Bounded Model Checking of Graph Transformation Systems via SMT Solving

Tobias Isenberg, Dominik Steenken, and Heike Wehrheim

{isenberg,dominik,wehrheim}@mail.upb.de
Universität Paderborn
Institut für Informatik
33098 Paderborn, Germany

Abstract. Bounded model checking (BMC) complements classical model checking by an efficient technique for checking error-freedom of bounded system paths. Usually, BMC approaches reduce the verification problem to propositional satisfiability. With the recent advances in SAT solving, this has proven to be a fast analysis.
In this paper we develop a bounded model checking technique for graph transformation systems. Graph transformation systems (GTSs) provide an intuitive, visual way of specifying system models and their structural changes. An analysis of such models – however – remains difficult since GTSs often give rise to infinite state spaces. In our BMC technique we use *first-order* instead of propositional logic for encoding complex graph structures and rules. Today's off-the-shelf SMT solvers can then readily be employed for satisfiability solving. The encoding heavily employs the concept of *uninterpreted function* symbols for representing edge labels. We have proven soundness of the encoding and report on experiments with different case studies.

Keywords: verification, graph transformation systems, bounded model checking, satisfiablility modulo theories

1 Introduction

Graph transformation systems (GTSs) are an intuitive and powerful way of modeling dynamic systems. They describe the states of a system as graphs and state changes as graph transformation rules. This makes them particularly suitable for modeling structural aspects of systems. GTSs have been applied in diverse areas, such as model transformation and refactorings [9], the modeling of dynamic self-adaptive systems [6] or web services [14]. Some of these application areas are safety critical, raising the need for formal verification of such systems. When modeling these systems using GTSs, the analysis often suffers from the state space explosion problem. A number of approaches aim at fighting this problem, most often using *abstraction* techniques [25, 2].

Bounded model checking [8] is a technique avoiding the state space explosion problem by focusing on bounded paths. This technique is incomplete, and thus primarily used for finding errors, not for correctness proofs. This idea has been adapted for the verification of GTSs [17, 3], however, few of the existing solutions directly utilizes the progress of SAT/SMT solvers seen in the last years. In particular, none of the verification techniques for GTSs have proposed to reduce the complete analysis directly to satisfiability checking. An encoding of GTS in logical formulae is only used in [20], which however gives a SAT encoding and uses the result to guide the state space exploration of a (non-SAT) analysis tool for GTS.

In this paper we present a bounded model checking (BMC) technique for GTSs using satisfiability checking of quantifier-free first-order logic with uninterpreted functions (QF_UF). We transform the BMC instance over GTSs into a satisfiability modulo theories (SMT) instance encoding only the initial graph and the transformation rules, as well as the property to be checked.

In the present paper, the property will – for simplicity – always be a (forbidden) graph; thus we just aim at checking reachability of error states. Our approach, detailed in [16], does however cover more general properties specified in LTL. The generated SMT formula describes bounded-length paths of the GTS that exhibit the forbidden graph. The formula is given to an off-the-shelf SMT solver which checks for satisfiability. A satisfying interpretation represents a violating path and is used to display it.

We have implemented our approach and tested it with a number of SMT solvers (MathSAT[12], SMTInterpol[11], veriT[10], Z3[21]). The performance of the approach heavily depends on the structure of rules (e.g., number of nodes, NACs), and only to a minor extent on the solver. We exemplify our technique on the car platooning case study of the Transformation Tool Contest 2010 [1].

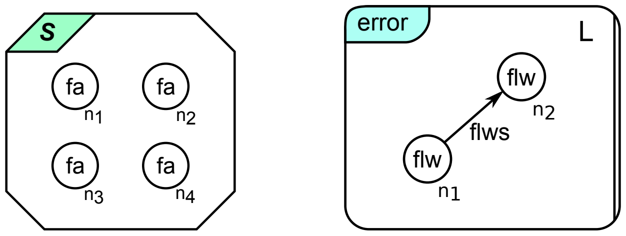
2 Background

We start by defining *Graph Transformation Systems* (GTSs) and the reachability problem of a forbidden pattern in this context, while introducing the running example. The definitions are similar to those in [13, 15, 22].

Definition 1 (Graph). *Let L_E be a set of edge labels and L_N be a set of node labels. A* graph *$G = (N_G, E_G, U_G)$ over L_E and L_N consists of a set N_G of nodes, a set $E_G \subseteq N_G \times L_E \times N_G$ of labeled edges and a set $U_G \subseteq L_N \times N_G$ of unary edges labeling the nodes.*

A graph is a set of nodes with edges between pairs of them and on individual nodes. These edges are labeled over L_E and L_N respectively.

For illustration purposes, consider the car platooning GTS specified by Backes and Reineke [1], modeling a protocol for cars to build a platoon, i.e., a connected entity. Initially the cars are unconnected and in the state *free agent*(*fa*). During the process of connecting there are several intermediate states, as described by the protocol. However, the final states of a car are *leader*(*ld*) and *follower*(*flw*). A platoon is structured such that there are a number of *follower*s following one *leader*.



(a) Initial graph S (b) Forbidden pattern

Fig. 1: Part of the car platooning GTS [1]

In our example this protocol is modeled as a GTS with an initial graph containing four nodes labeled *fa*, modeling cars in state *free agent*(see Figure 1a). Note that we model node labels as "unary" edges, i.e., edges with a target but without a source. Additionally, a GTS has rules that describe possible changes and under which conditions these can be applied. For these transformation rules and their applicability consider the following definitions.

Definition 2. *A* total graph morphism $f : G \to G'$ *from graph* G *to graph* G' *is a tuple* $f = \langle f_N, f_E, f_U \rangle$ *of total functions* $f_N : N_G \to N_{G'}$, $f_E : E_G \to E_{G'}$ *and* $f_U : U_G \to U_{G'}$ *satisfying the following conditions:*

$$\begin{aligned} f_E(n_1, l, n_2) &= (f_N(n_1), l, f_N(n_2)) && \forall (n_1, l, n_2) \in E_G \\ f_U(l, n_1) &= (l, f_N(n_1)) && \forall (l, n_1) \in U_G \end{aligned}$$

A partial graph morphism from graph G to graph G' is a total graph morphism of a subgraph S of G to G'. A total graph morphism specifies a mapping that maintains the structure. Intuitively speaking, such a morphism exists, if a subgraph similar to G can be found in G'. This construct is fundamental for applying graph transformation rules. Such a rule consists of a left hand side graph L and a right hand side graph R, related by a partial graph morphism.

Definition 3 (Transformation Rule and Match). *Let* G *be a graph, called host graph. A* graph transformation rule $r = \langle L, R \rangle$, *described by a partial graph morphism* $r : L \to R$, *is applicable to* G, *if and only if there exists a total, injective graph morphism* $m : L \to G$, *called a* match.

For simplicity, in the following we assume $r = id_{L \cap R}$ (i.e., L and R need not be disjoint, e.g., see the nodes in Figure 2). Intuitively speaking, a match specifies at which location in the graph a rule can be applied. This application itself is done obeying the well known SPO-semantics [13] explained below.

Let G, r and m be as in Def. 3. We define $N_{del} := N_L \setminus N_R$ and N_{add} as a disjoint copy of $N_R \setminus N_L$. By setting $m_N := m \cup id_{N_{add}}$ we extend m onto N_{add}.

The following sets are useful in defining the effect of a rule application.

$$E_{del} := \{(m_N(n_1), l, m_N(n_2)) \mid (n_1, l, n_2) \in E_L \setminus E_R \vee n_1 \in N_{del} \vee n_2 \in N_{del}\}$$
$$E_{add} := \{(m_N(n_1), l, m_N(n_2)) \mid (n_1, l, n_2) \in E_R \setminus E_L\}$$

Corresponding sets U_{del} and U_{add} for unary edges are defined analogously. Each of these sets contains the objects deleted / added through the rule application.

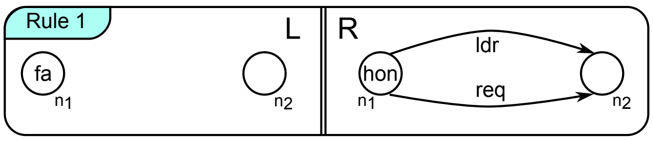


Fig. 2: Rule 1 of the car platooning GTS [1]

This gives rise to the following definition.

Definition 4 (Rule Application). *Given a host graph G, a graph transformation rule $r = \langle L, R\rangle$ and a match $m : L \to G$. The* application of *r to graph G via the match m, written $G \overset{r,m}{\Longrightarrow} H$, creates a result graph H specified by $N_H := N_G \setminus m_N(N_{del}) \cup N_{add}$, $E_H := E_G \setminus E_{del} \cup E_{add}$, and $U_H := U_G \setminus U_{del} \cup U_{add}$.*

The resulting graph is constructed such that the images (w.r.t. the match) of the nodes and edges of the LHS, which are not part of the RHS have to be deleted. Afterwards, new nodes and edges are added for the ones contained in the RHS and not in the LHS. All dangling edges, i.e., edges that are left without a source- or target node after application, are deleted.

Definition 5 (GTS). *A* graph transformation system *$\mathcal{G} = (S, P)$ consists of a start graph S and a set $P = \{r_1, ..., r_n\}$ of graph transformation rules.*

The car platooning GTS [1] consists of 14 rules. For simplicity we will only present the encoding of two of them (*Rules 1 and 13*) illustrated in the next section (see Figures 2,3). The LHS of *Rule 1* requires two distinct nodes in the host graph, one of which is labeled *fa*(*free agent*) and changes its state to *hon*(*hand over nothing*). Additionally, two edges are added as shown in Fig. 2. This rule models the initialization of a connection started by a *free agent* car.

For *Rule 13*, we will need an extension of the above formalism. We need to allow Negative Application Conditions (NACs) within the rules [15]. A NAC constrains the application of a rule, s.t. it is only applicable if the structures described by the NAC are not present in the host graph.

Definition 6 (NAC). *A* negative application condition *for a graph transformation rule $r = \langle L, R\rangle$ is a set C of total graph morphisms with domain L. Let*

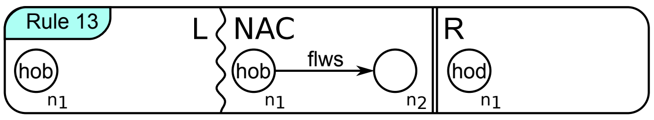


Fig. 3: Rule 13 of the car platooning GTS [1]

$l : L \to \hat{L}$ be such a morphism and let m be a match of r to G. The application of r with match m to the host graph G is allowed w.r.t. l, if and only if no match $n : \hat{L} \to G$ exists with $n \circ l = m$ (concatenation operator $\circ$). A NAC allows the application of r with match m to the host graph G, if and only if the application is allowed w.r.t. all total graph morphisms of the NAC.

For simplicity, in the following we assume l to be an embedding, i.e., $L \subset \hat{L}$. Rule 13 (Figure 3) contains a NAC. It states that the match of the node n_1 must not have a *flws*-labeled edge to any other node. When two platoons join, one leader hands over all his followers to the other leader. While having at least one follower left, he is not allowed to proceed to the next step of the protocol, but must finish the handover first. This is modeled by the NAC in Rule 13. Thus, car n_1 must not have a follower, when changing its state from *hand over back* to *hand over done*. With this understanding of the application of rules, we define the paths of a GTS and the reachability problem.

Definition 7. *Given a GTS $\mathcal{G} = (S, P)$. A state of the GTS is a graph. Thus, a* path *$G_0, G_1, \ldots$ of $\mathcal{G}$ is a finite or infinite sequence of graphs starting with the initial graph ($G_0 = S$) such that we have $G_i \overset{r_i,m_i}{\Longrightarrow} G_{i+1}$ for all consecutive graphs G_i,G_{i+1}. The* state space *of a GTS is the union of all of its paths, joined at isomorphic graphs.*

While usually, GTSs do not distinguish between isomorphic graphs, our encoding does not have that capability. Thus, each isomorphic instance of a graph is explored separately. In the following we state the reachability problem.

Definition 8. *Given a GTS $\mathcal{G}$ and a graph F, called the forbidden pattern. The pattern is* reachable*, if there exists a finite path $G_0, \ldots, G_k$ of $\mathcal{G}$, such that F matches G_k.*

Intuitively speaking, reachability means the existence of a path containing a graph in which the forbidden pattern can be found. Within the car platooning protocol [1], no follower should ever follow a follower. This can be expressed as a forbidden pattern (see Figure 1b). If that pattern would be reachable, the model of the protocol would not meet the desired behavior.

We propose a BMC technique to check reachability of such a forbidden graph. Our approach directly encodes the initial graph, the transformation rules and the forbidden pattern as a satisfiability modulo theories (SMT) instance. SMT describes the problem of searching for a satisfying interpretation of a first-order logic formula with a background theory. Intuitively, a theory can be seen as constraints on the possible interpretations, as it describes a priori interpretations of functions. Our formulae use the theory of uninterpreted functions with equality, i.e., only the equality sign has an a priori interpretation. This approach relocates the computational complexity of exploring the bounded state space into solving an SMT formula, which can be a reasonable trade-off, as current research continuously improves efficiency of SMT solvers (SMT-COMP [4]).

The technique presented here is subject to several restrictions, which we summarize here for clarity.

1. we only allow simple, labeled graphs (no multigraphs, no hypergraphs)
2. we disallow node merging
3. we disallow non-injective matches
4. we only support reachability analysis

Restrictions 3 and 4 are removed in our full technique, detailed in [16]. In the following, we will only give the definitions required for the restricted version.

3 Encoding of GTSs in first-order logic

In the following we describe our BMC technique, which encodes bounded paths of a GTS (S, P) as an SMT formula to check the reachability of a forbidden pattern in the bounded state space. The concept of BMC via SAT-solving was originally introduced in 1999 [8]. In hardware verification it is a natural choice to represent circuits as boolean formulas. This idea was then extended to check error freedom of a bounded subset of the state space. The state space is only examined up to a predefined number k of steps, called the bound. The initial state, k transitions and the error are encoded as a boolean formula, that is satisfiable if the error exists within the bounded state space.

Our approach likewise encodes the start graph, k transitions and a forbidden pattern (or LTL formula). A transition in this context denotes the application of one of the graph transformation rules of the GTS.

Let $[\![S]\!]$ be the encoding of the initial graph $S = G_0$ and $[\![T]\!]_1$ to $[\![T]\!]_k$ be the encodings of the k transition steps. The complete formula is defined as $[\![S]\!] \wedge [\![T]\!]_1 \wedge ... \wedge [\![T]\!]_k \wedge [\![F]\!]$ with $[\![F]\!]$ being the encoding of the forbidden pattern.

Below we describe the process in detail. Let L_E be the set of all edge labels occurring in the initial graph, the rules or the forbidden pattern and let L_N be the set of all node labels, respectively. For representing all edges with label $l \in L_E$ of a graph $G_i(i \in \{0, ..., k\})$ along the path $G_0, ..., G_k$ we use binary predicates l_i. For tagging the nodes with label $l \in L_N$ we use unary predicates, respectively. Let $\mathcal{U}$ be the universe consisting of the nodes of the initial graph. Unary predicates have domain $\mathcal{U}$, binary ones have domain $\mathcal{U} \times \mathcal{U}$.

Definition 9. *Let $L_{E,i} = \{l_i | l \in L_E\}$ and $L_{N,i} = \{l_i | l \in L_N\}$ be the sets of binary and unary predicates with index i. An i-*interpretation *$\mathcal{I}_i$ assigns a total function with image true or $false$ to every such predicate. Formally, $\mathcal{I}_i$:*

- $L_{E,i} \rightarrow \{\mathcal{U} \times \mathcal{U} \rightarrow \{true, false\}\}$

– $L_{N,i} \rightarrow \{\mathcal{U} \rightarrow \{true, false\}\}$.

An i-interpretation $\mathcal{I}_i$ *represents a graph* G_i *(written* $\mathcal{I}_i(G_i)$*), iff:*

– $\forall l \in L_E \, \forall n_1, n_2 \in N_{G_i} : \;\; \mathcal{I}_i(l_i)(n_1, n_2) = true \Leftrightarrow (n_1, l, n_2) \in E_{G_i}$
– $\forall l \in L_N \, \forall n \in N_{G_i} : \;\; \mathcal{I}_i(l_i)(n) = true \Leftrightarrow (l, n) \in U_{G_i}$.

This idea of how a satisfying interpretation represents a graph is used throughout the complete encoding process.

Encoding of the initial graph. Given the initial graph $S = G_0 = (N_{G_0}, E_{G_0}, U_{G_0})$, we encode each edge $(n_1, l, n_2) \in E_{G_0}$ with a positive literal $l_0\,(n_1, n_2)$ and each nonexistent edge $(n_1, l, n_2) \in (\mathcal{U} \times L_E \times \mathcal{U}) \setminus E_{G_0}$ with a negated literal $\neg l_0\,(n_1, n_2)$. Unary edges are handled analogously. Furthermore, we include literals $\neg\,(n_i = n_j)$ for all pairings of distinct nodes n_i, n_j. This ensures that the graph itself and not a smaller graph it would be homomorphic to is encoded, and provides support for isolated nodes. These literals are combined via conjunction.

Example 1. Our example GTS has an initial graph as shown in Figure 1a and two transformation rules. These can be seen in Figures 2 and 3. The sets of labels in the example GTS are $L_E = \{\mathit{flws}, \mathit{ldr}, \mathit{req}\}$ and $L_N = \{\mathit{fa}, \mathit{hob}, \mathit{hod}, \mathit{hon}, \mathit{flw}\}$. $\mathcal{U}$ consists of the nodes from the start graph, called n_1 to n_4. The encoding of the initial graph is straightforward as there are no binary edges in the initial graph and the nodes are only labeled with *fa*:

$$[\![G_0]\!] = \bigwedge_{i=1}^{4} \mathit{fa}_0\,(n_i) \wedge \bigwedge_{\substack{l \in L_N \\ l \neq fa}} \bigwedge_{i=1}^{4} \neg l_0\,(n_i) \wedge \bigwedge_{\substack{i=1 \\ j=1}}^{4} \bigwedge_{l \in L_E} \neg l_0\,(n_i, n_j) \wedge \bigwedge_{\substack{i=1 \\ j=1 \\ i \neq j}}^{4} \neg\,(n_i = n_j)$$

The following lemma states the correctness of this encoding.

Lemma 1. *Given a graph* G*,* $[\![G]\!]$ *is satisfiable only with* $\mathcal{I}_0(G)$ *representing* G*.*

Encoding of a transition. The i-th transition ($i \in \{1, ..., k\}$) can be caused by the application of any of the graph transformation rules to the host graph G_{i-1}. For a GTS with the set $P = \{r_1, ..., r_n\}$ of rules, the i-th transition is encoded as $[\![T]\!]_i = [\![r_1]\!]_i \vee ... \vee [\![r_n]\!]_i$ with $[\![r]\!]_i = [\![r]\!]_i^{Cond} \wedge [\![r]\!]_i^{App}$. Thus, we have to encode the applicability and the application of each rule for each transition step.

Encoding $[\![r]\!]_i^{Cond}$ *of the applicability of a transformation rule* r*.* We first encode the check whether a rule is applicable to a host graph G_j (Note that $j = i - 1$ for the i-th transition). For each node $n \in N_L$ create a variable m_n^j, to which we want the SMT-Solver to assign a node from the universe $\mathcal{U}$. In general, SMT uses sorts with an unbounded number of members. Thus, we have to assure that only nodes out of $\mathcal{U}$ are used. We do this with the disjunction $(m_n^j = n_1) \vee ... \vee (m_n^j = n_2)$ for $\mathcal{U} = \{n_1, ..., n_2\}$. The assignment of the variables to the actual nodes in the universe represents the match. Since the match must be injective, for each two distinct nodes $n_1, n_2 \in N_L$ the literal $\neg\left(m^j_{n_1} = m^j_{n_2}\right)$ is added. In addition, each edge $(n_1, l, n_2) \in E_L$ of the LHS has to exist within the matched nodes of G_j. This is encoded by the literal $l_j\left(m^j_{n_1}, m^j_{n_2}\right)$. The analog holds true for the unary edges. Thus, we encode a unary edge $(l, n_1) \in U_L$ by a literal $l_j\left(m^j_{n_1}\right)$. All these literals are combined via conjunction.

Encoding of a NAC. If the transformation rule has Negative Application Conditions, these are encoded and added to the conjunction as well. Each total graph morphism of the NAC is encoded separately and combined afterwards by conjunction. Given such a morphism $L \to \hat{L}$, a variable m^j_n is created for each node $n \in \hat{L} \setminus L$. These new variables are bound by a universal quantifier and have to be distinct from each other and the previously generated variables of this rule. The quantified formula for an injective match is true if the negation of the conjunction of literals representing the binary and unary edges specified in $\hat{L} \setminus L$ holds ($l_j\left(m^j_{n_1}, m^j_{n_2}\right)$ for a given edge $(n_1, l, n_2) \in E_{\hat{L}} \setminus E_L$ and $l_j\left(m^j_n\right)$ for a unary edge $(l, n) \in U_{\hat{L}} \setminus U_L$). Within this negation we ensure an injective match with the tests that each newly created variable is disjoint from all other variables ($m^j_{n_1} = m^j_{n_2}$ for $n_1 \in \hat{L} \setminus L$ and $n_2 \in \hat{L}$).

Example 2. We continue the example and choose $k = 2$. We encode the injective applicability of both transformation rules (see Figures 2, 3) for the first transition step. (The encoding for the second transition step only differs by the indices.)

$$\begin{aligned}
[\![Rule1]\!]^{Cond}_1 =& fa_0\left(m^0_{n_1}\right) \wedge \neg\left(m^0_{n_1} = m^0_{n_2}\right) \wedge \bigvee_{n\in\mathcal{U}} \left(m^0_{n_1} = n\right) \wedge \bigvee_{n\in\mathcal{U}} \left(m^0_{n_2} = n\right) \\
[\![Rule13]\!]^{Cond}_1 =& hob_0\left(m^0_{n_1}\right) \wedge \bigvee_{n\in\mathcal{U}} \left(m^0_{n_1} = n\right) \wedge \\
& \forall m^0_{n_2} \in \mathcal{U} : \neg\left(\neg\left(m^0_{n_2} = m^0_{n_1}\right) \wedge \left(flws_0\left(m^0_{n_1}, m^0_{n_2}\right)\right)\right)
\end{aligned}$$

Note that, while we use quantifiers here to make the presentation more concise, these can be (and are) eliminated in the encoding by explicit enumeration of their range, thus making the resulting formula quantifier-free.

Lemma 2. *Given a graph transformation rule $r = \langle L, R\rangle$ and a host graph G_j. $[\![r]\!]^{Cond}_{j+1}$ is satisfiable with the interpretation $\mathcal{I}_j(G_j)$ representing G_j iff there exists a match $m : L \to G_j$ and the assignment of the variables is according to that match ($m^j_n = m_N(n)$).*

In the following we describe the encoding of the actual application of a graph transformation rule. Note, that we do not describe graph transformation rules with node creation/deletion here to keep illustration simple. Nevertheless, at the end of this section we briefly discuss these features.

Encoding $[\![r]\!]^{App}_i$ of the application of a transformation rule r. Let graph G_j ($j = i-1$) be the host graph as before. We use the previously created variables. Added binary edges $(n_1, l, n_2) \in E_R \setminus E_L$ have to exist in the resulting graph, encoded by a literal $l_{j+1}\left(m^j_{n_1}, m^j_{n_2}\right)$. Analogously, added unary edges are encoded by the corresponding positive unary literal. Similarly, deleted edges must not be present in the resulting graph and are encoded via the corresponding negated literals.

All other edges, which are neither deleted, nor added, exist in the resulting graph G_{j+1} if and only if they exist in the host graph G_j. This is represented by a universally quantified equivalence for each pair of two nodes from $\mathcal{U}$ from which we exclude the changed edges. Let $\{(n_1, l, n_2), ..., (n_3, l, n_4)\} \subseteq (E_R \setminus E_L \cup E_L \setminus E_R)$ be the set of changed edges with label l, which we need to exclude. For each label $l \in L_E$ we encode as follows. (The same holds true analogously for unchanged unary edges. All these parts are combined by conjunction.)

$$\begin{aligned}
\forall n_s, n_t \in \mathcal{U} :&(l_j\left(n_s, n_t\right) \Leftrightarrow l_{j+1}\left(n_s, n_t\right) && \text{same interpretation}\\
&\vee \left(\left(n_s = m^j_{n_1}\right) \wedge \left(n_t = m^j_{n_2}\right)\right) \vee ... && \text{except for all the}\\
&\vee \left(\left(n_s = m^j_{n_3}\right) \wedge \left(n_t = m^j_{n_4}\right)\right)). && \text{changed edges}
\end{aligned}$$

Example 3. We continue the example. As mentioned before, the encodings for the second transition step differs only by the indices.

$$\begin{aligned}
[\![Rule1]\!]^{App}_1 =&hon_1\left(m^0_{n_1}\right) \wedge ldr_1\left(m^0_{n_1}, m^0_{n_2}\right) \wedge req_1\left(m^0_{n_1}, m^0_{n_2}\right) \wedge \neg fa_1\left(m^0_{n_1}\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : \left(flws_0\left(n_s, n_t\right) \Leftrightarrow flws_1\left(n_s, n_t\right)\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : \left(\left(ldr_0\left(n_s, n_t\right) \Leftrightarrow ldr_1\left(n_s, n_t\right)\right) \vee\right.\\
&\qquad\qquad \left.\left(\left(n_s = m^0_{n_1}\right) \wedge \left(n_t = m^0_{n_2}\right)\right)\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : \left(\left(req_0\left(n_s, n_t\right) \Leftrightarrow req_1\left(n_s, n_t\right)\right) \vee\right.\\
&\qquad\qquad \left.\left(\left(n_s = m^0_{n_1}\right) \wedge \left(n_t = m^0_{n_2}\right)\right)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(\left(fa_0(n) \Leftrightarrow fa_1(n)\right) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(hob_0(n) \Leftrightarrow hob_1(n)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(hod_0(n) \Leftrightarrow hod_1(n)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(\left(hon_0(n) \Leftrightarrow hon_1(n)\right) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(flw_0(n) \Leftrightarrow flw_1(n)\right)
\end{aligned}$$

$$\begin{aligned}
[\![Rule13]\!]^{App}_1 =&\neg hob_1\left(m^0_{n_1}\right) \wedge hod_1\left(m^0_{n_1}\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : \left(flws_0\left(n_s, n_t\right) \Leftrightarrow flws_1\left(n_s, n_t\right)\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : \left(ldr_0\left(n_s, n_t\right) \Leftrightarrow ldr_1\left(n_s, n_t\right)\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : \left(req_0\left(n_s, n_t\right) \Leftrightarrow req_1\left(n_s, n_t\right)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(fa_0(n) \Leftrightarrow fa_1(n)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(\left(hob_0(n) \Leftrightarrow hob_1(n)\right) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(\left(hod_0(n) \Leftrightarrow hod_1(n)\right) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(hon_0(n) \Leftrightarrow hon_1(n)\right)\\
&\wedge \forall n \in \mathcal{U} : \left(flw_0(n) \Leftrightarrow flw_1(n)\right)
\end{aligned}$$

Lemma 3. *Given a graph transformation rule $r = \langle L, R\rangle$ and a host graph G_j. $[\![r]\!]^{Cond}_{j+1} \wedge [\![r]\!]^{App}_{j+1}$ is satisfiable with an interpretation representing graphs G_j and G_{j+1} (i.e., $\mathcal{I}_j(G_j)$ and $\mathcal{I}_{j+1}(G_{j+1})$), iff $G_j \overset{r,m}{\Longrightarrow} G_{j+1}$ holds and the assignment of the variables is according to that match ($m^j_n = m_N(n)$).*

Encoding of the forbidden pattern. In order to check reachability of the forbidden pattern for our path of length k, we have to check its applicability to G_k. Thus, we encode the reachability check for the pattern F by $[\![F]\!]^{Cond}_k$.

Dynamics of nodes. For illustration purposes, we omitted the aspects of added and deleted nodes. These are more difficult to handle, as they can have many side effects. We summarize the additional encoding efforts necessary. Given that we cannot handle a changing universe in our formula, we need a universe $\mathcal{U}$ that can capture all possibilities. Thus, we search the transformation rule with the highest number *max* of new nodes and add $k \cdot max$ new nodes to $\mathcal{U}$. This trick enables us to apply k transitions without the need to change the universe.

Of course this introduces the new problem of having "potential" nodes in $\mathcal{U}$ that do not actually belong to the graph. To solve this problem, we introduce the *dead*-predicate, initially valued 1 for all potential nodes and 0 for all nodes in the initial graph. The label set, the rules (including NACs) and their encodings can be automatically adjusted in a fairly straightforward manner to emulate the expected semantics of that predicate. In addition, one has to encode the deletion of all edges adjacent to deleted nodes to ensure the non-existence of dangling edges. For details, please refer to [16].

The central theorem of this paper states the correctness of our approach.

Theorem 1. *Given a GTS $\mathcal{G} = (S, P)$, a forbidden pattern F and a bound k. The encoding $[\![S]\!] \wedge [\![T]\!]_1 \wedge ... \wedge [\![T]\!]_k \wedge [\![F]\!]^{Cond}_k$ is satisfiable iff F is reachable in $\mathcal{G}$ in k steps. The satisfying interpretation represents graphs G_0 to G_k (i.e., $\mathcal{I}_i(G_i)$) in a path $G_0, ..., G_k$ of $\mathcal{G}$ containing the forbidden pattern.*

As the satisfying interpretation represents the graphs along a path with the forbidden pattern, we can directly use this to present this path.

From Reachability to LTL. The version presented above of encoding paths of length k of a graph transformation system lacks the ability of describing finite paths of length smaller k. If a path has length $i < k$, then there is no transformation rule applicable to host graph G_i. Thus, the encoding $[\![T]\!]_{i+1}$ of the transition $i + 1$ is not satisfiable, which means the path cannot be found and therefore not be checked for reachability. In consequence of this, we have to iteratively check all paths of length i with $1 \leq i \leq k$. This however forces the satisfiability checking of up to k instances, resulting in a long runtime. To overcome this issue, we introduce a self-loop, which is only applicable, if and only if no transformation rule is applicable. In order to encode this self-loop, we developed an encoding of the non-applicability of a rule. The self-loop enables a path of length smaller than k to be represented by a path of length k. As strictly smaller paths are not checked separately, we adapt the encoding of the forbidden pattern to $[\![F]\!]_0^{Cond} \vee ... \vee [\![F]\!]_k^{Cond}$. Thus, we do not need to check for strictly smaller paths. We will compare both approaches in the next section.

This self-loop and the encoding of the non-applicability allow us to completely encode the BMC of LTL formulae. The semantics and the encoding of the LTL formula are the same as for standard BMC of Kripke structures, except that atomic propositions here mean the applicability of a pattern and negated atomic propositions mean the non-applicability. In general, our technique is slightly more expressive than LTL, as we can check whether a transformation rule actually was applied in a transition, not just its potential applicability to a host graph.

Theorem 1 describes the iterative version for a forbidden pattern. All previously mentioned lemmata and a more general theorem for LTL formulae without iterative checking are described and proved by Isenberg [16].

4 Implementation and Evaluation

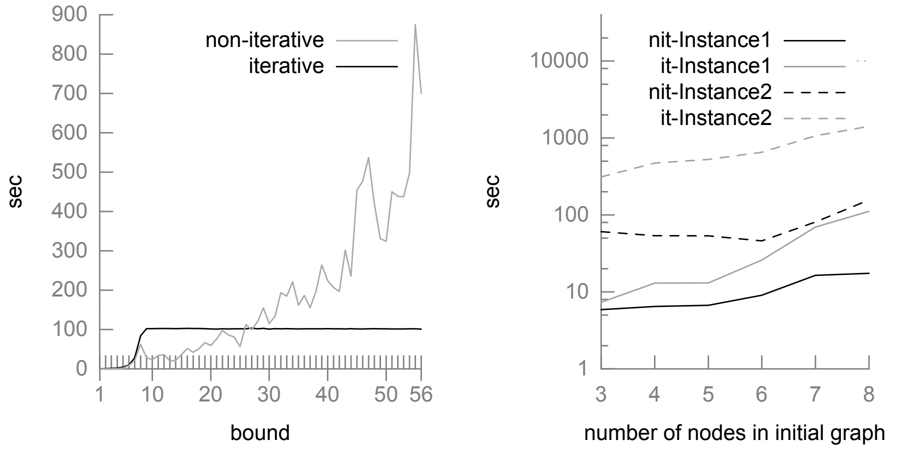
In the following, we shortly describe our implementation and present results of the evaluation of our technique.

We built a prototypical implementation of our BMC technique as an extension to GROOVE [24], which can generate the encoding in two ways, the first of which is described in this paper. The second encoding does not use different *predicates* (numbering) to differentiate between the graphs in different steps of a path, but different *universes*. A first result of the evaluation is that these encodings are basically equally well suited; none can outperform the other. In both cases the result of the encoding is an SMT formula written according to the SMT-Lib v2.0 format [5], which is then fed into the SMT solver Z3 [21]. If a satisfiable interpretation is found, the represented counterexample is displayed.

We tested our approach on several examples, including the car platooning GTS [1]. The forbidden pattern used by this example is shown in Figure 1b and does not occur within the state space of the original GTS. To check our approach, we also created a faulty specification of car platooning, essentially by omitting the NAC of rule 13 (Instance 1). With this rule modified, the graph transformation system yields several paths containing the forbidden pattern. To further increase the complexity of verification, we added a rule creating new cars as to make the state space infinite and to have an instance with node dynamics (Instance 2). These instances have 15, respectively 16 transformation rules.

On these two instances, we compare the iterative (*it*) reachability check and the non-iterative one (*nit*) using the self-loop. For this, we use several initial graphs with different numbers of nodes (see Figure 4b). In addition, we compared both the iterative and non-iterative reachability check for different bounds using Instance 1 with an initial graph consisting of 8 nodes (see Figure 4a). Note, that the shortest path containing the forbidden pattern has length 9.

Our technique works most efficiently in its non-iterative version when the guess of the bound is close to the size of the counterexample. If the bound is



(a) Instance 1 with 8 nodes in initial graph (b) Different Instances with bound 10

Fig. 4: Results of our experiments

chosen much higher than the length of the smallest counterexample, the iterative can outrun the non-iterative version, as it only evaluates paths up to the length of the smallest counterexample (see Figure 4a). One further observation gained from looking at different case studies is that the order of the encodings of the transformation rules matters (with respect to solving time), in particular when having only one path containing the forbidden pattern in a huge set of paths.

We also compared our tool with GROOVE's full state space exploration as well as its BMC technique. The BMC technique in its version as online in March 2013 cannot (strangely) deal with the car platooning GTS of Instance 2 which is infinite state. Unfortunately, we do not see the reason for this, and thus a meaningful comparison is not possible. A comparison with GROOVE's full state space exploration shows that GROOVE is faster for the finite state Instance 1, but – because of the very nature of the exploration technique – cannot deal with the infinite state Instance 2. The conclusion of our evaluation is thus that our technique can be seen as a complement to existing approaches, especially useful for bug finding in GTSs with infinite state spaces. We moreover think that our approach will prove its strength in cases when the error path is already approximately known, for instance when checking for spurious counterexamples.

5 Related Work

BMC [8] is a standard technique originally from the field of circuit verification. It unfolds the transition relation up to a predefined boundary to check for errors. One of its main concepts is the encoding of the problem as a SAT instance.

While the encoding of problems as a SAT/SMT problem is a well known standard technique in the field of planning [18, 26], this is not the case for graph transformation systems. However, there are some approaches transforming GTSs into the planning language PDDL [29] to use recent progress in that field.

There is also research focusing on transformations into other target logics to verify graph properties, e.g., using rewriting logic [30, 7].

However, some approaches use only the concept of bounding the state space without the encoding as a SAT problem. Kastenberg [17] constructs a Büchi automaton on-the-fly and uses a nested-DFS algorithm to check non-emptiness by iteratively searching the state space up to predefined boundaries.

Baresi [3] transforms a given GTS into Alloy, a simple structural modeling language, which is then automatically transformed into a propositional formula. The generated formula is afterwards fed into a SAT-solver. This approach, however, doesn't utilize SAT-solving directly, but uses an intermediate tool.

Another approach using propositional formulae is the one presented by Kreowski [20], which is very similar to ours, but differs in some major points. We use an encoding in first-order predicate logic, allowing us to represent the graph transformation system in a more compact and readable manner. This can be of great help with respect to planned future research, where we are interested in interpolation. In addition, the representation as SMT-formula gives us the ability to let the SMT-solver handle the matches itself. In contrast, Kreowski enumerates all possible matches in his encoding. The overhead of using SMT instead of SAT should be small for the used theory. Thus, our approach can be fast if the solver is able to restrict the search space quickly.

Other approaches to solving the problem with state space explosion and infinite growth focus on overapproximation, rather than on bounded state spaces (underapproximation). The upside to this is that positive results extend to the entire state space, rather than just a small subset that has been examined. The downside is that negative results might not be definite and there can be no guarantee of termination without excluding some types of inputs.

Into this class of approaches falls the work by Barbara König et al. [19], which uses a combined formalism of graphs and Petri nets. Other overapproximation approaches, inspired by shape analysis [27], use a more direct encoding of abstraction. Examples are the work by Rensink and Zambon [23, 31] as well as Steenken, Wonisch and Wehrheim [28].

Another approach that performs a kind of overapproximation is given by Giese, Beyer et al. [6]. Here the idea is to prove inductive invariants of GTS by starting at the error pattern and working backwards.

6 Conclusion

In this paper, we presented a technique for BMC of graph transformation systems via SMT solving. To this end we encoded the reachability problem of a forbidden pattern in a GTS as an SMT formula. The presented approach can easily be extended to cover properties specified in LTL, and we have already implemented this extension as well.

One idea for further research could be to optimize the way of handling new nodes in transformation rules. In addition, one could try to find good heuristics for arranging the encodings of the transformation rules to improve the speed even further. Our main objective for the future is, however, the usage of this approach for a fast counter example analysis in our shape analysis based verification technique for GTS [28].

References

1. Backes, P., Reineke, J.: A graph transformation case study for the topology analysis of dynamic communication system. In: Transformation Tool Contest 2010. CTIT Workshop Proceedings, vol. WP10-03, pp. 107–118. University of Twente (2010)
2. Baldan, P., König, B., Rensink, A.: Summary 2: Graph grammar verification through abstraction. In: König, B., Montanari, U., Gardner, P. (eds.) Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems. Dagstuhl Seminar Proceedings, vol. 04241 (2004)
3. Baresi, L., Spoletini, P.: On the use of Alloy to analyze graph transformation systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT. LNCS, vol. 4178, pp. 306–320. Springer (2006)
4. Barrett, C., Deters, M., de Moura, L., Oliveras, A., Stump, A.: 6 Years of SMT-COMP. Journal of Automated Reasoning pp. 1–35 (2012), 10.1007/s10817-012-9246-5
5. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
6. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE. pp. 72–81. ACM (2006)
7. Bergmann, G., Boronat, A., Heckel, R., Torrini, P., Ráth, I., Varró, D.: Advances in model transformations by graph transformation: Specification, execution and analysis. In: Wirsing, M., Hölzl, M. (eds.) Rigorous Software Engineering for Service-Oriented Systems, LNCS, vol. 6582, pp. 561–584. Springer Berlin Heidelberg (2011)
8. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
9. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE. Lecture Notes in Computer Science, vol. 4961, pp. 347–361. Springer (2008)
10. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient smt-solver. In: Schmidt, R. (ed.) Automated Deduction – CADE-22, LNCS, vol. 5663, pp. 151–156. Springer Berlin Heidelberg (2009)
11. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. In: Donaldson, A.F., Parker, D. (eds.) SPIN. LNCS, vol. 7385, pp. 248–254. Springer (2012)
12. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) Proceedings of TACAS. LNCS, vol. 7795. Springer (2013)

13. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars. pp. 247–312. World Scientific (1997)
14. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring consistency of business process models and web services using visual contracts. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE. LNCS, vol. 5088, pp. 17–31. Springer (2007)
15. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. 26(3/4), 287–313 (1996)
16. Isenberg, T.: Bounded Model Checking für Graphtransformationssysteme als SMT-Problem. Master's thesis, University of Paderborn, Germany (2012)
17. Kastenberg, H.: Graph-based software specification and verification. Ph.D. thesis, University of Twente, Enschede (October 2008)
18. Kautz, H.A., Selman, B.: Planning as satisfiability. In: ECAI. pp. 359–363 (1992)
19. König, B., Kozioura, V.: Augur 2 - a new version of a tool for the analysis of graph transformation systems. Electronic Notes in Theoretical Computer Science 211(0), 201 – 210 (2008), proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)
20. Kreowski, H.J., Kuske, S., Wille, R.: Graph Transformation Units Guided by a SAT Solver. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT. Lecture Notes in Computer Science, vol. 6372, pp. 27–42. Springer (2010)
21. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
22. Rensink, A.: The joys of graph transformation. Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica 9 (2005)
23. Rensink, A., Zambon, E.: Neighbourhood abstraction in GROOVE. Electronic Communications of the EASST 32 (2011)
24. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE. LNCS, vol. 3062, pp. 479–485. Springer (2003)
25. Rensink, A., Zambon, E.: Pattern-based graph abstraction. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) ICGT. LNCS, vol. 7562, pp. 66–80. Springer (2012)
26. Rintanen, J.: Planning and sat. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 483–504. IOS Press (2009)
27. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (May 2002)
28. Steenken, D., Wehrheim, H., Wonisch, D.: Sound and Complete Abstract Graph Transformation. In: Brazilian Symposium on Formal Methods. pp. 92–107 (2011)
29. Tichy, M., Klöpper, B.: Planning self-adaption with graph transformations. In: Schürr, A., Varró, D., Varró, G. (eds.) Applications of Graph Transformations with Industrial Relevance, LNCS, vol. 7233, pp. 137–152. Springer (2012)
30. Vandin, A., Lluch-Lafuente, A.: Towards a maude tool for model checking temporal graph properties. ECEASST 41 (2011)
31. Zambon, E., Rensink, A.: Graph subsumption in abstract state space exploration. arXiv preprint arXiv:1210.6413 (2012)