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 elements 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 elements while facilitating 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 · Higher-order model transformation · Graph transformation · Model checking · Formalization

1 Introduction

In today's fast-paced business environment, organizations with complex workflows require a powerful means to accurately map, analyze, and optimize their processes. Business Process Modeling Notation (BPMN) [16] is a widely used standard to define these workflows. However, the informal description of the BPMN execution semantics leads to different interpretations of BPMN elements and difficulties in checking behavioral properties [4]. Formalizing BPMN would reduce the cost of business process automation drastically by facilitating the detection of errors and optimization potentials in process models already during design time. To this end, we propose a formalization that covers most of the BPMN elements used in practice and supports checking behavioral properties.

In this paper, we consider two fundamental concepts when formalizing the execution semantics of BPMN. First, *state structure*, i.e., how models are represented during execution. The state structure corresponds to the type graph in Graph Transformation (GT) systems. Second, *state-changing elements*, i.e., which elements in a model encode state changes. These elements are implemented using GT rules. In our approach, we automatically generate GT rules based on a Higher-Order model Transformation (HOT) for each specific BPMN model, as shown in Figure 1.

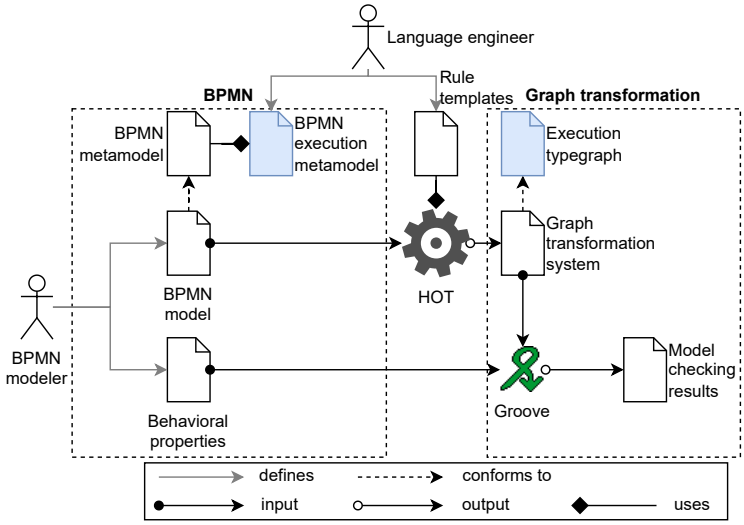

Fig. 1. Overview of the approach

To begin the BPMN modeling process, a modeler first defines the BPMN model and its corresponding behavioral properties for evaluation. This model must adhere to the BPMN metamodel as outlined in the BPMN specification by the Object Management Group [16]. To create the state structure for BPMN, the BPMN execution metamodel is defined by language engineers, utilizing the BPMN metamodel as a foundation. Typically, an execution metamodel is created by extending the structural BPMN 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 [21]. The HOT creates a GT system, i.e., GT rules and a start graph for a given BPMN model. It is defined using 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, which corresponds to the BPMN execution metamodel. In the figure, we have colored both artifacts blue to visualize that they contain the same information. Ultimately, we use Groove as an execution engine for the GT system and check the behavioral properties defined earlier.

Our approach has been implemented in a user-friendly, open-source web-based tool, the *BPMN Analyzer*, which can be used online without needing installation. The BPMN Analyzer was validated using a comprehensive test suite. Additionally, our approach is versatile as it can be applied to formalize other behavioral languages as well. To define the execution semantics of an alternate behavioral language, one simply needs to establish a new execution metamodel and HOT (see the language engineer in Figure 1).

The contribution of this paper is twofold. First, we introduce a new approach utilizing a HOT to generate GT rules instead of providing fixed GT rules to formalize the semantics of a behavioral language. Second, we apply our approach to BPMN, resulting in a formalization covering most BPMN elements that supports property checking.

The remainder of this paper is structured as follows. First, in section 2, we introduce BPMN and point out the theoretical background of this contribution. 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, we present BPMN Analyzer implementing our approach in section 5. Finally, we discuss related work regarding BPMN element coverage in section 6 and conclude in section 7.

2 Preliminaries

In this section, we will briefly introduce the execution semantics of BPMN, and readers are encouraged to consult [10] or the BPMN specification [16] for more in-depth information. Furthermore, our application of GTs to formalize the execution semantics of BPMN will be outlined in addition to a brief overview of the theoretical principles that underlie our use of GTs.

2.1 BPMN

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

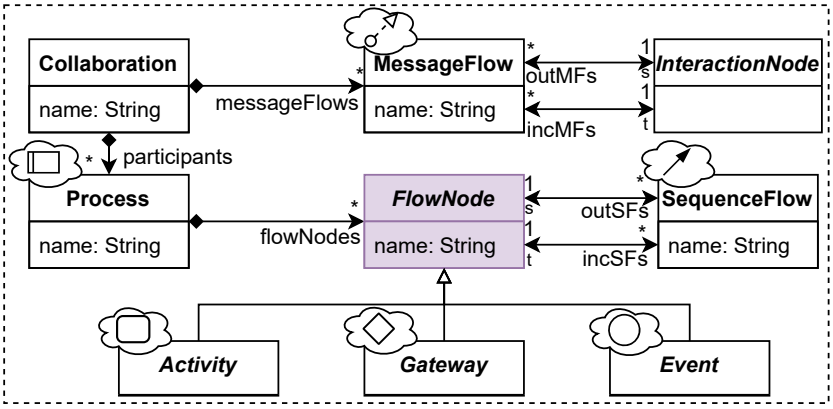

Fig. 2. Excerpt of the BPMN metamodel [16]

A BPMN model is represented by a Collaboration that has participants and messageFlows between InteractionNodes. 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 [10].

The BPMN execution semantics is described using the concept of *tokens* [16], 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. The FlowNode is colored purple in Figure 2 since it represents the state-changing elements of BPMN, as described in section 3.

A BPMN process is triggered by one of its start events, leading to a token at each outgoing flow of the triggered start event. Activities can start when at least one token is on an incoming sequence flow. The start of an activity will move the incoming token to the activity. When an activity terminates, 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 like 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 we introduce in section 3.

Regarding GT, we utilize the single-pushout (SPO) approach with negative application conditions (NAC) [8], as implemented in Groove [18]. In addition, we utilize *nested rules* with quantification to make parts of a rule repeatedly applicable or optional [19,20]. Moreover, we utilize NACs to implement more intricate parts in the BPMN execution semantics, such as the termination of processes.

3 BPMN semantics formalization

The approach supports all the BPMN elements depicted in Figure 3. These BPMN elements are divided into Events, Gateways, Activities, and Edges. Events and Activities are further divided into subgroups. Although all these elements have been implemented and tested (see [14]), due to space limitations, we only explain the realization of the elements marked with a green background. In the following, first, we define the BPMN execution metamodel to represent the BPMN state structure, then we explain our formalization of the elements in Figure 3.

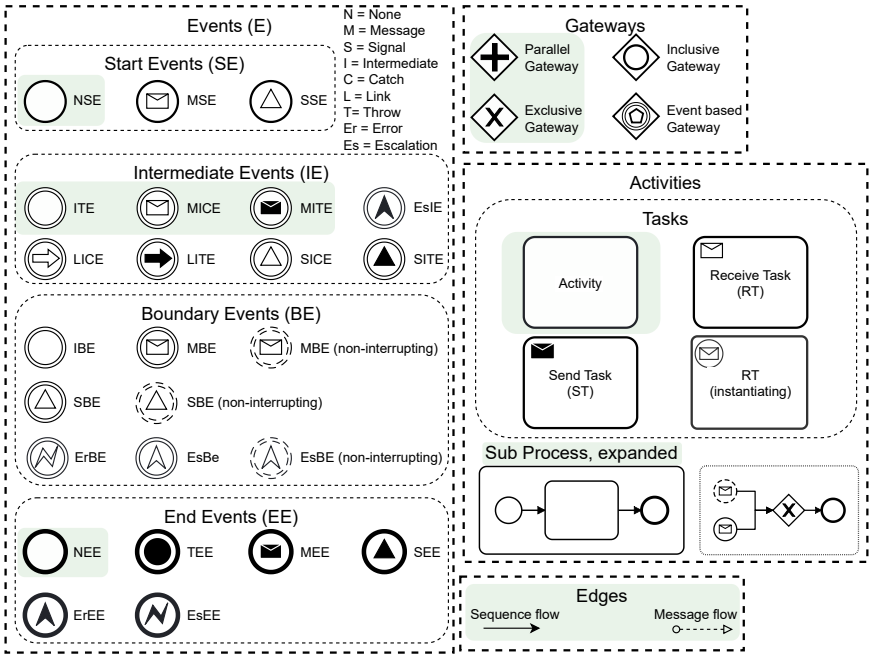

Fig. 3. Overview of the supported BPMN elements (structure adapted from [12])

3.1 BPMN execution metamodel

In our formalization of BPMN, we utilize a token-based representation of the execution semantics, similar to the approach used in the informal description of the BPMN specification [16]. To describe processes holding tokens during execution, we use the execution metamodel shown in Figure 4, depicted as a UML class diagram.

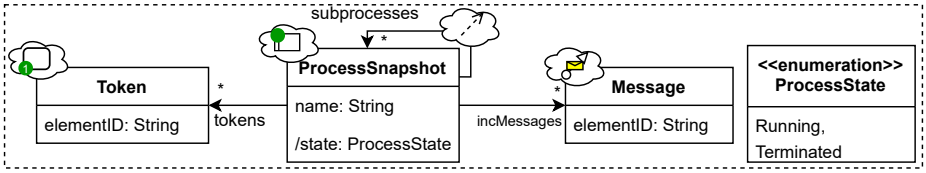

Fig. 4. BPMN execution metamodel

We use ProcessSnapshot to denote a running BPMN process with a specific token distribution that describes one state in the history of the process 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 [2].

The execution metamodel is a UML class diagram without operations, which can be seen as an attributed type graph [11]. Using the execution metamodel as the type graph, we can now define how the start graph and GT rules for the different BPMN elements are created.

Since our approach is based on a HOT from BPMN to GT systems, we generate a *start graph* and *GT rules* for each given BPMN model. Generating 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). An NSE describes a start event without a trigger (none). Then, for each NSE, we add one token to each outgoing sequence flow. An example of a start graph is shown in Figure 5 using abstract and concrete syntax.

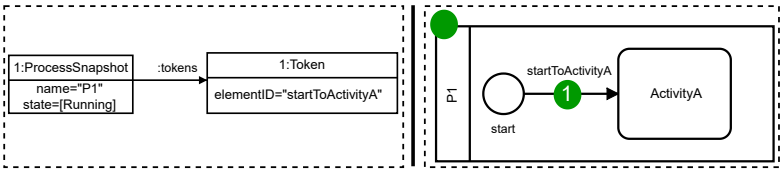

Fig. 5. Example start graph in abstract (left) and concrete syntax (right)

The HOT generates one or more GT rules for each FlowNode, i.e., state-changing element in a BPMN model. In order to provide a better understanding of the transformation process, we will begin by presenting two example results, namely the generated rules for an activity (as shown in Figure 3). Following this, we will delve into an explanation of how our HOT creates these rules as well as others.

Figure 6 depicts an example GT rule ($L \to R$) to start an activity in abstract syntax. The rule is straightforward, moving a token from the incoming sequence flow to the activity itself.

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 on

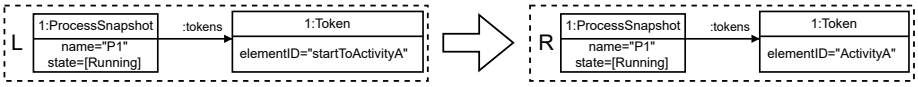

Fig. 6. Example GT rule to start an activity (abstract syntax)

the left in Figure 7. The rule on the right in Figure 7 implements the termination of an activity, which will move one token from the activity to the outgoing sequence flow.

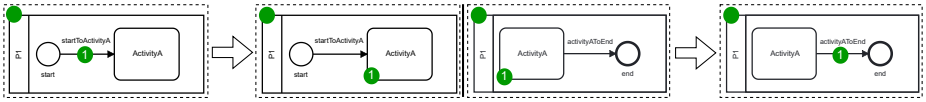

Fig. 7. Example GT rule to start (left) and terminate (right) an activity

To summarize, we described two 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.

3.2 Process instantiation and termination

Start events do not need GT rules since the generated start graph of the GT system will contain a token for each outgoing sequence flow of an NSE. Other types of start events are triggered in corresponding throw event rules.

Figure 8 depicts the rule generation template for end events (NEE in Figure 3). All rule generation templates show a state-changing element (FlowNode) with surrounding flows in the left column and the applicable rule generation in the right column. The left column shows instances of the BPMN metamodel (Figure 2), and the right column shows the generated rules typed by the BPMN execution metamodel (see Figure 4). If more than one rule is generated from a FlowNode, an expression defines how each rule is generated. For example, the expression $\forall sf \in E.incSFs$ for the rule generation template of end events (see Figure 8) 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 shown in Figure 2. In the example, E.incSFs means following all incSFs links for a FlowNode object, resulting in a set of SequenceFlow objects.

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 9

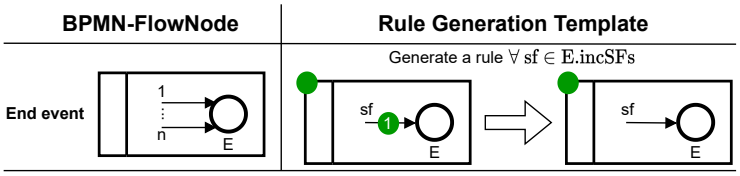

Fig. 8. Rule generation templates for start and end events

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.

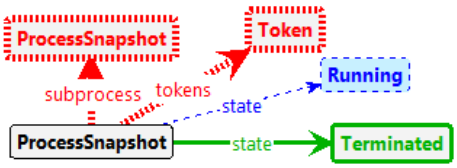

Fig. 9. Termination rule in Groove

3.3 Activities & Subprocesses

Figure 10 depicts the rule generation templates for activities and subprocesses (see Figure 3). Activity execution is divided into two steps implemented in two parts in the first rule template. The upper part 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. This part generates the sample rule on the right of Figure 7. Having multiple incoming or outgoing sequence flows for a flow node is considered bad practice since the implicitly encoded gateways should be explicit to avoid confusion. Our formalization still supports this behavior, but we recommend using static analyzers to avoid such models [3].

The lower part generates one rule that terminates the activity. It deletes a token at the activity and adds one at each outgoing sequence flow. This implicitly encodes a parallel gateway (see Figure 11) but should be avoided, as described earlier.

Subprocess execution is like activity execution. The upper part of the template generates one rule for each incoming sequence flow. The rule deletes an incoming token and adds a process snapshot representing a subprocess. The created process snapshot is represented with a colored circle on the top left corner of the subprocess with a token at each outgoing sequence flow of its start events

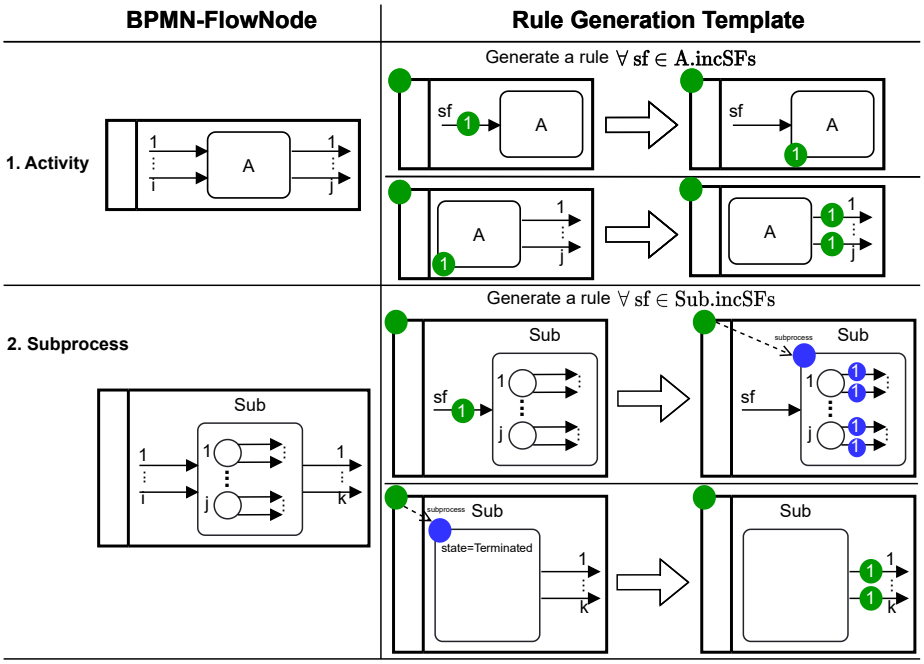

Fig. 10. Rule generation template for activities and subprocesses

(similar to start graph generation). There is a *subprocess* link between the process snapshots to depict the subprocesses relation in Figure 4. If the subprocess has no start events, a token will be added to every activity and gateway with no incoming sequence flows.

The bottom part of the template generates one rule to delete a terminated process snapshot and adds tokens at each outgoing sequence flow. Subprocesses are terminated by the termination rule (see section 3.2).

3.4 Gateways

Figure 11 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.

Exclusive Gateways are triggered by exactly one incoming sequence flow, and exactly one outgoing sequence flow will be triggered as a result. 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 one to an outgoing sequence flow.

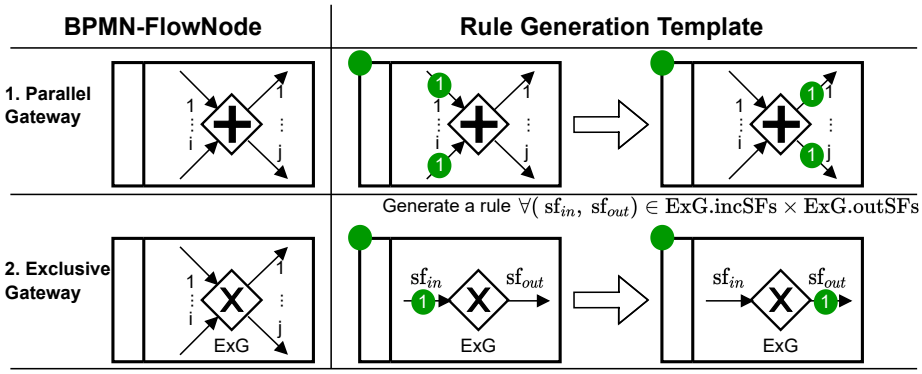

Fig. 11. Rule generation template for gateways

3.5 Message Events

Figure 12 depicts the rule generation templates for *message intermediate throw events* and *message intermediate catch events* (MITE and MICE in Figure 3). The first rule template describes how MITEs interact with MICEs. A MITE deletes an incoming token and adds one at each outgoing sequence flow. In addition, it sends one message to each 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. A process can consume a message if its MICE has at least one token at an incoming sequence flow (see rule template two in Figure 12). We implement optional message sending using nested rules with quantification. Concretely, we use an optional existential quantifier [19] (see blue dotted rectangle marked with optional in Figure 12) to send a message only if the receiving process is ready to consume it.

The second rule template in Figure 12 shows the behavior of 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, as well as add a token at each outgoing sequence flow.

4 Model checking BPMN

Model checking—and verification in general—of BPMN models is necessary to ensure the correctness and reliability of business processes, which ultimately leads to increased efficiency, reduced costs, and improved user satisfaction. Using our approach, model checking a BPMN model is possible using the generated GT system and a set of temporal properties with atomic propositions. Properties are defined using a temporal logic, such as CTL and LTL. In this paper, we will use CTL. An atomic proposition is formalized as a graph and holds in a given

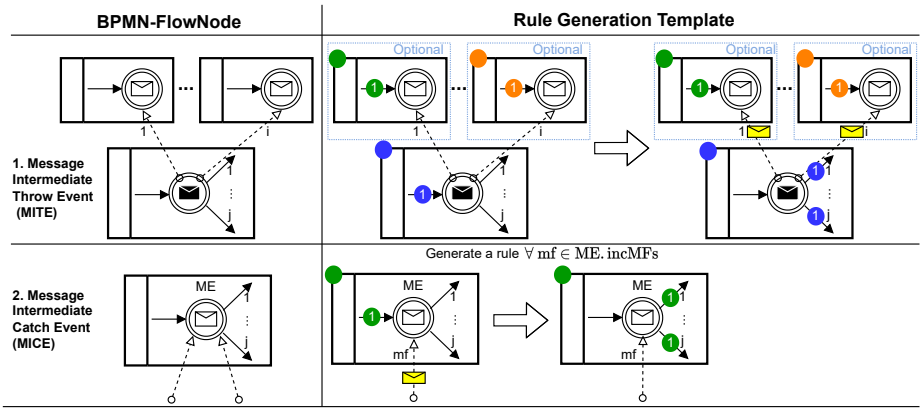

Fig. 12. Rule generation templates for message events

state if a match exists from the graph representing the proposition to the graph representing the state [13].

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 BPMN) 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 defined and checked.

4.1 General BPMN properties

Safeness and *Soundness* properties are defined for BPMN in [6]. A BPMN model is *safe* if, during its execution, at most one token occurs along the same sequence flow [6]. Soundness is further decomposed into (i) *Option to complete*: any running process instance must eventually complete, (ii) *Proper completion*: after completion, each token of the process instance must be consumed by a different end event, as well as (iii) *No dead activities*: each activity can be executed in at least one process instance [6]. In the following, we will describe how to implement the *Safeness* and *Option to complete* properties.

We specify *Safeness* as 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. It is shown in Figure 13.

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. AllTerminated is given in [14].

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

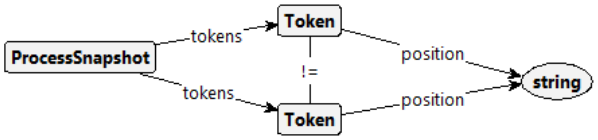

Fig. 13. The atomic proposition *Unsafe* in Groove.

Checking the properties *Safeness*, *Option to Complete*, and *No Dead Activities* is implemented in our tool [14]. 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. Therefore, to define an atomic proposition, a modeler adds process snapshots and tokens to 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 activity 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 activity A simultaneously by creating an LTL/CTL property. Thus, a modeler does not need to know the GT semantics used for execution.

However, the 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 amount of knowledge required from the modeler [15].

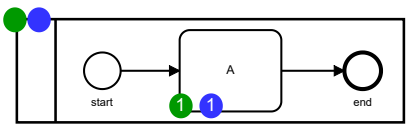

Fig. 14. Token distribution defining an atomic proposition.

5 Implementation

In this section, we will present our tool and then describe experiments regarding its performance.

5.1 BPMN Analyzer tool

Our approach is implemented in a web-based tool called *BPMN Analyzer*, which is open-source, publicly available, and does not require any installation [14]. Figure 15 depicts a screenshot of the BPMN Analyzer.

The modeler can create or upload a BPMN model, which can then be verified using either BPMN-specific properties or custom CTL properties in the verification section. BPMN Analyzer can generate a GT system for the supplied BPMN model and run model checking in Groove [13]. To evaluate the correctness of our HOT, we have created a comprehensive test suite that verifies the correct generation of rules for the implemented BPMN elements [14]. Additionally, we have conducted performance experiments for our approach, as described in the next section.

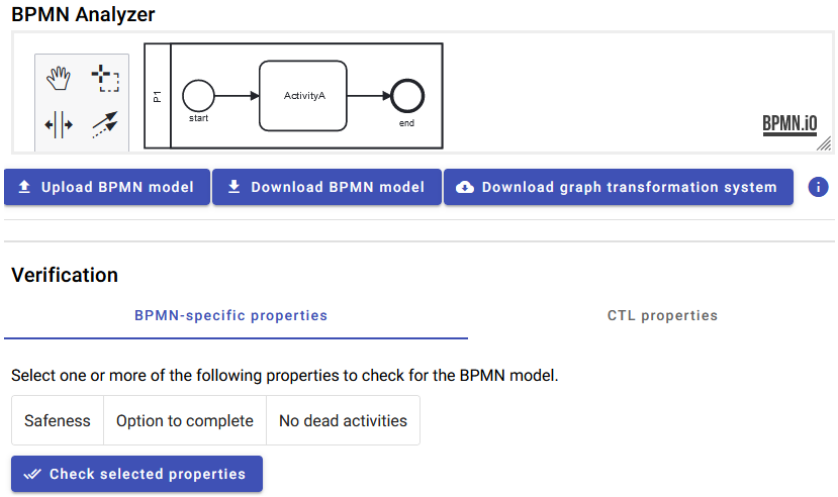

Fig. 15. Screenshot of the BPMN Analyzer tool

5.2 Experiments

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 experimented with ten different BPMN models from [12] to assess the performance of our implementation.

We picked the models at random, besides disregarding some models that were similar. The models include realistic business process models (001, 002, and 020) [12].

To calculate the average runtime, we used the hyperfine benchmarking tool [17] (version 1.15.0), which ran state space exploration for each BPMN model ten times. The experiment was run on Windows 11 (AMD Ryzen 7700X processor, 32 GB RAM) using Groove version 5.8.1 [14].

First, we ran our HOT for the BPMN models. The HOT took less than one second to generate a GT system for each model. Thus, the generation of the GT systems is fast enough. Furthermore, we estimate most of the time is spent writing the GT system to disk.

Second, we ran 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 comprehensive benchmark, including a detailed comparison to other tools, is left for future work.

BPMN model	Processes	Nodes (gw.)	States	Transitions	Total time
001	2	17(2)	68	118	$\sim$ 1.00 s
002	2	16(2)	62	108	$\sim$ 0.97 s
007	1	8(2)	45	81	$\sim$ 0.92 s
008	1	11(2)	49	85	$\sim$ 0.93 s
009	1	12(2)	137	308	$\sim$ 1.01 s
010	1	15(2)	162	357	$\sim$ 1.04 s
011	1	15(2)	44	69	$\sim$ 0.97 s
015	1	14(2)	53	86	$\sim$ 0.95 s
016	1	14(2)	44	68	$\sim$ 0.94 s
020	1	39(6)	3060	8584	$\sim$ 1.75 s

Table 1. Experimental results for a full state space exploration in Groove

6 Related work

A BPMN formalization based on in-place GT rules is given in [22]. The formalization covers a substantial part of the BPMN specification, including complex concepts such as inclusive gateways and compensation. In addition, the GT rules are visual and thus can be aligned with the informal description of the execution semantics of BPMN. A key difference to our approach is that the rules in [22]

are general and can be applied to every BPMN model, while we generate specific rules for each BPMN model using our HOT. Thus, our approach can be seen as a program specialization compared to [22] since we process a concrete BPMN model before its execution. However, they do *not* support property 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 [4]. 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 [12]. 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 time-related constructs [7,12] and data flow [5,9].

Table 2 shows which BPMN elements are supported by our approach and the approaches mentioned above. We cover most BPMN elements when compared to other approaches. The coverage of BPMN elements greatly impacts how useful each approach is to check properties in practice. In addition, we cover the most important elements found in practice since we come close to the element coverage of popular process engines such as Camunda [1].

7 Conclusion & future work

This paper makes two main practical contributions. First, we conceptualize 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 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 element coverage (compared to the literature and industrial process engines) that supports checking behavioral properties. Furthermore, our contribution is implemented in an open-source web-based tool to make our ideas easily accessible to other researchers and practitioners.

Future work targets both of our main contributions. First, we plan a detailed comparison of our HOT approach with approaches that utilize 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 elements used in practice and want to turn the modeling environment of our tool into an interactive simulation environment. In addition, we can use this environment to visualize potential counterexamples in cases where behavioral properties are violated.

Table 2. BPMN elements supported by different formalizations (based on [22]).

BPMN element/feature	Van Gorp et al. [22]	Corradini et al. [4]	Houhou et al. [12]	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.

References

1. Camunda Services GmbH: BPMN 2.0 Implementation Reference. https://docs.camunda.org/manual/7.16/reference/bpmn20/ (Mar 2023)
2. Camunda Services GmbH: Bpmn-js Token Simulation. https://github.com/bpmn-io/bpmn-js-token-simulation (Mar 2023)
3. Camunda Services GmbH: Bpmnlint. https://github.com/bpmn-io/bpmnlint (Mar 2023)
4. 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
5. 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
6. 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
7. 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
8. 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
9. El-Saber, N.A.S.: CMMI-CM Compliance Checking of Formal BPMN Models Using Maude. Ph.D. thesis, University of Leicester (Jan 2015)
10. 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)
11. 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
12. 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
13. 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)
14. Kräuter, T.: Artifacts - ICGT. https://github.com/timKraeuter/ICGT-2023 (Oct 2023)
15. 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

16. Object Management Group: Business Process Model and Notation (BPMN), Version 2.0.2. https://www.omg.org/spec/BPMN/ (Dec 2013)
17. Peter, D.: Hyperfine (2022)
18. 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)
19. 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
20. 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
21. 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
22. 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 examples of the BPMN Analyzer checking general BPMN properties. Our tool and the example models are available as artifacts [14].

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, which might not be intended. This could be a simple mistake of picking an exclusive instead of a parallel gateway.

8.2 Option to complete example

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

Option to complete does not hold for the BPMN model since the scan (second exclusive gateway) might never be successful. This leads to an infinite loop. Thus, not every process might terminate.

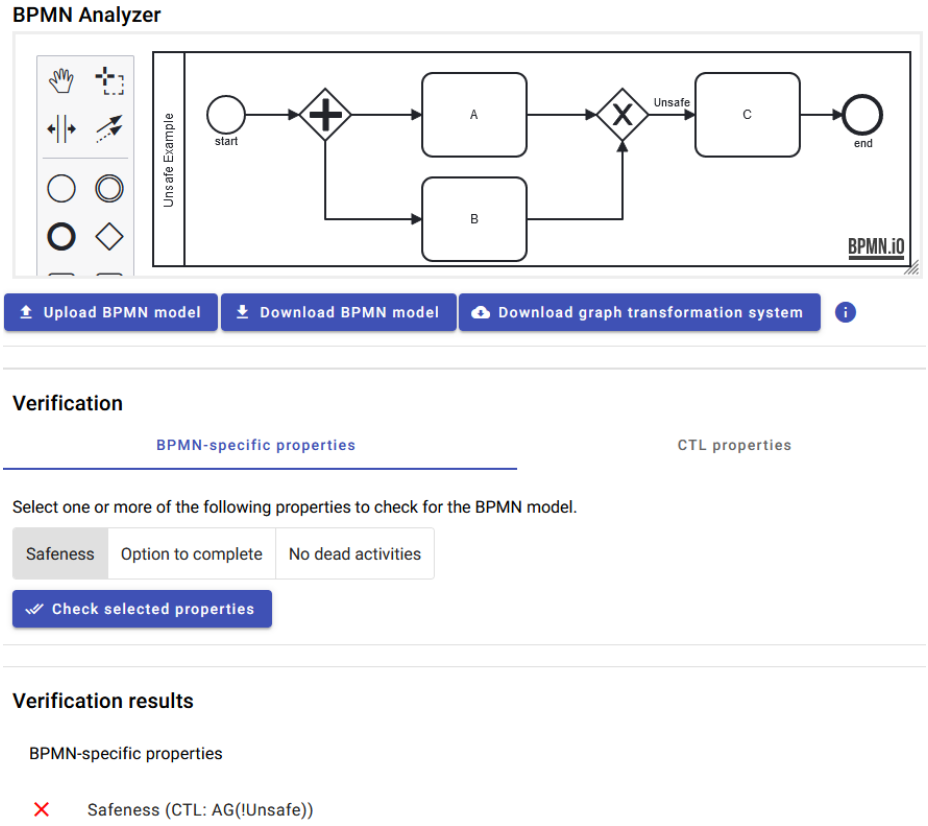

Fig. 16. Screenshot of detecting an unsafe situation

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. The mistake can be fixed by using one gateway type consistently. However, in practice, erroneous situations might be much more complex when multiple processes communicate using messages, signals, and other events.

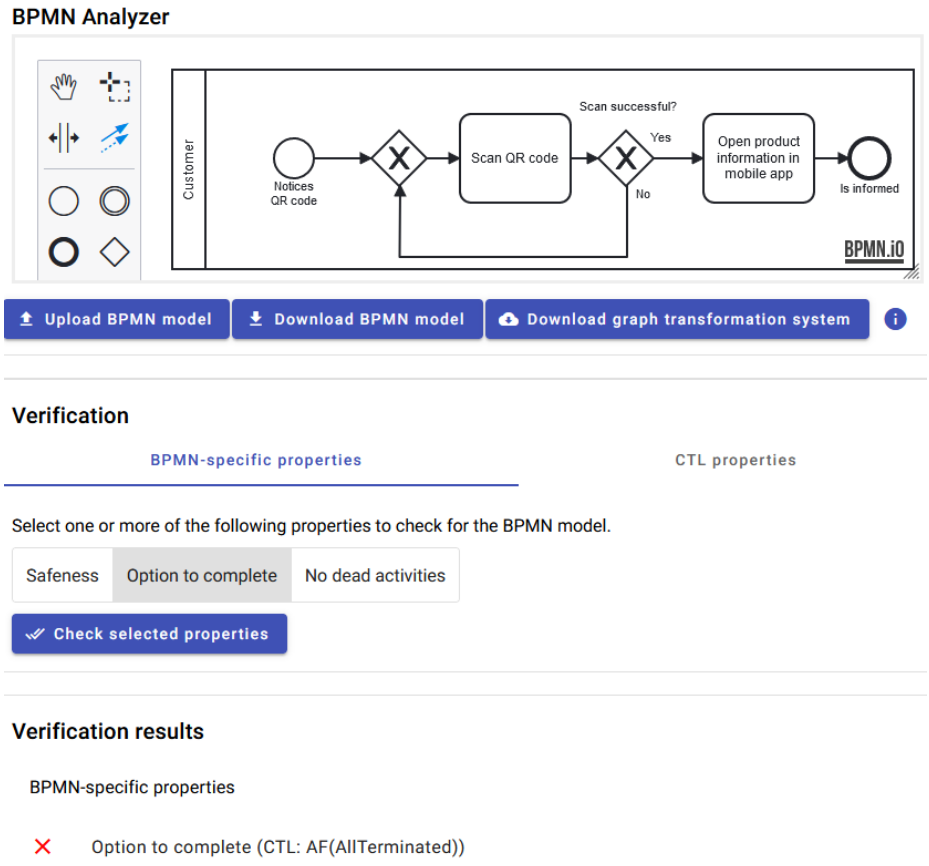

Fig. 17. Screenshot of checking *Option to complete*

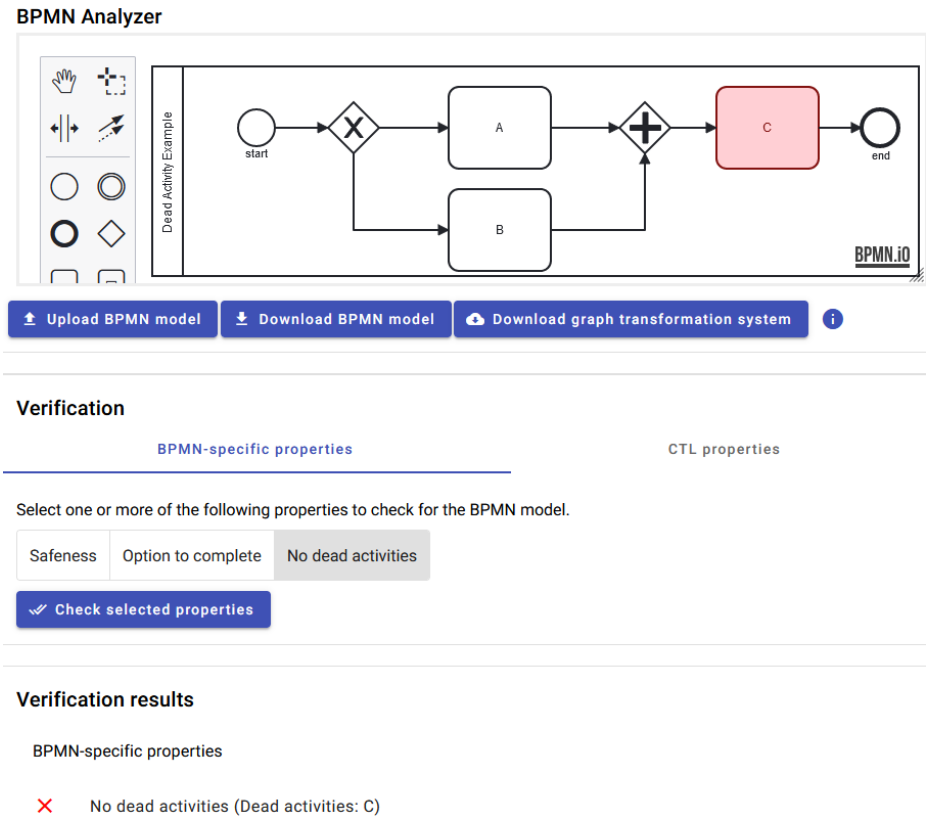

Fig. 18. Screenshot of detecting a dead activity