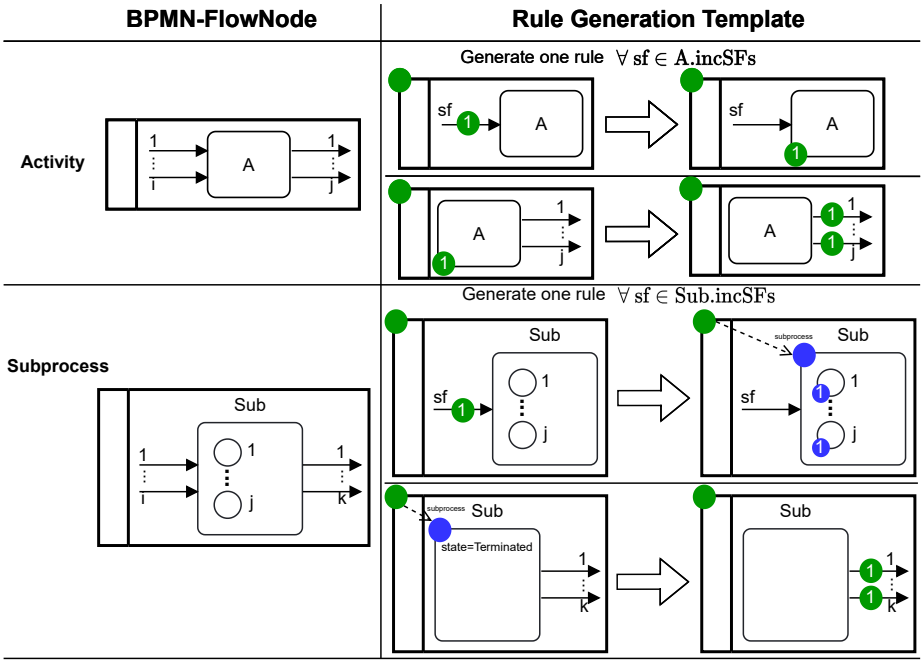

Fig. 11. Rule generation template for activities and subprocesses

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. We added a *subprocess* link between the process snapshots to depict the subprocesses relation in Figure 4. 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 3.2) if they neither have tokens nor subprocesses.

3.4 Gateways

Figure 12 depicts the rule generation templates for parallel and exclusive gateways (see Figure 3). A parallel gateway can synchronize and fork the control flow simultaneously. Thus, one rule is generated that deletes one token from each incoming sequence flow and adds one token to each outgoing sequence flow.

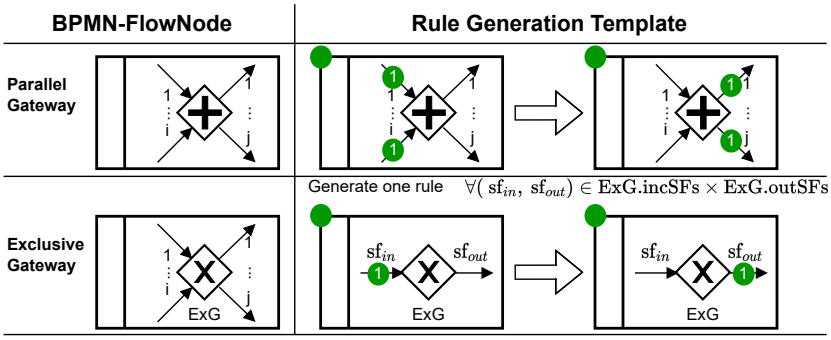

Fig. 12. Rule generation template for gateways

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.

3.5 Message Events

Figure 13 depicts the rule generation templates for *message intermediate throw events* and *message intermediate catch events* (MITE and MICE in Figure 3). The upper rule describes how MITEs interact with *message intermediate catch events* (MICE in Figure 3). 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 (see blue dotted rectangle marked with optional in Figure 13) to send a message only if the receiving process runs and is ready to receive it [16].

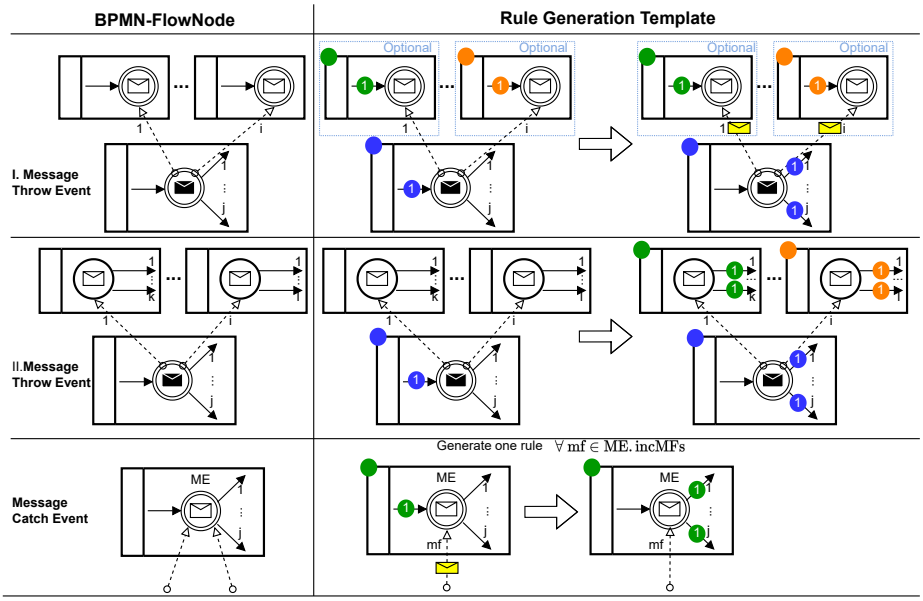

Fig. 13. Rule generation templates for message events

The middle 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. We chose to create a process snapshot immediately rather than creating a message and then consume this message and create a process snapshot in a separate rule. This decision keeps the rule generation simpler and the state space of the GT system smaller. It is worth noting that a MITE might interact with MICEs and *message start events simultaneously*. Thus, the rule generation templates in Figure 8 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.

The bottom rule in Figure 13 depicts the rule generation template for MICEs. To trigger a MICE, only one message at an incoming *message flow* is needed. Thus, one rule is generated for each incoming *message flow*. The 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. Furthermore, *receive tasks* behave similarly regarding message consumption.

4 Model checking BPMN

Model checking a BPMN model is possible using the generated GT system. Besides a GT 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 using the defined atomic propositions.

We differentiate between *general BPMN properties* defined for all BPMN models and *custom properties* tailored towards a particular BPMN model. We do not consider structural properties (like conformance to the syntax of PBMN) since they can be checked using a standard modeling tool without implementing execution semantics. We will now give an example of two predefined general BPMN properties and show how they can be checked using our approach. Then, we describe how custom properties can be featureed and checked.

4.1 General BPMN properties

Safeness and *Soundness* properties are defined for BPMN in [3]. A BPMN model is *safe* if, during its execution, at most one token occurs along the same sequence flow [3]. 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 [3]. For example, we will describe how to implement the *Safeness* and *Option to complete* properties.

Safeness is specified using the CTL property defined in (1). The atomic property Unsafe is true if two tokens of one process snapshot point to the same sequence flow. Groove rules for all the atomic propositions are included in [11].

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

$$AG(\neg \text{Unsafe}) \qquad (1) \qquad\qquad AF(\text{AllTerminated}) \qquad (2)$$

Checking the properties *Safeness*, *Option to Complete*, and *No Dead Activities* is implemented in our tool [11]. The property *Proper Completion* is not yet implemented, but all the information needed can be found in the GT systems state space.

4.2 Custom properties

To make model checking user-friendly, we envision modelers defining atomic propositions in the extended BPMN syntax, i.e., the concrete syntax introduced

in Figure 4. Thus, to define an atomic proposition, a modeler adds process snapshots and tokens for a BPMN model, which we can automatically convert to a graph representing an atomic proposition.

For example, the token distribution shown in Figure 14 defines two running process snapshots with a token at task A. Differently colored tokens define different process snapshots. A modeler could use this atomic proposition, for example, to check if, eventually, two processes are executing task A simultaneously by creating a LTL/CTL property. Thus, a modeler does not need to know the GT semantics used for execution.

However, a modeler must still know the temporal logic, such as LTL and CTL to express his properties. In the future, a domain-specific property language for BPMN would further lessen the knowledge required from a modeler [12].

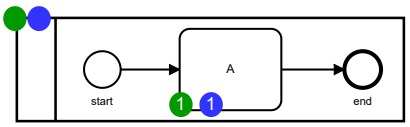

Fig. 14. Token distribution defining an atomic proposition.

5 Implementation

Our approach is implemented in a web-based tool. The tool is open-source, publicly available, and does not require any installation [11]. Figure 15 depicts a screenshot of the implemented tool.

First, one has to model or upload a BPMN process. Then, in the verification section one can choose to check either BPMN-specific properties or custom CTL properties. Finally, our tool can generate a GT system for the supplied BPMN model and run model checking in Groove [10]. Furthermore, to evaluate the correctness of our HOT, we created a comprehensive test suite, which verifies correct rule generation for the implemented BPMN features [11]. As described in the next section, we also ran a performance benchmark for our approach.

5.1 Performance benchmark

Model checking is a useful technique but often falls short in practice due to insufficient performance. Poor performance might have many reasons, most notably large models leading to state space explosion. We ran a benchmark for ten different BPMN models from [9] to assess the performance of our implementation. The models include realistic business process models (001, 002, and 020), and more information can be found in [9].

To calculate the average runtime, we used the hyperfine benchmarking tool [14] (version 1.15.0), which ran state space exploration for each BPMN model ten

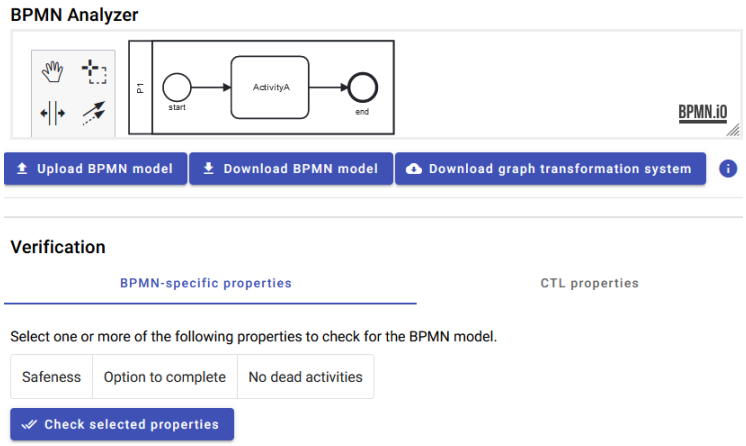

Fig. 15. Screenshot of the tool

times. The benchmarks were run on Windows 11 (AMD Ryzen 7700X processor, 32 GB RAM) using Groove version 5.8.1 (detailed results and reproduction steps are given in [11]).

First, we benchmarked our HOT for the BPMN models. The HOT took less than one second to generate a GT system for every model. Thus, the generation of the GT systems is fast enough. Furthermore, we suspect most of the time is spent writing files to disk for Groove to consume.

Second, we benchmarked a full state exploration using the resulting ten GT systems, see Table 1. The exploration takes roughly one second for most of the models. Only model *020* needs nearly two seconds due to its larger state space. Furthermore, we estimate that up to one second is spent before state space exploration, most likely reading the GT system files. For example, Groove reports only 722 ms for state space exploration for model *020*.

We conclude that our approach is sufficiently fast for models of normal size. In addition, there is still room for optimization, such as avoiding costly I/O to disk. A more comprehensive benchmark and detailed comparison to other implementations are left for future work.

6 Related work

A BPMN formalization based on in-place GT rules is given in [19]. The formalization covers a substantial part of the BPMN specification, including complex concepts such as inclusive gateway merge and compensation. In addition, the GT 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 [19] are general and can be applied to any BPMN model, while we generate specific rules for every BPMN model using our HOT. Thus, our approach can be seen as a program specialization compared to [19] since we process

BPMN model	Processes	Nodes (gw.)	States	Transitions	Total time
001	2	17(2)	78	132	$\sim$ 1.00 s
002	2	16(2)	63	109	$\sim$ 0.97 s
007	1	8(2)	46	82	$\sim$ 0.92 s
008	1	11(2)	51	87	$\sim$ 0.93 s
009	1	12(2)	208	474	$\sim$ 1.01 s
010	1	15(2)	245	543	$\sim$ 1.04 s
011	1	15(2)	83	157	$\sim$ 0.97 s
015	1	14(2)	66	111	$\sim$ 0.95 s
016	1	14(2)	56	91	$\sim$ 0.94 s
020	1	39(6)	3121	8726	$\sim$ 1.75 s

Table 1. Full state space exploration in Groove

a concrete BPMN model before its execution. The GT rules are implemented in a prototype using GrGen.NET. Moreover, they do *not* support model checking since their goal is only formalization.

The tool *BProVe* is based on formal BPMN semantics given in rewriting logic and implemented in the Maude system [1]. Using this formal semantics, they can verify custom LTL properties and general BPMN properties, such as Safeness and Soundness.

The verification framework fbpmn uses first-order logic to formalize and check BPMN models [9]. This formalization is then realized in the TLA$^+$ formal language and can be model-checked using TLC. Like BProVe, fbpmn allows checking general BPMN properties, such as Safeness and Soundness. Furthermore, they focus on different communication models besides the standard in the BPMN specification and support time-related constructs. We currently disregard data flow (see [2,6]) and time-related constructs (see [4,9]).

Table 2 shows which BPMN features are supported by the approaches mentioned above compared to ours. We investigated these three approaches since they support a significant subset of the BPMN features and have accessible and well-documented tools. The coverage of BPMN features greatly impacts how useful each approach is in practice.

Our approach covers most of the BPMN features compared to other current approaches. Thus, we conclude that our formalization is comprehensive but can still be improved. In addition, it covers the most important features found in practice since we come close to the feature coverage of popular process engines such as Camunda[4].

7 Conclusion & future work

This paper makes two main practical contributions. First, we conceptualized a new approach utilizing a HOT to formalize the semantics of behavioral languages. Our approach moves complexity from the GT rules to the rule templates

[4] `https://docs.camunda.org/manual/7.16/reference/bpmn20/`

Table 2. features supported by different BPMN formalizations (overview based on [19]).

Feature	Van Gorp et al. [19]	Corradini et al. [1]	Houhou et al. [9]	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				X
Error Events	X			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, not subprocesses.
[3] Only supports message and timer events.

making up the HOT. Furthermore, the approach can be applied to any behavioral language if one can define its state structure and identify its state-changing elements.

Second, we apply our approach to BPMN, resulting in a comprehensive formalization regarding feature coverage (compared to the literature and industrial process engines) that supports checking behavioral properties. Furthermore, our contribution resulted in an open-source web-based tool to make our ideas easily accessible to other researchers and potential practitioners.

Potential future work can target both of our main contributions. First, we plan a detailed comparison of our HOT approach with an approach with fixed rules. It will be interesting to investigate how the two approaches differ, for example, in runtime during state space generation. Second, we aim to improve our formalization and the resulting tool in multiple ways. We intend to extend our formalization to support the remaining few BPMN features used in practice. For example, we want to support cancel and compensation events. Finally, we want to turn the modeling environment of our tool into an interactive simulation environment driven by our formal semantics. In addition, we can use this environment to visualize potential counterexamples of behavioral properties.

References

1. Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., Vandin, A.: A formal approach for the analysis of BPMN collaboration models. Journal of Systems and Software **180**, 111007 (Oct 2021). https://doi.org/10.1016/j.jss.2021.111007
2. Corradini, F., Muzi, C., Re, B., Rossi, L., Tiezzi, F.: Formalising and animating multiple instances in BPMN collaborations. Information Systems **103**, 101459 (Jan 2022). https://doi.org/10.1016/j.is.2019.101459
3. Corradini, F., Muzi, C., Re, B., Tiezzi, F.: A Classification of BPMN Collaborations based on Safeness and Soundness Notions. Electronic Proceedings in Theoretical Computer Science **276**, 37–52 (Aug 2018). https://doi.org/10.4204/EPTCS.276.5
4. Durán, F., Salaün, G.: Verifying Timed BPMN Processes Using Maude. In: Jacquet, J.M., Massink, M. (eds.) Coordination Models and Languages, vol. 10319, pp. 219–236. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_12
5. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: ALGEBRAIC APPROACHES TO GRAPH TRANSFORMATION – PART II: SINGLE PUSHOUT APPROACH AND COMPARISON WITH DOUBLE PUSHOUT APPROACH, pp. 247–312. World Scientific (Feb 1997). https://doi.org/10.1142/9789812384720_0004
6. El-Saber, N.A.S.: CMMI-CM Compliance Checking of Formal BPMN Models Using Maude. Ph.D. thesis, University of Leicester (Jan 2015)
7. Freund, J., Rücker, B.: Real-Life BPMN: Using BPMN and DMN to Analyze, Improve, and Automate Processes in Your Company. Camunda, Berlin, fourth edn. (2019)
8. Heckel, R., Taentzer, G.: Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-43916-3

9. Houhou, S., Baarir, S., Poizat, P., Quéinnec, P., Kahloul, L.: A First-Order Logic verification framework for communication-parametric and time-aware BPMN collaborations. Information Systems **104**, 101765 (Feb 2022). https://doi.org/10.1016/j.is.2021.101765
10. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) Model Checking Software. pp. 299–305. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
11. Kräuter, T.: Artifacts - ICGT. https://github.com/timKraeuter/ICGT-2023 (Jan 2023)
12. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: A Framework for Generating Domain-Specific Property Languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) Software Language Engineering, vol. 8706, pp. 1–20. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_1
13. Object Management Group: Business Process Model and Notation (BPMN), Version 2.0.2. https://www.omg.org/spec/BPMN/ (Dec 2013)
14. Peter, D.: Hyperfine (2022)
15. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance. pp. 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
16. Rensink, A.: Nested Quantification in Graph Transformation Rules. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) Graph Transformations. pp. 1–13. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11841883_1
17. Rensink, A.: How Much Are Your Geraniums? Taking Graph Conditions Beyond First Order. In: Katoen, J.P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd, vol. 10500, pp. 191–213. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_10
18. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the Use of Higher-Order Model Transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) Model Driven Architecture - Foundations and Applications, vol. 5562, pp. 18–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02674-4_3
19. Van Gorp, P., Dijkman, R.: A visual token-based formalization of BPMN 2.0 based on in-place transformations. Information and Software Technology **55**(2), 365–394 (Feb 2013). https://doi.org/10.1016/j.infsof.2012.08.014

8 Appendix

This section shows some simple examples of how our tool is used to check general BPMN properties. Our tool and the example models are available as artifacts [11].

8.1 Safeness example

Figure 16 shows a screenshot of the tool detecting an unsafe situation. Unfortunately, Groove does not provide a counterexample when running CTL model

checking through the console. Thus, we cannot highlight where the model is unsafe. In this case, the sequence flow *Unsafe* is unsafe.

The exclusive gateway merges the sequence flows but does not synchronize. Thus, the outgoing sequence flow can hold two tokens, and Activity C is executed twice. Again this could be a simple mistake not obvious to people unfamiliar with the BPMN execution semantics.

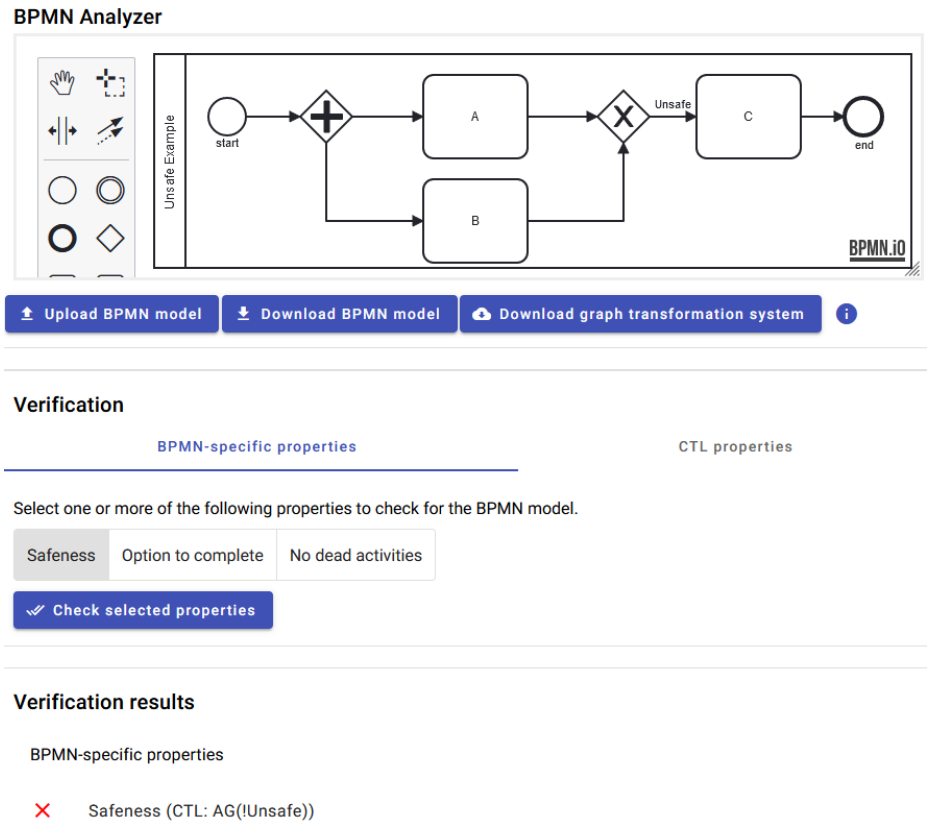

Fig. 16. Screenshot of the tool detecting an unsafe situation

8.2 Option to complete example

Figure 17 shows a screenshot of the tool checking *Option to complete*.

The parallel gateway cannot synchronize since no split happened before. Thus, the process cannot terminate. Again this could be a simple mistake not obvious to people unfamiliar with the BPMN execution semantics.

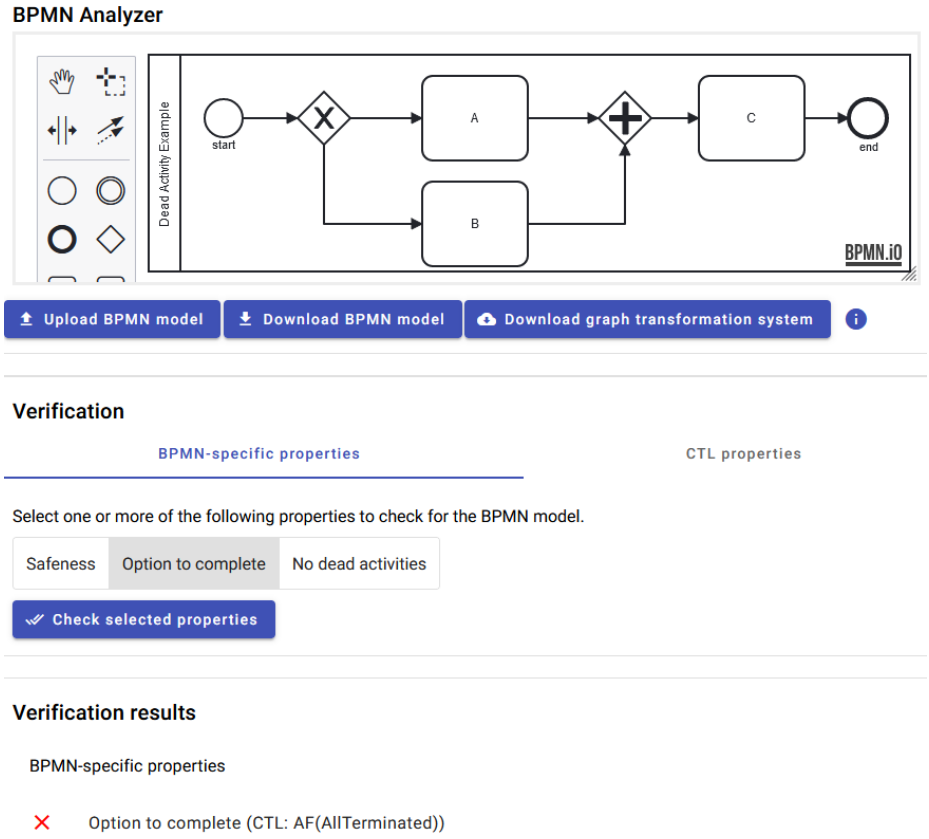

Fig. 17. Screenshot of the tool checking *Option to complete*

8.3 No dead activities example

Figure 18 shows a screenshot of the tool detecting a dead activity. Activity C is *dead*, which is highlighted when checking for dead activities.

The parallel gateway is incorrect since it cannot synchronize two sequence flows that never split. This could be a simple mistake not obvious to people unfamiliar with the BPMN execution semantics.

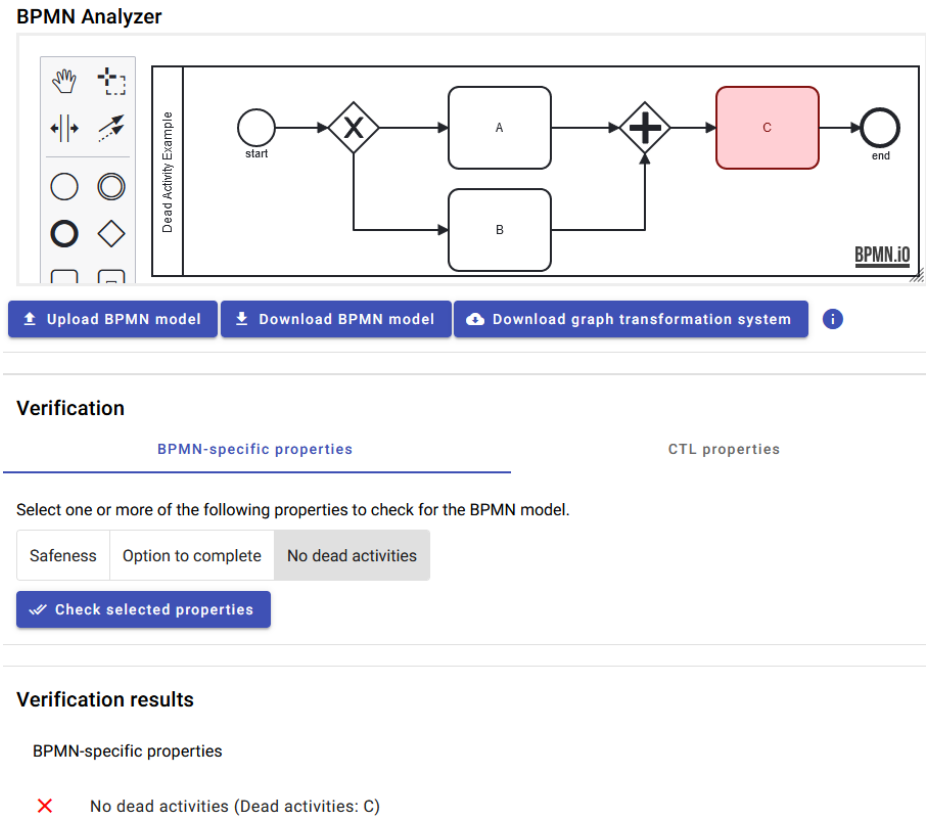

Fig. 18. Screenshot of the tool detecting a dead activity