# A HIGHER-ORDER TRANSFORMATION APPROACH TO THE FORMALIZATION AND ANALYSIS OF BPMN USING GRAPH TRANSFORMATION SYSTEMS *

TIM KRÄUTER [a], ADRIAN RUTLE [a], HARALD KÖNIG [a,b], AND YNGVE LAMO [a]

[a] Western Norway University of Applied Sciences, Bergen, Norway
*e-mail address*: tkra@hvl.no, aru@hvl.no, hkoe@hvl.no, yla@hvl.no

[b] University of Applied Sciences, FHDW, Hanover, Germany
*e-mail address*: 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 article, we propose a formalization of the execution semantics of BPMN that, compared to existing approaches, covers more BPMN elements while also facilitating property checking. Our approach is based on a higher-order transformation from BPMN models to graph transformation systems. To show the capabilities of our approach, we implemented it as an open-source web-based tool.

## 1. Introduction

In today's fast-paced business environment, organizations with complex workflows require powerful means to accurately map, analyze, and optimize their processes. Business Process Modeling Notation (BPMN) [Obj13] is a widely used standard to define these workflows. However, the informal description of the BPMN execution semantics leads to different interpretations of BPMN models and difficulties in checking behavioral properties [CFP+21]. Various studies have shown that business process models suffer from control-flow errors [Men09]. Formalizing BPMN can drastically reduce the cost of business process automation by facilitating the detection of errors and optimization potentials in process models already during design time. For example, general behavioral properties such as *Safeness* and *Soundness* were adapted to BPMN in [CMRT18]. They can uncover control-flow errors in BPMN models leading to deadlocks, dead activities or other undesirable execution states. To this end, we propose a formalization that covers nearly all of the BPMN elements used in practice and supports checking behavioral properties to uncover control-flow errors in a reasonable amount of time.

*Key words and phrases:* BPMN, Higher-order model transformation, Graph transformation, Model checking, Formalization.

* This article is an extended version of [KRKL23].

Preprint submitted to
Logical Methods in Computer Science

© T. Kräuter, A. Rutle, H. König, and Y. Lamo
Creative Commons

In this article, we consider two fundamental concepts when formalizing the execution semantics of BPMN. First, *state structure*, i.e., how model instances 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, which we automatically generate based on a Higher-Order model Transformation (HOT) [TJF+09] for each specific BPMN model, as shown in Figure 1. Our HOT defines a formal execution semantics of BPMN, similar to other approaches that formalize BPMN by mapping to Petri Nets or other formalisms [DDO08].

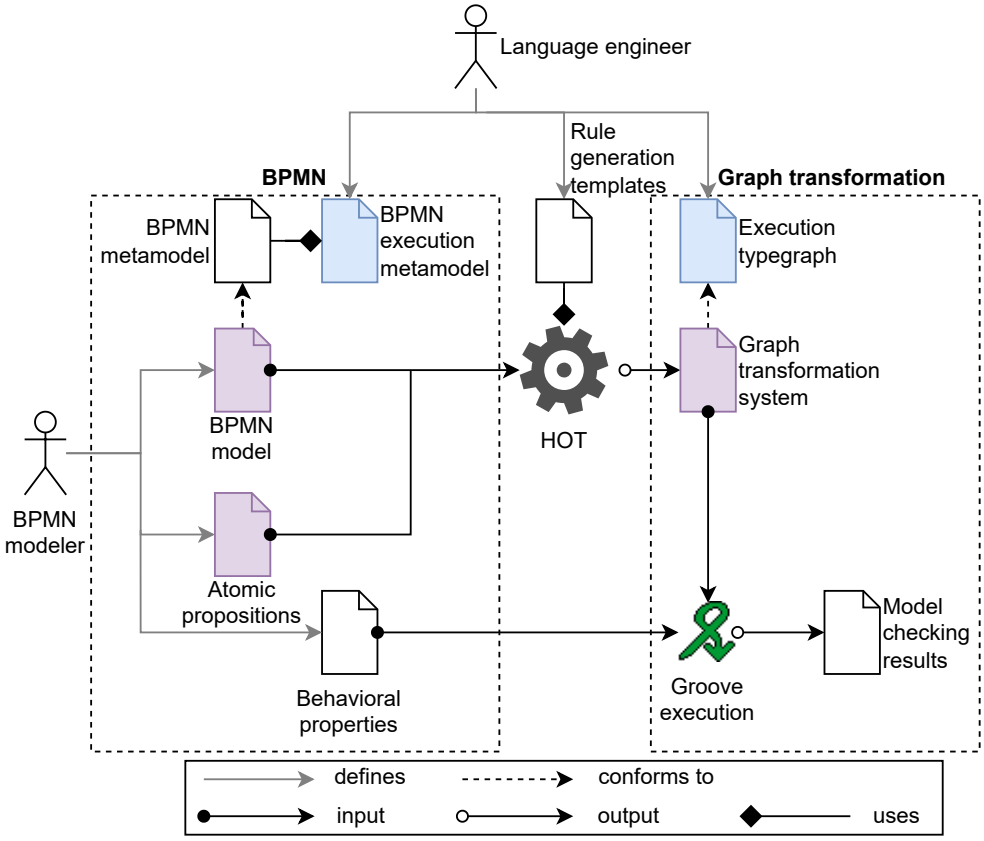


Figure 1: Overview of the approach

To begin the BPMN modeling process, a modeler first defines the BPMN model. The BPMN model may be checked against a predefined list of general behavioral properties, such as safeness and soundness. Furthermore, the modeler may also define custom behavioral properties specifically defined for the BPMN model. Custom properties require atomic propositions to describe desirable or undesirable states during execution, which the modeler can define using our concrete syntax based on the BPMN syntax. The defined BPMN model must adhere to the BPMN metamodel as outlined in the BPMN specification by the Object Management Group [Obj13]. The BPMN execution metamodel is defined by language engineers, utilizing the BPMN metamodel as a foundation to create the state structure for executing BPMN models.

We define a HOT from BPMN models and atomic propositions to GT systems (see purple-colored elements in Figure 1). We call the transformation *higher-order* since the resulting graph-transformation systems represent model-transformations themselves [TJF+09]. 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 2). The obtained GT system conforms to the execution type graph, which corresponds to the BPMN execution metamodel. In the figure, we have used the same color for artifacts that correspond to each other. Ultimately, we use Groove to execute the GT system and check the behavioral properties defined earlier. To facilitate model checking of custom behavioral properties, we create specific GT rules for the corresponding atomic propositions during the HOT.

The overview in Figure 1 is divided into two separated parts, denoted by dashed rectangles, to indicate the versatility of the approach as it can be applied to formalize other behavioral languages, such as activity diagrams and state charts [SSHK15, Obj17]. This formalization will require the language engineer to establish a new execution metamodel and a HOT for the new language. One could even change the *target* of the HOT from GT to a different formalism (term rewriting, Petri Nets, process algebras) if this makes sense for a given behavioral language [KKR+23].

This article consists of two main contributions. First, we introduce a new approach utilizing a HOT to generate GT rules — instead of providing fixed model-independent 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 behavioral property checking. Furthermore, our formalization is implemented as a user-friendly, open-source web-based tool, the *BPMN Analyzer*, which can be used online without needing installation [Krä23].

Our contributions are practical, not theoretical. We build upon the comprehensive theory and tools available in the GT research field. Concretely, we utilize the single-pushout (SPO) approach with negative application conditions (NAC) [EHK+97], as implemented in Groove [Ren04]. In addition, we utilize *nested rules* with quantification to make parts of a rule repeatedly applicable or optional [Ren06, Ren17]. Moreover, we utilize the NACs to implement more intricate parts in the BPMN execution semantics, such as the termination of processes. Formal definitions of SPO rules, their application, and the corresponding extensions of the theory (NACs, nested rules) are well-known, see [EHK+97, Ren06]. We do not repeat them and instead focus on our practical contribution.

This article extends [KRKL23] as follows. (i) We explain many more BPMN elements, which are covered by our approach (see elements highlighted in blue in Figure 3). (ii) We enhance the explanation of the custom properties in section 3 by using an order handling process to illustrate use cases for these properties. (iii) We detail the extensively improved BPMN analyzer tool in section 4 in which modelers can use our new atomic proposition editor. (iv) We test the scalability of our approach with 300 synthetically generated BPMN models of increasing size in section 4.

**Outline** The remainder of this article is structured as follows. First, we describe the BPMN semantics formalization using the HOT (section 2) before explaining how this can be utilized for model checking general BPMN and custom properties (section 3). Then, we detail the BPMN Analyzer, which implements our approach in section 4 and describe how we tested its performance and scalability. Finally, we discuss related work regarding BPMN element coverage in section 5 and conclude in section 6.

## 2. BPMN syntax & BPMN semantics formalization

Figure 2 depicts the structure of BPMN models with the corresponding concrete syntax BPMN symbols contained in clouds. A BPMN model is represented by a Collaboration that has participant Processes and MessageFlows between InteractionNodes. Each participant is a Process containing FlowElements. A FlowElement is either a FlowNode or SequenceFlow. A FlowNode is either an Activity, a Gateway, or an Event and can be connected to other FlodNodes using SequenceFlows. Many types of activities, gateways, and events exist, such as call activities, parallel gateways, and start events. 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 [FR19].

Figure 2 is a simplified excerpt of the BPMN metamodel described in the BPMN specification [Obj13]. Each class depicted in Figure 2 maintains consistent naming with the BPMN metamodel, and all associations are directly found in [Obj13] or simplified, i.e., given by compositions of multiple existing associations.

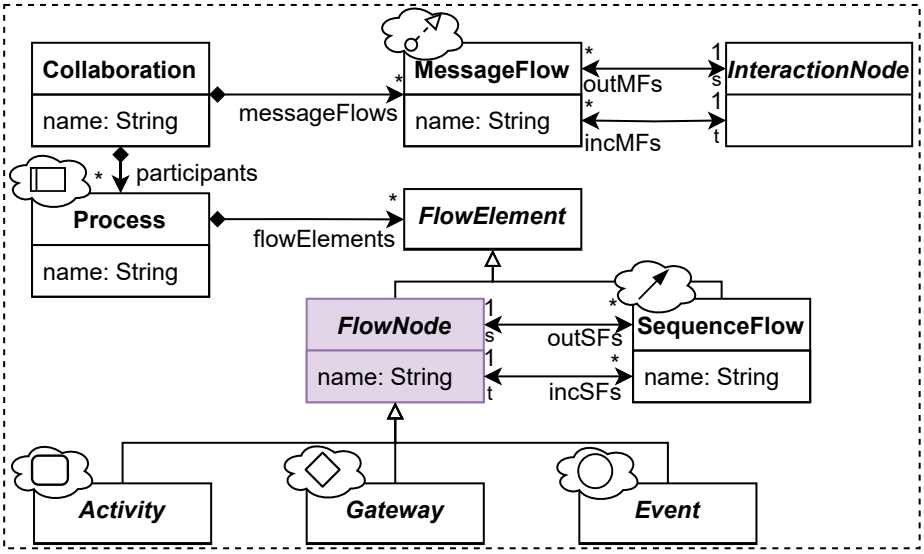


Figure 2: Simplified excerpt of the BPMN metamodel [Obj13]

Our 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. An extensive overview of BPMN and its elements can be found in [FR19]. Although all these elements have been implemented and tested (see [Krä23]), we only explain the realization of the elements marked with a blue background due to space limitations. In the following, first, we define the BPMN execution metamodel to represent the BPMN state structure, and then we explain our formalization of the elements in Figure 3.

### 2.1. BPMN execution metamodel.

The BPMN execution semantics is described using the concept of *tokens* [Obj13, FR19], which can be located at sequence flows and specific flow nodes. Tokens are consumed and created by flow nodes according to the flow node's type and

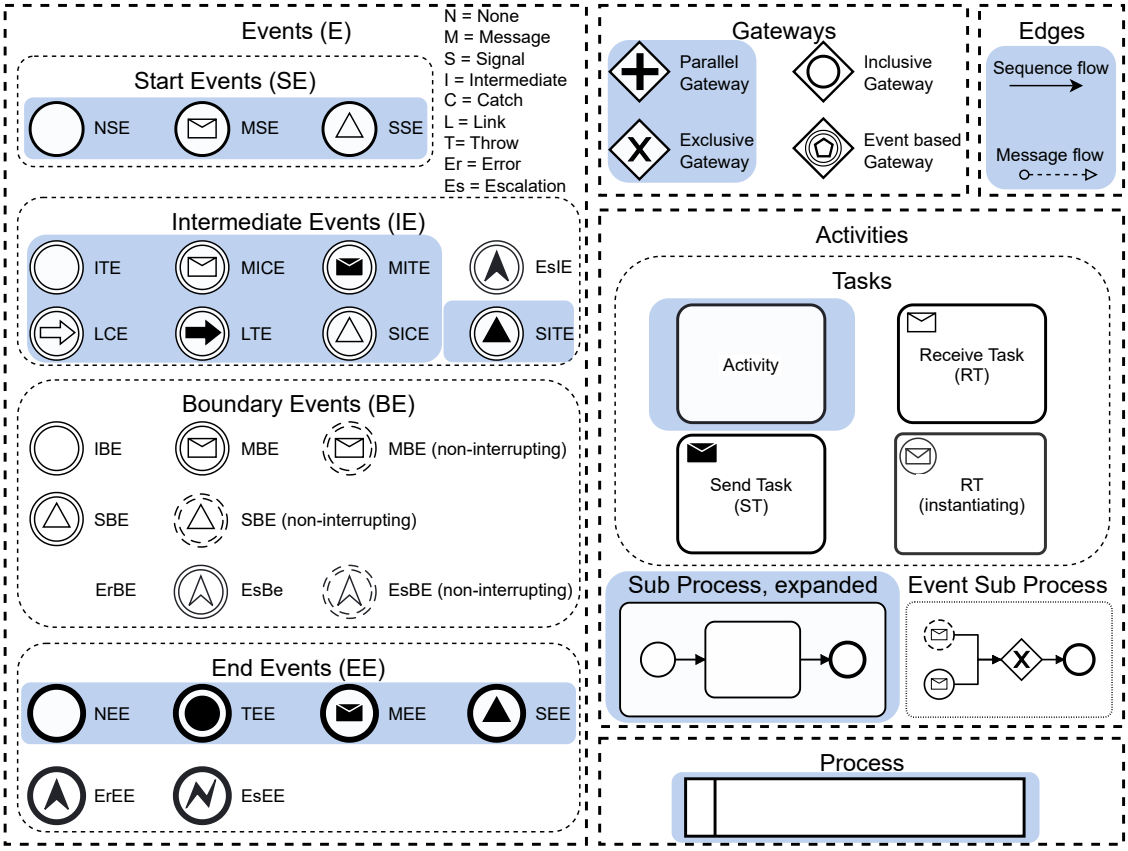


Figure 3: Overview of the supported BPMN elements (structure adapted from [HBP+22])

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 2. In our formalization of BPMN, we follow this token-based representation of the execution semantics.

To describe processes holding tokens during execution, we define the execution metamodel shown in Figure 4, depicted as a UML class diagram. The first task of a language engineer in our approach is to define the execution metamodel (see Figure 1).

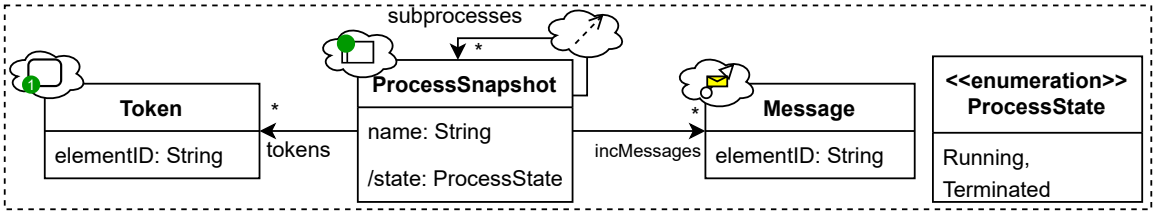


Figure 4: BPMN execution metamodel

The BPMN execution metamodel was not created by extending the BPMN metamodel and adding missing concepts such as tokens and messages. We created a minimal execution model that only contains concepts needed during execution. This is only possible since the

HOT generates our rules for each specific BPMN model such that the structure of each model is already implicitly encoded in the rules. This design choice leads to smaller states in the GT system compared to an execution metamodel that extends the BPMN metamodel.

In Figure 4, 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 [Cam23c].

The execution metamodel in Figure 4 is a UML class diagram without operations, which can be seen as an attributed type graph [HT20]. We keep the execution metamodel and the execution type graph separate (see Figure 1) because the execution metamodel should be independent of the formalism used to define the execution semantics. One can reuse the execution metamodel when changing the formalism or concrete tool implementing the formalism (in our case, Groove) by adjusting how the execution metamodel is transformed. 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 (see Figure 1). 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.

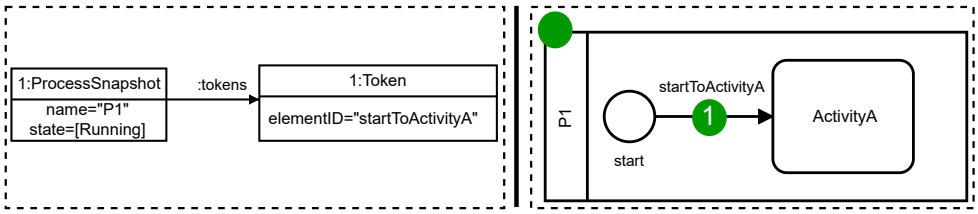


Figure 5: Example start graph in abstract (left) and concrete syntax (right)

The HOT generates one or more GT rules for each FlowNode, i.e., state-changing element in a BPMN model. To better understand the transformation process, we will begin by presenting example results, namely the generated rules for an activity. Following this, we will explain how our HOT creates these rules and rules for the other elements in Figure 3.

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

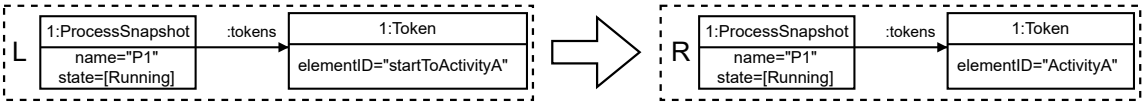


Figure 6: Example GT rule to start an activity (abstract syntax)

For the rest of the article, we will depict all rules in the concrete syntax introduced earlier. The rule from Figure 6 depicted in concrete syntax is shown on the top in Figure 7. The rule on the bottom in Figure 7 implements the termination of an activity, which will move one token from the activity to the outgoing sequence flow.

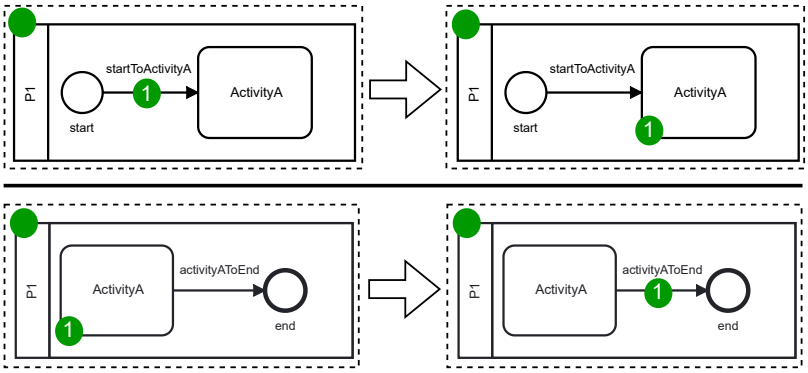


Figure 7: Example GT rules to start (top) and terminate (bottom) an activity

In the following subsections, we use our concrete syntax to define how our HOT generates these rules and rules for other flow nodes. Elements of the HOT are depicted using rule generation templates that show how specific rules are created for various flow nodes. Defining the rule generation templates and, thus, the HOT from BPMN to GT systems is the second task of the language engineer in our approach (see Figure 1).

2.2. **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 *None End Events* (NEEs 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.

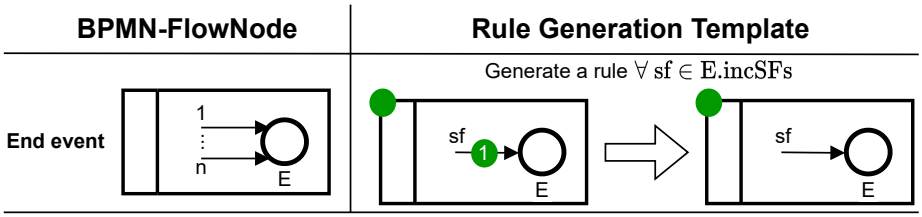


Figure 8: Rule generation template for *None End Events*

For example, if a BPMN model contains one NEE with two incoming sequence flows, the HOT will generate two GT rules as defined in the rule generation template in Figure 8.

The generated end event rules delete tokens individually 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 applies to all process snapshots. The termination rule in Figure 9 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. We use Groove Syntax instead of our concrete syntax since it is a special, more complex rule, and we do not want the concrete syntax to become too complex.

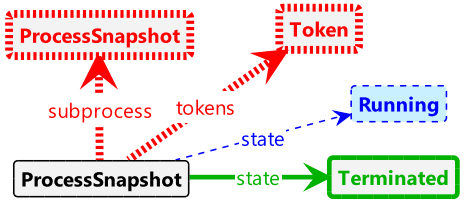


Figure 9: Termination rule in Groove

The Groove syntax is the following. The thin black elements in Figure 9 need to be present and will be preserved during transformation, while the dashed blue elements need to be present but will be removed. Furthermore, the fat green elements will be created, and the dashed fat red elements represent the NACs, whose presence prevents the rule from being applied.

2.3. **Activities & Subprocesses.** Activities represent work to be performed within a BPMN process, while subprocesses group parts of a BPMN model together, allowing for reusability and separation of concerns [Obj13].

Figure 10 depicts the rule generation templates for activities and subprocesses (see Figure 3). Activity execution is divided into two steps implemented in parts **(a)** and **(b)** in the rule generation template **(1)**. Part **(a)** 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 left 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 those models not to force modelers to rewrite them, but we recommend using static analyzers to avoid such models [Cam23e].

Part **(b)** 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.

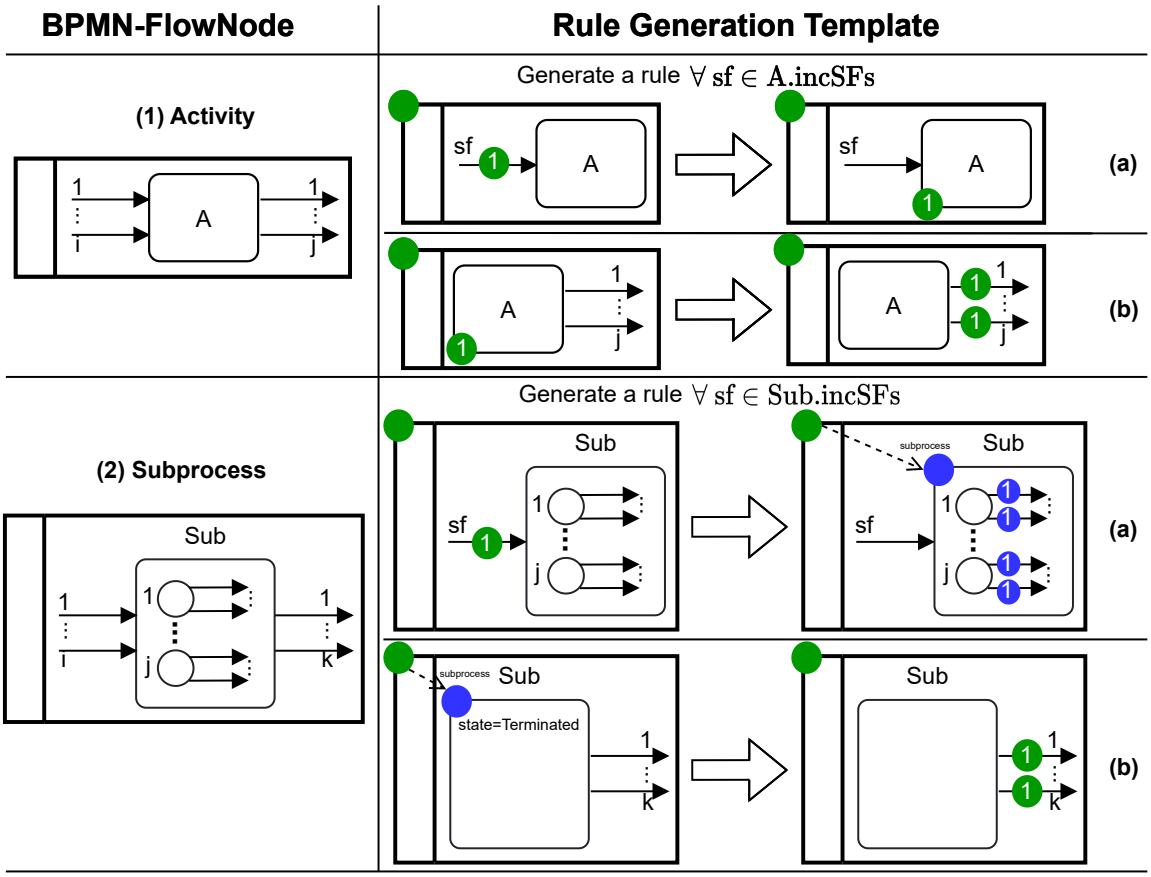


Figure 10: Rule generation template for activities and subprocesses

Subprocess execution is like activity execution. Part **(a)** 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 (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.

Part **(b)** 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 2.2).

2.4. **Gateways.** Parallel gateways represent forking and joining in the sequence flow. Exclusive gateways represent exclusive choices and merges in the sequence flow [Obj13].

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 to each outgoing sequence flow.

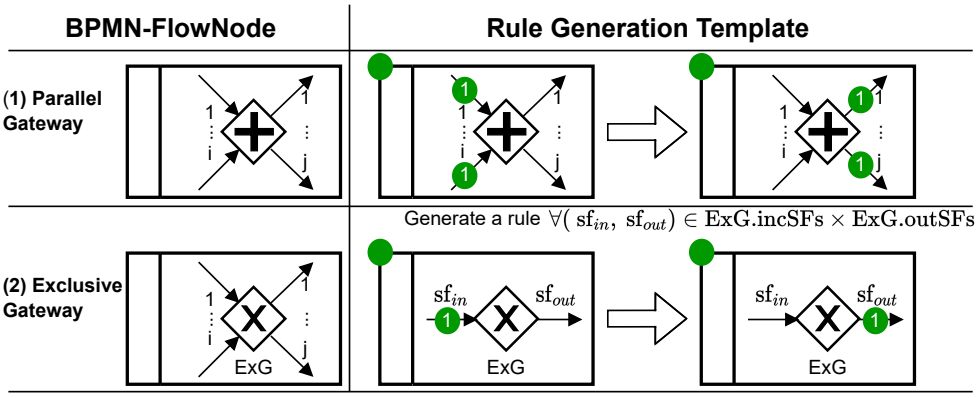


Figure 11: Rule generation template for gateways

Exclusive Gateways are triggered by exactly one incoming sequence flow, and exactly one outgoing sequence flow will be triggered as a result. In practice, boolean conditions using data attached to the process are attached to exclusive gateways that decide which outgoing sequence flow to follow. We do not model data flow in our formalization and instead allow each outgoing sequence flow to be explored. 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.

2.5. **Message Events.** Message events are events directed at a single recipient. Thus, they are unicast compared to broadcast signal events discussed later in subsection 2.6. Figure 12 depicts the rule generation templates for *Message Intermediate Throw Events* (MITE in Figure 3). Rule generation template **(1)** describes how MITEs interact with *Message Intermediate Catch Events* (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 (1) in Figure 12). We implement optional message sending using nested rules with quantification. Concretely, we use an optional existential quantifier [Ren06] (see dotted rectangle labeled Optional in Figure 12) to send a message only if the receiving process is ready to consume it.

Rule generation template **(2)** describes how MITEs interact with *Message Start Events* (MSEs). For each MSE, a new process snapshot is created with tokens located at its outgoing sequence flows. We split the interaction of MITEs with MICEs and MSEs into two rule templates for better understanding. However, a MITE might interact with MICEs and MSEs simultaneously. Thus, our HOT implements a merge of both templates. *Message End*

*Events* (MEE) behave similarly to MITEs but only delete incoming tokens and do not add outgoing tokens.

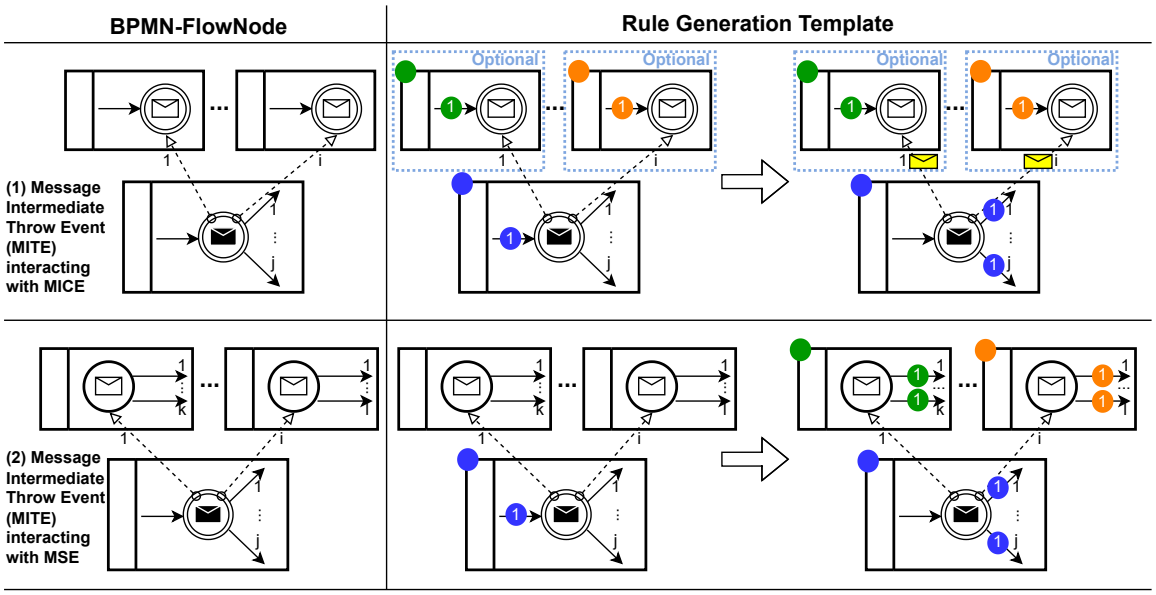


Figure 12: Rule generation templates for MITEs interacting with MICEs **(1)** and MSEs **(2)**

The rule generation template in Figure 13 shows the behavior of MICE (see MICE in Figure 3). 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.

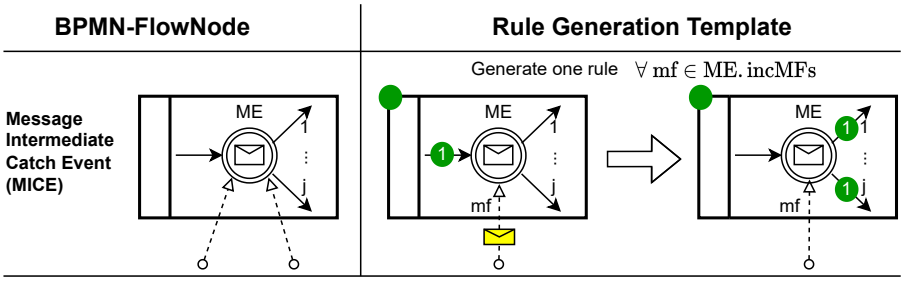


Figure 13: Rule generation templates for MICEs

2.6. **Link Events.** Link events are similar to "Go To" statements since they move tokens from link throw events to link catch events in the same process level (cannot link to subprocesses). They are meant to avoid long sequence flows and connect BPMN models spanning multiple pages but can also be used to create loops due to their "Go To" nature [Obj13]. Figure 14 depicts the rule generation template for *Link Throw Events* (LTEs), see LTE in Figure 3. It shows how LTEs interact with *Link Catch Events* (LCEs).

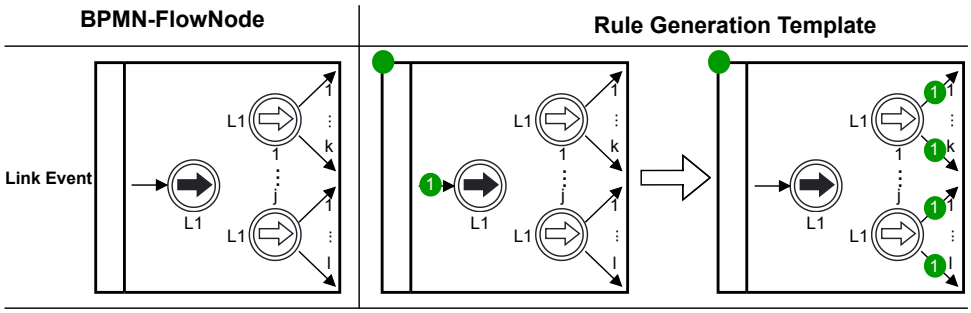


Figure 14: Rule generation template for LTEs interacting with LCEs

Each rule deletes a token at that sequence flow and adds tokens to all outgoing sequence flows of matching LTEs. An LTE matches an LCE if they have the same name or event definition (see [Obj13]). Our HOT automatically finds matching LTEs during transformation and then applies the rule template shown in Figure 14.

2.7. **Signal Events.** Each signal event is assigned a signal name. Signal throw events *broadcast* to all signal catch events with the same signal name. Signal broadcasts have a global scope, i.e., they can communicate across process levels and pools [Obj13].

Figure 15 depicts the rule generation template for *Signal Intermediate Throw Events* which interact with *Signal Intermediate Catch Events* and *Signal Start Events* (SITE, SICE, and SSE in Figure 3). *Signal End Events* (SEE) behave similarly to SITEs but only consume incoming tokens and do not add outgoing tokens.

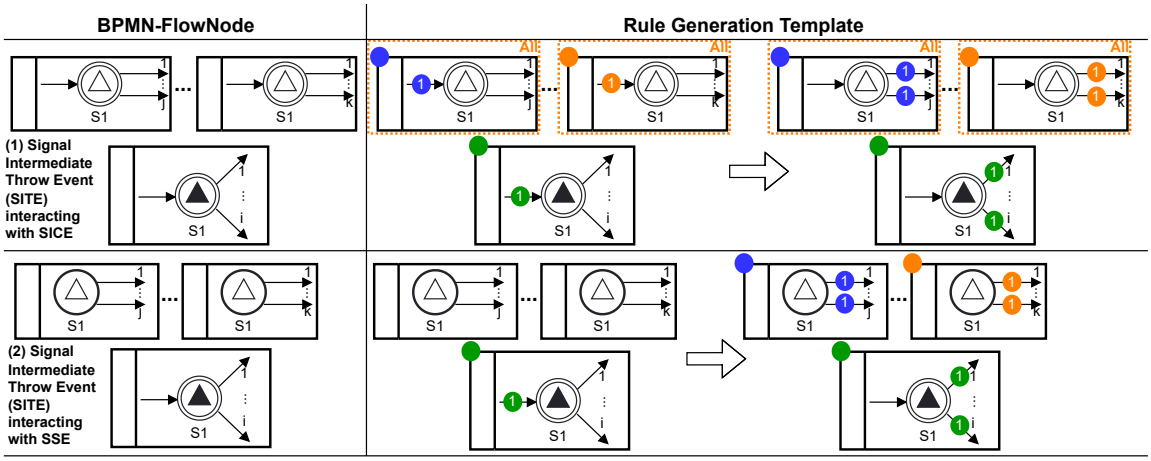


Figure 15: Rule generation templates for SITEs interacting with SICE **(1)** and SSE **(2)**

Rule generation template **(1)** describes how SITEs interact with SICE. Like other intermediate events, the incoming token is consumed while one token is added for each outgoing sequence flow. Due to its broadcast semantics, a SITE interacts with all matching SICEs with an incoming token. A SITE and SICE match if they have the same signal name. In our templates, we assume that the signal name is the SITE/SICE name. For each

matching SICE, a universally quantified nested rule consumes the incoming token and adds a token for each outgoing sequence flow. We use a universal quantifier (All in Figure 15) since one process snapshot might have multiple tokens waiting before a SICE. Then, a SITE should trigger this SICE multiple times.

Rule generation template **(2)** describes how SITEs interact with SSEs. Analogous to MITEs and MSEs, new process snapshots with tokens at the outgoing sequence flows of the SSEs are added for each matching SSE. Each matching SSE is only triggered once, meaning we do not need any quantified nested rules. We split the interaction of SITEs with SICEs and SSEs into two rule templates for better understanding. However, a SITE might interact with SICEs and SSEs simultaneously. Thus, our HOT implements a merge of both templates.

2.8. **Terminate Events.** A *Terminate End Event* (TEE) abnormally terminates the running process [Obj13], meaning the process changes its state to terminated, and all its tokens are consumed. Figure 16 depicts the rule generation template for TEEs (see TEE in Figure 3).

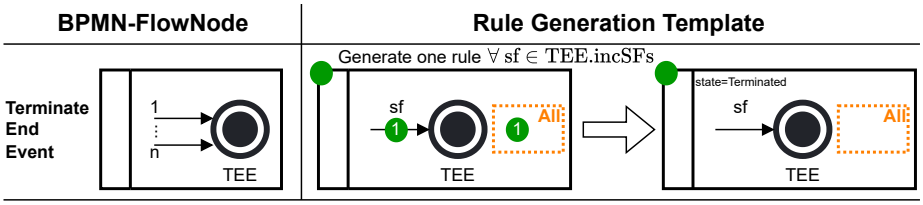


Figure 16: Rule generation template for terminate end events

One rule is generated for each incoming sequence flow of a TEE. The rule consumes the incoming token, similar to the rules for end events, but also changes the process snapshot state to Terminated. In addition, the rule deletes all other tokens of the process snapshot using a universally quantified nested rule (see dotted rectangle labeled **All** in Figure 16). Terminating a process must also terminate its subprocesses, which is not shown in the rule template in Figure 16 for brevity; it is described in our wiki [Krä23].

## 3. 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 user satisfaction. Using our formalization approach, BPMN models may be verified against behavioral properties —both general and custom— by utilizing the generated GT system (see Figure 1). These behavioral properties are defined using temporal logic, such as Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) [BK08, CHVB18]. As mentioned in section 1, modelers may use our approach to specify custom properties consisting of atomic propositions and operators from CTL/LTL. The atomic propositions are transformed to graph conditions by the HOT. A graph condition is a GT rule which does not delete or add elements. A proposition holds in a given state if a match of the graph condition exists in the graph representing the state [KR06].

We differentiate between two types of behavioral properties: *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 BPMN syntax) since they can be checked using a standard modeling tool without implementing execution semantics. We will now give an example of predefined general BPMN properties and show how our approach can check them. Then, we describe how custom properties can be defined and checked.

3.1. **General BPMN properties.** *Safeness* and *Soundness* properties are defined for BPMN in [CMRT18]. A BPMN model is *safe* if, during its execution, at most one token occurs along the same sequence flow [CMRT18]. 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 [CMRT18]. Process completion is synonymous with process termination. In the following, we will describe how to implement the *Safeness* and *Option to complete* using CTL, as well as *Proper completion* and *No dead activities* by analyzing the GT system's state space.

We specify *Safeness* as the following CTL property:

$$AG(\neg \text{Unsafe}) \tag{3.1}$$

The path quantifier $A$ means the following proposition $G(\neg \text{Unsafe})$ should hold for *all* paths starting from the current state. The temporal operator $G$ means the following proposition $\neg$ Unsafe should hold at all states in the future [CHVB18]. More detailed information about CTL can be found in [CHVB18, BK08]. Combining the path quantifier $A$ and temporal operator $G$ in (3.1) means $\neg$ Unsafe should hold for all states in all paths starting from the initial state. Thus, (3.1) describes that a state labeled as Unsafe should not be reachable. The atomic proposition Unsafe is true if two tokens of one process snapshot point to the same sequence flow. Atomic propositions are either fulfilled Figure 17 shows how Unsafe is represented as a graph condition in Groove.

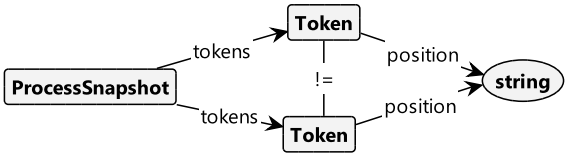


Figure 17: The atomic proposition Unsafe as a Groove graph condition.

*Option to complete* is specified using the following CTL property:

$$AF(\text{AllTerminated}) \tag{3.2}$$

The temporal operator $F$ means the following proposition AllTerminated should hold in some state in the future [CHVB18]. Thus, (3.2) describes that a state labeled as AllTerminated should be reached for all paths starting from the initial state. The atomic proposition AllTerminated is true if there exists no process snapshot in the state Running, i.e., all process snapshots are Terminated. Figure 18 shows how Terminated is represented as a graph condition in Groove.

Checking the properties *Safeness* and *Option to complete* is implemented by checking the CTL properties above using Groove [KR06, Ren08]. The property *Proper Completion* is


Figure 18: The atomic proposition AllTerminated as a Groove graph condition.

implemented by checking the GT system's state space for two executions of an end event in the same path. Similarly, *No dead activities* is implemented by analyzing the GT system's state space to see if each activity has been executed at least once [Krä23].

3.2. **Custom properties.** To make model checking user-friendly, we enable modelers to define atomic propositions using the concrete syntax of the extended BPMN execution metamodel introduced in Figure 4 (see Figure 19). An atomic proposition is defined as a process snapshot with a token distribution, which we can automatically convert to a graph condition in Groove (see Figure 20). Recall that graph conditions are GT rules that do not add or delete elements. Atomic propositions may be connected by CTL operators to create temporal formulas that should hold in the given BPMN model. Furthermore, modelers may forbid certain states in the BPMN model by specifying that a certain token distribution should not exist. These situations would lead to Negative Application Conditions (NACs) in the graph conditions.

For example, the token distribution shown in Figure 19 defines a process snapshot with two tokens at activity *Ship goods*. A modeler could use this atomic proposition to check if the activity *Ship goods* is executed twice by creating an appropriate CTL property. Shipping goods twice but only receiving one payment during an order-handling process would be a critical error for a business. The order handling process in Figure 19 is taken from [Rüc21] but changed to contain a modeling error. It contains an exclusive gateway instead of a parallel gateway. The modeling error could lead to shipping goods twice if the process is not corrected before deployment.

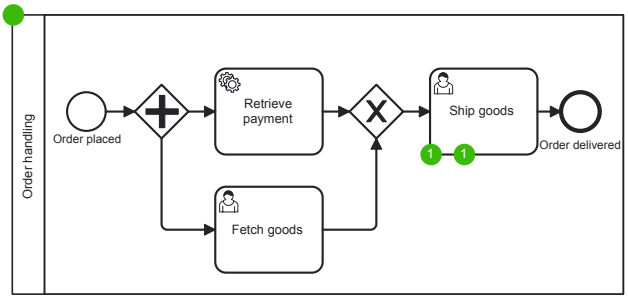


Figure 19: Atomic proposition *shipGoodsTwice* defining shipping goods twice.

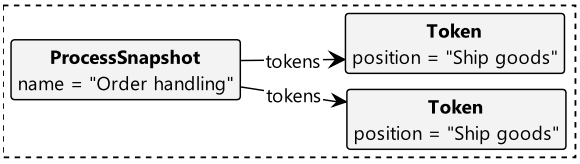


Figure 20: Generated Groove graph condition describing the atomic proposition in Figure 19.

Another proposition for the same process with a different error is shown in Figure 21. The proposition *noShipment* defines that the activity *Ship goods* should not run (has no token). "Has no token" is depicted by crossing out the token symbol and represents an extension of our concrete syntax introduced for defining propositions. This proposition can be used to define a CTL property to check if shipping always occurs. In this case, the error in the order handling process prevents shipping from occurring. The GT systems for both variants of the order handling process containing the propositions can be found in [Krä23].

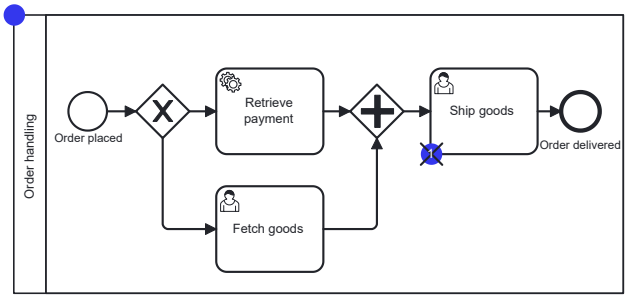


Figure 21: Atomic proposition *noShipment* defining no ongoing shipping of goods.

Using an atomic proposition editor based on the BPMN concrete syntax, modelers do not need extensive knowledge about the GT-based execution semantics. Although the expressiveness is not as powerful as in the GT-based execution semantics in Groove—e.g., one can use nested rules with quantification in graph conditions—we favor *simplicity* over *expressiveness*. In addition, we attempt to stay as independent as possible from the framework and tools used for the execution semantics (see the right part in Figure 1).

Finally, the modeler must still know temporal logic to specify custom properties. To combat this problem, we added temporal logic templates to our tool to generate commonly occurring propositions without knowledge about temporal logic. The next section about the BPMN Analyzer discusses this feature in detail. A domain-specific property language for BPMN would further lessen the knowledge required from the modeler [MDL+14].

## 4. BPMN Analyzer

Our approach is implemented as a web-based tool called *BPMN Analyzer*, which is open-source, publicly available, and does not require any installation [Krä23, KRKL23]. A demonstration of the BPMN Analyzer is available online[1]. Figure 22 depicts a screenshot of the BPMN Analyzer. We use the order handling process from [Rüc21] as an example. It is the same BPMN model as in Figure 19 and Figure 21 but without modeling errors.

The modeler can create or upload a BPMN model, which can then be verified using either general BPMN properties or custom properties formulated in CTL. BPMN Analyzer generates a GT system for the supplied BPMN model and runs model checking against the specified properties in Groove [KR06, Ren08]. We have created a comprehensive test suite [Krä23], which verifies that rules are generated as defined by the rule generation templates in the previous section. The test suite covers over 90% of our source code.

[1] `https://youtu.be/MxXbNUl6IjE`

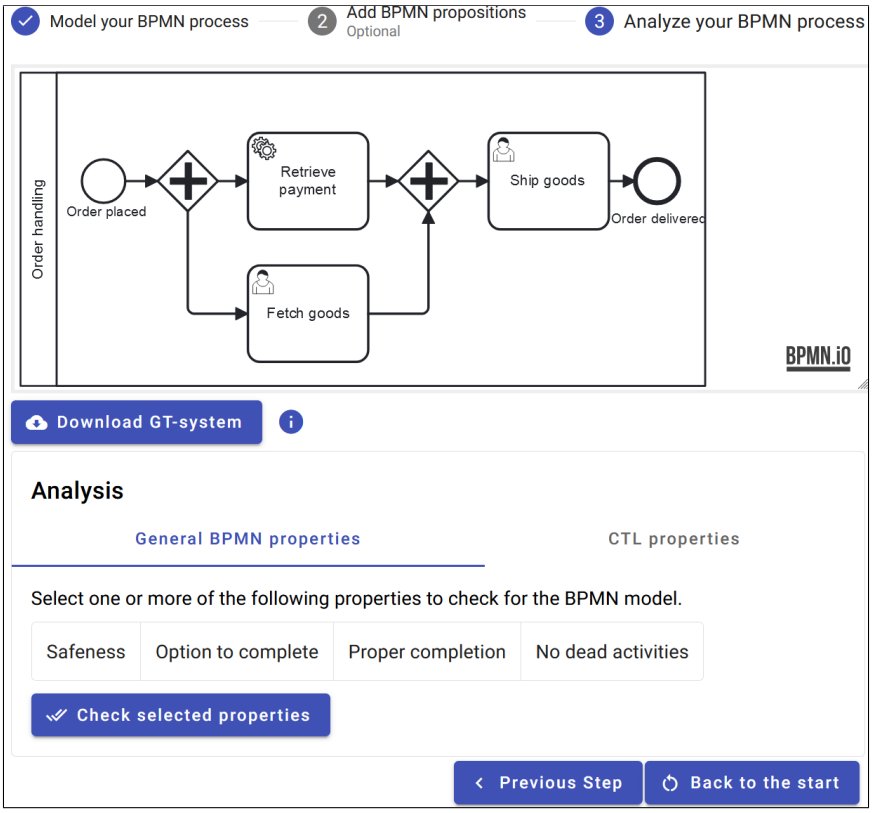


Figure 22: Screenshot of the *analysis step* in the BPMN Analyzer

The BPMN Analyzer interface is structured into three steps that guide the user transparently through the modeling and analysis process.

(1) The **Modeling** step lets users upload or define the BPMN model. We utilize a properties panel in the modeling step so that IDs of BPMN elements can be viewed and edited directly in the model editor. This allows for better traceability between BPMN elements and generated GT rules if a user inspects the GT system.

(2) The **BPMN Propositions** step contains our custom *Token Editor* and is shown in Figure 23. In this step, users may create atomic propositions, which can be used as ingredients in the custom CTL properties in the analysis step. In Figure 23, the user is editing one of two created propositions. Users who are only interested in general BPMN properties may skip this step. Atomic propositions are created using the concrete syntax detailed in subsection 3.2, implemented in our *Token Editor*. As mentioned, users attach tokens and process snapshots to the BPMN model created in the modeling step to create a proposition.

(3) In the **Analysis** step, users may check the general BPMN properties and build custom CTL properties using the atomic propositions. The properties builder utilizes the textual CTL syntax implemented in Groove to specify custom properties using the atomic propositions from the previous step, see Figure 24. The two atomic propositions created in the previous step (see Figure 23) are available to the user. The CTL properties builder comes with CTL templates to facilitate commonly occurring CTL properties. These

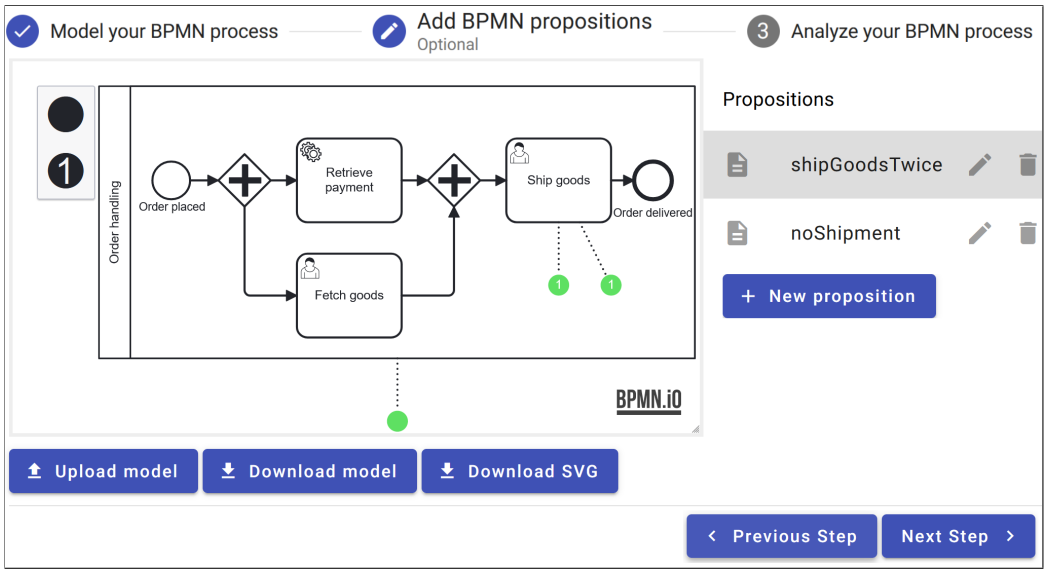


Figure 23: Screenshot of the *propositions step* in the BPMN Analyzer

templates allow users to check whether a state (described by an atomic proposition) can be reached or is never reached (see Figure 24). Thus, simple safety and liveness properties can be checked using these templates. Model-checking experts in an organization can define more templates in the future and share them for reuse. Users who download the generated GT system may inspect and edit the graph conditions generated from the atomic propositions; they may also specify more properties and check them using Groove.

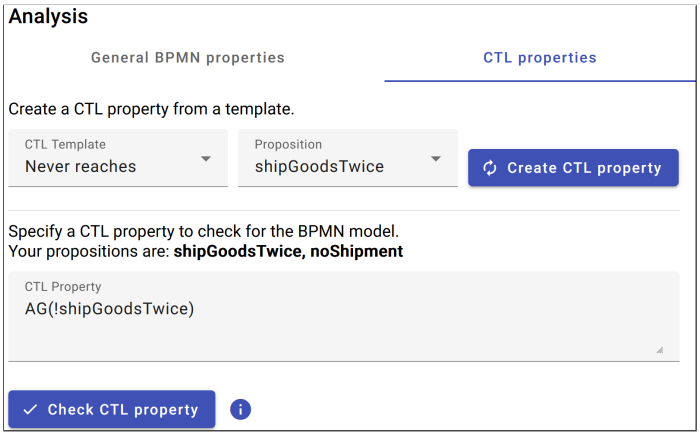


Figure 24: CTL properties builder in the *analysis step* of the BPMN Analyzer

4.1. **Reusable libraries.** In addition to the BPMN Analyzer, we published multiple parts of our application as libraries that can be reused seamlessly by other researchers and practitioners. The BPMN metamodel extension needed to define atomic propositions is

published as an npm module (*token-bpmn-moddle*) [Krä23]. Npm is the default package manager for the JavaScript programming language. In addition, the Token Editor to create atomic propositions is also published as an npm module (*token-bpmn*) [Krä23]. Furthermore, we published our *graph-rule-generation* library[2] to generate Groove GT systems to the Maven central repository [Krä23]. The Maven central repository is the standard repository for developing JVM-based applications. We explain each library in detail in the following sections.

4.1.1. *BPMN metamodel extension.* Our implementation *token-bpmn-moddle* [Krä23] extends *bpmn-moddle* [Cam23d], which implements the BPMN specification. Our extension adds the Token and ProcessSnapshot types from the BPMN execution metamodel shown in Figure 4 to the BPMN metamodel. Listing 1 shows an example BPMN XML snippet, where a token and process snapshot was added.

Listing 1: XML snippet showing the BPMN metamodel extension (simplified)

```
1 <process id="Process_1">
2   <extensionElements>
3     <bt:processSnapshot id="ProcessSnapshot_1" />
4     <bt:token id="Token_1" processSnapshot="ProcessSnapshot_1"
5         elementID="Task_A"/>
6   </extensionElements>
7   <task id="Task_A" />
8 </process>
```

The library allows one to create tokens and process snapshots and stores them in the BPMN extension elements (lines 2-7 in the XML example in Listing 1). This is the recommended way to extend the BPMN metamodel [Obj13].

The extension of the BPMN metamodel is realized by letting Token and ProcessSnapshot extend from Element, which is the type of elements of the extension elements, see Figure 25. Each Token points to the FlowElement it is currently positioned at, which can be an Activity or SequenceFlow, see BPMN metamodel in Figure 2. In Listing 1, this is realized using the attribute elementID.

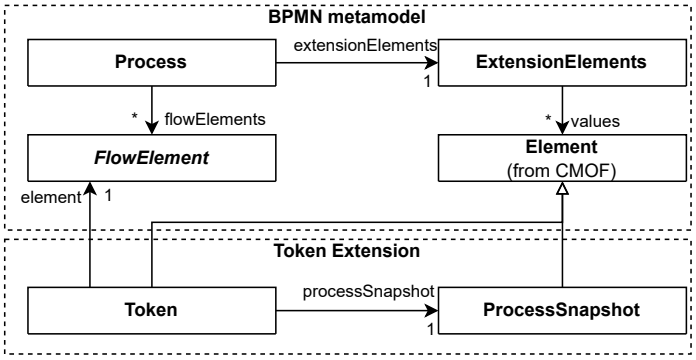


Figure 25: Token Extension of the BPMN metamodel (simplified)

[2] https://mvnrepository.com/artifact/io.github.timKraeuter/graph-rule-generation

The XML model in Listing 1 can be used by our HOT to generate atomic propositions for Groove (see Figure 1). However, following sound model-driven principles and creating an extended metamodel, the XML model could also be used in other applications or for model checking with different tools.

4.1.2. *Token Editor.* The Token Editor implements the concrete syntax for tokens and process snapshots described in Figure 4. Using our Token Editor, a user does not need to write XML, which hides the complexity of extending the BPMN metamodel in the graphical editor.

Figure 23 shows the Token Editor embedded in the second step of the BPMN Analyzer. We simplify the user interface so users can only edit tokens and process snapshots, not the underlying BPMN model. Process snapshots are automatically assigned distinct colors, and tokens held by a process snapshot have the same color (see concrete syntax in Figure 4). Tokens can be assigned to process snapshots, which changes the token's color to match the snapshot.

Our implementation is based on *bpmn-js* [Cam23b], which provides a BPMN rendering toolkit and uses the *token-bpmn-moddle* library described in the previous section to persist and load our models. Since both implementations are published as libraries, they can be easily reused in other applications.

4.1.3. *Graph rule generation.* The graph-rule-generation library offers various Java classes to generate graphs, GT rules, or entire GT systems, following the *builder pattern* [GHJV95]. Listing 2 shows an example code snippet to generate a GT rule using the GT rule builder implemented for Groove. One could also implement the GT rule builder for a different GT tool than Groove, which would only result in changes in the first two lines of Listing 2.

Listing 2: Code snippet to generate a GT rule using the GT rule builder

```
GraphTransformationRuleBuilder ruleBuilder
    = new GrooveRuleBuilder();
// Start a new GT rule with the name "sampleRule"
ruleBuilder.startRule("sampleRule");
// Create context nodes A and B and edge AB from A->B.
GraphNode a = ruleBuilder.contextNode("A");
GraphNode b = ruleBuilder.contextNode("B");
ruleBuilder.contextEdge("AB", a, b);
// Delete nodes C and D and edge CD from C->D.
GraphNode c = ruleBuilder.deleteNode("C");
GraphNode d = ruleBuilder.deleteNode("D");
ruleBuilder.deleteEdge("CD", c, d);
// Add nodes E and F and edge EF from E->F.
GraphNode e = ruleBuilder.addNode("E");
GraphNode f = ruleBuilder.addNode("F");
ruleBuilder.addEdge("EF", e, f);
// Create NAC nodes G and H and edge GH from G->H.
GraphNode g = ruleBuilder.nacNode("G");
GraphNode h = ruleBuilder.nacNode("H");
```

```
20 ruleBuilder.nacEdge("GH", g, h);
21 // Build the GT rule.
22 GraphTransformationRule gtRule = ruleBuilder.buildRule();
```

Using the rule builder, one can construct a GT rule by defining which nodes and edges should be present (lines 6-9), deleted (lines 10-13), added (lines 14-17), or NACs (lines 18-21), see Listing 2. Figure 26 shows the resulting GT rule specified by Listing 2 in Groove syntax.

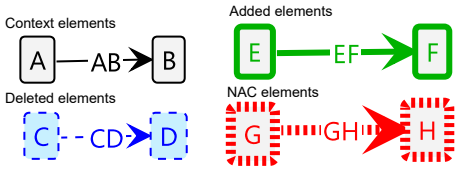


Figure 26: Groove GT rule generated by the code snippet in Listing 2

Similarly to the groove rule builder in Listing 2, we also provide a builder for constructing graphs to create start graphs of GT systems [Krä23]. Finally, the library provides a builder for GT systems using the graph and GT rule builders. It also automatically lays out graphs and GT rules using the Eclipse Layout Kernel (ELK). The Groove UI and the Groove command-line tools can then consume the generated GT systems.

4.2. **Performance testing.** Model checking is a valuable technique but can fall short when applied in the industry due to insufficient performance [CHVB18]. Inadequate performance might have many reasons, most notably large models leading to state space explosion. We assess the performance of our implementation for two sets of BPMN models. First, we pick ten BPMN models from the literature, including realistic ones. Second, we generated ten BPMN models with exponential state space growth to test how our tool deals with increasing complexity. We explain each benchmark and its results in the following subsections and provide all necessary information and artifacts to reproduce them in [Krä23]

To calculate the average runtime, we use the hyperfine benchmarking tool [Pet23] (version 1.18.0), which runs the HOT/state space exploration for each BPMN model ten or more times. The experiment was run on Windows 11 (AMD Ryzen 7700X processor, 32 GB RAM) using Groove version 6.1.0 [Krä23].

4.2.1. *BPMN models from the literature.* We randomly picked ten different BPMN models from [HBP+22] to assess the performance of our implementation. The models include realistic BPMN models (e.g., 001, 002, and 020) [HBP+22].

First, we ran our HOT for the BPMN models. The HOT takes approximately half a second to generate a GT system for each model. Thus, the generation of the GT systems for these models is fast enough. In addition, Table 1 states how many GT rules are generated for each BPMN model.

Second, we ran a full state exploration using the resulting ten GT systems, see Table 1 (runtime only includes state space exploration). The exploration takes around one second for most of the models. Only model *020* needs nearly two seconds due to its larger state

Table 1: Results for a full state space exploration of realistic models

| BPMN model | Processes | Nodes (gw.) | GT Rules | States | Transitions | Runtime |
|---|---|---|---|---|---|---|
| 001 | 2 | 17 (2) | 26 | 68 | 118 | $\sim$ 1.00 s |
| 002 | 2 | 16 (2) | 24 | 62 | 108 | $\sim$ 0.97 s |
| 007 | 1 | 8 (2) | 14 | 45 | 81 | $\sim$ 0.92 s |
| 008 | 1 | 11 (2) | 17 | 49 | 85 | $\sim$ 0.93 s |
| 009 | 1 | 12 (2) | 17 | 137 | 308 | $\sim$ 1.01 s |
| 010 | 1 | 15 (2) | 20 | 162 | 357 | $\sim$ 1.04 s |
| 011 | 1 | 15 (2) | 20 | 44 | 69 | $\sim$ 0.97 s |
| 015 | 1 | 14 (2) | 20 | 53 | 86 | $\sim$ 0.95 s |
| 016 | 1 | 14 (2) | 19 | 44 | 68 | $\sim$ 0.94 s |
| 020 | 1 | 39 (6) | 59 | 3060 | 8584 | $\sim$ 1.75 s |

space. Furthermore, up to one second is spent on startup, not model checking. For example, Groove reports only 722 ms for state space exploration for model *020*.

4.2.2. *BPMN models with increasing complexity.* To increase the state space complexity, we generate BPMN models with a growing number of parallel branches, similar to [CFP+21]. Figure 27 shows our schema to generate models. The possible interleavings of executing activities in parallel lead to an exponential increase in the state space. Concretely, we generated ten BPMN models with one to ten parallel branches containing one activity [Krä23]. We use these models to benchmark our implementation.

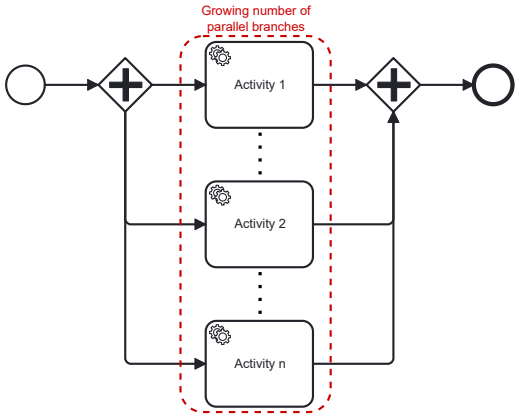


Figure 27: BPMN model generation with an increasing number of parallel branches

First, we ran our HOT for the BPMN models. The HOT takes less than a second to generate a GT system for each model. Thus, the generation of the GT systems for these models is fast enough.

Second, we ran a full state exploration using the resulting ten GT systems, see Table 2 (runtime only includes state space exploration). The models take one to nine seconds to explore. One can see an exponential increase in runtime due to the exponential increase in state space complexity.

Table 2: Results for a full state space exploration of models with increasing complexity

| Branches | Nodes (gw.) | GT Rules | States | Transitions | Runtime |
|---|---|---|---|---|---|
| 1 | 5 (2) | 9 | 7 | 7 | ∼ 0.87 s |
| 2 | 6 (2) | 11 | 13 | 17 | ∼ 0.86 s |
| 3 | 7 (2) | 13 | 31 | 59 | ∼ 0.88 s |
| 4 | 8 (2) | 15 | 85 | 221 | ∼ 0.95 s |
| 5 | 9 (2) | 17 | 247 | 815 | ∼ 1.03 s |
| 6 | 10 (2) | 19 | 733 | 2921 | ∼ 1.15 s |
| 7 | 11 (2) | 21 | 2119 | 10.211 | ∼ 1.49 s |
| 8 | 12 (2) | 23 | 6.565 | 34.997 | ∼ 2.13 s |
| 9 | 13 (2) | 25 | 19.687 | 118.103 | ∼ 3.85 s |
| 10 | 14 (2) | 27 | 59.053 | 393.665 | ∼ 9.16 s |

We conclude that our approach is sufficiently fast for models of average size and complexity. In the next section, we test the scalability of our approach when models increase in size. Furthermore, we discuss potential performance and scalability improvements. However, a comprehensive benchmark, including a detailed comparison to other tools, is left for future work.

4.3. **Scalability testing.** In this section, we test the scalability of our approach by applying it to 300 heterogeneous BPMN models with increasing model sizes.

4.3.1. *Setup.* We generated 300 BPMN models to test the scalability of our approach. We used the following strategy to include different BPMN elements in the models. We generated the models incrementally, increasing the number of *blocks* they contain. Thus, model one contains one block, model two contains two blocks, and so forth until the last model contains 300 blocks. A block is defined as one of the three BPMN model parts shown in Figure 28. During the generation, we alternate between the three different blocks.

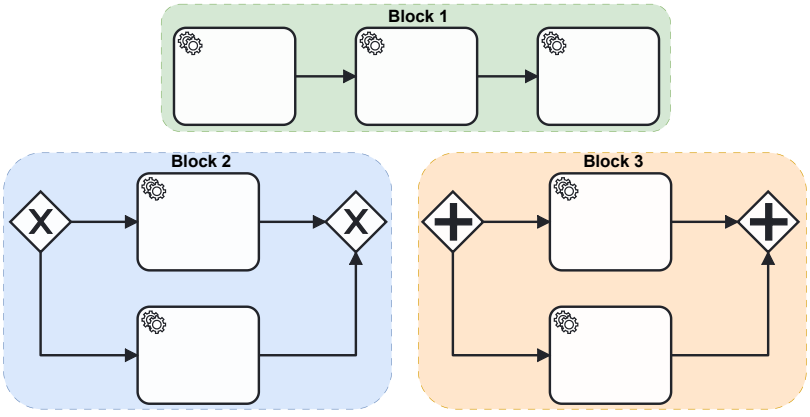


Figure 28: The three different blocks used for BPMN model generation

For example, the BPMN model with three blocks is depicted in Figure 29. Blocks two and three are shown in a new line for better visualization. However, the generated models

are expanding horizontally in one line. We then repeat adding one block at a time for each new model until we reach 300 models.

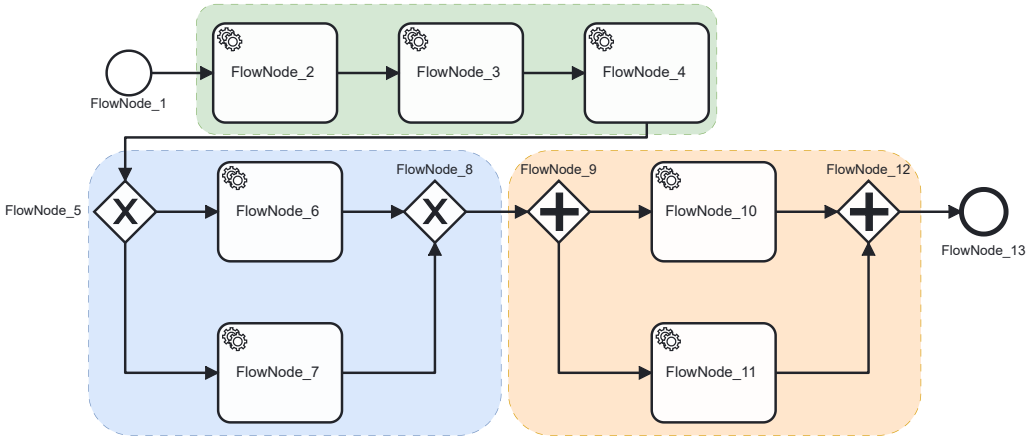


Figure 29: A generated BPMN model with three blocks

Table 3 states the characteristics of the generated BPMN models, such as the number of gateways, flow nodes, and sequence flows. One can deduce from the table that adding fifty blocks adds around 400 BPMN elements to a model. All models, including their characteristics and how to generate them, can be found in [Krä23]. Our BPMN model generation uses the camunda BPMN model API [Cam23f].

BPMN models in practice tend to be much smaller since large models are usually divided into smaller submodels [FFK+11], i.e., subprocesses, to ensure they are understandable by modelers. Each of these subprocesses can then be analyzed independently. From our experience and referring to other studies [FFK+11], this best practice leads to models with less than 400 total elements (comparable to less than 50 blocks in Table 3). We ran our scalability test for models with up to 300 blocks since we wanted enough data to see trends in the average runtime. We did not go beyond 300 blocks since the whole test should still run in a reasonable time.

Table 3: Characteristics of the generated BPMN models

| BPMN model / Blocks | Gateways | Flow nodes | Sequence flows | Total elements |
|---|---|---|---|---|
| 1 | 0 | 5 | 4 | 9 |
| 50 | 66 | 185 | 217 | 402 |
| 100 | 132 | 368 | 433 | 801 |
| 150 | 200 | 552 | 651 | 1203 |
| 200 | 266 | 735 | 867 | 1602 |
| 250 | 332 | 918 | 1083 | 2001 |
| 300 | 400 | 1102 | 1301 | 2403 |

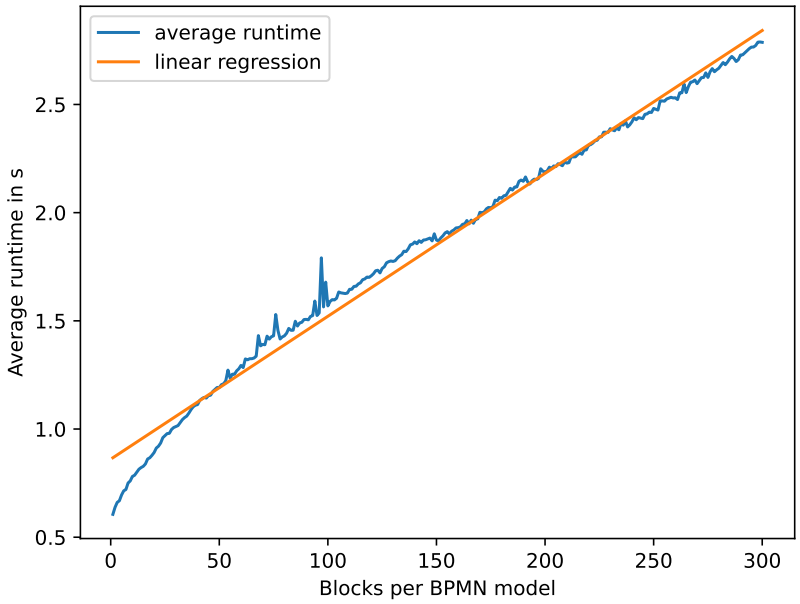


Figure 30: Scalability testing result of the GT system generation

4.3.2. *Results.* Figure 30 depicts the results of benchmarking our HOT with the generated BPMN models. It shows the average runtime of five runs for transforming each BPMN model into a GT system using our HOT. We used the same machine and setup as discussed for our performance experiments in section 4.

The average HOT runtimes data fits the linear regression shown in Figure 30 well. This makes sense since the HOT algorithm has *linear runtime complexity* because it iterates over all flow nodes of a BPMN model to generate GT rules. We conclude that the HOT is fast enough (around one second or less) for models of reasonable size (50 blocks or less).

Figure 31 depicts the results of benchmarking the state space generation in Groove for the GT systems obtained by our HOT. It shows the average runtime of five runs, calculated by hyperfine [Pet23].

The increase in the runtime of the state space generation looks worse than linear. However, models of reasonable size (50 blocks or less) are handled in less than two seconds. To summarize, using our approach, we can conservatively estimate that these models can be checked (HOT followed by a full state space generation) in around three seconds or less. In addition, full-state space exploration might not be needed if *on-the-fly* model checking is used [CHVB18].

*On-the-fly* model checking allows temporal properties to be checked incrementally while the state space is constructed. If a property violation is detected, there is no need to further complete the construction of the state space. This can considerably reduce the time and memory required for verification [CHVB18].

Plenty of optimization potential exists, starting from the HOT and ending with the state space generation in Groove. Currently, neither our HOT nor Groove are specifically optimized for performance with this use case in mind. Regarding the HOT, multiple

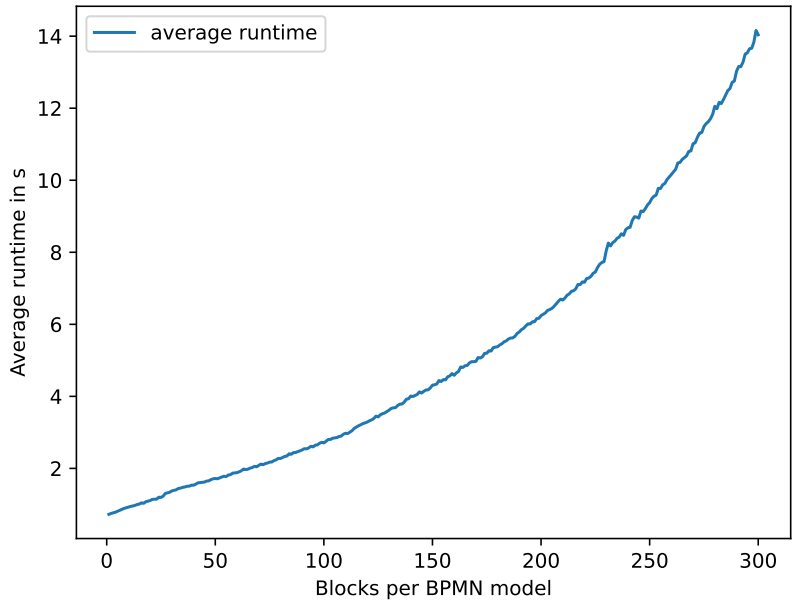


Figure 31: Scalability testing result of the state space generation in Groove

optimizations come to mind. First, one can parallelize the generation of GT rules since each rule is independent. Second, one can change the rule-generation templates to reduce the number of generated rules and the state space. For example, two rules are currently generated to represent starting and finishing an activity (see Figure 10), which fits the description in the BPMN specification. However, one could instead generate only one rule, which represents the whole execution of an activity. Thus, one less rule is generated, and the intermediate state representing the activity executing is no longer part of the state space. If there are many activities, especially when they are executed in parallel, this can lead to large reductions in the state space. However, a less granular state space could prohibit checking certain properties. A trade-off exists between staying close to the BPMN execution specification and overall runtime (HOT and state space generation).

Groove is a powerful tool with good out-of-the-box performance. However, there is still optimization potential. First, GT rules should not be written to and read from the disk to interact with Groove. Each GT rule is saved to a new file, and the number of generated GT rules increases with the size of a BPMN model. Integrating our HOT and Groove more tightly can eliminate these costly I/O operations since the generated rules can stay in the main memory. Second, *partial order reduction* methods could greatly reduce the time and space required for model checking [CHVB18]. Third, Groove could use *on-the-fly* model checking as mentioned by the Groove authors [KR06]. If one combines verification and state space exploration and finds a counterexample, there is no need to continue the state space generation [KR06, CHVB18].

## 5. Related work

The most common formalizations of BPMN execution semantics use Petri Nets. For example, [DDO08] formalizes a subset of BPMN elements by defining a mapping to Petri Nets conceptually close to our HOT-based formalization. Especially Petri net model checkers such as LoLA show great performance when analysing business process models [FFK+11] but supporting the same variety of BPMN elements can be problematic. Encoding basic BPMN modeling elements into Petri Nets is generally straightforward, but for some advanced elements, it can be complicated to define [HA02]. For example, representing *Termination End Events* and *Interrupting Boundary Events*, which interrupt a running process, is usually unsupported because of the complexity of managing the non-local propagation of tokens in Petri Nets [CFP+21]. We solve these situations by using nested graph conditions, for example, to remove all tokens when reaching a *Termination End Event*.

A BPMN formalization based on in-place GT rules is given in [VGD13]. 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 [VGD13] 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 [VGD13] since we process a concrete BPMN model before its execution. However, they do *not* support property checking since their goal is only to formalize the BPMN execution semantics.

The tool *BProVe* is based on formal BPMN semantics given in rewriting logic and implemented in the Maude system [CFP+21]. Using this formal semantics, *BProVe* can verify custom LTL properties and general BPMN properties, such as Safeness and Soundness. However, *BProVe* only supports the most common BPMN elements, as shown later. Regarding performance, [CFP+21] describes a timeout (runtime greater than 600 seconds) for state space generation of a model with five parallel branches. In subsection 4.2, we show that our tool takes less than ten seconds for the state space exploration of a model with ten parallel branches.

The verification framework fbpmn uses first-order logic to formalize and check BPMN models [HBP+22]. This formalization is then realized in the TLA+ formal language, which can be model-checked using TLC. TLC is an explicit state model checker for TLA+ specifications. Like BProVe, fbpmn allows checking general BPMN properties, such as Safeness and Soundness. Furthermore, *fbpmn* focuses on different communication models besides the standard in the BPMN specification and supports time-related constructs. In our approach, we currently disregard time-related constructs [DS17, HBP+22] and data flow [CMR+22, El-15] but rather support more BPMN elements. Regarding performance, [HBP+22] report 3.66-10.26s on a machine with less powerful hardware compared to our 1-1.75s for the models in section subsection 4.2. Thus, including the previous comparison to BProVe, we conclude that our tool performs well. However, assessing tool performance is difficult when there are no standardized benchmarks that allow for a direct comparison of results in the same environment.

Table 4 shows which BPMN elements are supported by our and the abovementioned approaches. Compared to the other approaches, we cover most BPMN elements. The coverage of BPMN elements significantly impacts how practical each approach is in checking properties in real life [FFK+11]. 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 [Cam23a].

The BPMN elements that our approach does not support, compared to Camunda, are transactions, cancel events, and compensation events. These elements are rather complex, but [VGD13] shows how cancel and compensation events can be formalized. We plan to support these elements by extending our implementation and test suite in the future.

## 6. Conclusion & future work

This article reports two main practical contributions. First, we conceptualize a new approach utilizing a Higher-Order model Transformation (HOT) to formalize the semantics of behavioral languages. Our approach moves complexity from the GT rules to the rule templates, which constitute the HOT. Furthermore, the approach can be applied to other behavioral languages as long as one can define the *state structure* and identify *state-changing elements* of the language.

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. In addition, our performance and scalability testing indicates that the tool can handle most BPMN models found in practice.

Future work targets both of our main contributions. First, we plan a detailed comparison of our HOT approach with approaches that utilize fixed model-independent 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.

## References

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Cam23a] Camunda Services GmbH. BPMN 2.0 Implementation Reference. https://docs.camunda.org/manual/7.19/reference/bpmn20/, October 2023.

[Cam23b] Camunda Services GmbH. Bpmn-js. https://github.com/bpmn-io/bpmn-js, October 2023.

[Cam23c] Camunda Services GmbH. Bpmn-js Token Simulation. https://github.com/bpmn-io/bpmn-js-token-simulation, October 2023.

[Cam23d] Camunda Services GmbH. Bpmn-moddle. https://github.com/bpmn-io/bpmn-moddle, October 2023.

[Cam23e] Camunda Services GmbH. Bpmnlint. https://github.com/bpmn-io/bpmnlint, October 2023.

[Cam23f] Camunda Services GmbH. Camunda BPMN model API. https://github.com/camunda/camunda-bpm-platform/tree/master/model-api/bpmn-model, October 2023.

[CFP+21] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, Francesco Tiezzi, and Andrea Vandin. A formal approach for the analysis of BPMN collaboration models. *Journal of Systems and Software*, 180:111007, October 2021. doi:10.1016/j.jss.2021.111007.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8.

Table 4: BPMN elements supported by different formalizations (based on [VGD13]).

| BPMN element/feature | Dijkman [DDO08] | Van Gorp [VGD13] | Corradini [CFP+21] | Houhou [HBP+22] | This article |
|---|---|---|---|---|---|
| *Instantiation and termination* | | | | | |
| Start event instantiation | X | 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 | X |
| *Activities* | | | | | |
| Activity | X | X | X | X | X |
| Loop activity | X | X | | | |
| Multiple instance activity | | | | | |
| Subprocess | X | X | | X | X |
| Event subprocess | | | | | X |
| Transaction | | | | | |
| Ad-hoc subprocesses | | | | | |
| *Gateways* | | | | | |
| Parallel gateway | X | X | X | X | X |
| Exclusive gateway | X | X | X | X | X |
| Inclusive gateway (split) | X | 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 | X |
| Message events | X | X | X | X | X |
| Timer Events | | | | X | |
| Escalation Events | | | | | X |
| Error Events | X | 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 |

[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.

[CMR+22] Flavio Corradini, Chiara Muzi, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. Formalising and animating multiple instances in BPMN collaborations. *Information Systems*, 103:101459, January 2022. doi:10.1016/j.is.2019.101459.

[CMRT18] Flavio Corradini, Chiara Muzi, Barbara Re, and Francesco Tiezzi. A Classification of BPMN Collaborations based on Safeness and Soundness Notions. *Electronic Proceedings in Theoretical Computer Science*, 276:37–52, August 2018. doi:10.4204/EPTCS.276.5.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, November 2008. doi:10.1016/j.infsof.2008.02.006.

[DS17] Francisco Durán and Gwen Salaün. Verifying Timed BPMN Processes Using Maude. In Jean-Marie Jacquet and Mieke Massink, editors, *Coordination Models and Languages*, volume 10319, pages 219–236. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-59746-1_12.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach.*, pages 247–312. World Scientific, February 1997. doi:10.1142/9789812384720_0004.

[El-15] Nissreen A. S. El-Saber. *CMMI-CM Compliance Checking of Formal BPMN Models Using Maude*. PhD thesis, University of Leicester, January 2015.

[FFK+11] Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data & Knowledge Engineering*, 70(5):448–466, May 2011. doi:10.1016/j.datak.2011.01.004.

[FR19] Jakob Freund and Bernd Rücker. *Real-Life BPMN: Using BPMN and DMN to Analyze, Improve, and Automate Processes in Your Company*. Camunda, Berlin, 4th edition, 2019.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[HA02] Arthur Hofstede and Wil Aalst. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In *Proceedings of Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560, August 2002.

[HBP+22] Sara Houhou, Souheib Baarir, Pascal Poizat, Philippe Quéinnec, and Laïd Kahloul. A First-Order Logic verification framework for communication-parametric and time-aware BPMN collaborations. *Information Systems*, 104:101765, February 2022. doi:10.1016/j.is.2021.101765.

[HT20] Reiko Heckel and Gabriele Taentzer. *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-43916-3.

[KKR+23] Tim Kräuter, Harald König, Adrian Rutle, Yngve Lamo, and Patrick Stünkel. Behavioral consistency in multi-modeling. *The Journal of Object Technology*, 22(2):2:1, 2023. doi:10.5381/jot.2023.22.2.a9.

[KR06] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In Antti Valmari, editor, *Model Checking Software*, volume 3925, pages 299–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi:10.1007/11691617_19.

[Krä23] Tim Kräuter. LMCS-2024: Artifacts. Zenodo, October 2023. doi:10.5281/ZENODO.10018457.

[KRKL23] Tim Kräuter, Adrian Rutle, Harald König, and Yngve Lamo. Formalization and Analysis of BPMN Using Graph Transformation Systems. In Maribel Fernández and Christopher M. Poskitt, editors, *Graph Transformation*, volume 13961, pages 204–222. Springer Nature Switzerland, Cham, 2023. doi:10.1007/978-3-031-36709-0_11.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, volume 8706, pages 1–20. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-11245-9_1.

[Men09] Jan Mendling. Empirical Studies in Process Model Verification. In Kurt Jensen and Wil M. P. Van Der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460, pages 208–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-00899-3_12.

[Obj13] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0.2. https://www.omg.org/spec/BPMN/, December 2013.

[Obj17] Object Management Group. Unified Modeling Language, Version 2.5.1. https://www.omg.org/spec/UML, December 2017.

[Pet23] David Peter. Hyperfine. https://github.com/sharkdp/hyperfine/, October 2023.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062, pages 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-25959-6_40.

[Ren06] Arend Rensink. Nested Quantification in Graph Transformation Rules. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 1–13, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11841883_1.

[Ren08] Arend Rensink. Explicit state model checking for graph grammars. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, pages 114–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-68679-8.

[Ren17] Arend Rensink. How Much Are Your Geraniums? Taking Graph Conditions Beyond First Order. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd*, volume 10500, pages 191–213. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-68270-9_10.

[Rüc21] Bernd Rücker. *Practical Process Automation: Orchestration and Integration in Microservices and Cloud Native Architectures*. O'Reilly, Beijing Boston Farnham Sebastopol Tokyo, first edition edition, 2021.

[SSHK15] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom*. Undergraduate Topics in Computer Science. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-12742-2.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562, pages 18–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-02674-4_3.

[VGD13] Pieter Van Gorp and Remco Dijkman. A visual token-based formalization of BPMN 2.0 based on in-place transformations. *Information and Software Technology*, 55(2):365–394, February 2013. doi:10.1016/j.infsof.2012.08.014.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany