A Lightweight Approach for Model Checking Variability-Based Graph Transformations

Downloaded from: https://research.chalmers.se, 2026-09-14 20:42 UTC

Citation for the original published paper (version of record):

Albers, M., Damasceno, C., Strüber, D. (2022). A Lightweight Approach for Model Checking Variability-Based Graph Transformations. GCM 2022 Graph Computation Models 13th International Workshop Proceedings

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

A Lightweight Approach for Model Checking Variability-Based Graph Transformations

Mitchell Albers
Radboud University
Nijmegen, The Netherlands
mitchell.albers@ru.nl

Carlos Diego N. Damasceno
Radboud University
Nijmegen, The Netherlands
d.damasceno@cs.ru.nl

Daniel Strüber
Chalmers | University of Gothenburg, SE
Radboud University Nijmegen, NL
danstru@chalmers.se

Graph transformation systems often contain large numbers of similar rules, leading to maintenance issues as well as performance bottlenecks during rule applications. Previous work introduced variability-based graph transformations as a paradigm for explicitly managing variability in rules, successfully addressing these issues. However, no previous work investigated whether variability-based graph transformations can also lead to benefits during the automated analysis of graph transformations, particularly during model checking, in which the main performance bottleneck is the combinatorial explosion arising during state space exploration.

In this paper, as an initial approach for model checking of variability-based graph transformations, we present an extension of an existing symbolic model checking technique. The existing technique, called Gryphon, converts the graph transformation system into a symbolic encoding and, from there, into the input format of a hardware model checker. We adapt Gryphon's encoding to incorporate information on variability, which reduces the size and complexity of the overall encoding since it is now derived from a smaller set of rules (some of them being variability-based rules that represent several similar rules). In a preliminary evaluation, we show that our extension leads to performance benefits in a standard model checking scenario.

1 Introduction

Model-Driven Engineering (MDE) is a software engineering paradigm that promotes the use of models and transformations as primary artefacts, allowing for the abstraction of non-essential aspects [23]. MDE promotes model transformations as a key enabling technique for automatically generating models to reduce errors and save implementation efforts. One of the main paradigms in model transformation is algebraic graph transformation (AGT [16]) which allows the specification of transformation rules in a high-level, declarative manner using model transformation languages. Sometimes, one wants to specify many similar transformation rules that have many commonalities but differ in some parts. In that case, these similarly-structured rules can be merged while preserving their structure by specifying variability points on diverging elements, introducing variability on transformation rules. Therefore, by expressing variability points on diverging graph elements in transformation rules, we can merge rules and keep the number of total rules minimal. Doing so, one obtains a new concept of graph transformation rules with variability, called *variability-based graph transformation* in previous work [41]. As a further benefit, one can improve the performance of such graph transformations by providing a variability-aware execution mode that can make the rule application procedure faster and more scalable [41].

When verifying the correctness of regular AGT rules, model checking has proven to be a powerful tool of increasing interest [36]. However, current state-of-the-art model checking tools do not support variability-based graph transformation. But why would it be desirable if they did so? Model checking tools currently do not consider variability, implying that each rule variant has to be made explicit in

© Albers, Damasceno & Strüber
This work is licensed under the
Creative Commons Attribution License.

a separate rule. From a model checking perspective, this leads to redundant computational effort when dealing with rules with many shared actions, resulting in a sub-optimal model checking approach in terms of runtime behaviour. A model checking approach that addresses variability as part of the model checking procedure without considering each rule variant as a separate rule could lead to runtime improvements.

To our knowledge, no previous work has addressed variability in model checking of graph transformation systems. Related work (discussed in Section 6) addresses variability in model checking for other formalisms, such as *labelled transition systems* [14, 13] and *Markov decision processes* [11].

This paper introduces an initial approach for model checking graph transformations with variability. We propose an extension of the open-source, lightweight symbolic model verification technique Gryphon [19]. Gryphon uses a symbolic encoding to represent graph transformation systems to bounded first-order relational logic. Gryphon assumes a bounded universe, implying that it does not support arbitrary creation and deletion of nodes. However, it does support negative application conditions (NACs) as well as arbitrary creation and deletion of edges. We study the working assumption that addressing variability explicitly, as part of the model checking procedure, may help to increase the performance of the model checker, compared to enumerating all rule variants explicitly and feeding them as input to the model checker. Thereby, we will be answering the following research question:

RQ: *Does the direct support for variability-based rules in Gryphon decrease the execution time for model checking of graph transformations of the kind supported by Gryphon?*

To answer this research question, we propose an extension of Gryphon's symbolic encoding that supports variability-based graph transformations. We further present an implementation of the resulting encoding, leading to the first implementation of a variability-based model checking approach for graph transformations. Like Gryphon, the resulting approach is restricted to a certain type of graph transformations: double-pushout graph transformations, restricted to rules that can create and delete edges and can have NACs. The considered transformation kind is expressive enough to support previous scenarios from the literature on model checking of graph transformations, such as *dining philosophers*, *interlocking railway systems*, and *pacman* (provided as examples in Gryphon's presentation [18, 19]). However, several other examples from the literature cannot be handled by our approach, such as *circular buffers* [35], *malaria surveillance* [8], *knowledge graph management* [9], and *health information system* [7] that require creation, deletion, cloning and/or merging of nodes.

Using the encoding, we answer the research question in an empirical performance evaluation. This evaluation compares the execution time of model checking with standard Gryphon and a Gryphon version using the modified encoding.

2 Preliminaries

Graph transformations

We consider the paradigm of algebraic graph transformations (AGT), in particular the formalization of graph morphisms and graph transformation rules based on the *double-pushout approach* [16]. A graph morphism $m : G \to H$ is a structure-preserving mapping between two graphs, in the sense that edges are mapped in a way that respects their source and target. A transformation rule t is defined as $t = (L \xleftarrow{l} I \xrightarrow{r} R,\ Nac)$, consisting of the following elements: a left-hand side graph L and a right-hand side graph R, along with two graph morphisms l and r, both being inclusions, an interface graph I, satisfying $I \subseteq L$ and $I \subseteq R$, and a set Nac (negative application conditions) of graph morphisms of the

form $n : L \to N$. In the rest of this paper, we consider a restricted notion of rules in which the node sets of L, I, and R are identical.

Rules are applied to a given host graph in order to obtain a result graph. The application of a transformation rule t on a given start graph G assumes an available injective graph morphism $m : L \to G$ that matches L to G. To support node deletions in a consistent way, m needs to fulfill a *dangling edge condition*: if a node n is to be deleted by the rule application, the rule application has to delete all adjacent edges of the node as well. For the restricted kind of rules we consider (which cannot delete or create nodes), this condition is always fulfilled. Furthermore, m needs to satisfy the NACs. Intuitively, each NAC graph N is interpreted as a pattern whose existence in the host graph is forbidden. More formally, it has to be checked that for none of the NACs there exists morphism $n' : N \to G$ s.t. $m = n' \circ n$. If m fulfills these conditions, it is called a match.

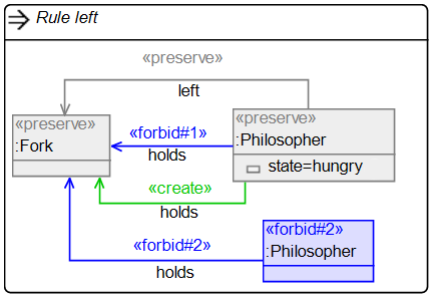
Given a match, a rule application is executed as follows: First, for each graph element that is part of L and that is not part of the interface graph I, the element identified by m is deleted. Second, for each graph element that is part of R and that is not part of interface graph I, a graph element is created; new graph elements are "glued" to existing ones as specified by m and r.

The graphs supported by our technique can have types and attributes. We omit providing details of the formalization of these concepts; the interested reader is referred to Ehrig et al.'s [16] formalization.

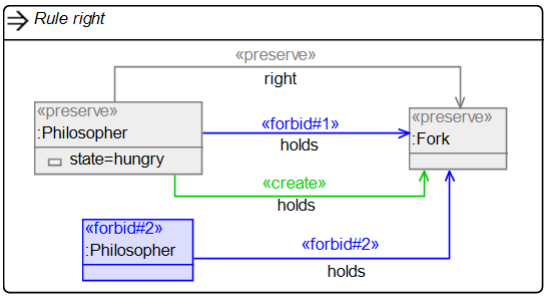
Example. We consider the Dining Philosopher's problem as presented in [19], which was originally designed to illustrate challenges concerning deadlocks, making it a well-suited problem for model checking. The Dining Philosopher's problem starts out with a group of n philosophers around a table on which there are n plates and n forks (i.e., one fork on the left, and one fork on the right of each philosopher). A philosopher can either be *hungry*, *thinking*, or *eating*. Whenever a philosopher transitions from *thinking* to *hungry*, it wants to eat. In order for a philosopher to be able to eat, it is required to have 2 forks assigned to that philosopher. Whenever a philosopher is done with eating, it releases both forks so that another philosopher can take them. Figure 1 captures a standard specification of the Dining Philosopher's problem in terms of graph transformation rules, inspired by [19]. In this example, the following five graph transformation rules are considered:

- Rule *hungry* transitions a philosopher's state from *thinking* to *hungry*.
- Rule *left* leads to a philosopher picking up their left fork (indicated by *holds*), provided that no philosopher (including themselves) already holds that fork.
- Rule *right* leads to a philosopher picking up their right fork (indicated by *holds*), provided that no philosopher (including themselves) already holds that fork.
- Whenever a philosopher is assigned to both a left and right fork, they can start eating. This behaviour is specified in rule *eating*.
- Whenever a philosopher is done eating, they release both forks, which is specified in rule *release*.

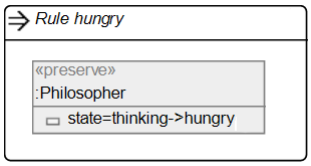
Note that these transformation rules follow an integrated syntax with respect to the formal definition; graph elements are annotated with labels. The labels *delete*, *preserve*, and *create* represent the sets $L \setminus I$, I, and $R \setminus I$, respectively. The label *forbid*, together with a hash symbol and the index of a NAC, represents the set $N \setminus L$ for the given NAC. In the example, rule *left* has two NACs, the first of them (*forbid#1*) ensuring that philosophers cannot pick up a fork that they already hold.



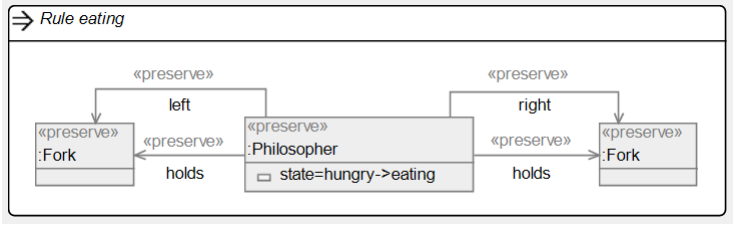
(a) Rule *left*



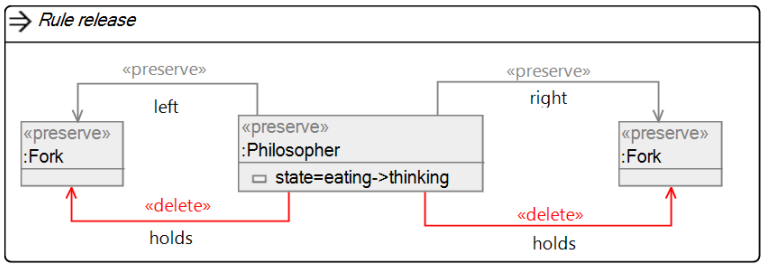
(b) Rule *right*



(c) Rule *hungry*



(d) Rule *eating*



(e) Rule *release*

Figure 1: Graph transformation rules of the Dining Philosopher's problem

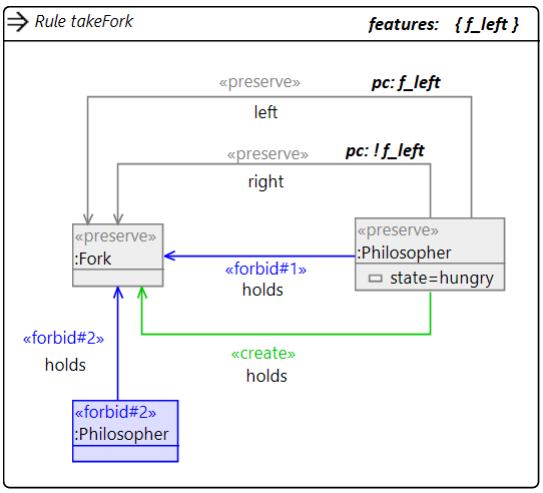


Figure 2: Variability-based rule *takeFork*

Variability-Based Graph Transformations

Variability-based graph transformations are an extension of standard graph transformations, which allows to express several similar rules in a "single-copy" representation. Individual, diverging elements (in the scope of this paper, edges) that are specific to some, but not all of the rules are annotated with *presence conditions*. A presence conditions is a boolean formula over a given set of features $\mathscr{F}$ (a.k.a. variation points), specifying a condition under which an annotated element is present. Given a variability-based rule, flat *rules* can be derived by binding each feature from $\mathscr{F}$ to either *true* or *false*, and removing those elements whose presence condition evaluates to *false*.

A variability-based transformation rule $\hat{t}$ is formalized as $\hat{t} = (t,\ \mathscr{F},\ pc)$, where t is a standard graph transformation rule (the "maximal rule" with all elements), and $pc : (L_t \cup R_t) \rightarrow Bool(\mathscr{F})$ is a *presence condition* function which maps rule elements to a propositional formula over the set of features $\mathscr{F}$ (defaulting to *true*, unless explicitly stated). To apply a variability-based graph transformation rule, intuitively, the first step is to *configure* it, that is, bind each of its features to either *true* or *false*, and remove all elements from t whose presence conditions evaluate to false. Configuring a variability-based graph transformation rule that way yields a flat rule that can be applied in the standard way (described above). The configuration can be either done manually by the user, or automatically by the configuration engine, where the latter leads to the notion of a variability-aware execution engine described in [41, 43]. Editing variability-based rules in a user-friendly way is supported by a dedicated extension of Henshin [42].
Example. When taking a closer look at rules *left* and *right* from Figure 1, we observe that these rules share a similar structure with an exception on the edges that are annotated with '*left*' and '*right*'. Therefore, we can merge these similar-structured rules and introduce a variability-based rule $takeFork = (t,\ \mathscr{F},\ pc)$ to represent the designation of forks to a philosopher. Transformation rule t consists of all the rule elements from rule *left* and *right*, such that it is "maximal" in the sense that it contains all rule elements. We can then annotate the diverging elements from the other rules with presence conditions over the set of features $\mathscr{F} = \{f_{left}\}$. Thereby, we annotate edges *left* and *right* with presence conditions $pc(left) = f_{left}$ and $pc(right) = \neg f_{left}$, respectively as shown in Figure 2. To be able to produce the original rules, both presence conditions may not evaluate to false or true at the same time—which is the case here, because they are mutual exclusive. For more complicated examples, one would need a *configuration constraint* to avoid illegal configurations (a.k.a. feature selections). Nonetheless, the concept of configuration constraints is out of scope of this paper.

Gryphon

The verification technique Gryphon [19] provides a model checking approach for model-driven software systems, whose static structure is defined by models built with the Eclipse Modeling Framework and whose behaviour is defined by graph transformation rules built with the Henshin API [2, 39]. From this static structure and behaviour models, Gryphon constructs a relational transition system (an encoding of a transition system using first-order logic formulas) which then can be checked by hardware model checkers to verify both safety and reachability properties. Gryphon does not support for the specification of custom LTL/CTL properties, but considers only one CTL reachability property in the form of $\exists\Diamond\phi$, were ϕ is a graph state to be reached. By duality, we may negate this formula and check for the unreachability or safety of ϕ, which can be expressed by $\forall\Box\neg\phi$.

Gryphon uses a symbolic encoding to encode an input graph, along with graph transformation rules, into relational formulas (i.e., first-order logic) that describe the model transformations. In essence, the encoding consists of two steps.

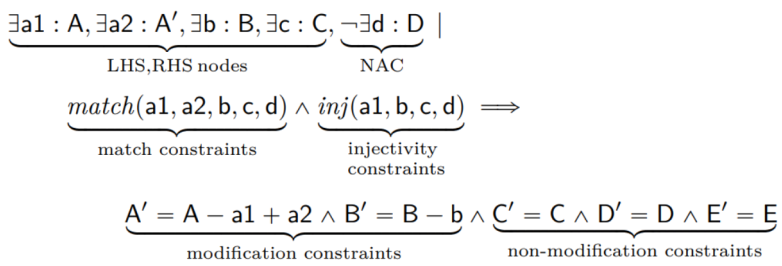


Figure 3: Scheme of a relational formula produced from a graph transformation rule (based on [19])

First, relational variables are generated. This entails assumes that a type graph $G_T = (V_T,\ E_T)$ is given with nodes V_T representing node types and edges E_T representing edge types. The creation of these relational variables is done by using the function $relgen : V_T \cup E_T \to Rel$ which generates for each node type a unary relational variable, and for each edge type a binary relational variable. The unary relational variables are included as atoms in a fixed universe, which consists of a sequence of uninterpreted atoms $\mathbb{A}$. This universe is initially derived from the initial model, such that for every object in the initial model, there exists a corresponding atom in the universe. The Gryphon technique considers a bounded, first-order logic, meaning that each relational variable is assigned an upper bound, and optionally a lower bound. Bounds are specified over the set of atoms in the universe [19]. In order to assign upper and lower bounds to unary relational variables, we use the function $\sqcup : Rel \to \mathscr{P}(\mathbb{A})$; upper bounds of binary relational variables arise from the product of edge source's and target's upper bounds.

Then, from these relational variables, graph transformation rules are translated into first-order, relational formulas. These relational formulas are generated by deriving a formula[1]

$$\mathrm{F}_t := Pre(L, Nac, R) \Longrightarrow Post(L, R)$$

for each graph transformation rule $t : (L \leftarrow I \rightarrow R,\ Nac)$. The function $Pre : G \times G \times G \times G \to \mathbb{F}$ takes a quadruple of graphs and produces a conjunction of relational formulas that mimic the match conditions of the transformation's left-hand side and NACs. Moreover, *Pre* also takes injectivity constraints into account. The function $Post : G \times G \to \mathbb{F}$ takes a pair of graphs and produces a conjunction of relational formulas that mimic how the rule application changes the assumed host graph via deletion and addition. The resulting formulas can be modification or non-modification constraints, in the sense that specify what needs to be changed in the host graph or what needs to remain unchanged. The formula F_t can then be expressed by the scheme shown in Figure 3, which is based on an example with the relational variables $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$. A more comprehensive introduction of the relational encoding is found in [19].

Internally, Gryphon constructs the first-order relational formula from Figure 3 by using the KODKOD API [30], which is then used to translate it to a propositional formula (exploiting the assumption that the universe is bounded, which makes this translation possible). After that, this propositional formula is rewritten into an and-inverter graph (AIG)—a boolean circuit consisting of only 'and' and 'not' gates—and stored in the AIGER format (see Figure 4, [19]). This step is aimed to foster interoperability as several model checkers can read this format. For the scope of this research, we consider the Incremental Inductive model checker (IIMC, [24]). Properties are specified at the graph level, as Henshin "check rules" (non-modifying rules), and translated to the same format.

[1]This formula originates from Gryphon's encoding [19], which, in addition, considers positive application conditions–out of the scope of this research.

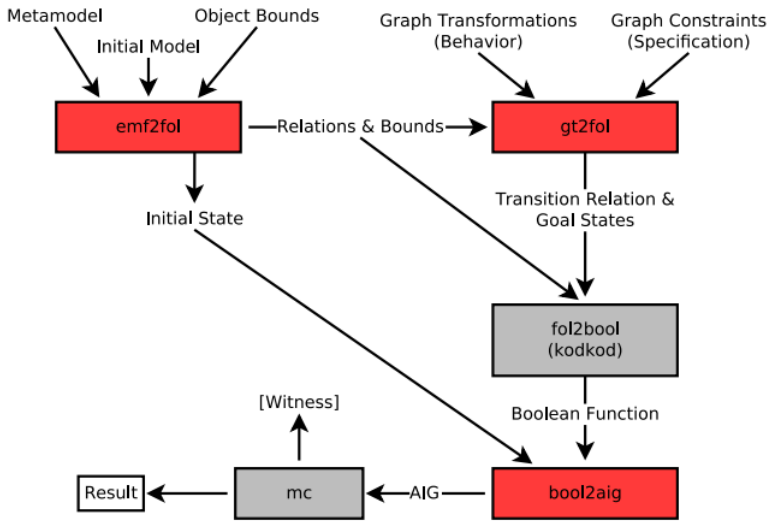


Figure 4: Gryphon's workflow architecture (from [19]). Grayed components represent external tools

3 Motivating Example

In this section, we will give a motivating example to illustrate how model checking of graph transformations with variability can be improved by explicitly using variability-related information. For this example, we consider the Dining Philosopher's problem with variability as presented in Section 2.

For model checking, we consider a very simple reachability property in which we want to check if it is possible whether a philosopher can be in the state *eating*, expressed as the following CTL property: $\exists \Diamond eating$. When considering the variability-based rule *takeFork* from Figure 2 in which we have specified presence conditions $pc(left) = f_{left}$ and $pc(right) = \neg f_{left}$, we observe that this can be easily done by applying the rule variants for picking up the left and the right fork. This can be achieved by applying rule *takeFork* twice; once in which f_{left} is set to true, and once by applying *takeFork* in which f_{left} is set to false. This evaluates the corresponding presence conditions to true, making it possible for a philosopher to take the corresponding fork(s). Consequently, given this information, the model checker is able to conclude that it is now possible for a philosopher to eat as both forks can be assigned to it, which means that rule *eating* can be applied, satisfying the reachability property. However, when taking a closer look at the rule *takeFork*, we observe that this rule tries to match a single fork as a left and a right fork to a philosopher. Therefore, we consider presence conditions to preserve the semantics relative to having two separate rules (i.e., one for picking a left fork and one for a right fork). These considerations would lead to a *variability-aware model checking algorithm* that explicitly addresses the rule variants expressed by a variability-based rule as part of its internal workings.

Without dedicated support for variability-based rules as part of the model checking algorithm, one could still supporting *takeFork*, by enumerating all possible configurations for the features $\mathscr{F}$, and creating a rule for each of them, and feeding the rules as input to a standard model checker. In general, this leads to an exponential increase in the number of rules, depending on the number of features and presence conditions that are annotated inside the rules. Feeding more rules with lots of redundant information into the model checker is likely to make the model checking procedure less efficient.

Therefore, by instead using the information contained in the variability-based rules explicitly during

the model checking procedure, we expect to obtain efficient support for variability in graph transformation rules, as we can keep the number of rules minimal and avoid expressing information redundantly.

4 Variability Encoding

Gryphon's encoding encompasses the use of relational formulas to mimic the behaviour of graph transformations. This section describes an extension of this encoding to support graph transformation rules with variability.

Recall from Section 2 that a variability-based graph transformation rule consists of a "maximal rule" t, a presence condition function pc, and a set of features $\mathscr{F}$ that can be used to annotate graph elements with presence conditions in the form of a boolean formula over the set of features $\mathscr{F}$. The intention of annotations on a graph transformation rule with presence conditions is to specify an additional constraint on the rule's matching behaviour. Thereby, the evaluation of a rule's set of presence conditions, determines which rule variant is applied.

To specify this variability-based matching behaviour within Gryphon's encoding, we first introduce a relational variable for each feature. Similarly to the function *relgen* as described in Section 2, we introduce the function $f_{relgen} : \mathscr{F} \rightarrow Rel$, which maps features to unary relational variables. These relational variables are then also included in the set of atoms $\mathbb{A}$. This step is required for constructing relational formulas, in which we can refer to the atoms in the universe that correspond to the features as relational variables.

Recall from Section 2 that relational variables must be bounded. Since feature variables are used to construct presence conditions in the form of propositional formulas, they are upper bounded by boolean values (i.e., true and false). As we are now able to generate and bound relational feature variables from graph annotations, we consider the function $pc_{relgen} : Bool(\mathscr{F}) \rightarrow Bool(Rel)$ that maps a boolean formula of features to a relational formula, representing a presence condition.

Gryphon's encoding as discussed in Section 2, considers the formula F_t, following the formula generation scheme as exemplified in Figure 3. This formula uses the function *Pre* to create a relational formula that mimics the matching conditions of the left-hand side of a graph transformation rule t. The function *Pre* consists of an explicit formula *match* that contains the matching constraint of a transformation rule. Since we are interested in embedding the presence conditions into the matching constraints, we imply that we want our presence condition to be a dependent variable of this matching. More concretely, we want to make the matching constraints of graph elements dependent on these presence conditions such that: whenever the presence conditions can be met, then the actual matching can also be done. This can be easily specified as an implication between the presence condition and the matching constraints of the corresponding graph element. For the scope of this research, we only consider edges to be annotated with presence conditions. Therefore, we translate every edge e with $src(e) = c$ and $trg(e) = d$ to the following matching constraint: $(c \rightarrow d) \subseteq C_e$, where C_e is a binary relational variable that represents the edge. However, whenever an edge annotated with a presence condition, we want to embed this information in this edge. Therefore, we add our presence condition pc to the matching constraint of this edge as follows: $pc \rightarrow ((c \rightarrow d) \subseteq C_e)$. By doing so, the matching constraint is dependent on the presence condition; if the presence condition is met, then this reference can be matched.

We argue for the soundness of our encoding informally, leaving a rigorous soundness proof to future work: In [41], we introduced a notion of *variability-based rule applications*, which supports the application of a variability-based rule to a graph. We showed the soundness of variability-based rule application, i.e., its equivalence to applying each of the rule variants expressed by the variability-based rule

to the graph, using traditional rule applications. We argue that Gryphon's standard encoding captures the notion of traditional rule application, whereas our modified encoding captures variability-based rule application. More specifically, including presence conditions of edges into the encoding as we do allows *variability-based matches* to be identified. Hence, we can benefit from our earlier soundness proof.

Example. Consider the dining philosopher problem with variability-based rules, as discussed in the motivating example in Section 3. In this example, we consider $\mathscr{F} = \{f_{left}\}$ as the set of features that are present in rule *takeFork*. These features are also added as uninterpreted atoms in the universe. From this, we can construct the following relational feature variable $F_{left} = f_{relgen}(f_{left})$ and bind it by $\sqcup(F_{left}) = \{true,\ false\}$. In this case, rule *takeFork* is annotated with f_{left} on edges *left* and *right*. Therefore, we have presence conditions $pc_{relgen}(pc(left)) = pc_{relgen}(f_{left}) = F_{left}$ and $pc_{relgen}(pc(right)) = pc_{relgen}(\neg f_{left}) = \neg F_{left}$ for edge *left* and *right*, respectively. The references for edge $left$ and *right* are translated as the following matching constraints:

$$F_{left} \rightarrow \big((p \rightarrow f) \subseteq Phil_{left}\big),\ src(left) = p \in Phil,\ trg(left) = f \in Fork$$

$$\neg F_{left} \rightarrow \big((p \rightarrow f) \subseteq Phil_{right}\big),\ src(right) = p \in Phil,\ trg(right) = f \in Fork$$

Note that *Phil* corresponds to the class '*Philosopher*', and *Fork* to the class '*Fork*' in the transformation rule *takeFork*, where p is a philosopher object and f is a fork object in the matching host graph.

5 Evaluation

This section concerns an evaluation in the form of a runtime comparison between the model checking procedure with variability encoding and the standard Gryphon approach without variability-based graph transformation rules. For this evaluation, we again consider the dining philosopher's problem as described in Section 2. Thereby, we consider the system of transformation rules from Figure 1, consisting of rules: *left*, *right eating*, *hungry*, *release*. For fair comparison, we compare this system of transformation rules with the same system, but instead of using the rules *left* and *right*, we consider the variability-based rule *takeFork* from Figure 2. Moreover, we consider one single reachability property that checks whether it is possible for a philosopher to be in an *eating* state (i.e., $\exists\Diamond eating$). We reuse the provided specification of this property from the original Gryphon implementation of the case, using a non-changing rule to specify the graph pattern part of the constraint.

For this runtime comparison, we are interested in the following two metrics: (1) the time of the actual model checking to conclude the property and (2) the total execution time of the tool (i.e., including rule translation, etc.). We can then gain insight into how much time was spent on solving and how much time was spent on translating the models. During this comparison, we consider five different input graphs, consisting of 10, 20, 30, 40, and 50 philosophers. For accuracy, we have executed the model checking tool several times over each input model and for each model type; 30 times with variability-based graph transformation rules (VAR), and 30 times without variability-based rules (STD).

The runtime comparison between both model checking approaches is depicted in Figures 5, 6, and 7, covering total runtime, solving time, and translation time, respectively. From these figures, we generally observe that model checking with support for variability outperforms model checking without that support on larger input models in terms of solving time, but not on small input models. For translation time, we observe that a variability-based rules increases performance for every input model. Moreover, the main contribution to total runtime comes from solving, whereas rule translation affects total runtime performances non-significantly.

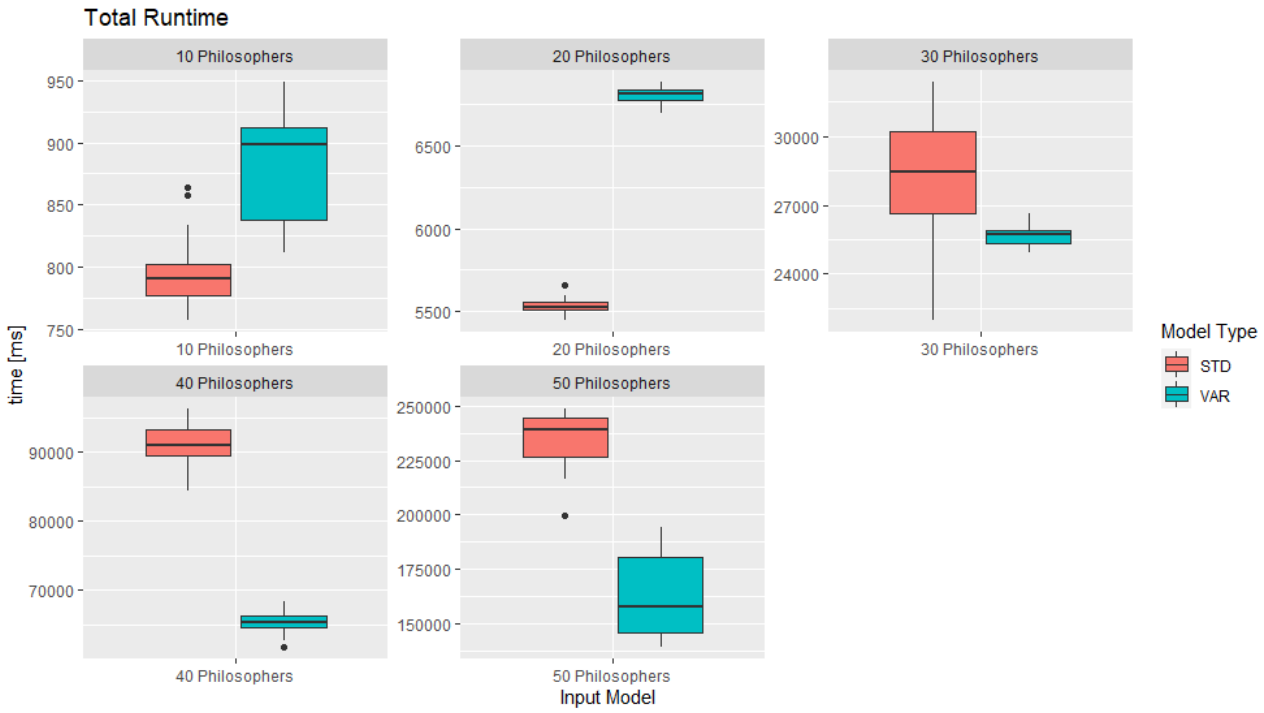


Figure 5: Total runtime for standard (STD) and variability-aware (VAR) model checking.

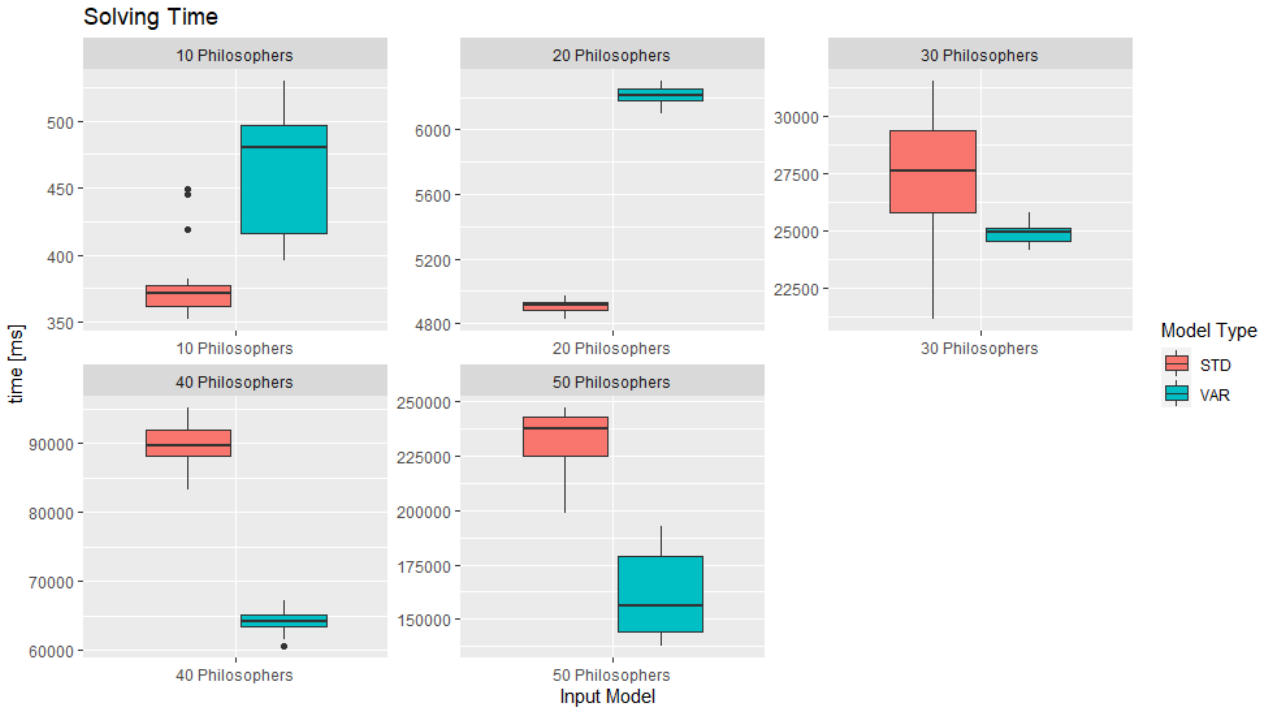


Figure 6: Solving time for standard (STD) and variability-aware (VAR) model checking.

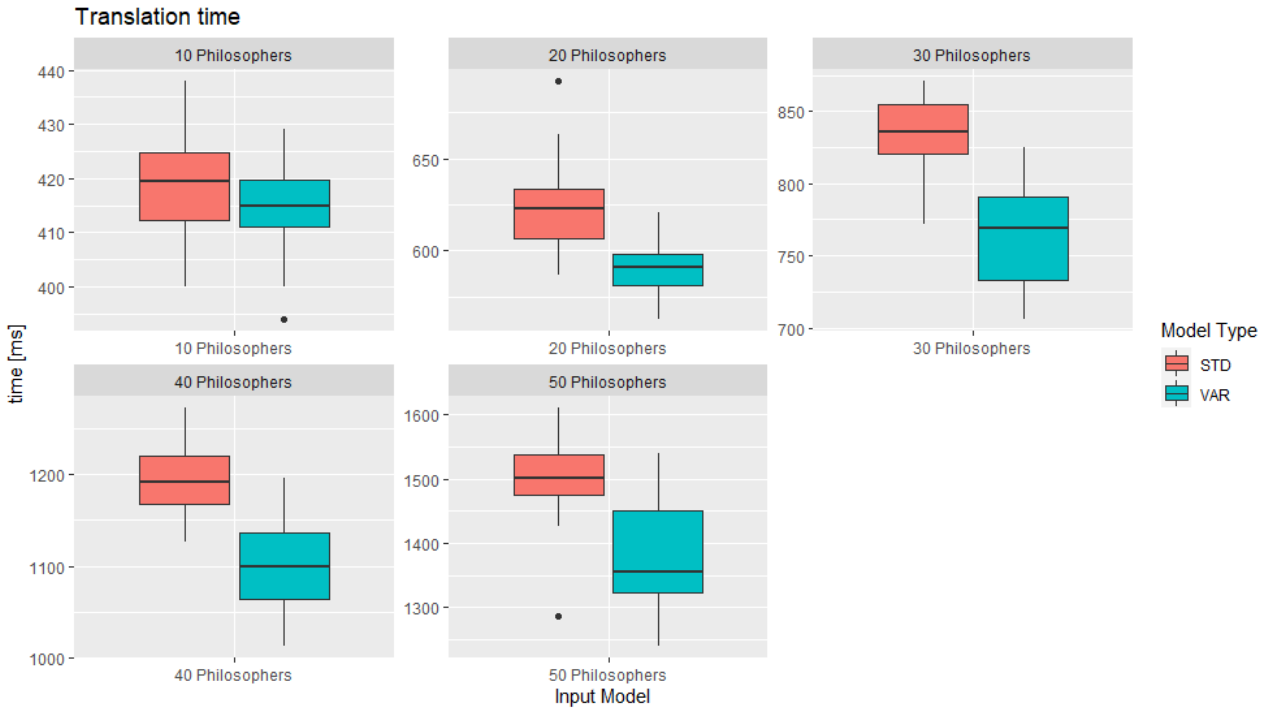


Figure 7: Translation time for standard (STD) and variability-aware (VAR) model checking.

When taking a closer look at the metrics of our runtime comparison from Figures 5, 6, and 7, we consider the mean μ and standard deviation σ to give insightful information regarding our obtained evaluation result (see Table 1). Note that lines that are written in bold represent the metrics that had the best overall performance for a given model type (i.e., STD or VAR) and the corresponding input model. Based on the mean average total runtime, we observe a slowdown of 11% on VAR relative to STD for an input model of 10 philosophers. This slowdown increases to 23% for an input model that considers 20 philosophers. At 30 philosophers, we start to observe a turning point as we see a speedup of 11%. This speedup increases substantially for an input model that considers 40 philosophers towards 39% and, for an input model with 50 philosophers, to 45%.

Regarding our research question **RQ**, we can conclude that variability-aware model checking can significantly improve performance, up to a speedup of 45%, even for a seemingly simple case with only two similar rules that were represented as one variability-based rule. These savings can be explained from the specifics of the case that, however, might be transfer to other similar cases as well: The variability is included in the most complicated (because NAC-including) pair of rules (i.e., by merging rules *left* and *right* into *takeFork*), which also leads to a particular complicated encoding of these rules as part of the overall encoding. By representing these two rules using one variability-based rule, we get a shorter and less complex overall encoding.

While a formal performance argumentation based on the encoding's complexity would need to address the internal workings of the underlying hardware model checker, which is outside our scope, we empirically observe that the less complex encoding leads to performance improvements for larger instances. Most of the observed savings are made in the *solving* time, which, especially for larger instances, is the computational bottleneck of the overall process. We expect these observed savings to increase even

		Total Runtime (ms)		*Solving Time (ms)*		*Translation time (ms)*	
Input Model	*Model Type*	μ	σ	μ	σ	μ	σ
10 Philosophers	**STD**	**795**	**25**	**371**	**23**	**385**	**10**
	VAR	879	43	477	43	376	8
20 Philosophers	**STD**	**5534**	**44**	**4901**	**35**	**539**	**23**
	VAR	6803	50	6691	54	489	15
30 Philosophers	STD	28385	2423	27396	2426	686	25
	VAR	**25662**	**459**	**24368**	**451**	**654**	**34**
40 Philosophers	STD	91031	2653	89880	2646	939	35
	VAR	**65291**	**1472**	**59473**	**1472**	**812**	**48**
50 Philosophers	STD	234842	11868	232038	11830	1377	63
	VAR	**162486**	**18771**	**142951**	**18774**	**1213**	**80**

Table 1: Performance evaluation results: runtime comparison between STD and VAR.

more for cases with a greater amount of variability.

6 Related Work

Studies related to ours are on model checking for software product lines [46], in particular product-based [29, 28, 15, 1] and family-based analysis [22, 12, 6]; and model checking [25, 37, 36, 35] and analysis [21, 26, 34] of graph transformations.

Product-Based Model Checking. In product-based model checking, product-specific models of a product line are generated as separate entities and individually verified, each using a standard verification technique which may be optimized or not. In unoptimized approaches, product variants are verified each time they are derived [29]. To improve scalability and reduce redundant computations, model checking techniques can be optimized for incremental verification [46] with richer notions of features, e.g. *conservative features* that do not remove behaviour [15]; or aspects, e.g., spectative, regulative, and invasive [28]. Sampling-based model checking aims at reducing the verification problem by selecting a subset of valid products [47]. They are likely to find defects quickly, but may miss defects due to its incompleteness [1]. Instead, our technique for model checking graph transformations is family-based.

Family-Based Model Checking. The main problem with product-based analyses are redundant computations over shared assets. Then, to achieve a more efficient verification, family-based model checking incorporates domain artifacts, such as feature models, to analyze a family model with respect to the variability model and one or more properties. For a given property, a family-based model checker analyzes whether the property is fulfilled by all products. If not, the model checker provides a propositional formula specifying those products that violate the property [22]. As standard model checking, their family-based variants can operate directly on source code or on an abstraction of a system [6], such as featured transition systems [12]. Our work differs from the aforementioned literature on family-based analysis by operating on graph transformations as our abstraction.

Model Checking Graph Transformations. The basics of verifying graph transformation systems by model checking have been studied thoroughly by Heckel in [25]. More recently, approaches for model checking graph transformation systems have been also extended to more complex scenarios, such as compositionality, and probabilistic, timed behaviour [33, 34]. Typically, graphs can be interpreted as states and rule applications as transitions in a transition system [36]. From a graph transformation perspective,

there are two main approaches: CheckVML [37] and Groove [35]. The main idea of CheckVML [37] is to exploit off-the-shelf model checker tools, like SPIN [5], by translating a graph transformation system and property graphs into their Promela and temporal logic equivalents to carry out the formal analysis. In contrast, in the GROOVE approach [35], the core concepts of graphs and graph transformations are used all the way through model checking by explicitly representing and storing states as graphs, and transitions as applications of transformation rules. Also, since properties are specified in a graph-based logic, the theory and tool support of standard model checkers [3] may not be applied immediately and graph-specific model checking algorithms should be developed. We contribute the first approach to address variability in the model checking of graph transformations.

Analyzing Graph Transformations. In addition to model checking, there are various other techniques to analyze graph transformation systems. For an overview in this topic, we refer the interested reader to [26]. According to Heckel and Taentzer [26], techniques to analyze graph transformation systems may address *conflict and dependency analysis* – to determine the possibility of conflicts or dependencies between rules, *termination analysis* – to establish the absence of infinite transformation sequences, *constraint verification and enforcement* – to derive and check weakest preconditions, and *graph parsing* – to construct a derivation for a graph based on grammar rules. One of the severe issues in analyzing graph transformations is that graphs are typically specified monolithically and, for large models, it can quickly undermine the advantage of visualisation, and lead to state space explosion [21]. To address this issue, the analysis of graph transformation systems can also be enriched with notions of compositionality [21, 34]. Our work addresses this problem by matching and merging similar-structured transformation rules and annotating their divergences with presence conditions. Within the scope of infinite-state or potentially-large systems, description logic [8], symbolic execution [38], and over/under-approximation [4, 20] contribute to the design of push-button technologies for verifying graph transformation systems in a timely manner. Our technique is suitable for cases where the explicit representation of states is practical, in which our preliminary evaluation shows reduced computational effort. It can be seen as a complement to verification techniques suitable for infinite-state spaces [27].

7 Conclusion

In this paper, we have presented a variability encoding as an extension of the lightweight symbolic model checking technique Gryphon. We have introduced a minimal model checking approach that can use variability information when it is explicitly encoded in graph transformations rules, introducing an initial approach to model checking of variability-based graph transformations. This variability encoding tries to mimic the matching behaviour of multiple rule variants. Moreover, we have given a motivating example that shows how including features as part of the model-checking procedure can result to increase performance relative to enumerating all rule variants of a given transformation rule.

Considering our research question, we have empirically evaluated the resulting model checking approach for variability-based graph transformations. This evaluation was performed by a runtime comparison with a standard model checking approach without variability-based rules (see Table 1). From this evaluation, we observe moderate improvements in terms of performance on solving time on larger input models. Therefore, we may conclude that a variability-based execution mode shows potential in model checking of graph transformations. We expect that this improvement will become more significant when including more variability-based graph transformation rules with more features and presence conditions for a given system of transformation rules.

8 Limitations and Future Work

Our presented model checking approach is far from complete and is to be considered a proof of concept. Important future directions for extending our present, Gryphon-based, approach include:

- A rigorous argumentation for the soundness and performance of our approach. For soundness, this entails proving that our variability-based execution mode leads to the same results as the classical one—a conjecture currently supported by our informal argumentation and automated testing in our evaluation scenario. A performance proof addressing the internal workings of the underlying hardware model checker could be used to explain our empirically observed performance benefits.
- Increasing the expressiveness of supported variability-based transformations, to support configuration constraints as well as arbitrary propositional formulas as presence conditions. While the current implementation does not support constraints and only supports simple presence conditions consisting of a singular feature or its negation, we do not foresee any particular limitations towards improving the offered support as described.
- A more exhaustive empirical performance evaluation. The present evaluation considered a small graph transformation system, including a limited number of features and presence conditions. While we have shown that including variability in transformation rules can bring performance improvements even in this small setting, we foresee that the improvement can be more significant when considering more rule variability and larger input models.

The abovementioned directions could be addressed by further extensions of and experimentation with the Gryphon model checker. However, any future extensions of Gryphon are likely to retain the same limitations regarding the supported kinds of graph transformations that we currently have. In particular, they would be focused on graph transformation rules that create and delete edges.

To lift this limitation, a promising direction is to extend other graph-based model checking techniques that may benefit from introducing variability as well. As a starting point, one might want to introduce a variability-based execution mode in the GROOVE [35] model verification technique. As described in Section 6, GROOVE explicitly represents states as graphs and transitions as applications of transformation rules. Therefore, a promising idea is to include a variability-based rule application mode to make GROOVE variability-aware. To this end, one could use some of the concepts of dealing with variability as presented in this paper.

Finally, while the current work is focused on addressing variability in rules, graphs could be affected by variability as well. In particular, this is the case when modeling and analyzing a *software product line* [46] using graphs, which during model checking would lead to a further dimension of combinatorial explosion. Graph transformation of software product lines has been addressed in previous work [45, 40, 10], motivating future work on model checking as a means of validating such transformations.

References

[1] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger & Dirk Beyer (2013): *Strategies for product-line verification: Case studies and experiments*. In: *2013 35th International Conference on Software Engineering (ICSE)*, pp. 482–491.

[2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause & Gabriele Taentzer (2010): *Henshin: advanced concepts and tools for in-place EMF model transformations*. In: *International Conference on Model Driven Engineering Languages and Systems*, Springer, pp. 121–135.

[3] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking*. MIT Press, Cambridge, MA, USA.

[4] Paolo Baldan, Andrea Corradini & Barbara König (2008): *A framework for the verification of infinite-state graph transformation systems*. Information and Computation 206(7), pp. 869–907.

[5] Mordechai Ben-Ari (2008): *Principles of the Spin Model Checker*. Springer, London.

[6] Harsh Beohar, Mahsa Varshosaz & Mohammad Reza Mousavi (2016): *Basic behavioral models for software product lines: Expressiveness and testing pre-orders*. Science of Computer Programming 123, pp. 42–60.

[7] Jon Haël Brenas, Rachid Echahed & Martin Strecker (2016): *Ensuring Correctness of Model Transformations While Remaining Decidable*. In Augusto Sampaio & Farn Wang, editors: Theoretical Aspects of Computing – ICTAC 2016, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 315–332.

[8] Jon Haël Brenas, Rachid Echahed & Martin Strecker (2018): *Verifying Graph Transformation Systems with Description Logics*. In Leen Lambers & Jens Weber, editors: Graph Transformation, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 155–170.

[9] Jon Haël Brenas & Arash Shaban-Nejad (2021): *Proving the Correctness of Knowledge Graph Update: A Scenario From Surveillance of Adverse Childhood Experiences*. Frontiers in Big Data 4. Available at `https://www.frontiersin.org/article/10.3389/fdata.2021.660101`.

[10] Marsha Chechik, Michalis Famelis, Rick Salay & Daniel Strüber (2016): *Perspectives of model transformation reuse*. In: International Conference on Integrated Formal Methods, Springer, pp. 28–44.

[11] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz & Christel Baier (2018): *ProFeat: feature-oriented engineering for family-based probabilistic model checking*. Formal Aspects of Computing 30(1), pp. 45–75.

[12] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay & Jean-Francois Raskin (2013): *Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking*. IEEE Transactions on Software Engineering 39(8), pp. 1069–1089.

[13] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens & Axel Legay (2011): *Symbolic model checking of software product lines*. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, pp. 321–330.

[14] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay & Jean-François Raskin (2010): *Model checking lots of systems: efficient verification of temporal properties in software product lines*. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10, 1, ACM Press, p. 335.

[15] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans & Axel Legay (2012): *Towards an incremental automata-based approach for software product-line model checking*. In: Proceedings of the 16th International Software Product Line Conference - Volume 2, SPLC '12, Association for Computing Machinery, New York, NY, USA, pp. 74–81.

[16] Hartmut Ehrig, Karsten Ehrig, Ulrike Golas & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. XIV, Springer.

[17] Andre W.B. Furtado, Andre L.M. Santos, Geber L. Ramalho & Eduardo Santana de Almeida (2011): *Improving Digital Game Development with Software Product Lines*. IEEE Software 28(5), pp. 30–37. Conference Name: IEEE Software.

[18] Sebastian Gabmeyer (2015): *New model checking techniques for software systems modeled with graphs and graph transformations*. Ph.D. thesis.

[19] Sebastian Gabmeyer & Martina Seidl (2016): *Lightweight Symbolic Verification of Graph Transformation Systems with Off-the-Shelf Hardware Model Checkers*. In Bernhard K. Aichernig & Carlo A. Furia, editors: Tests and Proofs, 9762, Springer International Publishing, pp. 94–111.

[20] Fabio Gadducci, Alberto Lluch Lafuente & Andrea Vandin (2012): *Exploiting Over- and Under-Approximations for Infinite-State Counterpart Models*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Graph Transformations*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 51–65.

[21] Amir Hossein Ghamarian & Arend Rensink (2012): *Generalised Compositionality in Graph Transformation*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Graph Transformations*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 234–248.

[22] Alexander Gruler, Martin Leucker & Kathrin Scheidemann (2008): *Modeling and Model Checking Software Product Lines*. In Gilles Barthe & Frank S. de Boer, editors: *Formal Methods for Open Object-Based Distributed Systems: 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 113–131.

[23] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp & Dániel Varró (2005): *Using Graph Transformation for Practical Model-Driven Software Engineering*. In Sami Beydeda, Matthias Book & Volker Gruhn, editors: *Model-Driven Software Development*, Springer Berlin Heidelberg, pp. 91–117.

[24] Matthias Güdemann (2022): *mgudemann/iimc*. Available at `https://github.com/mgudemann/iimc`.

[25] Reiko Heckel (1998): *Compositional verification of reactive systems specified by graph transformation*. In Egidio Astesiano, editor: *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 138–153.

[26] Reiko Heckel & Gabriele Taentzer (2020): *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer International Publishing, Cham.

[27] Tobias Isenberg, Dominik Steenken & Heike Wehrheim (2013): *Bounded Model Checking of Graph Transformation Systems via SMT Solving*. In Dirk Beyer & Michele Boreale, editors: *Formal Techniques for Distributed Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 178–192.

[28] Shmuel Katz (2006): *Aspect categories and classes of temporal properties*. In: *Transactions on Aspect-Oriented Software Development I*, Springer-Verlag, Berlin, Heidelberg, pp. 106–134.

[29] Tomoji Kishi & Natsuko Noda (2006): *Formal verification and software product lines*. *Communications of the ACM* 49(12), pp. 73–77.

[30] Kodkod (2017): *Kodkod: About*. Available at `https://emina.github.io/kodkod/`.

[31] Christian Krause (2021): *Henshin | The Eclipse Foundation*. Available at `https://www.eclipse.org/henshin/`.

[32] Sonja Maier & Daniel Volk (2008): *Facilitating language-oriented game development by the help of language workbenches*. In: *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, Association for Computing Machinery, New York, NY, USA, pp. 224–227.

[33] Maria Maximova, Holger Giese & Christian Krause (2018): *Probabilistic timed graph transformation systems*. *Journal of Logical and Algebraic Methods in Programming* 101, pp. 110–131.

[34] Maria Maximova, Sven Schneider & Holger Giese (2021): *Compositional Analysis of Probabilistic Timed Graph Transformation Systems*. In Esther Guerra & Mariëlle Stoelinga, editors: *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 196–217.

[35] Arend Rensink (2004): *The GROOVE Simulator: A Tool for State Space Generation*. In John L. Pfaltz, Manfred Nagl & Boris Böhlen, editors: *Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 479–485.

[36] Arend Rensink, Ákos Schmidt & Dániel Varró (2004): *Model Checking Graph Transformations: A Comparison of Two Approaches*. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce & Grzegorz Rozenberg, editors: *Graph Transformations*, 3256, Springer Berlin Heidelberg, pp. 226–241.

[37] Ákos Schmidt & Dániel Varró (2003): *CheckVML: A Tool for Model Checking Visual Modeling Languages*. In Perdita Stevens, Jon Whittle & Grady Booch, editors: *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 92–95.

[38] Sven Schneider, Johannes Dyck & Holger Giese (2020): *Formal Verification of Invariants for Attributed Graph Transformation Systems Based on Nested Attributed Graph Conditions*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 257–275.

[39] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf & Matthias Tichy (2017): *Henshin: A usability-focused framework for EMF model transformation development*. In: *International Conference on Graph Transformation*, Springer, pp. 196–208.

[40] Daniel Strüber, Sven Peldszus & Jan Jürjens (2018): *Taming Multi-Variability of Software Product Line Transformations*. In: *FASE*, pp. 337–355.

[41] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer & Jennifer Plöger (2018): *Variability-based model transformation: formal foundation and application*. *Formal Aspects of Computing* 30(1), pp. 133–162.

[42] Daniel Strüber & Stefan Schulz (2016): *A tool environment for managing families of model transformation rules*. In: *International Conference on Graph Transformation*, Springer, pp. 89–101.

[43] Daniel Strüber, Julia Rubin, Marsha Chechik & Gabriele Taentzer (2015): *A Variability-Based Approach to Reusable and Efficient Model Transformations*. In Alexander Egyed & Ina Schaefer, editors: *Fundamental Approaches to Software Engineering*, 9033, Springer Berlin Heidelberg, pp. 283–298.

[44] Eugene Syriani & Hans Vangheluwe (2008): *Programmed Graph Rewriting with Time for Simulation-Based Design*. In Antonio Vallecillo, Jeff Gray & Alfonso Pierantonio, editors: *Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 91–106.

[45] Gabriele Taentzer, Rick Salay, Daniel Strüber & Marsha Chechik (2017): *Transformations of software product lines: A generalizing framework based on category theory*. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, pp. 101–111.

[46] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer & Gunter Saake (2014): *A Classification and Survey of Analysis Strategies for Software Product Lines*. *ACM Comput. Surv.* 47(1), pp. 6:1–6:45.

[47] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi & Ina Schaefer (2018): *A Classification of Product Sampling for Software Product Lines*. In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, Association for Computing Machinery, New York, NY, USA, pp. 1–13.

[48] Meng Zhu & Alf Inge Wang (2019): *Model-driven Game Development: A Literature Review*. *ACM Computing Surveys* 52(6), pp. 123:1–123:32.