Lightweight Symbolic Verification of Graph Transformation Systems with Off-the-Shelf Hardware Model Checkers

Sebastian Gabmeyer[1] and Martina Seidl[2]

[1] Security Engineering Group, TU Darmstadt, Germany
[2] Institute for Formal Models and Verification, JKU Linz, Austria

Abstract. We present a novel symbolic bounded model checking approach to test reachability properties of model-driven implementations of software. Given a concrete initial state of a software system, a type graph and, respectively, a set of graph transformations describing the system's structure and, respectively, behavior, the reachability properties are expressed in terms of graph constraints. Without any user intervention, our approach exploits state-of-the-art model checking technologies successfully used in hardware industry. The efficiency of our approach is demonstrated in two case studies.

1 Introduction

The growing demand for more sophisticated functionality considerably increases the complexity of state-of-the-art software and the complexity of today's software development [2,12,20]. With the rise in complexity, more defects tend to get introduced into the code [32]. To counter this challenge, graphical and textual modeling languages, like the Unified Modeling Language (UML) [22], began to permeate the modern software development process. The motivation of lifting models to first-class development artifacts is twofold: first, models abstract away irrelevant details and, second, they express ideas and solutions in the language of the problem domain offering a focused view to the developers.

In the context of the model-driven engineering (MDE) paradigm, *model transformations* play a pivotal role [10,27] when rewriting the models, e.g., to generate executable code or to perform refactorings, or when specifying the behavior of models. Unfortunately, software development based on models and model transformations is not immune to defects. In fact, errors introduced at the modeling layer might propagate to the executable code and might be hard to detect. Baurry et al. [5] emphasize that model transformations show characteristics that lead to challenging barriers to systematic testing. One particular problem encountered when testing is the complexity of the resulting output models whose correctness cannot be easily validated. A natural solution is a constructive approach where first the models are constructed by executing the model transformation and then checking constraints the output model has to satisfy. If the model transformation contains errors, however, this approach does not provide any debugging support,

because the intermediate models are not considered and hence the defect might be hard to trace. Also, the testing is usually non-exhaustive and model transformations are often applied nondeterministically. As a consequence, defects might be overseen. A solution is the lightweight, i.e., depth-bounded, integration of verification approaches like model checking into the testing process. In model checking, a specification is tested against the system and, in case a violation is found, an error trace is returned. Model checking is successfully used to verify hardware systems; for software, however, there are still many challenges that have to be overcome. Models and model transformations miss many of the features that make software model checking difficult and, thus, model checking is a promising technique to overcome the barriers of testing model transformations.

In this paper, we propose a lightweight verification approach based on symbolic model checking for testing the correctness of graph transformations. It is "lightweight" in the sense that it only allows to verify systems up to a user-defined object count. Among the multitude of available model transformation languages, we choose graph transformations [13,24], which offer a formal and concise language to describe modifications on graphs and, hence, on models. Graph transformations can be shown to be Turing complete [19] and, therefore, they are as expressive as any other conventional programming language. Any software system may thus formally be described by a graph transformation system (GTS). In contrast to previous work, we present a symbolic model checking approach in this paper. For the symbolic encoding of graph transformation systems we employ relational logic because of (a) the high resemblance between sentences of relational logic, on the one hand, and graphs and graph transformations on the other hand, and (b) the tool support available to convert bounded, first-order relational logic into propositional logic [30]. We thus *describe* the execution semantics of graph transformation systems by means of bounded, first-order relational logic. On this basis we construct a relational transition system (RTS) which is then automatically checked by state-of-the-art hardware model checkers.

This paper is structured as follows. We start with presenting the running example in Section 2. Then we introduce the required preliminaries in Section 3 and review related approaches in Section 4. In Section 5 we show how our approach works and discuss our symbolic model checking encoding. In Section 6 we outline the relational semantics of graph transformations in depth. Then we present the results of two case studies in Section 7 and summarize our results with an outlook on future work in Section 8.

2 Running Example

In the Dining Philosophers (DP) problem's setup [11,14] a group of philosophers sits around a table with a plate in front of each of them and a fork on each side of the plate. Philosophers transition through a sequence of three

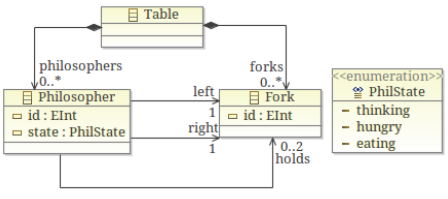
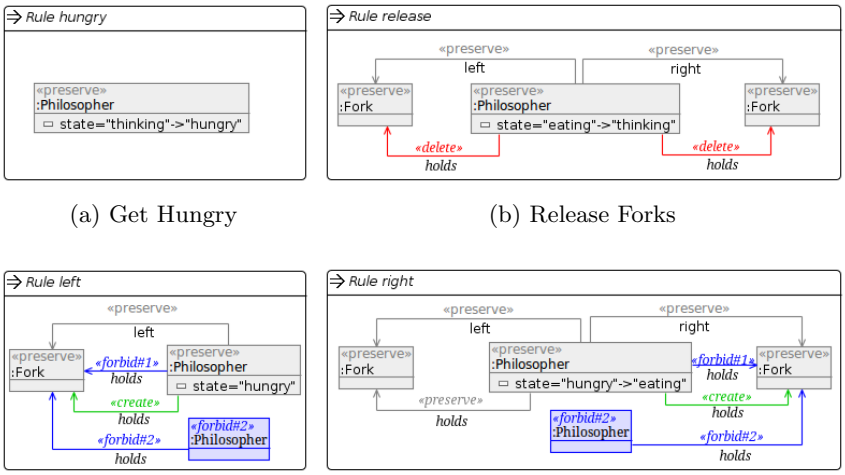


Fig. 1: Metamodel of DP.



(a) Get Hungry

(b) Release Forks

(c) Get Left Fork

(d) Get Right Fork & Eat

Fig. 2: Graph Transformations for the Dining Philosophers in Henshin

states: *thinking*, *hungry*, and *eating*. Once they finish eating they go back into thinking. Each philosopher requires two forks to start eating. These forks, however, are shared with the philosophers sitting to the philosopher's left and right.

The static structure of the DP problem's graph transformation system is given in terms of the metamodel shown in Figure 1. The behavior of each philosopher is defined by four graph transformations depicted in Figure 2. The *Get Hungry* transformation rewrites the philosopher's `state` attribute to transition from state *thinking* to state *hungry*. A hungry philosopher needs to eat and attempts to acquire a left fork first; this is achieved by applying the *Get Left Fork* transformation that establishes a `hold` reference between a *hungry* philosopher and her `left` fork. The negative application conditions (NAC), *forbid#1* and *forbid#2*, ensure that a `holds` reference my only be established to those forks not held by the philosopher herself (*forbid#1*) or any other philosopher (*forbid#2*). Once a philosopher holds the left fork, the *Get Right Fork & Eat* transformation picks up the appropriate right fork and changes the philosopher's `state` from *hungry* to *eating*. Again, the NACs *forbid#1* and *forbid#2* prohibit forks from being picked up if held by a philosopher. If a philosopher is done eating, the *Release Forks* transformation puts back the forks on the table and switches the philosopher's `state` to *thinking* again.

3 Preliminaries

Graph Transformations. Graphs and graph transformations are a popular choice to formally describe models and model transformations. For this purpose the theory of graph transformations has been extended to support rewriting of

problem := univ relation* formula

univ := {obj[, obj]*}
relation := $\text{rel}_{:\text{arity}}$[(lower,)? upper]

varDecl := rel : expr
lower := constant
upper := constant
constant := {tuple*}
tuple := ⟨obj[, obj]*⟩

arity := $\mathbb{N}$
obj := ID
rel := ID

expr := rel | unary | binary | comprehension
unary := expr unop
unop := $^+$ | $^{-1}$
binary := expr binop expr
binop := ∪ | ∩ | \ | . | ×
comprehension := {varDecl | formula}

formula := atomic | composite | quantified
atomic := expr ⊆ expr
composite := ¬formula | formula logop formula
logop := ∧ | ∨
quantified := quantifier varDecl | formula
quantifier := ∀ | ∃

(a) Syntax

P: problem → binding → boolean
R: relation → binding → boolean
M: formula → binding → boolean
E: expr → binding → constant
binding: rel → constant

$\mathrm{P}[\![\ U\ \ r_1 \ldots r_n\ \ \mathrm{F}\ ]\!]\mathrm{b} = \mathrm{R}[\![r_1]\!]\mathrm{b} \wedge \cdots \wedge \mathrm{R}[\![r_n]\!]\mathrm{b} \wedge \mathrm{M}[\![\mathrm{F}]\!]\mathrm{b}$

$\mathrm{R}[\![r\ :\ [\mathrm{L}]]\!]\mathrm{b} = \mathrm{R}[\![r\ :\ [\mathrm{L},\mathrm{L}]]\!]\mathrm{b}$
$\mathrm{R}[\![r\ :\ [\mathrm{L},\mathrm{U}]]\!]\mathrm{b} = \mathrm{L} \subseteq \mathrm{b}(r) \subseteq \mathrm{U}$

$\mathrm{M}[\![\mathrm{p} \subseteq \mathrm{q}]\!]\mathrm{b} = \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b} \subseteq \mathrm{E}[\![\mathrm{q}]\!]\mathrm{b}$
$\mathrm{M}[\![\neg\mathrm{F}]\!]\mathrm{b} = \neg\mathrm{M}[\![\mathrm{F}]\!]\mathrm{b}$
$\mathrm{M}[\![\mathrm{F}\ op\ \mathrm{G}]\!]\mathrm{b} = \mathrm{M}[\![\mathrm{F}]\!]\mathrm{b}\ op\ \mathrm{M}[\![\mathrm{G}]\!]\mathrm{b}$ where $op = \{\wedge, \vee\}$
$\mathrm{M}[\![\forall v : \mathrm{p} \mid \mathrm{F}]\!]\mathrm{b} = \bigwedge_{x \in \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b}} \mathrm{M}[\![\mathrm{F}]\!](\mathrm{b} \oplus [v \mapsto x])$
$\mathrm{M}[\![\exists v : \mathrm{p} \mid \mathrm{F}]\!]\mathrm{b} = \bigvee_{x \in \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b}} \mathrm{M}[\![\mathrm{F}]\!](\mathrm{b} \oplus [v \mapsto x])$

$\mathrm{E}[\![\mathrm{p}\ op\ \mathrm{q}]\!]\mathrm{b} = \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b}\ op\ \mathrm{E}[\![\mathrm{q}]\!]\mathrm{b}$ where $op = \{\cup, \cap, \setminus\}$
$\mathrm{E}[\![\mathrm{p}\ .\ \mathrm{q}]\!]\mathrm{b} = \{\langle \mathrm{p}_1, \ldots, \mathrm{p}_{n-1}, \mathrm{q}_2, \ldots, \mathrm{q}_m \rangle \mid$
$\quad \langle \mathrm{p}_1, \ldots, \mathrm{p}_n \rangle \in \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b} \wedge \langle \mathrm{q}_1, \ldots, \mathrm{q}_m \rangle \in \mathrm{E}[\![\mathrm{q}]\!]\mathrm{b}$
$\quad \wedge \mathrm{p}_n = \mathrm{q}_1\}$
$\mathrm{E}[\![\mathrm{p} \times \mathrm{q}]\!]\mathrm{b} = \{\langle \mathrm{p}_1, \ldots, \mathrm{p}_n, \mathrm{q}_1, \ldots, \mathrm{q}_n \rangle \mid$
$\quad \langle \mathrm{p}_1, \ldots, \mathrm{p}_n \rangle \in \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b} \wedge \langle \mathrm{q}_1, \ldots, \mathrm{q}_n \rangle \in \mathrm{E}[\![\mathrm{q}]\!]\mathrm{b}\}$
$\mathrm{E}[\![\mathrm{p}^{-1}]\!]\mathrm{b} = \{\langle \mathrm{p}_2, \mathrm{p}_1 \rangle \mid \langle \mathrm{p}_1, \mathrm{p}_2 \rangle \in \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b}\}$
$\mathrm{E}[\![\mathrm{p}^{+}]\!]\mathrm{b} = \{\langle x, y \rangle \mid \exists \mathrm{p}_1, \ldots, \mathrm{p}_n \mid$
$\quad \langle x, \mathrm{p}_1 \rangle, \langle \mathrm{p}_1, \mathrm{p}_2 \rangle, \ldots, \langle \mathrm{p}_n, y \rangle \in \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b}\}$
$\mathrm{E}[\![\{v : \mathrm{p} \mid \mathrm{F}\}]\!]\mathrm{b} = \{x \in \mathrm{E}[\![\mathrm{p}]\!]\mathrm{b} \mid \mathrm{M}[\![\mathrm{F}]\!](\mathrm{b} \oplus [v \mapsto x])\}$
$\mathrm{E}[\![r]\!]\mathrm{b} = \mathrm{b}(r)$

(b) Semantics

Fig. 3: Syntax and semantics of relational logic [30]

attributed, typed graphs with inheritance and part-of relations [6]. In the following, we summarize the concepts relevant for this work. For details see [24,13]. A *graph* $G = (V_G, E_G)$ consists of a set V_G of nodes, a set E_G of edges. Further, we define a source and a target function, $src : E_G \to V_G$ and $tgt : E_G \to V_G$, that map edges to their source and target vertices. A *morphism* $m : G \to H$ is a structure preserving mapping between graphs G and H. A double pushout graph transformation $p : L \leftarrow K \rightarrow R$, with morphisms $l : K \to L$ and $r : K \to R$, describes how the left-hand side (LHS) graph L is transformed into the right-hand side (RHS) graph R via an interface graph K. A graph transformation $p : L \leftarrow K \rightarrow R$ is *applied* to a *host graph* G if there exists a morphism $m : L \to G$, called a *match*, that maps the LHS graph L into the host graph G. The application of transformation p at match m rewrites graph G to the *result graph* H. A transformation p preserves those nodes and edges that are both in the domain of morphisms l and r, while it deletes those nodes and edges in L that are not in the co-domain of l and creates those nodes and edges in R that are not in the co-domain of r.

Relational Logic. Bounded, first-order relational logic[3] extends propositional logic with relational *variables* of a given arity, a finite *universe* of objects, and

[3] Our presentation of the logic follows the one presented for Alloy [15] and Kodkod [30] that in turn are based on Tarski's exposition of the relational calculus [28].

quantifiers. Each relational variable is assigned an upper bound and, optionally, a lower bound, which are sets of tuples over the set of objects in the universe. Syntax and semantics is provided in Figure 3. A relational problem P in bounded, first-order relational logic is a tuple $(Rel, F, U, ar, \sqcap, \sqcup)$ consisting of

1. a set Rel of relations,
2. a relational formula F,
3. a finite *universe* of discourse U, i.e., a set of uninterpreted *objects*,
4. a map $ar : Rel \to \mathbb{N}$ that assigns an arity to each relational variable $r \in Rel$,
5. maps $\sqcap : Rel \to \mathcal{P}(U^n)$ and $\sqcup : Rel \to \mathcal{P}(U^n)$ that define n-ary lower and upper bounds for relations.

A relational *constant* is a set of n-ary tuples including the empty set. The set of relational *expressions* is recursively defined as the smallest set consisting of the empty set and the set of all atoms, i.e., the universe U, the relations $r \in Rel$, and all expressions resulting from applying either (i) a unary operator like *transitive closure* $(^+)$ or *transposition* $(^{-1})$ to another expression or (ii) a binary operator, *union* $(\cup)$, *intersection* $(\cap)$, *join* $(.)$, *difference* $(\setminus)$, or *product* $(\times)$, to the former and another expression (see Fig. 3). The evaluation of an expression yields a set of tuples over U. An atomic *relational formula* is a sentence constructed over two relational *expressions* connected by the subset $\subseteq$ operator. Formulas can be quantified and composed into composite formulas using the usual logical connectives, *and* $(\wedge)$, *or* $(\vee)$, and *not* $(\neg)$. A *model* of a relational problem is an assignment, i.e., a *binding*, of tuples to relational variables such that (1) the assigned tuples lie within the lower and upper bounds of the relational variable and (2) the formula evaluates to true. Note that our treatment of relational logic is untyped; admissible bindings to relations are solely defined by their lower and their upper bounds. Further note, that the logic also supports reasoning over bit-vectors that represent integer values.

A *relational transition system* (RTS) extends the a relational problem by a set Rel' of next state relational variables, an initial binding ι for the relational variables in Rel, and a relational formula g.

Definition 1. *A bounded, first-order* relational transition system S *is a tuple* $(Rel, Rel', T, U, ar, \sqcap, \sqcup, \iota, g)$ *that consists of*

1. *a set* Rel *of unary and binary relational variables,*
2. *a set* $Rel' = \{r' | r \in Rel\}$ *of next state relational variables,*
3. *a transition relation* T*, i.e., a conjunction of first-order relational formulas over* $Rel \cup Rel'$*,*
4. *a finite universe of discourse* U*,*
5. *a map* $ar : Rel \cup Rel' \to \mathbb{N}$ *that assigns an arity to each relational variable* $r \in Rel \cup Rel'$*,*
6. *an initial binding* ι *that represents the initial state of the transition system,*
7. *a lower and an upper bound map* $\sqcap : (Rel) \to \mathcal{P}(U^n)$ *and* $\sqcup : (Rel \cup Rel') \to \mathcal{P}(U^n)$*, and*
8. *a relational formula* g *that represents either a desired or an undesired state.*

A *state* in an RTS is a binding b of a set of tuples to each relational variable $r \in \mathit{Rel}$. A *trace* is a finite sequence of states $b_0 b_1 \dots b_n$ with $\iota = b_0$ such that

$$\mathrm{P}[\![\ U\ r_1 \dots r_n r'_1 \dots r'_n\ \mathit{true}\ ]\!]\mathrm{b}_0 \wedge \mathrm{P}[\![\ U\ r_1 \dots r_n r'_1 \dots r'_n\ T\ ]\!]\mathrm{b}_0 \cup \mathrm{b}'_1 \wedge \dots \wedge$$
$$\mathrm{P}[\![\ U\ r_1 \dots r_n r'_1 \dots r'_n\ T\ ]\!]\mathrm{b}_{n-1} \cup \mathrm{b}'_n \wedge \mathrm{P}[\![\ U\ r_1 \dots r_n r'_1 \dots r'_n\ g\ ]\!]\mathrm{b}_n,$$

where $\mathrm{b}'_i, 0 < i \leqslant n$, is equivalent to binding b_i but applied to the next state relational variables in Rel', is satisfied.

4 Related Work

Model checking [9] on which we base the approach presented in this paper is technique which has found several applications in the validation of GTSs. CheckVML [26,31] verifies the behavior of a system that is defined by UML-like class diagrams, internally represented as an attributed type graph with inheritance relations, and a set of graph transformations. Together with a user-defined initial state, CheckVML encodes the resulting GTS and a set of reachability properties into Promela, the input language of the model checker spin. A similar approach is proposed in [3], where the encoding of the GTS produces code for the model checker Bogor [23]. In [4], the authors also present an encoding that allows to analyze graph transformations specified in Agg [25] with the Alloy analyzer [16]. Groove [18] verifies the behavioral correctness of an object-oriented system represented by an attributed type graph with inheritance relations. It implements its own model checker that enumerates the entire state space before checking the user-specified safety or liveness properties. Similar to Groove, the model checker MocOCL [7] enumerates the state space explicitly but in an iterative manner. It employs the Henshin API [1] to construct the state space and is capable of verifying safety and liveness properties that are expressed in a temporal extension of OCL. In this paper, we present a symbolic approach, i.e., we represent the state space by a logical formula.

5 Architecture

Our symbolic model checking approach tests a model-driven implementation of a software system against a reachability properties, called a *goal state*, up to a certain bound.[4] Here, a model-driven implementation of a system consists of

(a) an EMF[5] *model* $\mathbb{M}$ that describes the static structure of the system and
(b) a set $\mathcal{R}$ of *graph transformations* that describes the system's execution behavior.

[4] Formally, a goal state φ is a temporal reachability property $\mathsf{EF}\,\varphi$. Note that by duality, we can also assert a system safe from reaching a bad state $\overline{s} \equiv \neg\varphi$ by proving $\neg\varphi$ unreachable.

[5] Eclipse Modeling Framework (EMF): `eclipse.org/modeling/emf/`

For the verification our tool expects in addition the following components:

(c) an instance model M_ι of $\mathbb{M}$ that describes the *initial state* of the system,
(d) a graph constraint g, i.e., the *goal states*, that represents the reachability property, and
(e) an *object bound map* $\Gamma : V_T \to \mathbb{N}$ that defines the maximal number of objects per class in instances of $\mathbb{M}$.

Internally, the EMF model $\mathbb{M}$ is represented as an attributed type graph $G_T = (V_T, E_T)$ with inheritance and containment edges [6], whose vertices V_T represent the classes and whose edges E_T represent the references and the attributes of the system. Attribute types and values, however, are restricted to integers *Int*, Booleans *Bool*, and user-defined enumerations *Enum*. The initial state M_ι is a typed graph $G_M = (V_M, E_M)$ with a type morphism $type\colon G_M \to G_T$ that maps "objects" $v \in V_M$ to their types $t \in V_T$. A goal state is modeled as a graph constraint [13], which describes a desired or an undesired pattern that is matched against an instance of $\mathbb{M}$. Note that due to the use of double pushout graph transformations, which are free of side effects, we can straightforwardly derive all relational frame conditions for the RTS (cf. [21]). Next observe that the semantics of bounded, first-order relational logic is defined by propositional logic [29]. By applying these semantic definitions to the previously constructed RTS we construct a sequential circuit, or more specifically, an *and-inverter graph* (AIG), which is, roughly speaking, a Boolean circuit that uses only AND and NOT gates. We store the resulting AIG together with the reachability properties that we want to verify in the AIGER file format, which is the standardized input format of all model checkers that compete in the Hardware Model Checking Competition. By storing the GTS in an AIGER file, we are not limited to a specific model checker and we can directly exploit the most recent developments in hardware model checking like the successfull IC3 algorithm [8].

In the following we give a high-level description of the workflow that generates in three steps from a model-driven implementation, first, a symbolic, relational transition system (RTS), second, a propositional formula by instantiating the first-order quantifiers in the transition relation of the RTS, and third, an and-inverter graph (AIG) by rewriting the propositional formula (see Fig. 4). In particular, the `emf2fol` component first extracts unary and binary relational variables for each class and reference in $\mathbb{M}$, respectively. It then constructs the universe U from the object bound map Γ. Next, it assigns appropriate upper bounds, i.e., sets of atoms from U, to each relational variable. Finally, it extracts the initial state of the RTS from M_ι. The `gt2fol` component generates from the set of graph transformations the transition relation T as a conjunction of existentially quantified relational logic formulas and from the graph constraint the goal state g. Next, the `fol2bool` component uses the KODKOD API to instantiate[6] the bounded, first-order relational formulas of the transition relation and the goal state into Boolean functions, i.e., propositional formulas. The resulting Boolean

[6] Note that we do not use KODKOD's model finding capabilities but only use it to translate relational logic formulas into propositional formulas.

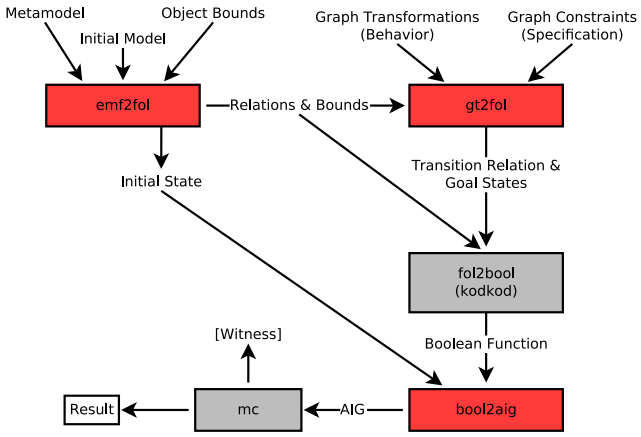


Fig. 4: Workflow of our symbolic model checking approach. Grayed components indicate external tools.

functions are rewritten into an AIG by the `bool2aig` component and stored in the AIGER format, which is readable by any model checker participating in the Hardware Model Checking Competition. Next, a model checker `mc` is used to verify the AIG. In case the model checker determines that a desired or undesired state is reachable it reports the result to the user together with a witness that may be used to re-construct a trace of the RTS. Otherwise, it returns "*unreachable*".

6 Encodings

In the following we detail the workings of `emf2fol` and `gt2fol`. Here, we denote the set of all relational formulas by $\mathbb{F}$ and we define the set $\mathbb{A}$ of atoms by $\mathbb{A} = \{(C_i) \mid C \in V_T, i \in \mathbb{N}\} \cup \{(j) \mid j \in \mathbb{Z}\}$.

Relational Variables, Universe, Bounds, and Initial State. The `emf2fol` component uses the function $\mathit{relgen} : V_T \cup E_T \to \mathit{Rel}$ to generate for each class C in V_T a unary relational variable C and likewise for each enumeration E in Enum a unary relational variable E. For each attribute $\mathit{attr} \in E_T$ the translation generates a binary relational variable $\mathsf{C\_attr}$ and for each reference $\mathit{ref} \in E_T$ from a source class C to a target class D it creates a binary relational variable $\mathsf{C\_ref}$. Moreover, relational variables Int and Bool are generated for the primitive types Int and Bool if necessary.

The universe U consists of a sequence of uninterpreted atoms, which is derived from the object bound map Γ and the initial model M_ι such that for every object in M_ι there exists a corresponding atom in U.

The function $\sqcup : \mathit{Rel} \to \mathcal{P}(\mathbb{A})$ assigns to each relational variable $r \in \mathit{Rel}$ an upper bound. Given a unary relational variable $\mathsf{C} = \mathit{relgen}(C), C \in V_T$ the upper bound is defined by $\sqcup(\mathsf{C}) = \{(C_i) \mid 0 < i \leqslant \Gamma(C)\}$. The upper bound of a relational variable for an attribute $\mathsf{C\_attr}$ is constructed from the

product of the upper bounds of class C and the domain of the attribute, i.e., $\sqcup(\mathsf{C\_attr}) = \sqcup(\mathsf{C}) \times \sqcup(\mathbb{D})$, where $\mathbb{D} \in \{\mathsf{Int}, \mathsf{Bool}, \mathsf{E}_1, \ldots, \mathsf{E}_n\}$. Likewise, the upper bound of a relational variable for reference is constructed from the product of the source and the target class's upper bounds, i.e., $\sqcup(\mathsf{C\_ref}) = \sqcup(\mathsf{C}) \times \sqcup(\mathsf{D})$ with source class C and target class D. Note that the the inheritance hierarchy is reflected by a relational variable's bounds. Let $S(C)$ denote the set of all subclasses of class C. The upper bound of C is then defined as above plus the union of all upper bounds of its subclasses, i.e., $\sqcup(C) = \{(C_i) \mid 0 < i \leqslant \Gamma(C)\} \cup \bigcup_{s \in S(C)} \sqcup(s)$.

Finally, the initial state map $\iota : \mathit{Rel} \to \mathcal{P}(\mathbb{A}^n)$ assigns sets of n-ary tuples to n-ary relational variables and it is derived from the initial model M_ι. Let $\mathit{atom} : V_M \cup E_M \to \mathbb{A}$ be a function that maps an object $v \in V_M$, an object reference or an attribute $e \in E_M$ to an atom in $\mathbb{A}$, then, ι is defined as follows.

$$\iota(\mathsf{r}) = \begin{cases} \{\mathit{atom}(v) \mid v \in V_M, \mathit{type}(v) = C\} & \text{if } \exists\, C \in V_T.\ \mathit{relgen}(C) = \mathsf{r} \\ \{(\mathit{atom}(v), \mathit{atom}(a)) \mid a \in E_M, \\ \qquad \mathit{type}(a) = \mathit{attr}, \mathit{src}_M(a) = v\} & \text{if } \exists\, \mathit{attr} \in E_T.\ \mathit{relgen}(\mathit{attr}) = \mathsf{r} \\ \{(\mathit{atom}(v), \mathit{atom}(e)) \mid e \in E_M, \\ \qquad \mathit{type}(e) = \mathit{ref}, \mathit{src}_M(e) = v\} & \text{if } \exists\, \mathit{ref} \in E_T.\ \mathit{relgen}(\mathit{ref}) = \mathsf{r} \end{cases}$$

Example. For the Dining Philosophers problem (see Sec. 2) `emf2fol` generates unary relational variables Table, Philosopher, Fork, and PhilState. As enumerations are mapped onto integers, `emf2fol` automatically infers the necessary bitwidth of 2 to represent the three literals *thinking*, *hungry*, and *eating* of enumeration *PhilState*, which are mapped to -2, -1, and 0, respectively. Further, binary relational variables are generated for each attribute and reference, e.g., for reference *forks* in class *Table* the relational variable Table_forks is generated. Note that we instruct `emf2fol` to omit relational variables for the *id* fields because they solely exist to aid the manual construction of instance models. Table 1 lists all generated relational variables and their upper bounds for the Dining Philosophers problem.

Given the object bound map $\Gamma = \{(\mathit{Table}, 1), (\mathit{Philosopher}, 2), (\mathit{Fork}, 2)\}$ `emf2fol` derives the universe $U = \{\mathit{Table}_1, \mathit{Philosopher}_1, \mathit{Philosopher}_2, \mathit{Fork}_1, \mathit{Fork}_2, -2, \ldots, 1\}$. Next, upper bounds are assigned to each of the generated relational variables. For example, the relation variable Philosopher is assigned the upper bound $\sqcup(\mathsf{Philosopher} = \{(\mathit{Philosopher}_1), (\mathit{Philosopher}_2)\}$. The upper bound of the Philosopher_right relational variable is defined as

$$\begin{aligned} \sqcup(\mathsf{Philosopher\_right}) &= \sqcup\,(\mathsf{Philosopher}) \times \sqcup(\mathsf{Fork}) \\ &= \{(\mathit{Philosopher}_1, \mathit{Fork}_1), (\mathit{Philosopher}_1, \mathit{Fork}_2) \\ &\quad\ (\mathit{Philosopher}_2, \mathit{Fork}_1), (\mathit{Philosopher}_2, \mathit{Fork}_2)\}. \end{aligned}$$

Class	Variable	Upper Bound
Table	Table	$\{(\mathit{Table}_1)\}$
philosophers	Table_philosophers	$\{(\mathit{Table}_1, \mathit{Philosopher}_1), (\mathit{Table}_1, \mathit{Philosopher}_2)\}$
forks	Table_forks	$\{(\mathit{Table}_1, \mathit{Fork}_1), (\mathit{Table}_1, \mathit{Fork}_2)\}$
Philosopher	Philosopher	$\{(\mathit{Philosopher}_1), (\mathit{Philosopher}_2)\}$
right	Philosopher_right	$\{(\mathit{Philosopher}_1, \mathit{Fork}_1), (\mathit{Philosopher}_1, \mathit{Fork}_2), (\mathit{Philosopher}_2, \mathit{Fork}_1), (\mathit{Philosopher}_2, \mathit{Fork}_2)\}$
left	Philosopher_left	$\{(\mathit{Philosopher}_1, \mathit{Fork}_1), \ldots, (\mathit{Philosopher}_2, \mathit{Fork}_2)\}$
holds	Philosopher_holds	$\{(\mathit{Philosopher}_1, \mathit{Fork}_1), \ldots, (\mathit{Philosopher}_2, \mathit{Fork}_2)\}$
state	Philosopher_state	$\{(\mathit{Philosopher}_1, -2), \ldots, (\mathit{Philosopher}_1, 0), (\mathit{Philosopher}_2, -2), \ldots, (\mathit{Philosopher}_2, 0)\}$
Fork	Fork	$\{(\mathit{Fork}_1), (\mathit{Fork}_2)\}$
PhilState	PhilState	$\{(-2), (-1), (0)\}$
Int	Int	$\{(-2), (-1), (0), (1)\}$

Table 1: Generated variables and bounds for the Dining Philosophers problem.

The initial state map for two dining philosophers is defined by the graph

$$\begin{aligned}\iota = \{&(\mathsf{Table}, \{(\mathit{Table}_1)\}), (\mathsf{Philosopher}, \{(\mathit{Philosopher}_1), (\mathit{Philosopher}_2)\}),\\ &(\mathsf{Fork}, \{(\mathit{Fork}_1), (\mathit{Fork}_2)\}), (\mathsf{Table\_philosophers}, \{(\mathit{Table}_1, \mathit{Philosopher}_1),\\ &(\mathit{Table}_1, \mathit{Philosopher}_2)\}), (\mathsf{Table\_forks}, \{(\mathit{Table}_1, \mathit{Fork}_1), (\mathit{Table}_1, \mathit{Fork}_2)\}),\\ &(\mathsf{Philosopher\_state}, \{(\mathit{Philosopher}_1, -2), (\mathit{Philosopher}_2, -2)\}), \ldots\}.\end{aligned}$$

Graph Transformations. Formally, `gt2fol` translates a graph transformation into a first-order, relational formula as follows. Given sets Rel and Rel' of current and next state relational variables generated for a (meta-)model $\mathbb{M}$ as described above, from each (double pushout) graph transformation $p : \mathit{Cond} \leftarrow \mathit{Lhs} \rightarrow \mathit{Rhs}$, where the application condition Cond can consist of positive application conditions (Pac) and/or negative application conditions (Nac), a formula

$$\mathrm{F}_p := \mathit{Pre}(\mathit{Lhs}, \mathit{Pac}, \mathit{Nac}, \mathit{Rhs}) \implies \mathit{Post}(\mathit{Lhs}, \mathit{Rhs}) \tag{1}$$

is derived where $\mathit{Pre} : G \times G \times G \times G \rightarrow \mathbb{F}$ is a function that generates from a quadruple of graphs a conjunction of relational formulas $f \in \mathbb{F}$ that mimic the match conditions of the transformation's LHS. Function $\mathit{Post} : G \times G \rightarrow \mathbb{F}$, on the other hand, generates a conjunction of relational formulas from a pair of graphs, i.e., the LHS and RHS, that mimic the effects of the transformation's RHS. Here, the set G of graphs is typed by $T \in \mathbb{M}$.

Function Pre generates the following conjuncts from the transformation's LHS. For each node obj_C of class C in the LHS we allocate a fresh, existentially quantified node variable c whose domain is bound by relational variable C that was generated for class C. This yields the relational formula $\exists\, \mathsf{c} : \mathsf{C}$. If obj_C of class

	LHS/RHS element	Formula
Preserve/Delete	Object $c \in C$	$\exists \mathsf{c} : \mathsf{C}$
	Reference *ref* with $src(ref) = c \in C$, $tgt(ref) = d \in D$	$(\mathsf{c} \rightarrow \mathsf{d}) \subseteq \mathsf{C\_ref}$
	Attribute *attr* with $src(attr) = c \in C$, expression e	$(\mathsf{c} \rightarrow expr(e)) \subseteq \mathsf{C\_attr}$
Forbid	Object $c \in C$	$\neg\exists \mathsf{c} : \mathsf{C}$
	Reference *ref*	*same as above*
	Attribute *attr*	*same as above*
Create	Object $c \in C$	$\exists \mathsf{c} : \mathsf{C}' \wedge \mathsf{c} \notin \mathsf{C}$
	Reference *ref*	—
	Attribute *attr*	—

Table 2: Relational formulas generated by function *Pre*

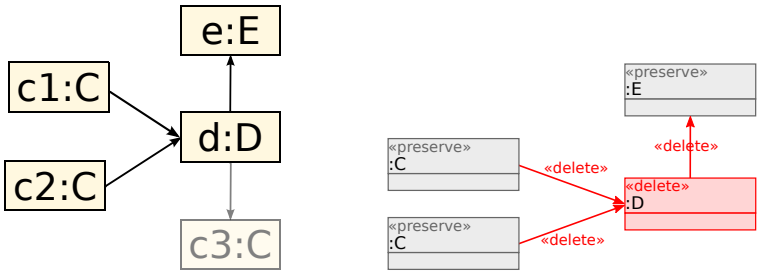
C has a reference *ref* to a target object obj_D of D and under the assumption that (relational) node variables c, d have been allocated for obj_C and obj_D, and relation $\mathsf{C\_ref}_{:2}$ was generated for reference *ref*, then the condition $(\mathsf{c} \rightarrow \mathsf{d}) \subseteq \mathsf{C\_ref}$ is generated where $(\mathsf{c} \rightarrow \mathsf{d})$ denotes the product of the tuples bound to c and d.

If an attribute *attr* of obj_C is assigned an expression e, then *Pre* generates formula $(\mathsf{c} \rightarrow expr(e)) \in \mathsf{C\_attr}$ where function $expr : Int \cup Bool \cup Enum \rightarrow$ Rexpr converts an integer, Boolean, or enumeration expression into a relational expression. This expression describes an additional constraint that a matching subgraph must satisfy. Thus, we generate a condition that requires the attribute value of the object that is bound to c to evaluate to the same value as $expr(e)$. For an overview of the conditions generated by *Pre* see Table 2.

If the graph transformation contains PAC patterns, they are translated into formulas of relational logic like the LHS pattern because they, too, demand the existence of nodes, edges, or matching attribute expressions. Thus, the translation of LHS and PAC patterns follow the same procedure as described above.

Negative application conditions, in contrast to LHS and PAC patterns, describe forbidden patterns that must not be satisfied by any matching subgraph. As such we generate equivalent relational formulas as for the LHS, but negate them such that the formula $\neg\exists\, \mathsf{n} : \mathsf{N}$ is generated assuming that node variable n was allocated for a NAC object obj_N. Note that none of the conditions generated for references and attributes in the NAC graph need to be negated and are thus equivalent to those generated for the LHS graph.

Finally, the injectivity and the dangling edge conditions, generated by *Pre*, are necessary to faithfully translate the graph modifying instructions into relational logic. The injectivity condition ensures that all elements of the LHS, the NACs, and the PACs are mapped to exactly one element in a matching host graph and each variable must be bound to a distinct object of the universe. The generated *injectivity* condition performs a pairwise test of inequality on all variables bound by the same expression. For example, given variables $\mathsf{c1}, \mathsf{c2}$, both of which are bound by C, the condition $\neg(\mathsf{c1} = \mathsf{c2})$ is generated to ensure that $\mathsf{c1}$ and $\mathsf{c2}$ are



(a) Object diagram (b) A graph transformation

Fig. 5: Dangling edge example

assigned to two different objects. The second condition ensures that no *dangling edges* are left behind after deleting nodes from the graph. This implies that all possible references to and from a node that is scheduled for deletion need to be deleted explicitly. We translate this requirement into a condition that checks whether the set of all possible references from and to an object that is scheduled for deletion coincides with the set of actually deleted references. For example, in Figure 5b object obj_D, an instance of class D, is deleted together with references coming from two objects, $obj_{C,1}$ and $obj_{C,2}$, and one reference to object obj_E. Class D may have references coming from objects of class C and references to object of class E. In the following we assume that the translation generates unary relational variables C, D, and E for classes C, D, and E, binary relational variables $\mathsf{C\_toD}$ and $\mathsf{D\_toE}$, and allocates existentially quantified variables $\mathsf{c1}$, $\mathsf{c2}$, d, and e (see Fig. 5 for a simplified object diagram). The formula $\mathsf{C\_toD.d} - \{\mathsf{c1}, \mathsf{c2}\} = \varnothing$ resembles the dangling edge condition for reference between objects of class C and class D. It consists of the following components:

- The relational variable $\mathsf{C\_toD}$ is bound to the set of all tuples (obj_c, obj_d) with $obj_c \in \sqcup(\mathsf{C})$ and $obj_d \in \sqcup(\mathsf{D})$ having a *toD* reference;
- the expression $\mathsf{C\_toD.d}$ represents the set of all *possible* objects of class C that have a *toD* reference to the object bound to d;
- the set $\{\mathsf{c1}, \mathsf{c2}\}$ represents the *actually deleted* objects.

Thus, the formula above checks whether the set of all actually deleted objects $\{\mathsf{c1}, \mathsf{c2}\}$ is equal to the set of all actual objects of class C that have a reference to the object bound to d. If, however, a third reference had pointed from $\mathsf{c3}$ to d, the expression $\mathsf{C\_toD.d} - \{\mathsf{c1}, \mathsf{c2}\}$ would evaluate to $\{\mathsf{c3}\}$ and thus violate $\{\mathsf{c3}\} \neq \varnothing$. In the latter case the graph transformation must not be applied.

The RHS describes the effects of the graph transformation; once a matching subgraph of the LHS is found it is rewritten according to the RHS that specifies which nodes, edges, and attributes are created and/or deleted. The function *Post* generates relational formulas over *Rel* and *Rel*′ that mimic the modifications of the transformation's RHS as follows. If the transformation creates an object obj_C of class C, two conditions are created, one by *Pre* and one by *Post*. First, function

	LHS/RHS element	Formula
Delete	Object $c \in C$	$\mathsf{C}' = \mathsf{C} - \mathsf{c}$
	Reference *ref* with $src(ref) = c \in C$, $tgt(ref) = d \in D$	$\mathsf{C\_ref}' = \mathsf{C\_ref} - (\mathsf{c} \rightarrow \mathsf{d})$
	Attribute *attr* with $src(attr) = c \in C$, expression e	$\mathsf{C\_attr}' = \mathsf{C\_attr} - (\mathsf{c} \rightarrow expr(e))$
Create	Object $c \in C$	$\mathsf{C}' = \mathsf{C} + \mathsf{c}$
	Reference *ref* with $src(ref) = c \in C$, $tgt(ref) = d \in D$	$\mathsf{C\_ref}' = \mathsf{C\_ref} + (\mathsf{c} \rightarrow \mathsf{d})$
	Attribute *attr* with $src(attr) = c \in C$, expression e	$\mathsf{C\_attr}' = \mathsf{C\_attr} + (\mathsf{c} \rightarrow expr(e))$

Table 3: Relational formulas generated by function *Post*

Pre checks for the non-existence of an object bound to relational variable c in the current state with i.e., $\exists \mathsf{c} : \mathsf{C}' \wedge \mathsf{c} \nsubseteq \mathsf{C}$, i.e., the formula asserts that the object bound to c is *inactive* in the current state relational variable C. Second, function *Post* generates a condition that adds the new object (bound to c) to relational variable C such that the next state relational variable is set to $\mathsf{C}' = \mathsf{C} + \mathsf{c}$. The procedure for the deletion of an object bound to c is similar except that (i) *Pre* checks for the existence of an object that is scheduled for deletion, i.e. $\exists \mathsf{c} : \mathsf{C}$ and (ii) *Post* updates the next state relational variable to reflect the removal of the object (bound to c), i.e., $\mathsf{C}' = \mathsf{C} - \mathsf{c}$. Addition and deletion of (multiple) objects to and from a relational variable can be combined, i.e., the condition $\mathsf{C}' = \mathsf{C} + \{\mathsf{c}_1, \ldots, \mathsf{c}_m\} - \{\mathsf{c}_{m+1}, \ldots, \mathsf{c}_n\}$ states that C' is equivalent to C except that all objects bound to variables $\mathsf{c}_1, \ldots, \mathsf{c}_m$ are added to C', while objects bound to variables $\mathsf{c}_{m+1}, \ldots, \mathsf{c}_n$ are removed from C. Note that the addition and deletion of references and attributes proceeds analogous to the addition and deletion of objects. For example, *Post* generates for the deletion of a reference *ref* from an object bound to c pointing to an object bound to d the formula $\mathsf{C\_ref}' = \mathsf{C\_ref} - (\mathsf{c} \rightarrow \mathsf{d})$. The formulas generated by *Post* are summarized in Table 3. In addition, the *Post* function also generates conditions for those relational variables that do not change, as otherwise arbitrary tuples could be assigned to these relational variables.

The encodings outlined in Tables 2 and 3 translate a graph transformation $p \in \mathcal{R}$ over *Rel*, which is fixed w.l.o.g to $Rel = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$ for the following explanations, into a relational formula following the scheme outlined in Figure 6. Here, function *match* returns constraints that mimic the transformation's LHS and control the creation of new nodes, while functions *inj* and *dec* generate injectivity constraints over nodes of the transformation's LHS and PACs/NACs and dangling edge conditions over LHS nodes, respectively.

Graph Constraints. In contrast to graph transformations, a graph constraint does not alter a matching host graph; it may thus be used to describe a desired

$$\underbrace{\exists \mathsf{a1} : \mathsf{A}, \exists \mathsf{a2} : \mathsf{A}', \exists \mathsf{b} : \mathsf{B}, \exists \mathsf{c} : \mathsf{C}}_{\text{LHS,RHS, and PAC nodes}}, \underbrace{\neg \exists \mathsf{d} : \mathsf{D}}_{\text{NAC}} \mid$$

$$\underbrace{\mathit{match}(\mathsf{a1}, \mathsf{a2}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\text{match constraints}} \wedge \underbrace{\mathit{inj}(\mathsf{a1}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\substack{\text{injectivity} \\ \text{constraints}}} \wedge \underbrace{\mathit{dec}(\mathsf{a1}, \mathsf{b}, \mathsf{c})}_{\substack{\text{dangling edge} \\ \text{constraints}}} \implies$$

$$\underbrace{\mathsf{A}' = \mathsf{A} - \mathsf{a1} + \mathsf{a2} \wedge \mathsf{B}' = \mathsf{B} - \mathsf{b}}_{\text{modification constraints}} \wedge \underbrace{\mathsf{C}' = \mathsf{C} \wedge \mathsf{D}' = \mathsf{D} \wedge \mathsf{E}' = \mathsf{E}}_{\text{non-modification constraints}}$$

Fig. 6: Scheme of a relational formula produced from a graph transformation

$$\underbrace{\exists \mathsf{a} : \mathsf{A}, \exists \mathsf{b} : \mathsf{B}, \exists \mathsf{c} : \mathsf{C}}_{\text{LHS and PAC nodes}}, \underbrace{\neg \exists \mathsf{d} : \mathsf{D}}_{\text{NAC}} \mid$$

$$\underbrace{\mathit{match}(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\text{match constraints}} \wedge \underbrace{\mathit{inj}(\mathsf{a1}, \mathsf{b}, \mathsf{c}, \mathsf{d})}_{\substack{\text{injectivity} \\ \text{constraints}}}$$

Fig. 7: Scheme of a relational formula produced from a graph constraint

or an undesired pattern in a graph, i.e., a good or a bad state of the system. Thus, graph constraints are graph transformations with identical left-hand and right-hand sides. Formally, a graph constraint $c : \mathit{Cond} \leftarrow \mathit{Lhs}, c \in F$ is translated into a relational formula

$$\mathrm{F}_c := \mathit{Pre}_c(\mathit{Lhs}, \mathit{Pac}, \mathit{Nac}), \tag{2}$$

where function $\mathit{Pre}_c : G \times G \times G \to \mathbb{F}$ translates the triple LHS, PAC, and NAC into a conjunction of relational formulas $f \in \mathbb{F}$. For this purpose, the encodings presented in Table 2 are re-used to translate a graph constraint into a relational formula. The scheme of the relational formula generated from graph constraint $c \in F$ is depicted in Figure 7. It coincides with that of a graph transformation (see Fig. 6) in all but two aspects, the absence of the implication, i.e., there is no RHS, and the dangling edge condition, which is omitted because a graph constraint may not delete elements.

7 Case Studies

In two case studies, we compare our tool Gryphon with the state-of-the-art tool Groove [18]. First, we consider the Dining Philosophers problem of Section 2 and second, we showcase the railway interlocking scenario inspired by [17]. The experiments were run on an Intel$^{\text{TM}}$Core i5 M580 2.67GHz CPU with 8GB of RAM running Gentoo 2.2 (Linux kernel 3.14.14). For the benchmarks we use the Oracle$^{\text{TM}}$Java$^{\text{TM}}$SE 7 Runtime Environment (build 1.7.0_71-b14), the Henshin API in version 0.0.1, and Groove in version 5.5.2 (build: 20150324114640). The

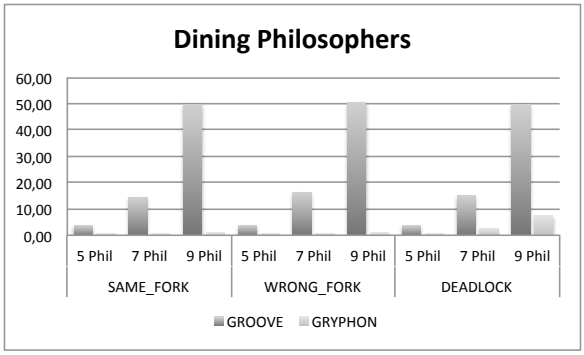


Fig. 8: Runtime comparison (sec) of Groove and Gryphon on the Dining Philosopher Benchmark.

heap size was set to 6GB and the timeout was set to 720 seconds. The runtimes of each benchmark were averaged over 10 consecutive runs.

Case Study 1: Dining Philosophers. For this benchmark with our tool, we modeled the Dining Philosophers problem as described in Section 2. In a similar manner, we modeled the metamodel, i.e., the type graph, used for the implementation of the graph transformation in Groove (see Appendix). Then we formulated the following three invariants: (i) *No two philosophers hold the same fork* (`SAME_FORK`), (ii) *if a philosopher holds a fork, its either her left or her right fork* (`LEFT_RIGHT`), and (iii) *the philosophers do not deadlock* (`DEADLOCK`).

For the benchmarks we use initial models with five, seven, and nine thinking philosophers. We consider the test case, where none of them holds a fork at the beginning. Figure 8 compares the runtimes of our tool with the runtimes of Groove. Interestingly, for Groove we observed a deviation of up to one third of its average runtime. Note that for neither of the tools, we experienced time or memory timeouts. The latter is especially remarkable for Groove as it performs an explicit search of the state space. Gryphon could solve all benchmarks of this case study in less than 10 seconds.

Case Study 2: Interlocking Railway Systems. The second set of benchmarks targets an interlocking railway systems [17]. A railway system is described by a *scheme plan* that consists of a *track plan*, a *control table*, and a set of *release tables*. The topology of the railway network is captured by the *track plan*, which displays tracks and their lengths, entry and exit tracks, and points. A *route* consists of a set of tracks, the first of which may be entered if the guarding signal shows proceed. Each row in the control table is associated with a route and specifies the tracks that need to be *cleared* in order for a train to pass the signal guarding the route. If the route passes a point, the control table specifies the required *position* of the point. A point is locked either in *normal* position, leading the train straight ahead, or in *reverse* position, in which case the train is routed

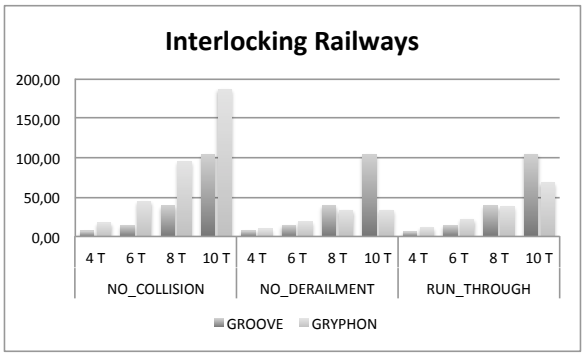


Fig. 9: Runtime comparison (sec) of Groove and Gryphon for the Interlocking Railway Benchmark with 4, 6, 8, and 10 trains (T).

to another line. A train must obtain a *lock* on a point prior to passing it and is required to release it after traversing the point. The *release table* associated with a point specifies the track, where a train must release the acquired lock.

The verification of the railway system then centers around three safety properties: (i) *collision freedom* prohibits two trains occupying the same track; (ii) *no-derailment* demands that a point does not change position while being occupied by a train; (iii)*run-through* requires a point to be set in position as specified by the control table for the specific route when a train is about to enter the point.

For the evaluation, we define four different initial states that instantiate the implementation with four, six, eight, and ten trains. The results are shown in Figure 9. On the `NO_COLLISION` property, Groove is faster by a factor of two to three. For the other two properties, Groove is (slightly) faster for the benchmarks on 4 and 6 trains, but with 8 and 10 trains, the symbolic approach outperforms the explicit approach on average.

8 Conclusion

We presented a novel model checking approach for the verification of graph transformations as used in model-driven engineering. Our approach is a completely automatic approach which allows modelers to benefit from the very efficient hardware model checkers like implementations of the successful IC3 algorithm. In this paper, we explained the internal realization of our tool which translates the modeling artifacts to sequential circuits. In two case studies we showed the potential of our tool by comparing it with Groove, a state-of-the-art model checker for graph transformations.

In future work, we plan to implement optimizations in the encodings from which we expect further improvements in the running times. Further, we plan to implement a visualization component for the witness which are returned in the case a property violation has been detected by our tool.

References

1. Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.
2. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
3. Luciano Baresi, Vahid Rafe, Adel Torkaman Rahmani, and Paola Spoletini. An efficient solution for model checking graph transformation systems. *Electr. Notes Theor. Comput. Sci.*, 213(1):3–21, 2008.
4. Luciano Baresi and Paola Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *Graph Transformations*, volume 4178 of *LNCS*, pages 306–320. Springer, 2006.
5. Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert B. France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.
6. Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and System Modeling*, 11(2):227–250, 2012.
7. Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. Model checking of ctl-extended OCL specifications. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *LNCS*, pages 221–240. Springer, 2014.
8. Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
9. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 1999.
10. Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
11. Edsger W. Dijkstra. Cooperating sequential processes, ewd 123. `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`.
12. Edsger W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, 1972.
13. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., 2006.
14. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
15. Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 130–139. ACM, 2000.
16. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.
17. Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve A. Schneider, and Helen Treharne. On modelling and verifying railway interlockings: Tracking train lengths. *Sci. Comput. Program.*, 96:315–336, 2014.
18. Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In *Model Checking Software*, volume 3925 of *LNCS*, pages 299–305. Springer, 2006.

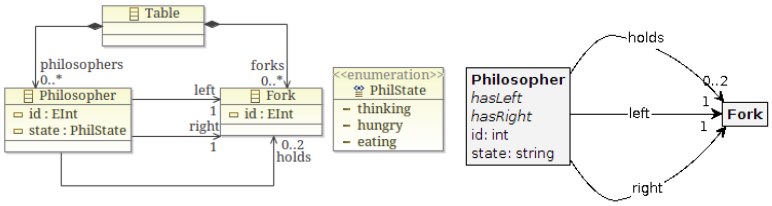
19. D. L. McBurney and M. Ronan Sleep. Graph Rewriting as a Computational Model. In *Concurrency: Theory, Language, And Architecture*, volume 491 of *Lecture Notes in Computer Science*, pages 235–256. Springer, 1989.
20. Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. NATO, 1969.
21. Philipp Niemann, Frank Hilken, Martin Gogolla, and Robert Wille. Assisted generation of frame conditions for formal models. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 309–312, San Jose, CA, USA, 2015. EDA Consortium.
22. Object Management Group OMG. OMG Unified Modeling Language (OMG UML), Infrastructure V2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, August 2011.
23. Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 267–276. ACM, 2003.
24. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
25. Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 - New Features for Specifying and Analyzing Algebraic Graph Transformations. In *Applications of Graph Transformations with Industrial Relevance*, volume 7233 of *LNCS*, pages 81–88. Springer, 2011.
26. Ákos Schmidt and Dániel Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In *UML 2003 - The Unified Modeling Language and Applications*, volume 2863 of *LNCS*, pages 92–95. Springer, 2003.
27. Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, September 2003.
28. Alfred Tarski. On the calculus of relations. *J. Symb. Log.*, 6(3):73–89, 1941.
29. Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, 2009. AAI0821754.
30. Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.
31. Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.
32. Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., 2nd edition, 2009.

A Artifacts of the Case Study

In following, we describe the artifacts used in the two case studies presented in Sec. 7. This include input (meta-)models for our tool and for GROOVE as well as the graph transformations describing the behavior of the system to be verified. Note that this appendix is for reviewing only. The artifacts will be available on our tool homepage.

A.1 Case Study 1: Dining Philosophers

In the first case study we considered the Dining Philosophers problem which served also as our running example in this paper (see Sec. 2). We described the problem in terms of a metamodel (Fig. 1) and the behavior in terms of Henshin graph transformations (Fig 2). Fig.10b shows the metamodel given to GROOVE. Fig. 11 contains the respective transformations for GROOVE.



(a) EMF Model (b) GROOVE Model

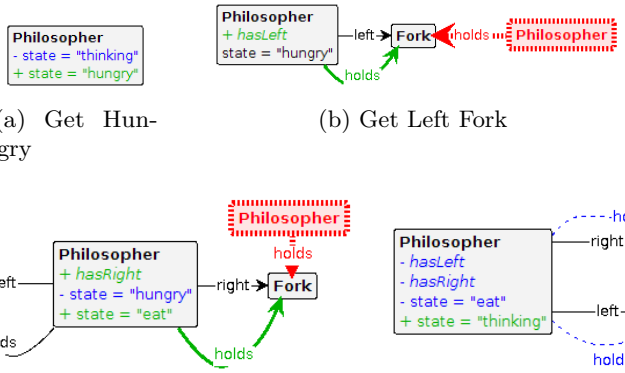
Fig. 10: Dining Philosophers Model

A.2 Details on Implementation of Case Study 2

In the following, we first give some details of the modeled railway system which is based on [17]. A railway system is described by a *scheme plan* that consists of a *track plan*, a *control table*, and a set of *release tables* as depicted in Figure 12. The conditions that change the *aspect* of a signal from red to green, thus allowing a train to pass and continue along the route, are provided by the *control table*.

For the performance evaluation we use the implementation of the station scheme plan that is depicted in Figure 12. We define four different initial states that instantiate the implementation with four, six, eight, and ten trains, and each route is assigned two, three, four, or five trains, respectively. The track plan of the station with four trains is depicted in Figure 13. The EMF and GROOVE metamodels are shown in Fig. 14.

The behavior of the interlocking and the trains is implemented by the graph transformations depicted in Figure 15 and in Figure 16.



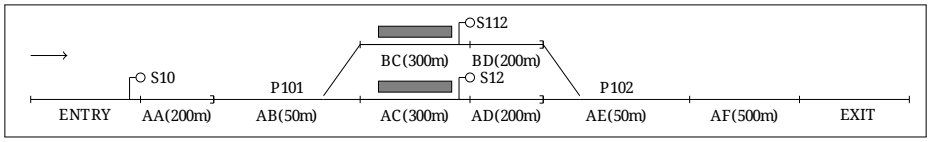
(a) Get Hungry

(b) Get Left Fork

(c) Get Right Fork & Eat

(d) Release Forks

Fig. 11: Graph Transformations for the Dining Philosophers implemented in GROOVE



Control Table

Route	Normal	Reverse	Clear
R10A	P101		AA, AB, AC, AD
R10B		P101	AA, AB, BC, BD
R12	P102		AD, AE, AF
R112		P102	BD, AE, AF

Release tables

P101	Occupied
R10A	AC
R10B	BC

P102	Occupied
R12	AF
R112	AF

Fig. 12: Station scheme plan [17]

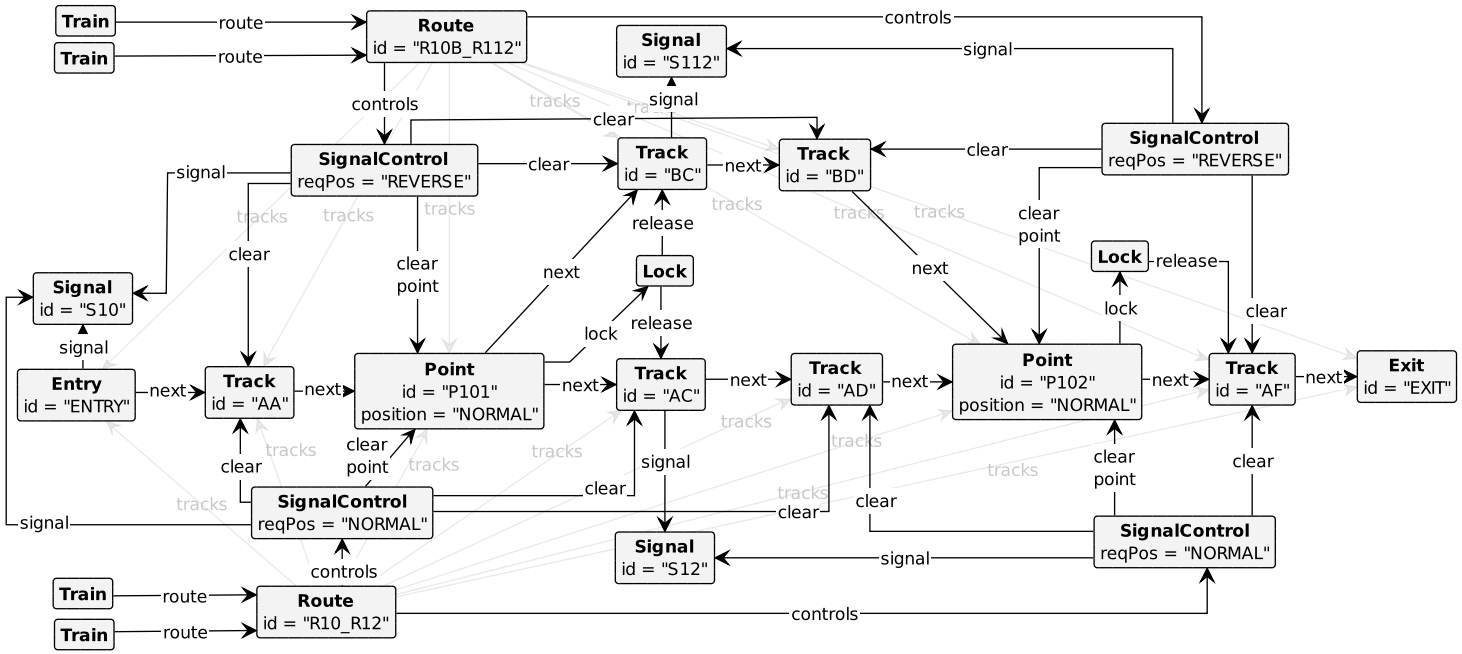
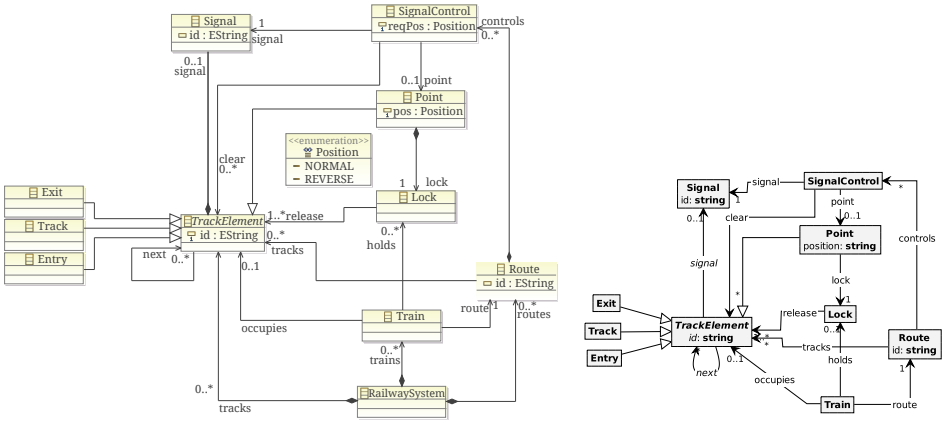
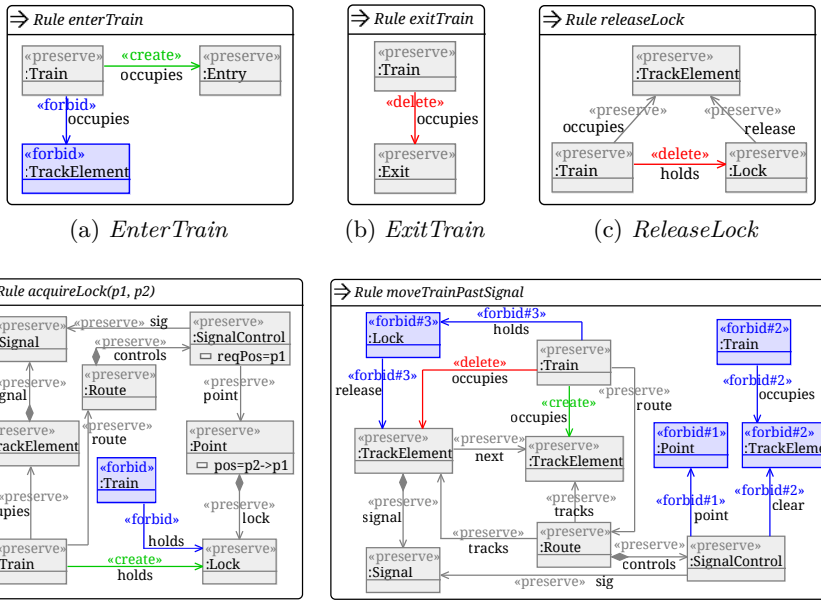


Fig. 13: Initial state of the station scheme plan with four trains modeled with Groove



(a) EMF (b) Groove

Fig. 14: Railway System

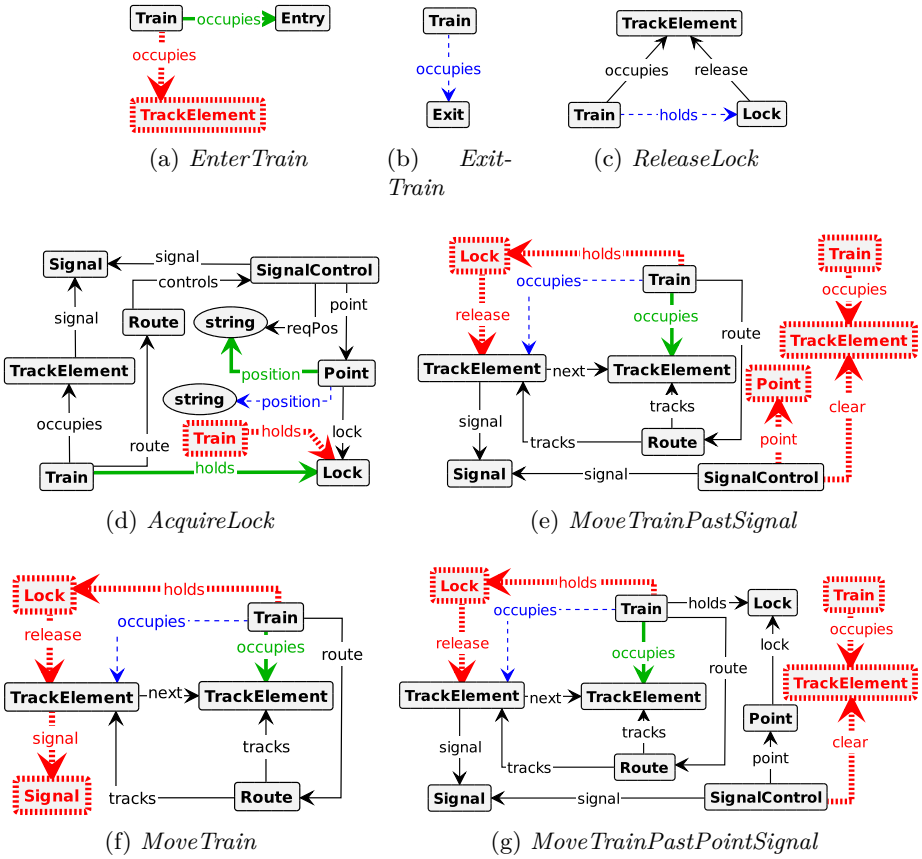


(a) *EnterTrain* (b) *ExitTrain* (c) *ReleaseLock*

(d) *AcquireLock* (e) *MoveTrainPastSignal*

(f) *MoveTrain* (g) *MoveTrainPastPointSignal*

Fig. 15: Behavior of the interlocking and train components modeled with HENSHIN



(a) *EnterTrain* (b) *ExitTrain* (c) *ReleaseLock*

(d) *AcquireLock* (e) *MoveTrainPastSignal*

(f) *MoveTrain* (g) *MoveTrainPastPointSignal*

Fig. 16: Behavior of the interlocking and train components modeled with GROOVE