k-Inductive Invariant Checking for Graph Transformation Systems

Johannes Dyck, Holger Giese

Technische Berichte Nr. 119

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 119

Johannes Dyck | Holger Giese

k-Inductive Invariant Checking for Graph Transformation Systems

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2017
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-406-7

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URN urn:nbn:de:kobv:517-opus4-397044
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-397044

While offering significant expressive power, graph transformation systems often come with rather limited capabilities for automated analysis, particularly if systems with many possible initial graphs and large or infinite state spaces are concerned. One approach that tries to overcome these limitations is inductive invariant checking. However, the verification of inductive invariants often requires extensive knowledge about the system in question and faces the approach-inherent challenges of locality and lack of context.

To address that, this report discusses k-inductive invariant checking for graph transformation systems as a generalization of inductive invariants. The additional context acquired by taking multiple (k) steps into account is the key difference to inductive invariant checking and is often enough to establish the desired invariants without requiring the iterative development of additional properties.

To analyze possibly infinite systems in a finite fashion, we introduce a symbolic encoding for transformation traces using a restricted form of nested application conditions. As its central contribution, this report then presents a formal approach and algorithm to verify graph constraints as k-inductive invariants. We prove the approach's correctness and demonstrate its applicability by means of several examples evaluated with a prototypical implementation of our algorithm.

1. Introduction

The expressive power of graph transformation systems often leads to rather limited capabilities for automated analysis, particularly if systems with many initial graphs and large or infinite state spaces are concerned. Model checkers can typically only be employed for the analysis of graph transformation systems with a finite state space of moderate size (e.g., [9, 14]). Other fully automatic approaches that can handle infinite state spaces by abstraction [2, 3, 11, 12, 16] are limited in their expressiveness, supporting only limited forms of negative application conditions at most. In some cases, additional limitations concerning the graphs of the state space apply (cf. [2]). In contrast to that, the SeekSat/ProCon tool [10, 13] is able to prove correctness of graph programs with respect to pre- and postconditions specified as nested graph constraints; however, it may require too expensive (cf. [5]) or infeasible computations.

One direction that tries to overcome these limitations is the automated verification of *inductive invariants* (cf. our own work [1, 5]), where we analyze the capability of system behavior (captured by a number of graph rules) to preserve or violate desired properties (captured by graph constraints) as inductive invariants. However, the technique faces the approach-inherent challenges of locality and lack of context information. The analysis of single transformation steps does not take the broader context, prior rule applications, or the state space into account, which is both the primary objective and a main challenge of the approach. Hence, in order to develop successfully verifiable inductive invariants (if the system is, indeed, safe) or establish meaningful counterexamples (if the system is not), additional knowledge encoded by additional properties may be required and often has to be accumulated by an iterative and manual procedure.

Therefore, this report applies the notion of *k-induction* [15] to graph transformation systems by extending our previous work in [1] and [5]. In particular, k-inductive invariants are a generalization of inductive invariants; conversely, the latter are a special case of the former for $k = 1$. Our approach takes paths of length k into account [15]: a k-inductive invariant is a property whose validity in a path of length $k - 1$ implies its validity in the subsequent step. By analyzing system behavior over multiple transformation steps, more context information is available and the resulting analysis will be more precise. While the idea of k-induction has been successfully employed in the field of software verification [4], to the best of our knowledge, no approach to automatically check k-inductive invariants for graph transformation systems has been developed so far.

In order to analyze possibly infinite systems in a finite fashion, we first introduce a symbolic encoding for transformation traces. Our main contribution is a formal approach and algorithm to verify a restricted form of graph constraints as

k-inductive invariants. We prove the approach's correctness and demonstrate its applicability by means of several examples evaluated with a prototypical implementation of our algorithm. While our technique takes care of the inductive step (verifying the *k*-inductive invariant), the base of induction for traces of length $k-1$ from an initial graph is established with the model checker GROOVE [9].

This report is organized as follows: In Section 2, we reintroduce the necessary foundations and our formal model. Section 3 defines our notion of *k*-inductive invariants and the symbolic encoding. We present our formal approach to *k*-inductive invariant checking in Section 4. In Section 5, we evaluate our algorithm and approach, before summarizing our results in Section 6. Omitted constructions and proofs can be found in the respective sources. More details to our examples can be found in Appendix A.

This report is an extended version of [6] and provides proofs to our lemmas and theorems and additional details to our example systems and evaluation. The numbering of definitions, theorems, examples, and lemmas follows the numbering in [6]; any such elements added in this report are not numbered.

2. Prerequisites

This section cites formal foundations [7, 8, 10], introduces our running example, and reintroduces the restricted formal model employed in our approach and tool.

2.1. Foundations

The formalism we use (see [7] and our previous work [5]) considers a *graph* to consist of node and edge sets V and E and source and target functions $s, t : E \to V$ assigning source and target nodes to edges. A *graph morphism* $f : G_1 \to G_2$ for graphs $G_i = (V_i, E_i, s_i, t_i)$ with $i = 1, 2$ consists of two functions mapping nodes and edges $f = (f_V, f_E)$ with $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve source and target functions. Injective graph morphisms (or *monomorphisms*) are graph morphisms with injective mapping functions and are denoted as $f : G_1 \hookrightarrow G_2$. A *typed graph* G is typed over a special type graph TG by a *typing morphism type* $: G \to TG$; *typed graph morphisms* must preserve the typing morphism.

Additionally, we require (nested) application conditions [8] and (nested) graph constraints [10] to describe more complex conditions over morphisms and graphs, respectively. Here, an application condition (graph constraint) can also be interpreted as describing the set of morphisms (graphs) that satisfy it.

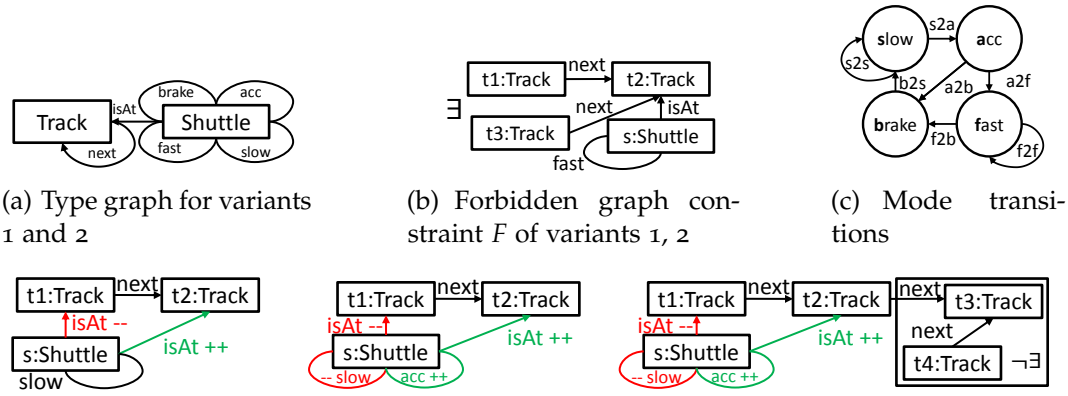
Application conditions (or *nested application conditions*) are inductively defined as in [8]: (1) for every graph P, true is an application condition over P; (2) for every morphism $a : P \hookrightarrow C$ and every application condition ac over C, $\exists(a, ac)$ is an application condition over P. Application conditions can also be extended over boolean combinations: (3) for application conditions ac, ac_i over P (for all index sets I), $\neg ac$ and $\bigwedge_{i \in I} ac_i$ are application conditions over P.

Satisfiability of application conditions is inductively defined as in [8]: (1) every morphism satisfies true; (2) a morphism $g : P \to G$ satisfies $\exists(a, ac)$ over P with $a : P \to C$ if there exists an injective $q : C \hookrightarrow G$ such that $q \circ a = g$ and q satisfies ac. Finally, (3) a morphism $g : P \to G$ satisfies $\neg ac$ over P if g does not satisfy ac and g satisfies $\bigwedge_{i \in I} ac_i$ over P if g satisfies each ac_i $(i \in I)$.

We write $g \vDash ac$ to denote that the morphism g satisfies ac. Two conditions ac and ac' are equivalent ($ac \equiv ac'$), if for all morphisms $g : P \to G$, $g \vDash ac$ if and only if $g \vDash ac'$. Also, $\exists p$ and $\forall(p, ac)$ abbreviate $\exists(p, \text{true})$ and $\neg\exists(p, \neg ac)$.

A *graph constraint* [10] is a condition over the empty graph $\varnothing$. A graph G then satisfies such a condition if the initial morphism $i_G : \varnothing \hookrightarrow G$ satisfies it.

We use graph rules to describe how graphs can be transformed by rule applications. As defined in [8], a *plain rule* $p = (L \hookleftarrow K \hookrightarrow R)$ consists of two injective morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$. L and R are called left- and right-hand side of



(a) Type graph for variants 1 and 2

(b) Forbidden graph constraint F of variants 1, 2

(c) Mode transitions

(d) Graph rule s2s of variants 1 and 2

(e) Graph rule s2a of variant 1 without NAC

(f) Graph rule s2a of variant 2 with NAC for switch ahead

Figure 1: Example type graph, graph constraint, mode transitions, and rules

p, respectively. A *rule* $b = \langle p, ac_L, ac_R \rangle$ consists of a plain rule p and a *left* (*right*) *application condition* ac_L (ac_R) over L (R).

A *transformation* (also [8]) consists of two pushouts (1) and (2) such that $m \vDash ac_L$ and $m' \vDash ac_R$. We write $G \Rightarrow_{b,m,m'} H$ and say that $m : L \to G$ ($m' : R \to H$) is the *match* (*comatch*) of b in G (in H). We write $G \Rightarrow_b H$ to express that there exist m and m' such that $G \Rightarrow_{b,m,m'} H$. For a set of rules $\mathcal{R}$, $G \Rightarrow_{\mathcal{R}} H$ expresses that there exist $b \in \mathcal{R}$ and m, m' such that $G \Rightarrow_{b,m,m'} H$. Also, given a rule $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, ac_R \rangle$, its *inverse rule* is denoted as b^{-1} and defined as $b^{-1} = \langle (R \hookleftarrow K \hookrightarrow L), ac_R, ac_L \rangle$.

$$\begin{array}{ccccc} ac_L \triangleright L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \triangleleft ac_R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle m'} \\ G & \xleftarrow{\ l'\ } & D & \xrightarrow{\ r'\ } & H \end{array}$$

Finally, a *typed graph transformation system* $GTS = (\mathcal{R}, TG)$ consists of a set of graph rules $\mathcal{R}$ and a type graph TG [7].

Example 1 (variant 1; see Appendix A.1 for all details). Our running example is a system where a single shuttle moves on a topology of connected tracks in different speed modes (slow, acc(elerate), fast, and brake), which follow a certain protocol (Fig. 1(c)). The system also has a forbidden property, which describes a shuttle driving on a switch in mode fast. Fig. 1 shows our system modeled as a typed graph transformation system: a type graph (Fig. 1(a)), the forbidden property as a graph constraint (F, Fig. 1(b)), and seven rules modeling shuttle movement and driving mode transitions. Two of those rules are depicted: s2s (slow to slow) in Fig. 1(d) and s2a (slow to acc(elerate)) in Fig. 1(e). All other rules (a2b, a2f, f2b, b2s, f2f) function analogously and follow the scheme of s2a or s2s, respectively. Graph rules

are pictured in a compact notation: deleted (created) elements are drawn in red (green) and annotated −− (++); unchanged elements are in black.

This example variant exhibits unsafe behavior in the sense of possible violations of our forbidden property, because rule s2a (Figure 1(e)) does not prevent the shuttle from accelerating from slow to acc when there is a switch two tracks ahead. Subsequent application of a2f could then lead to the violation. In our second variant, this error has been fixed:

Example 2 (variant 2; see Appendix A.2 for all details). While the type graph, forbidden property, and most rules remain the same, variant 2 modifies variant 1 by extending a number of rules with a left (negative) application condition: s2a as in Figure 1(f), a2f and f2f in a similar fashion. The application condition is designed to prevent the shuttle from acc(elerating) (and, for a2f and f2f, from driving fast) if a switch is two tracks ahead.

Since k-inductive invariant checking considers paths of transformations instead of single steps, we require the notion of transformation sequences. Given a set of rules $\mathcal{R} = \{b_i\}$, graphs G_0, G_i, and matches (comatches) m_i (m_i') for $i = 1, ..., k$, a *sequence of transformations to* $\mathcal{R}$ $trans = G_0 \Rightarrow_{b_1,m_1,m_1'} G_1 \Rightarrow_{b_2,m_2,m_2'} ... \Rightarrow_{b_k,m_k,m_k'} G_k$ denotes subsequent graph transformations $G_0 \Rightarrow_{b_1,m_1,m_1'} G_1$, $G_1 \Rightarrow_{b_2,m_2,m_2'} G_2, ..., G_{k-1} \Rightarrow_{b_k,m_k,m_k'} G_k$. Also, $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_k$ denotes $G_0 \Rightarrow_{\mathcal{R}} G_1, G_1 \Rightarrow_{\mathcal{R}} G_2, ..., G_{k-1} \Rightarrow_{\mathcal{R}} G_k$ and is abbreviated as $G_0 \Rightarrow_{\mathcal{R}}^{k} G_k$.

We say that a sequence of transformations $trans = G_0 \Rightarrow_{b_1} ... \Rightarrow_{b_k} G_k$ *leads* to a graph constraint F, if $G_k \vDash F$.

Example 3 (transformation sequence). Fig. 2 shows a transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m_1'} G_1 \Rightarrow_{b_2,m_2,m_2'} G_2$, where b_1 and b_2 are graph rules s2a and a2f from Example 1. Matches and comatches are not depicted. Here, G_2 contains a shuttle on a switch driving in mode fast, which matches our forbidden property F (Fig. 1(b)). Thus, we have $G_2 \vDash F$ and *trans* leads to F. Note that *trans* would not be a valid transformation sequence in variant 2 due to the additional negative application condition preventing the application of s2a (b_1).

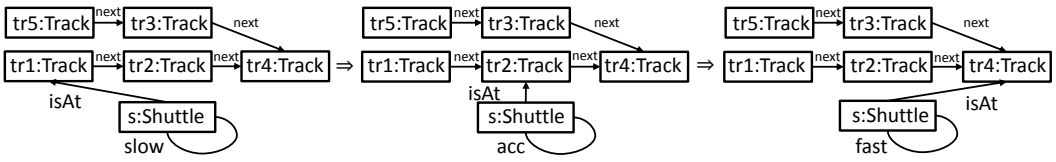


Figure 2: Example transformation sequence $trans = G_0 \Rightarrow_{b_1,m_1,m_1'} G_1 \Rightarrow_{b_2,m_2,m_2'} G_2$

Graph transformation systems appear in an executable form as part of (typed) graph grammars. A *typed graph grammar* [7] $GG = (GTS, G_0)$ consists of a typed graph transformation system GTS and an initial graph G_0. We define the *state space* of a graph grammar $GG = (GTS, G_0)$ with $GTS = (\mathcal{R}, TG)$ as $\text{REACH}(GG) = \{G \mid \exists n(G_0 \Rightarrow^n_{\mathcal{R}} G)\}$. We can also restrict the state space of a graph grammar by a graph constraint C and define the *state space under* C of a graph grammar GG as $\text{REACH}(GG, C) = \{G \mid \exists n(G_0 \Rightarrow^n_{\mathcal{R}} G \text{ and all traversed graphs satisfy } C)\}$.

In order to calculate the state space for a limited number of transformation steps, we define $\text{REACH}_k(GG) = \{G \mid \exists n(0 \leq n \leq k \wedge G_0 \Rightarrow^n_{\mathcal{R}} G)\}$ and $\text{REACH}_k(GG, C) = \{G \mid \exists n(0 \leq n \leq k \wedge G_0 \Rightarrow^n_{\mathcal{R}} G \text{ and all traversed graphs satisfy } C)\}$, respectively.

An important part of our algorithm is an adjusted form of the Shift-lemma and -construction [8], which transfers nested application conditions over morphisms to new context while preserving the condition's satisfiability.

Construction (Shift-construction, adjusted from [8])**.** *For each application condition ac over a graph P and for each morphism $b : P \to P'$, $\text{Shift}(b, ac)$ transforms ac via b into an application condition over P' such that, for each morphism $n : P' \hookrightarrow H$, it holds that $n \circ b \vDash ac \Leftrightarrow n \vDash \text{Shift}(b, ac)$.*

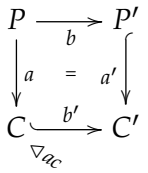
The Shift-construction is inductively defined as follows:

(1) $\text{Shift}(b, \text{true}) = \text{true}$,

(2) $\text{Shift}(b, \exists(a, ac)) = \bigvee_{(a',b') \in \mathcal{F}} \exists(a', \text{Shift}(b', ac))$ *for a non-empty set $\mathcal{F}$ of jointly surjective and injective morphism pairs (a', b') such that $b' \circ a = a' \circ b$, and* false*, if $\mathcal{F} = \varnothing$,*

(3) $\text{Shift}(b, \neg ac) = \neg \text{Shift}(b, ac)$*, and*

(4) $\text{Shift}(b, \bigwedge_{i \in I} ac_i) = \bigwedge_{i \in I} \text{Shift}(b, ac_i)$.



Furthermore, we employ the L-lemma and -construction [8, 10], which transforms application conditions over graph rules in reverse direction:

Construction (L-construction [8, 10])**.** *For each rule $b = \langle L \hookleftarrow K \hookrightarrow R \rangle$ and each application condition ac over R, $\text{L}(b, ac)$ transforms ac via b into an application condition over L such that, for each direct transformation $G \Rightarrow_{b,m,m'} H$, we have $m \vDash \text{L}(b, ac) \Leftrightarrow m' \vDash ac$.*

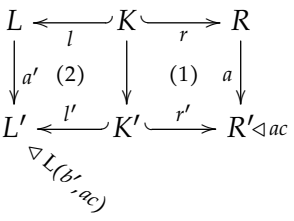
The L-construction is inductively defined:

(1) $\text{L}(b, \text{true}) = \text{true}$,

(2) $\text{L}(b, \exists(a, ac)) = \exists(a', \text{L}(b', ac))$ *if $b' = \langle L' \hookleftarrow K' \hookrightarrow R' \rangle$ constructed via the pushouts (1) and (2) exists and false, otherwise,*

(3) $\text{L}(b, \neg ac) = \neg \text{L}(b, ac)$*, and*

(4) $\text{L}(b, \bigwedge_{i \in I} ac_i) = \bigwedge_{i \in I} \text{L}(b, ac_i)$.



Both Shift and L produce finite results by construction.

2.2. Formal Model

As described in [5], our verification approach and tool impose certain restrictions on rules and properties in order to strike a balance between expressiveness and computational complexity while ensuring termination. In the following, we discuss those restrictions, starting with composed negative application conditions as a restricted form of nested application conditions.

Definition 4 (composed negative application condition [5]). *A* composed negative application condition *is an application condition of the form* $ac = \text{true}$ *or* $ac = \bigwedge_{i \in I} \neg \exists a_i$ *for monomorphisms* a_i*. An individual condition* $\neg \exists a_i$ *is called* negative application condition.

Our properties to be verified as k-inductive invariants are described by so-called forbidden patterns, which follow a restricted form of graph constraints.

Definition 5 (pattern [5]). *A* pattern *is a graph constraint of the form* $F = \exists(i_P : \varnothing \hookrightarrow P, ac_P)$*, with* P *being a graph and* ac_P *a composed negative application condition over* P*. A* composed forbidden pattern *is a graph constraint of the form* $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ *for some index set* I *and patterns* F_i*. Patterns* F_i *occurring in a composed forbidden pattern are also called* forbidden patterns.

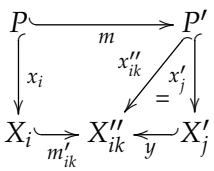
Besides forbidden patterns, we allow our systems to be equipped with *(composed) assumed forbidden patterns*, which are similar in form to (composed) forbidden patterns and which will be explained below (see Example 8).

In order to compare patterns, we reintroduce the notion of pattern implication in Definition 6 and our technique to perform implication checks in the theorem following below. More general approaches discussing implication of (unrestricted) graph constraints can be found in [10, 13].

Definition 6 (implication of patterns [5]). *Let* C *and* C' *be two patterns.* C' implies C*, denoted* $C' \vDash C$*, if, for all graphs* G*,* $G \vDash C'$ *implies* $G \vDash C$*.*

Theorem (implication of patterns, adjusted from [5]). *Let* $C = \exists(i_P : \varnothing \hookrightarrow P, ac)$ *and* $C' = \exists(i_{P'} : \varnothing \hookrightarrow P', ac')$ *be two patterns, with composed negative application conditions* $ac = \bigwedge_{i \in I} \neg \exists(x_i : P \hookrightarrow X_i)$ *and* $ac' = \bigwedge_{j \in J} \neg \exists(x'_j : P' \hookrightarrow X'_j)$ *for index sets* I, J*. Then* $C' \vDash C$*, if the following condition is fulfilled:*

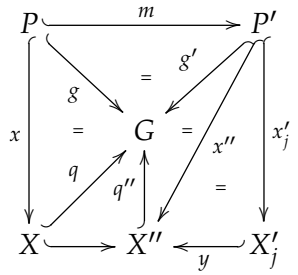
1′. *There exists a* $j \in J$ *such that* x'_j *is an isomorphism or*
1. *There exists a monomorphism* $m : P \hookrightarrow P'$ *such that:*
2. *With* $Shift(m, \neg \exists x_i) = \bigwedge_{k \in K_i} \neg \exists(x''_{ik} : P' \hookrightarrow X''_{ik})$ *for a number of corresponding index sets* K_i*, for each* x_i *it holds that* $\forall k (k \in K_i \Rightarrow \exists j \exists y (y : X'_j \hookrightarrow X''_{ik} \wedge x''_{ik} = y \circ x'_j))$.



Proof. Assuming that the above conditions hold, we have to show $\forall G(G \vDash C' \Rightarrow G \vDash C)$.

Assuming condition (1′) holds, consider an arbitrary graph G' with $g' : P' \hookrightarrow G'$ and thus, $G' \vDash \exists i_{P'}$. Since x'_j (for the specific j) is an isomorphism, there is a $q' : X'_j \hookrightarrow G'$ with $q' \circ x'_j = g'$. Hence, we have $g' \nvDash \exists x'_j$ and, more importantly, $G' \neg \vDash C'$. Thus, there does not exist a graph G' with $G' \vDash C'$ and consequently, $\forall G(G \vDash C' \Rightarrow G \vDash C)$ is trivially true.

Assuming condition (1′) does not hold, consider an arbitrary graph G with $G \vDash C'$. By definition of satisfaction, we have $i_G \vDash C'$, implying the existence of a monomorphism $g' : P' \hookrightarrow G$ with $g' \vDash \mathrm{ac}'$. By assumption, there is a monomorphism $m : P \hookrightarrow P'$. Then there exists a morphism $g : P \to G$ with $g = g' \circ m$. Since monomorphisms are closed under composition, g is a monomorphism.



We will show $g \vDash \mathrm{ac}$ by contradiction. Suppose $g \nvDash \mathrm{ac}$, implying the existence of a $x = x_i$ for some $i \in I$ and a corresponding monomorphism $q : X \hookrightarrow G$ with $g = q \circ x$, i.e. $g \nvDash \neg\exists x$. For $\mathrm{Shift}(m, \neg\exists x) = \bigwedge_{k \in K} \neg\exists(x''_k : P' \hookrightarrow X''_k)$ for an index set K depending on x, we have $g' \vDash \mathrm{Shift}(m, \neg\exists x) \Leftrightarrow g' \circ m \vDash \neg\exists(x_i)$. Since $g = g' \circ m$, we have $g' \nvDash \mathrm{Shift}(m, \neg\exists x)$ and thus $g' \nvDash \bigwedge_{k \in K} \neg\exists(x''_k : P' \hookrightarrow X''_k)$. This implies the existence of a $x'' = x''_k$ for some $k \in K$ with $g' \nvDash \neg\exists(x'' : P' \hookrightarrow X'')$. Then there exists a monomorphism $q'' : X'' \hookrightarrow G$ with $g' = q'' \circ x''$. By assumption, there exists a monomorphism $y : X'_j \hookrightarrow X''$ with $x'' = y \circ x'_j$ for some $j \in J$. This implies the existence of a monomorphism $q' : X'_j \hookrightarrow G$ with $q' = q'' \circ y$ and thus $q' \circ x'_j = q'' \circ y \circ x'_j = q'' \circ x'' = g'$. Thus, $g' \nvDash \neg\exists x'_j$ for the specific $j \in J$ and therefore $g' \nvDash \mathrm{ac}'$. This is a contradiction and hence, we have $g \vDash \mathrm{ac}$. With $g : P \hookrightarrow G$ and $g \vDash \mathrm{ac}$, we get $G \vDash C$, concluding the proof. □

In summary, our formal model is subject to the restrictions listed below [5]. However, only the requirements concerned with left application conditions and graph constraints actually result in a limitation of expressive power [7, 8, 10].

Morphisms in application conditions (see Section 2.1) must be injective.
Left application conditions (see Section 2.1) in rules are required to be composed negative application conditions.

Right application conditions (Section 2.1) in rules are required to be true.
Rule applicability (see Section 2.1) requires injective matches and comatches.
Graph constraints must be patterns (Definition 5).

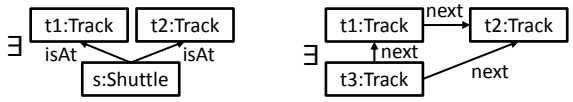
Since k-inductive invariants are a generalization of inductive invariants, we also reiterate the notion of inductive invariants as defined in our previous work.

Definition 7 (inductive invariant [5])**.** *Given a typed graph transformation system* $GTS = (\mathcal{R}, TG)$ *and graph constraints* $\mathcal{F}$ *and* $\mathcal{H}$*,* $\mathcal{F}$ *is an* inductive invariant *for GTS* under $\mathcal{H}$*, if, for each rule* b *in* $\mathcal{R}$*, it holds that:*

$$\forall G_0, G_1((G_0 \Rightarrow_b G_1) \Rightarrow ((G_0 \vDash \mathcal{F} \wedge G_0 \vDash \mathcal{H}) \Rightarrow (G_1 \vDash \mathcal{F} \vee G_1 \nvDash \mathcal{H})))$$

An inductive invariant (here: $\mathcal{F}$) is a property that, given its validity before rule application ($G_0 \vDash \mathcal{F}$), will also hold after rule application ($G_1 \vDash \mathcal{F}$). In addition, we allow our system to be equipped with an additional *assumed graph constraint* ($\mathcal{H}$). This constraint is assumed to be guaranteed by other means, such as additional verification steps. Typical examples include cardinality restrictions of the type graph, which, in our tool, are not automatically enforced otherwise. As defined above, rule applications involving violations of those properties are not considered as possible violations of the inductive invariant (cf. $G_0 \vDash \mathcal{H}$, $G_1 \nvDash \mathcal{H}$). In our approach, both types of constraints are required to be composed (assumed) forbidden patterns.

Example 8. Fig. 3(a) depicts an assumed forbidden pattern H_1 of our running example implementing a cardinality constraint resulting from the physical impossibility of a shuttle being located on two tracks. Fig. 3(b) (H_2) describes an undesired track topology. Both (and all other assumed forbidden patterns in our examples, see Appendix A) are negated and conjunctively joined in a composed assumed forbidden pattern $\mathcal{H} = \neg H_1 \wedge \neg H_2 \wedge ... \wedge \neg H_{15}$. $\mathcal{H}$ is an inductive invariant and can be separately verified as such by our existing algorithm [5]. In contrast to that, the composed forbidden pattern $\mathcal{F} = \neg F$ (Fig. 1(b)), which consists of just one forbidden pattern F, is not an inductive invariant for either variant.



(a) Cardinality restriction (b) Topology restriction

Figure 3: Assumed forbidden patterns

3. k-Induction and Symbolic Encoding of Sequences

With the foundations established, we can now define the notion of k-induction [15] and k-inductive invariants for graph transformation systems. We also introduce a symbolic encoding for transformation sequences.

As established, an inductive invariant (Def. 7) is a property that, given its validity before rule application, will also hold after rule application. Likewise, a k-inductive invariant is a property whose validity in a path of length $k-1$ ($G_z \vDash \mathcal{F}$, below) implies its validity after the next rule application ($G_k \vDash \mathcal{F}$).

Definition 9 (k-inductive invariant)**.** *Given a typed graph transformation system GTS = $(\mathcal{R}, TG)$ and graph constraints $\mathcal{F}$ and $\mathcal{H}$, $\mathcal{F}$ is a* k-inductive invariant *for GTS under $\mathcal{H}$, if, for all sequences of transformations to $\mathcal{R}$ trans = $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_k$ it holds that:*

$$\forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H}) \Rightarrow (G_k \vDash \mathcal{F} \vee G_k \nvDash \mathcal{H})$$

As with inductive invariants, which are k-inductive invariants with $k = 1$, our formal model requires the graph constraints $\mathcal{F}$ and $\mathcal{H}$ to be a composed (assumed) forbidden pattern, respectively.

In order to deduce from the existence of a k-inductive invariant its validity in all states of an executable system, we need to consider graph grammars [7] and their initial states and state spaces (cf. Section 2.1). In particular, a k-inductive invariant (inductive step) under a constraint ($\mathcal{H}$) holds in every reachable state of the grammar's state space under the constraint (REACH$(GG, \mathcal{H})$) if it is also valid in all transformation sequences (induction base) of length $k-1$ from the initial state that satisfy the constraint.

Lemma 10. *Let $GG = (GTS, G_0)$ be a graph grammar with a graph transformation system $GTS = (\mathcal{R}, TG)$ and let $\mathcal{F}$ and $\mathcal{H}$ be two graph constraints. Then, $\mathcal{F}$ is satisfied in all states of* REACH$(GG, \mathcal{H})$*, if the following conditions hold:*

1. $\forall G(G \in \text{REACH}_{k-1}(GG, \mathcal{H}) \Rightarrow G \vDash \mathcal{F})$.
2. *$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.*

Proof. Consider a graph G in the graph grammar's state space under the constraint, i.e. $G \in \text{REACH}(GG, \mathcal{H})$. Hence, there exists a transformation sequence $G_0 \Rightarrow^n_{\mathcal{R}} G$ such that all traversed graphs satisfy $\mathcal{H}$. If $n \leq k-1$, we have $G \vDash \mathcal{F}$ by precondition.

We will prove the case $n > k-1$ by induction:

Induction base. For $n = k$, there exist a transformation sequence $G_0 \Rightarrow^{k-1}_{\mathcal{R}} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$. Also, all graphs in the sequence satisfy $\mathcal{H}$ and there exist transformation sequences (whose length is smaller than k) from G_0 to all graphs

in the sequence such that all traversed graphs satisfy $\mathcal{H}$. Hence, by precondition (1), $G' \vDash \mathcal{F}$ and all graphs in the sequence satisfy $\mathcal{F}$. Then, since $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ (2) and holds in all graphs of the sequence $G_0 \Rightarrow_{\mathcal{R}}^{k-1} G'$ and since we have $G_0 \Rightarrow_{\mathcal{R}}^{k-1} G'$ and $G' \Rightarrow_{\mathcal{R}} G$, we get $G \vDash \mathcal{F}$.

Inductive step. Consider a sequence $G_0 \Rightarrow_{\mathcal{R}}^{n} G$ with $n > k$. Then, there is a transformation sequence $G_0 \Rightarrow_{\mathcal{R}}^{n-1} G'$ and a transformation $G' \Rightarrow_{\mathcal{R}} G$. By inductive hypothesis, $\mathcal{F}$ holds in all graphs of the sequence $G_0 \Rightarrow_{\mathcal{R}}^{n-1} G'$. Since $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$ (2) and $n > k$, we have $G \vDash \mathcal{F}$, concluding the inductive proof. □

This lemma disregards the specific part of a graph grammar's state space that cannot be reached without violating the assumed graph constraint (here: $\mathcal{H}$). Usually, this happens under the assumption that there exist additional measures (such as postprocessing) preventing the system (at runtime) from transitioning into such states. In the absence of such measures – and specifically, for our examples – we have to establish the assumed graph constraint's validity differently, as described by the following lemma:

Lemma. *Let $GG = (GTS, G_0)$ be a graph grammar with a graph transformation system $GTS = (\mathcal{R}, TG)$ and let $\mathcal{F}$ and $\mathcal{H}$ be two graph constraints. Then, $\mathcal{F}$ is satisfied in all states of* $\mathrm{REACH}(GG)$*, if the following conditions hold:*

1. *$G_0 \vDash \mathcal{H}$ and $\mathcal{H}$ is a 1-inductive invariant for GTS.*
2. *$\forall G(G \in \mathrm{REACH}_{k-1}(GG) \Rightarrow G \vDash \mathcal{F})$.*
3. *$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.*

Proof. Since $G_0 \vDash \mathcal{H}$ (1) and since $\mathcal{H}$ is a 1-inductive invariant for GTS (1), we have $REACH_{k-1}(GG) = REACH_{k-1}(GG, \mathcal{H})$ (A) and $REACH(GG) = REACH(GG, \mathcal{H})$ (B). $\forall G(G \in REACH_{k-1}(GG) \Rightarrow G \vDash \mathcal{F})$ (2) and (A) imply $\forall G(G \in REACH_{k-1}(GG, \mathcal{H}) \Rightarrow G \vDash \mathcal{F})$ and, with (3) and by Lemma 10, $\mathcal{F}$ is satisfied in all states of $REACH(GG, \mathcal{H})$. With (B), $\mathcal{F}$ is then satisfied in all states of $REACH(GG)$. □

While our verification approach and contribution focus on establishing k-inductive invariants, we will shortly discuss the base of induction in our evaluation. In the following, the notion that a transformation system is *safe* (*unsafe*) will refer to the inductive step; i.e. will mean that the respective composed forbidden pattern can (cannot) be established as a k-inductive invariant.

Since there is a possibly infinite amount of transformation sequences to be analyzed in order to establish a composed forbidden pattern as a k-inductive invariant, we require a symbolic encoding for sequences of transformations. Then, reasoning over transformation sequences can be reduced to reasoning over a finite set of representative symbolic encodings. To establish such an encoding, we first

require application conditions that can represent graph rule applications, similar to patterns and application conditions representing the set of graphs and morphisms that satisfy them. *Source (target) patterns* describe rule applications in an extended context (the application condition) beyond the left (right) rule side. *Target/source patterns* combine both and represent the context after one rule application and before another. Then, we combine source, target, and target/source patterns in *k-sequences of s/t (source/target) patterns* (Def. 12).

Definition 11 (source, target, and target/source pattern [5])**.** *A* source pattern *(*target pattern*) over a rule – specifically, over its left (right) side L (R) – is an application condition src (tar) of the form* false *or the form* $\exists(s : L \hookrightarrow S, ac_S)$ *(*$\exists(t : R \hookrightarrow T, ac_T)$*) with a composed negative application condition* ac_S *(*ac_T*) over S (T).*

A target/source pattern *is a pair of a target and a source pattern* (tar, src) *which share the same codomain and application condition, i.e.* $tar = \exists(t : R \hookrightarrow T, ac_T)$ *and* $src = \exists(s : L \hookrightarrow T, ac_T)$*. A pair of morphisms* (m', m) *with the same codomain (*$m' : R \hookrightarrow G$ *and* $m : L \hookrightarrow G$*) satisfies a target/source pattern* (tar, src)*, denoted* $(m', m) \vDash (tar, src)$*, if* $m' \vDash tar$ *and* $m \vDash src$ *by a common monomorphism* $y : T \hookrightarrow G$*, i.e. if there exists a monomorphism* $y : T \hookrightarrow G$ *such that* $y \circ t = m'$*,* $y \circ s = m$*, and* $y \vDash ac_T$*.*

Definition 12 (*k*-sequences of *s*/*t*-patterns)**.** *Given* $k \geq 1$*, a source pattern* src_1 *over a rule* b_1*, a target pattern* tar_k *over a rule* b_k *and a number of target/source patterns* (tar_i, src_{i+1}) *over a number of rules* b_i $(1 \leq i \leq k-1)$*,* $seq = src_1 \Rightarrow_{b_1} tar_1, src_2 \Rightarrow_{b_2} \ldots \Rightarrow_{b_k} tar_k$ *is a k-*sequence of *s*/*t*-patterns*.*

Satisfiability *of k-sequences of s/t-patterns is defined as follows:*

Given a sequence of transformations (of length k) $trans = G_0 \Rightarrow_{c_1,m_1,m_1'} \ldots \Rightarrow_{c_k,m_k,m_k'} G_k$ *and a k-sequence of s/t-patterns* $seq = src_1 \Rightarrow_{b_1} tar_1, src_2 \Rightarrow_{b_2} \ldots \Rightarrow_{b_k} tar_k$*, trans satisfies seq, denoted as* $trans \vDash seq$*, if, for all i with* $1 \leq i \leq k$*,* $c_i = b_i$*,* $m_i \vDash src_i$*,* $m_i' \vDash tar_i$ *and, in particular, for all i with* $1 \leq i \leq k-1$*,* $(m_i', m_{i+1}) \vDash (tar_i, src_{i+1})$*.*

Two k-sequences of s/t-patterns seq, seq' *are* equivalent *(*$seq \equiv seq'$*), if for all transformation sequences trans it holds that* $trans \vDash seq \Leftrightarrow trans \vDash seq'$*.*

The idea of *k*-sequences of *s*/*t*-patterns (or simply *s*/*t*-pattern sequences) is to not only describe subsequent transformations but, with source and target patterns, additional context in which those transformations occur. As such, an *s*/*t*-pattern sequence is a symbolic encoding for the set of transformation sequences that satisfy it. The construction of specific *s*/*t*-pattern sequences for the verification of *k*-inductive invariants will be explained in the next section.

Example 13 (*s*/*t*-pattern sequence). Fig. 4 shows a 2-sequence of *s*/*t*-patterns $seq = src_1 \Rightarrow_{b_1} tar_1, src_2 \Rightarrow_{b_2} tar_2$, where rules $b_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ $(i = 1, 2)$ are s2a and a2f from variant 1 (Example 1). In particular, $src_1 = \exists s_1$ and $src_2 = \exists s_2$ are source

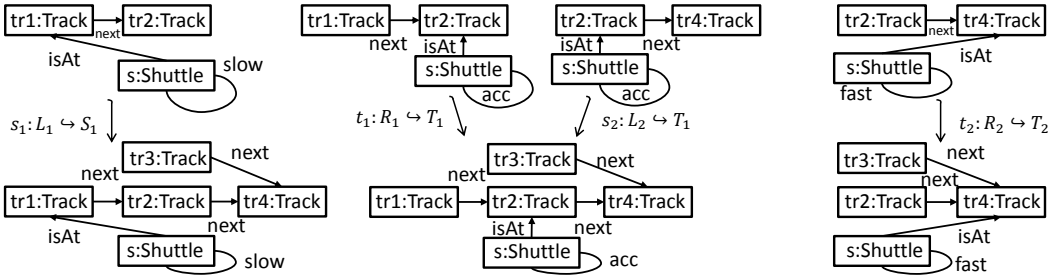


Figure 4: Example sequence of patterns $seq = src_1 \Rightarrow_{b_1} tar_1, src_2 \Rightarrow_{b_2} tar_2$

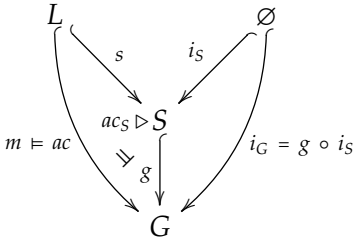
patterns, $tar_1 = \exists t_1$ and $tar_2 = \exists t_2$ are target patterns, and (tar_1, src_2) is a target/source pattern. The transformation sequence *trans* (Example 3, Figure 2) satisfies *seq* ($trans \vDash seq$). On the other hand, no transformation sequence from variant 2 (Example 2) could satisfy *seq* due to the negative application condition in s2a.

Note that $src_1 \Rightarrow_{b_1} tar_1$ and $src_2 \Rightarrow_{b_2} tar_2$ would also be valid 1-sequences of s/t-patterns.

Since we will need to compare elements of s/t-pattern sequences to (assumed) forbidden patterns, we establish a connection between source and target patterns and (forbidden) patterns with respect to pattern implication:

Lemma 14 (reduction to pattern [5])**.** *Let $ac = \exists(s : L \hookrightarrow S, ac_S)$ be an application condition over L with ac_S being a composed negative application condition. For the* reduction to a pattern *$ac_\varnothing = \exists(i_S : \varnothing \hookrightarrow S, ac_S)$ of ac we have: For each graph G with $m : L \hookrightarrow G$ such that $m \vDash ac$, we have $G \vDash ac_\varnothing$.*

Proof. Consider an arbitrary graph G with a monomorphism $m : L \hookrightarrow G$ such that $m \vDash ac$. By definition of satisfiability, there exists a monomorphism $g : S \hookrightarrow G$ such that $g \vDash ac_S$. For $i_G : \varnothing \hookrightarrow G$, we have $i_G = g \circ i_S$ and with $g \vDash ac_S$, we have $i_G \vDash ac_\varnothing$. By definition of satisfiability, this implies $G \vDash ac_\varnothing$, concluding the proof.



□

4. k-Inductive Invariant Checking

Our formal approach to verify a composed forbidden pattern as a k-inductive invariant consists of the following steps: We split the composed forbidden pattern into its individual forbidden patterns (step 1, section 4.1). Then, we construct a finite set of k-sequences of s/t-patterns per forbidden pattern such that these sequences represent all transformation sequences leading to the forbidden pattern (step 2, Section 4.2). Finally, we analyze each s/t-pattern sequence in each set for possible violations of (assumed) forbidden patterns earlier in the sequence (step 3, Section 4.3). Sequences with such violations can be discarded; all others present counterexamples with respect to the validity of the k-inductive invariant.

In addition to our formal approach to k-inductive invariant checking (Sections 4.1–4.3), we also explain the basic scheme of our implementation (Section 4.4).

4.1. Step 1: Separation of Forbidden Patterns

The following lemma is the formal basis for investigating individual forbidden patterns and transformation sequences that lead (see Section 2.1) to those patterns. It is based on the contraposition of Definition 9; its intention is to justify the procedure of finding all possible violations of individual forbidden patterns (the implication's precondition below) and trying to disprove them by finding violations earlier in the path (postcondition). The latter loosely corresponds to step 2 and 3 (Sections 4.2 and 4.3) below.

Lemma 15. *Given a typed graph transformation system $GTS = (\mathcal{R}, TG)$ and a composed (assumed) forbidden pattern $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$ ($\mathcal{H} = \bigwedge_{j\in J} \neg H_j$), $\mathcal{F}$ is a* k-inductive invariant for *GTS* under *$\mathcal{H}$, if the following holds for each k-sequence of transformations $trans = G_0 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_k$:*

$$\exists n(G_k \vDash F_n) \Rightarrow (\exists z, v(0 \leq z \leq k \wedge G_z \vDash H_v) \vee \exists z, v(0 \leq z \leq k-1 \wedge G_z \vDash F_v))$$

Proof. We can rearrange the formula from Definition 9 (for all sequences of transformations $trans = G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_k$):

$$\begin{aligned}
&\forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H}) \Longrightarrow (G_k \vDash \mathcal{F} \vee G_k \nvDash \mathcal{H}) \\
\Longleftrightarrow &\neg(G_k \vDash \mathcal{F} \vee G_k \nvDash \mathcal{H}) \Longrightarrow \neg\forall z(0 \leq z \leq k-1 \Rightarrow G_z \vDash \mathcal{F} \wedge \mathcal{H}) \\
\Longleftrightarrow &\ (G_k \nvDash \mathcal{F} \wedge G_k \vDash \mathcal{H}) \Longrightarrow\ \ \exists z(0 \leq z \leq k-1 \wedge G_z \nvDash \mathcal{F} \wedge \mathcal{H}) \\
\Longleftrightarrow &\ (G_k \nvDash \mathcal{F}) \Longrightarrow (\exists z(0 \leq z \leq k-1 \wedge G_z \nvDash \mathcal{F} \wedge \mathcal{H}) \vee G_k \nvDash \mathcal{H}) \\
\Longleftrightarrow &\ (G_k \nvDash \mathcal{F}) \Longrightarrow (\exists z(0 \leq z \leq k-1 \wedge (G_z \nvDash \mathcal{H} \vee G_z \nvDash \mathcal{F})) \vee G_k \nvDash \mathcal{H}) \\
\Longleftrightarrow &\exists n(G_k \vDash F_n) \Rightarrow \exists z, v(0 \leq z \leq k \wedge G_z \vDash H_v) \vee \exists z, v(0 \leq z \leq k-1 \wedge G_z \vDash F_v)
\end{aligned}$$

□

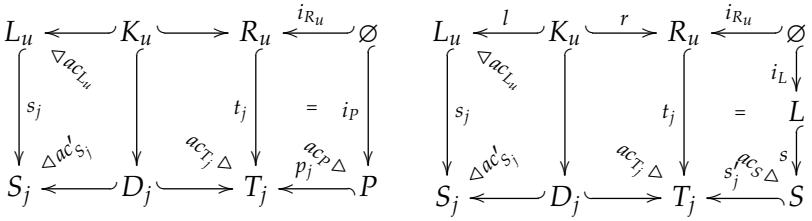
4.2. Step 2: Construction of k-Sequences and Context Propagation

The following theorem describes the construction of finite sets of s/t-pattern sequences, which represent all transformation sequences leading to a specific forbidden pattern. Those sequences are possible violations of the desired k-inductive invariant.

Theorem 16 (construction of sequences)**.** *There is a construction* Seq_k *such that for every pattern* $F = \exists(i_P, ac_P)$*, rule set* $\mathcal{R}$*, and* k*,* $\mathrm{Seq}_k(\mathcal{R}, F)$ *is a set of* k*-sequences of* s/t*-patterns and for each transformation sequence trans to* $\mathcal{R}$ *of length* k *that leads to* F*, there is a* $seq \in \mathrm{Seq}_k(\mathcal{R}, F)$ *with* $trans \vDash seq$*.*

Construction. Seq_k *is inductively constructed as follows (with appropriate indexes and index sets* u, j *and* U, J_u*, respectively), starting with* Seq_1 *(left figure):*

1. *For each rule* $b_u = \langle (L_u \hookleftarrow K_u \hookrightarrow R_u), ac_{L_u}, ac_{R_u} \rangle \in \mathcal{R}$*,* $\mathrm{Shift}(i_{R_u}, F) = \bigvee_{j \in J_u} tar_{u,j}$ *is a disjunction of target patterns over* R_u *of the form* $tar_{u,j} = \exists(t_j, ac_{T_j})$*.*
2. *For each such target pattern* $tar_{u,j}$*,* $src'_{u,j} = \mathrm{L}(b_u, tar_{u,j})$ *is a source pattern over* L_u *of the form* $src'_{u,j} = \mathrm{false}$ *or* $src'_{u,j} = \exists(s_j, ac'_{S_j})$*.*
3. *For the latter case,* $src_{u,j} = \exists(s_j, ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_{L_u}))$ *is a source pattern.*
4. *For each such pair of source and target pattern* $src_{u,j}$ *and* $tar_{u,j}$*,* $src_{u,j} \Rightarrow_{b_u} tar_{u,j}$ *is a* 1*-sequence of* s/t*-patterns.*
5. *Finally, we define* $\mathrm{Seq}_1(\mathcal{R}, F) = \{src_{u,j} \Rightarrow_{b_u} tar_{u,j} \mid b_u \in \mathcal{R} \wedge j \in J_u\}$ *as the set of these sequences.*



Given $\mathrm{Seq}_k(\mathcal{R}, F)$*, we construct* $\mathrm{Seq}_{k+1}(\mathcal{R}, F)$ *as follows (right figure).*

1. *For each sequence* $seq = src_1 \Rightarrow ... \Rightarrow tar_k \in \mathrm{Seq}_k(\mathcal{R}, F)$ *with* $src_1 = \exists(s : L \hookrightarrow S, ac_S)$ *and each* $b_u = \langle (L_u \hookleftarrow K_u \hookrightarrow R_u), ac_{L_u}, ac_{R_u} \rangle \in \mathcal{R}$*,* $\mathrm{Shift}(i_{R_u}, \exists(i_S, ac_S)) = \bigvee_{j \in J_u} tar_{u,j}$ *is a disjunction of target patterns over* R_u *with* $tar_{u,j} = \exists(t_j, ac_{T_j})$*.*

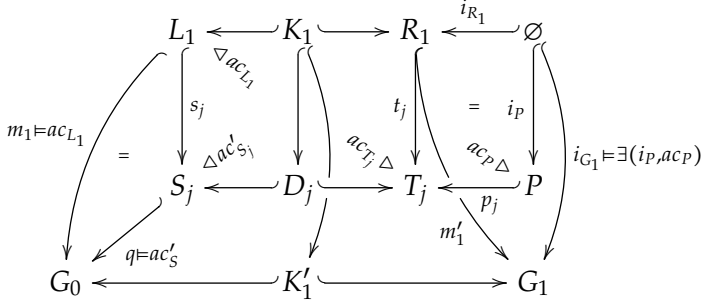
1′. *For each such target pattern* $tar_{u,j}$*,* $src^*_1 = \exists(s'_j \circ s, ac_{T_j})$ *is a source pattern.*

2. *For each such target pattern* $tar_{u,j}$*,* $src'_{u,j} = \mathrm{L}(b_u, tar_{u,j})$ *is a source pattern over* L_u *of the form* $src'_{u,j} = \mathrm{false}$ *or* $src'_{u,j} = \exists(s_j, ac'_{S_j})$*.*
3. *For the latter case,* $src_{u,j} = \exists(s_j, ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_{L_u}))$ *is a source pattern.*

4. *For each such pair of source and target pattern* $src_{u,j}$ *and* $tar_{u,j}$, $src_{u,j} \Rightarrow_{b_u} tar_{u,j}, src_1^* \Rightarrow ... \Rightarrow tar_k$ *is a* k*+1-sequence of s/t-patterns.*
5. *Finally, we define* $\mathrm{Seq}_{k+1}(\mathcal{R}, F) = \{src_{u,j} \Rightarrow_{b_u} tar_{u,j}, src_1^* \Rightarrow ... \Rightarrow tar_k \mid b_u \in \mathcal{R} \wedge j \in J_u \wedge seq \in \mathrm{Seq}_k(\mathcal{R}, F)\}$ *as the set of these sequences.*

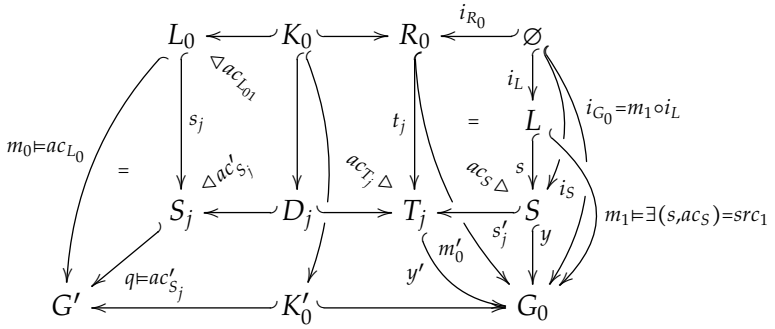
Also, given a set of rules $\mathcal{R}$ *and a composed forbidden pattern* $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ *with forbidden patterns* F_i*, we define* $\mathrm{Seq}_k(\mathcal{R}, \mathcal{F}) = \bigcup_{i \in I} \mathrm{Seq}_k(\mathcal{R}, F_i)$.

Proof. The statement is proven by induction as follows:



Induction base. Let $trans = G_0 \Rightarrow_{b_1, m_1, m'_1} G_1$ be a transformation sequence leading to $F = \exists(i_P : \varnothing \hookrightarrow P, ac_P)$, i.e. $G_1 \vDash F$. Hence, $i_{G_1} \vDash F$ and with $m'_1 \circ i_{R_1} = i_{G_1}$, we get $m'_1 \circ i_{R_1} \vDash F$. By the Shift-lemma, we have $m'_1 \vDash \mathrm{Shift}(i_{R_1}, F)$ and, considering the Seq-construction, $m'_1 \vDash tar_{1,j}$ for a specific $j \in J_1$ (and $tar_{1,j} = \exists(t_j, ac_{T_j})$).

$G_0 \Rightarrow_{b_1, m_1, m'_1} G_1$ implies $m_1 : L_1 \hookrightarrow G_0$ with $m_1 \vDash ac_{L_1}$. With $m'_1 \vDash tar_{1,j}$ and the L-construction, we have $m_1 \vDash \mathrm{L}(b_1, tar_{1,j})$ and, considering the construction above, $m_1 \vDash src'_{1,j}$ (where $src'_{1,j} = \exists(s_j, ac'_{S_j})$). More specifically, there exists a monomorphism $q : S_j \hookrightarrow G_0$ such that $q \circ s_j = m_1$ and $q \vDash ac'_{S_j}$. Because of $m_1 \vDash ac_{L_1}$ and the Shift-construction, $q \circ s_j \vDash ac_{L_1}$ yields $q \vDash \mathrm{Shift}(s_j, ac_{L_1})$, which leads to $q \vDash ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_{L_1})$ and $m_1 \vDash src_{1,j}$ with $src_{1,j} = \exists(s_j, ac'_{S_j} \wedge \mathrm{Shift}(s_j, ac_{L_1}))$. Then, $seq = src_{1,j} \Rightarrow_{b_1} tar_{1,j} \in \mathrm{Seq}_1(\mathcal{R}, F)$ and with $m'_1 \vDash tar_{1,j}$ and $m_1 \vDash src_{1,j}$, we have $trans \vDash seq$.



Inductive step. Let $\text{Seq}_k(\mathcal{R}, F)$ be a set of sequences such that for each k-sequence of transformations *trans* that leads to F, there is a k-sequence of s/t-patterns $seq \in \text{Seq}_k(\mathcal{R}, F)$ such that $trans \vDash seq$.

Consider a k+1-sequence of transformations $trans' = G' \Rightarrow_{b_0,m_0,m'_0} G_0 \Rightarrow_{b_1,m_1,m'_1} ... \Rightarrow_{b_k,m_k,m'_k} G_k$ that leads to F. Then, $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} ... \Rightarrow_{b_k,m_k,m'_k} G_k$ is a k-sequence of transformations that leads to F. By assumption, there is a k-sequence of s/t-patterns $seq = src_1 \Rightarrow_{b_1} ... \Rightarrow_{b_k} tar_k \in \text{Seq}_k(\mathcal{R}, F)$ such that $trans \vDash seq$ and hence, $m_1 \vDash src_1$ with $src_1 = \exists(s : L \hookrightarrow S, ac_S)$ and L being the left side of b_1. By Lemma 14, we have $G_0 \vDash src_{1,\varnothing}$ (where $src_{1,\varnothing} = \exists(i_S, ac_S) = \exists(s \circ i_L, ac_S)$). $G_0 \vDash src_{1,\varnothing}$ implies $i_{G_0} \vDash src_{1,\varnothing}$ and, with $i_{G_0} = m_1 \circ i_L = m'_0 \circ i_{R_0}$, we gain $m'_0 \circ i_{R_0} \vDash src_{1,\varnothing}$. By the Shift-lemma, we have $m'_0 \vDash \text{Shift}(i_{R_0}, src_{1,\varnothing})$.

Delving into the details of the Shift-construction, $m_1 \vDash src_1$ implies the existence of a monomorphism $y : S \hookrightarrow G_0$ such that $y \circ s = m_1$ and $y \vDash ac_S$. With $m'_0 : R_0 \hookrightarrow G_0$ and by $\mathcal{E}'$-$\mathcal{M}$ pair factorization [8], there exist a graph T and monomorphisms $t : R_0 \hookrightarrow T$, $s' : S \hookrightarrow T$, and $y' : T \hookrightarrow G_0$ such that $y' \circ s' = y$, $y' \circ t = m'_0$, and that (t, s') are jointly surjective. Then, we have $m'_0 \vDash \exists(t, \text{Shift}(s', ac_S))$ and, considering the Seq_{k+1}-construction, $m'_0 \vDash tar_{0,j}$ for a specific $j \in J_0$ (and $tar_{0,j} = \exists(t_j, ac_{T_j})$, $s' = s'_j$, $t = t_j$, $T = T_j$ for that j).

Furthermore, $y' \circ s'_j = y$ and $y \circ s = m_1$ imply $y' \circ s'_j \circ s = m_1$. Since $y \vDash ac_S$ and by the Shift-construction ($ac_{T_j} = \text{Shift}(s'_j, ac_S)$), we have $y' \vDash ac_{T_j}$. Thus, $m_1 \vDash src_1^*$ with $src_1^* = \exists(s'_j \circ s, ac_{T_j})$ and, by assumption, $trans \vDash seq^*$ with $seq^* = src_1^* \Rightarrow_{b_1} ... \Rightarrow_{b_k} tar_k$. Also, the existence of y' implies $(m'_0, m_1) \vDash (tar_{0,j}, src_1^*)$.

Because of $G' \Rightarrow_{b_0,m_0,m'_0} G_0$, we have $m_0 : L_0 \hookrightarrow G'$ with $m_0 \vDash ac_{L_0}$. With the L-construction, we get $m_0 \vDash L(b_0, tar_{0,j})$ and, considering the Seq_{k+1}-construction, $m \vDash src'_{0,j}$ (where $src'_{0,j} = \exists(s_j, ac'_{S_j})$) and more specifically, there exists a monomorphism $q : S_j \hookrightarrow G'$ such that $q \circ s_j = m_0$ and $q \vDash ac'_{S_j}$. Because of $m_0 \vDash ac_{L_0}$ and the Shift-construction, $q \circ s_j \vDash ac_L$ yields $q \vDash \text{Shift}(s_j, ac_{L_0})$, which leads to $q \vDash ac'_{S_j} \wedge \text{Shift}(s_j, ac_{L_0})$ and $m_0 \vDash src_{0,j}$ with $src_{0,j} = \exists(s_j, ac'_{S_j} \wedge \text{Shift}(s_j, ac_{L_0}))$.

By construction of $\text{Seq}_{k+1}(\mathcal{R}, F)$, we have $seq' = src_{0,j} \Rightarrow_{b_0} tar_{0,j}, src_1^* \Rightarrow_{b_1} ... \Rightarrow_{b_k} tar_k \in \text{Seq}_{k+1}(\mathcal{R}, \mathcal{F})$ and, with $m_0 \vDash src_{0,j}$, $(m'_0, m_1) \vDash (tar_{0,j}, src_1^*)$, and $trans \vDash seq^*$, we get $trans' \vDash seq'$. This concludes the inductive proof. □

To encode all situations in which a forbidden pattern is violated after a rule application, $\text{Seq}_1(\mathcal{R}, F)$ first builds target patterns for all overlappings of right rule sides and the forbidden pattern (cf. $\text{Shift}(i_{R_u}, F)$ in 1). Then, for each of those target patterns, the respective source pattern – the context before rule application – is computed ($\text{L}(b_u, tar_{u,j})$, 2). Finally, the left application condition of the applied rule is transferred from its left rule side to the context of the source pattern

(Shift(s_j, ac_{L_u}), 3). All pairs of a source and a target pattern thusly created then constitute a 1-sequence of s/t-patterns in $\text{Seq}_1(\mathcal{R}, F)$ (4, 5).

Given the set of sequences created by Seq_k, Seq_{k+1} repeats this process until the sequences reach a fixed length k. In particular, it creates all overlappings of right rule sides and leftmost source patterns of k-sequences (Shift($i_{R_u}, \exists(i_S, ac_S)$), 1) to create target patterns, then adds the newly accumulated context to the leftmost source patterns ($src_1^* = \exists(s_j' \circ s), ac_{T_j}$, 1′). 2–5 mirror the respective computations of Seq_1. Since all involved constructions (particularly Shift and L) produce finite results, Seq_k always yields finite results for a fixed k.

Our construction shows similarities to the notion of *E-concurrent rules* from [8]. In particular, consider a 1-sequence $seq = src \Rightarrow_b tar$ as an element of $\text{Seq}_1(F, \mathcal{R})$ and with $b = \langle (L \hookleftarrow K \hookrightarrow R), ac_L, \text{true} \rangle$, $F = \exists(\varnothing \hookrightarrow P, ac_P)$, $src = \exists(s : L \hookrightarrow S, ac_S)$, and $tar = \exists(t : R \hookrightarrow T, ac_T)$, and consider a hypothetical rule $b' = \langle (P \hookleftarrow P \hookrightarrow P), ac_P, \text{true} \rangle$. Then, given E isomorphic to T with the respective morphisms $t : R \hookrightarrow T$ and $p : P \hookrightarrow T$ as an E-dependency relation for b and b' (see [8]), we get the E-concurrent rule $b \star_E b'$ as $\langle (S \hookleftarrow D \hookrightarrow T), ac_S, \text{true} \rangle$. Put in a more declarative manner, satisfaction of *seq* by a transformation sequence $trans = G_0 \Rightarrow_{b,m,m'} G_1$ is equivalent to the existence of the transformation $G_0 \Rightarrow_{b \star_E b', m, m'} G_1$.

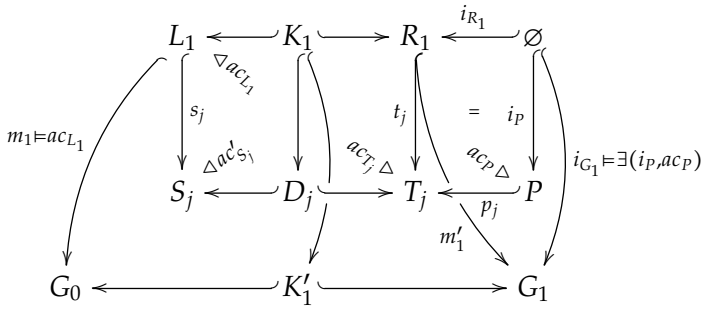
Similar analogies apply for longer sequences and the respective constructions. However, since our analysis specifically requires intermediate patterns in our s/t-pattern sequences, which are not explicitly represented in an E-concurrent rule, we do not use E-concurrent rules to represent transformation sequences. Instead, we rely on s/t-pattern sequences and the Seq-construction introduced above.

Theorem 16 states that all transformation sequences of length k that lead to a forbidden pattern F have a representative s/t-pattern sequence in $\text{Seq}_k(\mathcal{R}, F)$. We also proof that each s/t-pattern sequence is meaningful in the sense that every transformation sequence it represents actually leads to the forbidden pattern:

Lemma 17. *Given a set of rules $\mathcal{R}$, a pattern F, and the set $\text{Seq}_k(\mathcal{R}, F)$, for each k-sequence of s/t-patterns $seq \in \text{Seq}_k(\mathcal{R}, F)$, every transformation sequence $trans$ with $trans \models seq$ leads to F.*

Proof. The statement is proven by induction as follows:

Induction base. Let $seq = src_1 \Rightarrow_{b_1} tar_1$ be an arbitrary 1-sequence of s/t-patterns such that $seq \in \text{Seq}_1(\mathcal{R}, F)$ with $b_1 \in \mathcal{R}$ and $F = \exists(i_P, ac_P)$. Consider an arbitrary transformation sequence $trans = G_0 \Rightarrow_{b_1, m_1, m_1'} G_1$ such that $trans \models seq$. Hence, we have $m_1' \models tar_1$ and, by construction, $m_1' \models \text{Shift}(i_{R_1}, F)$. Then, $m_1' \circ i_{R_1} \models F$ and with $i_{G_1} : \varnothing \hookrightarrow G_1$ and $i_{G_1} = m_1' \circ i_{R_1}$, we gain $i_{G_1} \models F$, implying $G_1 \models F$ and thus, *trans* leads to F. Consequently, for every s/t-pattern sequence *seq* in $Seq_1(\mathcal{R}, F)$, every transformation sequence *trans* with $trans \models seq$ leads to F.



Inductive step. Let $\mathrm{Seq}_k(\mathcal{R}, F)$ be a set of s/t-pattern sequences such that for every s/t-pattern sequence in $\mathrm{Seq}_k(\mathcal{R}, F)$, each transformation sequence *trans* with $trans \vDash seq$ leads to F.

Consider $\mathrm{Seq}_{k+1}(\mathcal{R}, F)$ and $seq' = src_0 \Rightarrow_{b_0} tar_0, src_1^* \Rightarrow_{b_1} ... \Rightarrow_{b_k} tar_k$ as an arbitrary k+1-sequence of s/t-patterns with $seq' \in \mathrm{Seq}_{k+1}(\mathcal{R}, F)$. Consider an arbitrary transformation sequence $trans' = G'_0 \Rightarrow_{b_0,m_0,m'_0} G_0 \Rightarrow_{b_1,m_1,m'_1} ... \Rightarrow_{b_k,m_k,m'_k} G_k$ such that $trans' \vDash seq'$. By construction, there is a k-sequence of s/t-patterns $seq = src_1 \Rightarrow_{b_1} ... \Rightarrow_{b_k} tar_k$ with $seq \in \mathrm{Seq}_k(\mathcal{R}, F)$ and for $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} ... \Rightarrow_{b_k,m_k,m'_k} G_k$, we have $trans \vDash seq$. By assumption, *trans* leads to F and hence, $G_k \vDash F$. Consequently, $trans'$ leads to F, which concludes the inductive proof. □

Example 18. Fig. 4 in Example 13 also serves as an example of one sequence (of many) found in $\mathrm{Seq}_2(\mathcal{R}, F) - F$ as in Fig. 1(b) – for variant 1 of our example. However, the context of the second rule application described by the target pattern tar_2 does not yet take accumulated context of subsequent Seq_k-constructions into account; in particular, it lacks the fourth track (tr1) required in transformation sequences satisfying *seq*. Also, note that our transformation sequence *trans* with $trans \vDash seq$ (Example 13) leads to F (cf. Lemma 17).

For variant 2 of our example system, the Seq_2-construction for the case above would calculate $\mathrm{Shift}(s_1, ac_L)$, where ac_L is the additional negative application condition of graph rule s2a (Fig. 1(f)). Since ac_L – forbidding the existence of a subsequent switch – actually exists in S_1, the result of $\mathrm{Shift}(s_1, ac_L)$ would (upon evaluation) default to false. Since no transformation sequence can satisfy such a sequence of s/t-patterns, the sequence is invalid as a counterexample, which would become apparent in our analysis in step 3 (Section 4.3).

Repeating the Seq_k-construction only accumulates context in backward direction by reverse rule applications (via L). Similarly, acquired context can also be propagated in forward direction. In particular, our construction below uses L to recursively propagate context from the leftmost source pattern over the respective rules through the whole sequence. To justify the process, Lemma 19 establishes that the set of all transformation sequences represented by the s/t-pattern sequences in

Seq_k equals the set of transformation sequences represented by the propagated s/t-pattern sequences in Seq_k. Although that set of transformation sequences remains unchanged, *forward propagation* enriches our symbolic representation in order to discard false negatives in the subsequent analysis step.

Lemma 19 (forward propagation over sequences)**.** *Given a set of graph rules $\mathcal{R}$, a pattern F, and the set of sequences constructed by $\text{Seq}_k(\mathcal{R}, F)$, we describe forward propagation as a function* prop *such that for all $seq \in \text{Seq}_k(\mathcal{R}, F)$, we have $seq \equiv \text{prop}(seq)$.*

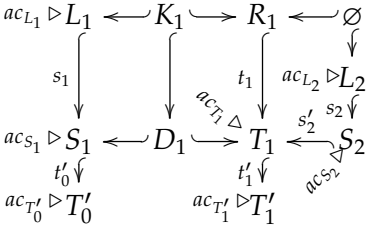
Construction. *We construct* prop *recursively as follows:*

$$\text{prop}(src_1 \Rightarrow_{b_1} tar_1) = src_1 \Rightarrow_{b_1} tar'_1$$

$$\begin{aligned}\text{prop}(src_1 \Rightarrow_{b_1} tar_1, \quad & src_2 \Rightarrow_{b_2} tar_2, ..., src_k \Rightarrow_{b_k} tar_k) \\ = & src_1 \Rightarrow_{b_1} tar'_1, \text{prop}(\text{comb}(tar'_1, src_2) \Rightarrow_{b_2} tar_2, ..., src_k \Rightarrow_{b_k} tar_k),\end{aligned}$$

where $tar'_1 = \text{L}(b_1^{-1}, src_1)$ *and*

$$\begin{aligned}&\text{comb}(\text{ false}, src_2) = \text{false} \\ &\text{comb}(tar'_1, \text{false}) = \text{false} \\ &\text{comb}(tar'_1, src_2) = \exists(t'_1 \circ s'_2 \circ s_2, ac_{T'_1})\end{aligned}$$



as in the diagram on the right, with $src_1 = \exists(t'_0 \circ s_1, ac_{T'_0})$, $tar_1 = \exists(t_1, ac_{T_1})$, $tar'_1 = \exists(t'_1 \circ t_1, ac_{T'_1}) = \text{L}(b_1^{-1}, src_1)$, *and* $src_2 = \exists(s'_2 \circ s_2, ac_S)$.

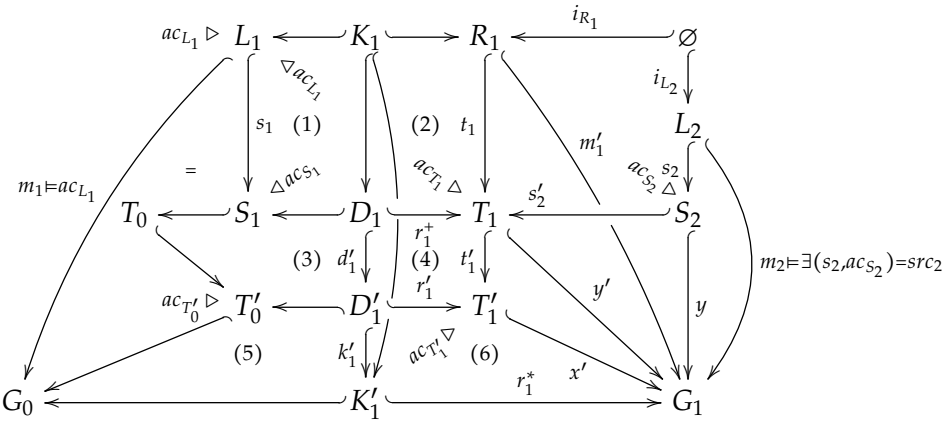
Note that for the first call of prop *on a sequence constructed by* Seq_k*, we have* $src_1 = \exists(s_1, ac_{S_1})$ *and* T'_0 *and* T'_1 *will not exist. For the purpose of* prop *and* comb*,* T'_0 *and* S_1 *can be treated as isomorphic (with* $ac_{T'_0} = ac_{S_1}$*); then,* T_1 *and* T'_1 *are isomorphic as well,* t'_1 *is an isomorphism, and* $ac_{T_1} = ac_{T'_1}$.

Proof. We will show the required statement by structural induction.

Induction base. Let $seq = src_1 \Rightarrow_{b_1} tar_1$ be a s/t-pattern sequence constructed as part of a Seq_k- and prop-construction. Then, $\text{prop}(seq) = src_1 \Rightarrow_{b_1} tar'_1$ with $tar'_1 = L(b_1^{-1}, src_1)$. Further, let $trans = G_0 \Rightarrow_{b_1, m_1, m'_1} G_1$ be a transformation sequence. We need to show $trans \vDash seq \Leftrightarrow trans \vDash \text{prop}(seq)$.

Only if. Assume $trans \vDash seq$. Then, $m_1 \vDash src_1$ and with the L-construction we have $m'_1 \vDash L(b_1^{-1}, src_1)$, i.e. $m'_1 \vDash tar'_1$. $m_1 \vDash src_1$ and $m'_1 \vDash tar'_1$ imply $trans \vDash \text{prop}(seq)$.

If. Assume $trans \vDash \text{prop}(seq)$. Then, $m'_1 \vDash tar'_1$, i.e. $m'_1 \vDash L(b_1^{-1}, src_1)$ and with the L-construction we have $m_1 \vDash src_1$. By construction of Seq_k, there is an underlying pattern $src'_1 = L(b_1, tar_1)$ (with $src'_1 = \exists(s_1, ac_{S_1})$ as in the diagram above) such that $m_1 \vDash src'_1$ and hence, $m'_1 \vDash tar_1$, implying $trans \vDash seq$.



Inductive step. Let $seq' = src_1 \Rightarrow_{b_1} tar_1, src_2 \Rightarrow_{b_2} ... \Rightarrow_{b_{k+1}} tar_{k+1}$ be a k+1-sequence of s/t-patterns constructed as part of a Seq_k-construction. Given a transformation sequence $trans' = G_0 \Rightarrow_{b_1,m_1,m_1'} G_1 \Rightarrow_{b_2,m_2,m_2'} ... \Rightarrow_{b_{k+1},m_{k+1},m_{k+1}'} G_{k+1}$, we need to show $trans' \vDash seq' \Leftrightarrow trans' \vDash \text{prop}(seq')$.

Only if. Assume $trans' \vDash seq'$. Then, we have $m_1 \vDash src_1$, which implies $m_1' \vDash L(b_1^{-1}, src_1)$ and $m_1' \vDash tar_1'$ with $tar_1' = \exists(t_1' \circ t_1, ac_{T_1'})$ and pushouts (3) and (4).

Furthermore, $trans' \vDash seq'$ implies $(m_1', m_2) \vDash (tar_1, src_2)$ (where $tar_1 = \exists(t_1, ac_{T_1})$ and $src_2 = \exists(s_2' \circ s_2, ac_{T_1})$). Thus, there exists a monomorphism $y' : T_1 \hookrightarrow G_1$ such that $y' \circ t_1 = m_1'$, $y' \circ s_2' \circ s_2 = m_2$, and $y' \vDash ac_{T_1}$.

By pushout decomposition, (G_1, r_1^*, y') is a pushout and hence, $y' \circ r_1^+ = r_1^* \circ k_1' \circ d_1'$. Since (4) is also a pushout, there is a monomorphism $x' : T_1' \hookrightarrow G_1$ such that $x' \circ r_1' = r_1^* \circ k_1'$ and $x' \circ t_1' = y'$. Then, by pushout decomposition, (G_1, r_1^*, x') is a pushout (6) over r_1' and k_1', implying $x' \vDash ac_{T_1'}$. Furthermore, we have $x' \circ t_1' \circ s_2' \circ s_2 = y' \circ s_2' \circ s_2 = m_2$, i.e. $m_2 \vDash \text{comb}(tar_1', src_2)$. With $x' \circ t_1' \circ t_1 = m_1'$, we get $(m_1', m_2) \vDash (tar_1', \text{comb}(tar_1', src_2))$ and, by inductive hypothesis, $trans' \vDash \text{prop}(seq')$.

If. Assume $trans' \vDash \text{prop}(seq')$. Then, we have $m_1 \vDash src_1$ and, for the respective target/source pattern, $(m_1', m_2) \vDash (tar_1', \text{comb}(tar_1', src_2))$, i.e. there exists a monomorphism $x' : T_1' \hookrightarrow G_1$ such that $x' \vDash ac_{T_1'}$, $x' \circ t_1' \circ t_1 = m_1'$, $x' \circ t_1' \circ \circ s_2' \circ s_2 = m_2$, and (3) and (4) are pushouts. There also exists a monomorphism $y' : T_1 \hookrightarrow G_1$ with $y' = x' \circ t_1'$ such that $y' \circ t_1 = m_1'$ and $y' \circ s_2' \circ s_2 = m_2$. Since $m_1 \vDash src_1$ and by construction of Seq_k, there is a monomorphism $q : S_1 \hookrightarrow G_0$ such that $q \vDash ac_{S_1}$. Since (4) + (6) and (3) + (5) are pushouts and by the L-construction, we get $y' \vDash ac_{T_1}$ and hence, $(m_1', m_2) \vDash (tar_1, src_2)$. With the inductive hypothesis, we have $trans' \vDash seq'$. □

Example 20. Fig. 5 shows the 2-sequence of s/t-patterns $seq' = \text{prop}(seq)$. The difference to seq (Fig. 4, Example 13) lies in the additional context (tr1) in target pattern tar_2. This also exemplifies the intention of calculating prop: additional information may make a difference in our subsequent analysis (step 3, below).

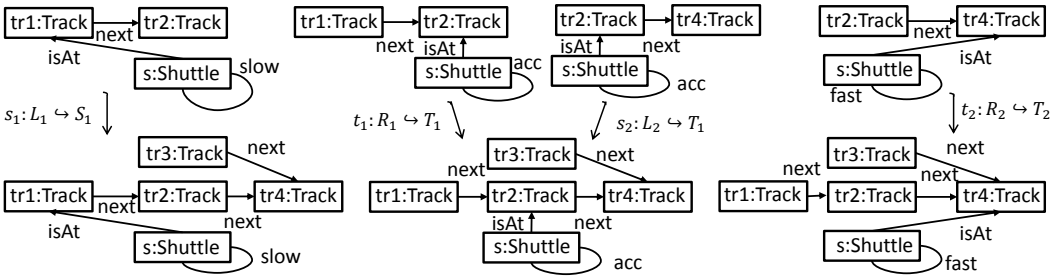


Figure 5: Sequence $seq' = \text{prop}(seq) = src_1 \Rightarrow_{b_1} tar_1, src_2 \Rightarrow_{b_1} tar'_2$, with $seq \equiv seq'$

4.3. Step 3: Analysis of Sequences

Our final and central theorem describes the main result of our approach and formalization: the analysis of s/t-pattern sequences created by our earlier steps.

Theorem 21 (k-inductive invariant checking)**.** *Let $GTS = (\mathcal{R}, TG)$ be a graph transformation system and $\mathcal{F} = \bigwedge_{i\in I} \neg F_i$ ($\mathcal{H} = \bigwedge_{j\in J} \neg H_j$) be composed (assumed) forbidden patterns. Let $\text{Seq}_k(\mathcal{R}, F)$ be the set of k-sequences constructed from the pattern F and $\text{Seq}_k(\mathcal{R}, \mathcal{F}) = \bigcup_{i\in I} \text{Seq}_k(\mathcal{R}, F_i)$. Let, for a source (target) pattern src_z (tar_z), $src_{z,\varnothing}$ ($tar_{z,\varnothing}$) be the reduction of src_z (tar_z) to a pattern.*

$\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$ if, for all sequences $\text{prop}(seq) = src_1 \Rightarrow_{b_1} ... \Rightarrow_{b_k} tar_k$ with $seq \in Seq_k(\mathcal{R}, \mathcal{F})$, one of the following conditions holds:

1. $\exists z, v(1 \le z \le k \wedge (src_{z,\varnothing} \vDash H_v \vee src_{z,\varnothing} \vDash F_v))$.
2. $\exists v(tar_{k,\varnothing} \vDash H_v)$.

Proof. According to Lemma 15, we need to show that for all k-sequences of transformations $G_0 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_k$, it holds that:

$$\exists n(G_k \vDash F_n) \Rightarrow \exists z, v(0 \le z \le k \wedge G_z \vDash H_v) \vee \exists z, v(0 \le z \le k-1 \wedge G_z \vDash F_v)$$

Consider an arbitrary k-sequence of transformations to $\mathcal{R}$ (with corresponding graphs) $trans = G_0 \Rightarrow_{\mathcal{R}} ... \Rightarrow_{\mathcal{R}} G_k$ such that $\exists n(G_k \vDash F_n)$ with, for ease of reading, $F_n = F$. More specifically, $trans = G_0 \Rightarrow_{b_1,m_1,m'_1} ... \Rightarrow_{b_k,m_k,m'_k} G_k$ for rules $b_i \in \mathcal{R}$ and matches (comatches) m_i (m'_i) and $trans$ leads to F.

By Theorem 16, there is a k-sequence of s/t-patterns $seq \in \text{Seq}_k(\mathcal{R}, F)$ with $trans \vDash seq$ and $trans \vDash \text{prop}(seq)$ (Lemma 19). By precondition, one of the following is true:

1. There exist z, v with $1 \le z \le k$ such that $src_{z,\varnothing} \vDash H_v$ or $src_{z,\varnothing} \vDash F_v$. Because of $trans \vDash \text{prop}(seq)$, we have $m_z \vDash src_z$ and, with $m_z : L_z \hookrightarrow G_{z-1}$ and Lemma 14, we gain $G_{z-1} \vDash src_{z,\varnothing}$ and thus, $G_{z-1} \vDash H_v$ or $G_{z-1} \vDash F_v$. Thus, $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.

2. There exists v such that $tar_{k,\varnothing} \vDash H_v$. Because of $trans \vDash \text{prop}(seq)$, we have $m'_k \vDash tar_k$ and, with $m'_k : R_k \hookrightarrow G_k$ and Lemma 14, we gain $G_k \vDash tar_{k,\varnothing}$ and thus, $G_k \vDash H_v$. Thus, $\mathcal{F}$ is a k-inductive invariant for GTS under $\mathcal{H}$.

□

Example 22. For $k = 2$, there is a counterexample for variant 1 (Fig. 5, Example 20). All sequences of length 2 for variant 2 would be discarded by Theorem 21. Hence, $\neg F$ is a 2-inductive invariant for variant 2, but not for variant 1.

Our approach is sound in the sense that for every violating transformation sequence, a symbolic counterexample (s/t-pattern sequence) will be found. It is not necessarily complete: spurious counterexamples can occur, because the theorem above only describes a sufficient condition. Addressing this approach-inherent drawback requires a more complex notion of pattern implication, which is, in general, an undecidable problem [13]. However, previous [5] and current evaluation show the applicability of our approach even without such extensions.

4.4. Implementation

Our implementation closely follows the formalization established above; its basic scheme is shown in Algorithm 1. Given a fixed value for k, a set of graph rules $\mathcal{R}$, and a composed forbidden pattern $\mathcal{F}$ to be verified as a k-inductive invariant under a composed assumed forbidden pattern $\mathcal{H}$, the tool first constructs all 1-sequences of s/t-patterns leading to forbidden patterns by applying Theorem 16 (line 5, first iteration). Next, the tool analyzes the sequences for (assumed) forbidden patterns (Theorem 21, lines 8–10) and discards invalid counterexamples (line 10). The algorithm then iterates (lines 3–10) over the process of prolonging the remaining sequences (Theorem 16, line 5), applying forward propagation (Lemma 19, lines 6–7), and analyzing the sequences (Theorem 21, lines 8–10) until the sequences' length reaches k. If all such sequences have been discarded, $\mathcal{F}$ is a k-inductive invariant under $\mathcal{H}$. Otherwise, the remaining sequences serve as counterexamples. Because of the finiteness of all involved constructions, this algorithm always terminates.

In order to convert our inductive or declarative constructions and theorems into imperative functions suitable to describe our algorithm, we introduced minor changes to their respective signatures. In particular, executing $\text{Seq}_i(\mathcal{R}, F)$ for a specific $i > 1$ requires the result of $\text{Seq}_{i-1}(\mathcal{R}, F)$ (cf. Theorem 16). Hence, in our algorithm, Seq_i has the set of s/t-pattern sequences of length $i - 1$ corresponding to $\mathcal{R}$ and F as a third parameter, the first and second remaining the set of rules and the forbidden pattern. For $i = 1$, that set should be empty. While the signature of prop remains unchanged, the function analyze implements the analysis formalized

by Theorem 21: accepting a s/t-pattern sequence, a set of forbidden patterns, and a set of assumed forbidden patterns as input, its Boolean result signals whether or not any (assumed) forbidden pattern occurs in the sequence in the way described by Theorem 21 such that the sequence can be discarded as a counterexample.

Algorithm 1: Basic scheme of verification algorithm

input : an integer k with $k \geq 1$, a set $\mathcal{R}$ of graph rules, sets $\mathcal{F}$ and $\mathcal{H}$ of (assumed) forbidden patterns
output: a set of k-sequences of s/t-patterns as counterexamples

```
1  foreach F ∈ 𝓕 do
2      sequences[F] ← ∅        /* initialization of map of sequence sets */
3  for i ← 1 to k do
4      foreach F ∈ 𝓕 do
5          sequences[F] ← Seq_i(F, 𝓡, sequences[F])          /* Theorem 16 */
6          foreach seq ∈ sequences[F] do
7              seq ← prop(seq)        /* Lemma 19, no effect for length 1 */
8          foreach seq ∈ sequences[F] do
9              if analyze(seq, 𝓕, 𝓗) then                    /* Theorem 21 */
10                 sequences[F] ← sequences[F] ∖ seq
11 return ⋃_{F∈𝓕} sequences[F]
```

5. Evaluation

In the following, we discuss the experimental evaluation of our approach, which we implemented as an extension of our tool described in [1] and [5]. We considered variants 1 (Example 1 and Appendix A.1) and 2 (Example 2 and Appendix A.2) of our running example as two cases where a k-inductive invariant cannot and can (for $k = 2$) be established, i.e., as cases for an unsafe and a safe system. Variants 3 (Appendix A.3) and 4 (Appendix A.4) present two more elaborate cases, which include sensor faults and a *single fault assumption*.

First, we used our existing tool for the verification of (1-)inductive invariants [5] for all example variants ($k = 1$). Then, we used our extensions implementing the algorithm formalized in this report ($k > 1$). We considered configurations with and without forward propagation[1] (Lemma 19), configurations that compute all counterexamples (denoted by *full*), and configurations that enforce termination as soon as one counterexample of length k has been found (*stop on ce*).

Our results[2] are shown in Table 1. The numbers in brackets denote the number of rules, forbidden properties, and assumed forbidden properties for the respective variants. Columns k, c, and t denote the length of the path for the inductive step, the number of counterexamples, and runtime in seconds, respectively. Column r shows the verification result, which can take the values *false* (*f*) for an unsafe system, *fn* for *false negatives* (spurious counterexamples), *f+fn* for a combination of both, *true* (*t*) for a safe system, or *na* (not applicable).

The term false negative refers to counterexamples, i.e. s/t-pattern sequences of the respective length k, for which there cannot exist a satisfying transformation sequence that describes an actual violation of the k-inductive invariant. Such counterexamples may occur (1) if forward propagation is not considered, which leads to incomplete information during the analysis, and (2) if a more complex (potentially undecidable) notion of pattern implication is required (cf. Section 4.3). However, since all forbidden and assumed forbidden patterns in our examples are of the simple form $F = \exists(i_P : \varnothing \hookrightarrow P, \text{true})$, the second type of false negatives cannot occur here. Hence, all counterexamples resulting from experiments that include forward propagation are true negatives (*f*) and could be instantiated as transformation sequences that violate the k-inductive invariant.

[1] To allow verification without forward propagation, Theorem 21 can be modified by considering all $seq \in \text{Seq}_k(\mathcal{R}, \mathcal{F})$ instead of all $\text{prop}(seq) \in \text{Seq}_k(\mathcal{R}, \mathcal{F})$.

[2] Setup: 64-bit system, two cores at 2.8 GHz, 8 GB main memory, Eclipse 4.5.1, Java 8, Windows 7. Java heap space limit was set to 1 GB, with the exception of variant 4 with forward propagation and $k = 6$, which required 4 GB.

Table 1: Results for 1- and *k*-inductive invariant checking ([5]/current approach)

example (# rules/ forbidden/ assumed)		without forward propagation						with forward propagation					
		full			stop on ce			full			stop on ce		
	k	c	t	r	c*	t	r	c	t	r	c*	t	r
variant 1 (7/1/15), unsafe	1	6	< 1	f	1	< 1	f			na			na
	2	9	< 1	f	1	< 1	f	9	< 1	f	1	< 1	f
	3	47	1.0	f	1	< 1	f	47	1.0	f	1	< 1	f
	4	217	2.8	f	1	< 1	f	217	2.9	f	1	< 1	f
	5	1102	15.4	f+fn	1	< 1	f/fn	1063	15.6	f	1	< 1	f
	6	6211	95.7	f+fn	1	< 1	f/fn	5551	94.1	f	1	< 1	f
variant 2 (7/1/15), safe	1	6	< 1	f	1	< 1	f			na			na
	2	0	< 1	t	0	< 1	t	0	< 1	t	0	< 1	t
	..	0	< 1	t	0	< 1	t	0	< 1	t	0	< 1	t
variant 3 (10/1/16), unsafe	1	9	< 1	f	1	< 1	f			na			na
	2	15	< 1	f+fn	1	< 1	f/fn	6	< 1	f	1	< 1	f
	3	128	1.5	f+fn	1	< 1	f/fn	27	1.1	f	1	< 1	f
	4	737	10.3	f+fn	1	< 1	f/fn	100	2.7	f	1	< 1	f
	5	4389	78.7	f+fn	1	< 1	f/fn	444	12.1	f	1	< 1	f
	6	28514	741.0	f+fn	1	< 1	f/fn	2011	74.2	f	1	1	f
variant 4 (10/1/16), safe	1	6	< 1	f	1	< 1	f			na			na
	2	9	< 1	fn	1	< 1	fn	0	< 1	t	0	< 1	t
	..	..	..	fn	1	< 1	fn	0	< 1	t	0	< 1	t

For the *full* case with forward propagation, the property is not a *k*-inductive invariant (up to $k = 6$) for the erroneous variant 1, as expected (cf. Examples 1 and 22). The corrections resulting in variant 2 (Example 2) lead to a safe system where the property can be established as a 2-inductive invariant. Likewise, variant 4 is a fixed version of the erroneous variant 3. While the computational effort for variants 2 and 4 is minimal, variants 1 and 3 show strongly increasing numbers for counterexamples and computation time. This will almost always be the case for erroneous systems. However, in both cases, execution with the *stop on ce* option will quickly return results for manual inspection of unsafe systems.

As explained before, execution without forward propagation can lead to false negatives. In particular, without forward propagation the safety property cannot be established even for the (actually safe) variant 4. Also, the high number of false negatives in variant 3 leads to even higher numbers of (false) counterexamples for subsequent values of *k* and significantly higher computation times.

To establish the base of induction (see Lemma 10) for a graph grammar with an initial graph, we used the model checker GROOVE [9] to (successfully) check all paths of length 1 ($k - 1 = 2 - 1$) from the initial graph for variants 2 and 4.

An important issue is the choice of *k* for the verification. There exist cases (variants 2 and 4) where the desired property can be established as a *k*-inductive

invariant for small k, but not as a 1-inductive invariant. Even for the (erroneous) variant 3 (with forward propagation), increasing k from 1 to 2 reduces the number of counterexamples. Closer inspection shows that several counterexamples for $k = 1$ rely on the application of rule a2f' and that the respective s/t-pattern sequences (of length 1) do not appear as part of sequences (counterexamples) of greater length (for increasing k). In other words, increasing the value of k (beyond 1) yields the information that rule a2f' is not (ultimately) responsible for the violation of our forbidden pattern. More generally, verification with increasing values for k may allow for a more precise analysis of an erroneous system.

Therefore, if the estimated value of k for an invariant is not known, we suggest to verify systems with increasing k, starting with 1, and to use counterexamples to fix system errors, as seen in variants 1 (fixed in variant 2) and 3 (fixed in 4). While we are confident that the technique is also applicable for different examples of similar size and values for k, we cannot yet generalize that claim for larger examples. However, since the approach's complexity is independent from a system's state space, it may be applied where approaches based on the state space are impractical.

6. Conclusion and Outlook

We presented an approach for automatic verification of k-inductive invariants that supports reasonably expressive graph rules and properties. We have proven and implemented our approach, which employs a finite symbolic encoding of traces. Further, our evaluation has demonstrated that k-inductive invariants can be established for some examples where inductive invariants are not sufficient.

Moving on, we plan to study further options to enrich k-sequences, optimize our algorithm, and apply suitable counterexample-guided refinement techniques.

Acknowledgments
This work was partially developed in the course of the project Correct Model Transformations II (GI 765/1-2), which is funded by the Deutsche Forschungsgemeinschaft.

We would like to thank Leen Lambers for her comprehensive feedback on a draft version of this report.

References

[1] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the 28th International Conference on Software Engineering (ICSE)*, New York, 2006. ACM.

[2] C. Blume, H. Bruggink, D. Engelke, and B. König. Efficient Symbolic Implementation of Graph Automata with Applications to Invariant Checking. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformations*, volume 7562 of *LNCS*, pages 264–278, Berlin/Heidelberg, 2012. Springer.

[3] I. B. Boneva, J. Kreiker, M. E. Kurban, A. Rensink, and E. Zambon. Graph Abstraction and Abstract Graph Transformations (Amended Version). Technical Report TR-CTIT-12-26, University of Twente, Enschede, 2012.

[4] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software Verification Using k-Induction. In E. Yahav, editor, *Static Analysis*, pages 351–368, Berlin/Heidelberg, 2011. Springer.

[5] J. Dyck and H. Giese. Inductive Invariant Checking with Partial Negative Application Conditions. In F. Parisi-Presicce and B. Westfechtel, editors, *Graph Transformation*, volume 9151 of *LNCS*, pages 237–253, Cham, 2015. Springer.

[6] J. Dyck and H. Giese. *k*-Inductive Invariant Checking for Graph Transformation Systems. In D. Plump and J. de Lara, editors, *Graph Transformation*, volume 10373 of *LNCS*, pages 142–158, Cham, 2017. Springer.

[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Secaucus, 2006.

[8] H. Ehrig, U. Golas, A. Habel, L. Lambers, and F. Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions. part 1: Parallelism, concurrency and amalgamation. *Math. Struct. Comput. Sci.*, 24, 2014.

[9] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *Int. J. Softw. Tools Technol. Transf.*, 14(1):15–40, 2012.

[10] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Math. Struct. Comput. Sci.*, 19:1–52, 2009.

[11] B. König and V. Kozioura. Augur 2 – A New Version of a Tool for the Analysis of Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 211:201–210, 2008.

[12] B. König and J. Stückrath. A General Framework for Well-Structured Graph Transformation Systems. In P. Baldan and D. Gorla, editors, *CONCUR 2014 – Concurrency Theory*, volume 8704 of *LNCS*, pages 467–481, Berlin/Heidelberg, 2014. Springer.

[13] K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, University of Oldenburg, 2009.

[14] A. Schmidt and D. Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 – The Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, Heidelberg, 2003. Springer.

[15] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144, Berlin/Heidelberg, 2000. Springer.

[16] D. Steenken. *Verification of lnfinite-State Graph Transformation Systems via Abstraction*. PhD thesis, University of Paderborn, 2015.

A. Running Example and Variants

In the following, we outline all details of the four variants considered in our running example and evaluation.

As established in Example 1, variant 1 (Section A.1) exhibits unsafe behavior, because its rules do not check for switches ahead of a shuttle. Variant 2 (Section A.2) is a fixed version of variant 1 such that our property can be established as a 2-inductive invariant. Variants 3 (Section A.3) and 4 (Section A.4), which were introduced in Section 5 for evaluation purposes, describe two more elaborate cases involving a single fault assumption and sensor faults when checking for switches. Similar to variants 1 and 2, variant 4 is a corrected version of the unsafe system in variant 3.

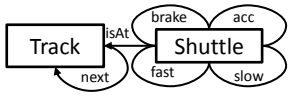
For all variants, we have an assumed forbidden property excluding the existence of multiple shuttles, which requires additional clarification. Since shuttle behavior is completely independent of other shuttles, the forbidden property could, in theory, also be established for multiple shuttles. Directly including the possibility of multiple shuttles collides with the consideration of subsequent reverse rule applications (of k-inductive invariant checking), because our sequential approach does not (yet) encode the parallel nature of behavior (here: shuttle movement).

A.1. Variant 1

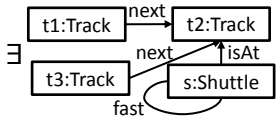
All elements of variant 1 are depicted in Figure 6, including:

- the type graph: Figure 6(a),
- the forbidden pattern: Figure 6(b),
- the state diagram denoting the shuttle's transitions between modes slow, acc(elerate), fast, and brake: Figure 6(c),
- the rules, realizing movement and mode transitions: Figures 6(d), 6(e), 6(f), 6(g), 6(h), 6(i), and 6(j),
- the assumed forbidden patterns (which, negated and conjunctively joined, are a 1-inductive invariant for the system): Figure 6(k).

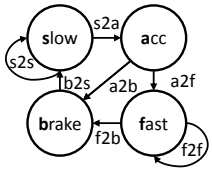
Due to the absence of negative application conditions checking for switches ahead of a shuttle, the forbidden pattern is not a k-inductive invariant for variant 1.



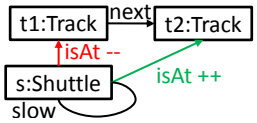
(a) Type graph for variants



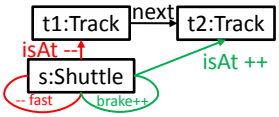
(b) Forbidden graph constraint F



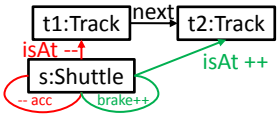
(c) Mode transitions



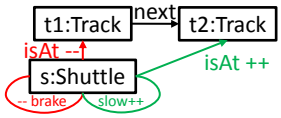
(d) Rule s2s



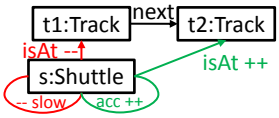
(e) Rule f2b



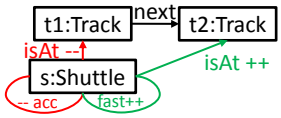
(f) Rule a2b



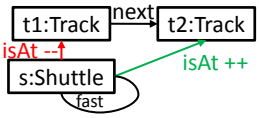
(g) Rule b2s



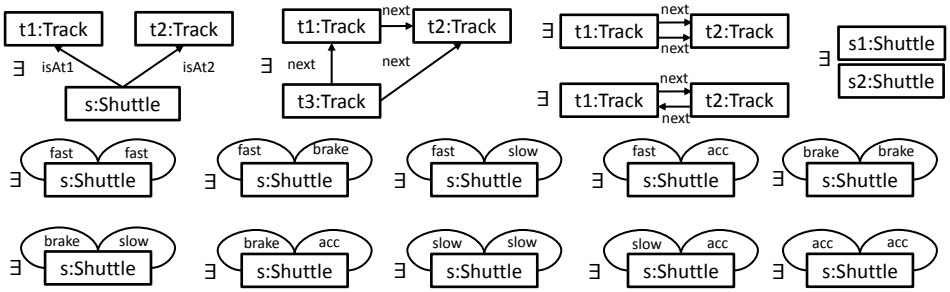
(h) Rule s2a



(i) Rule a2f



(j) Rule f2f



(k) Assumed forbidden patterns

Figure 6: Rules and properties of variant 1

A.2. Variant 2

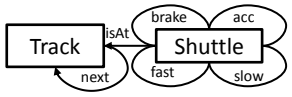
All elements of variant 2 are depicted in Figure 7, including:

- the type graph: Figure 7(a),
- the forbidden pattern: Figure 7(b),
- the state diagram denoting the shuttle's transitions between modes slow, acc(elerate), fast, and brake: Figure 7(c),
- the rules, realizing movement and mode transitions: Figures 7(d), 7(e), 7(f), 7(g), 7(h), 7(i), and 7(j),
- the assumed forbidden patterns (which, negated and conjunctively joined, are a 1-inductive invariant for the system): Figure 7(k).

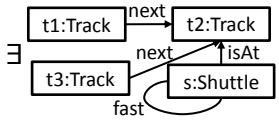
In contrast to variant 1, this variant's rules (specifically, s2a, a2f, and f2f) include negative application conditions to check for switches when transitioning to modes acc or fast. As shown in our evaluation, the forbidden property is a 2-inductive invariant for this variant.

To establish the forbidden property for a graph grammar starting from an initial graph, it has to be a k-inductive invariant (inductive step) and we have to check the property's absence in the first $k-1$ steps from the initial state (base of induction, see Lemma 10). As explained in Section 5, we used the model checker GROOVE to check the base of induction. We modeled the specific initial topology depicted in Figure 8(a), obtained the state space (Figure 8(b)) reachable by one step (by *bounded model checking* for $k-1 = 2-1$) from the specific initial topology, and found no occurrence of our forbidden property. This proves the respective graph grammar's safety for that initial topology: all reachable graphs will not have a shuttle driving on a switch while in mode fast.

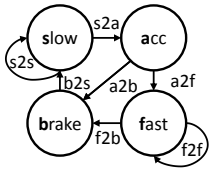
We can argue that beyond that specific topology, all graph grammars that share the same shuttle behavior in the first step ($k-1$) can be considered safe. The specific track topology beyond the tracks reachable by the shuttle in one step has no effect on the base of induction. By definition, it also has no effect on the validity of the 2-inductive invariant. Thus, we can even infer the absence of the forbidden property for an inifinite number of graph grammars.



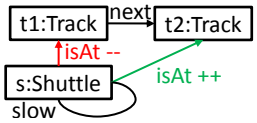
(a) Type graph for variants



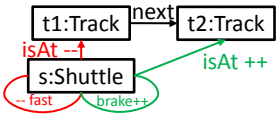
(b) Forbidden graph constraint F



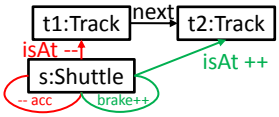
(c) Mode transitions



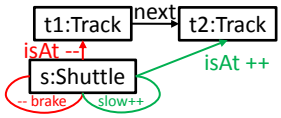
(d) Rule s2s



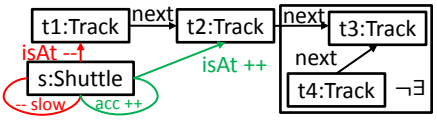
(e) Rule f2b



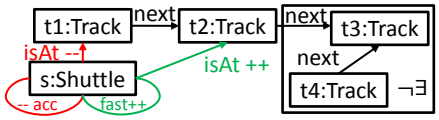
(f) Rule a2b



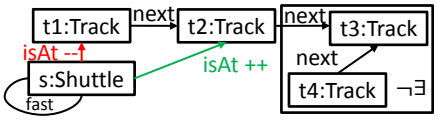
(g) Rule b2s



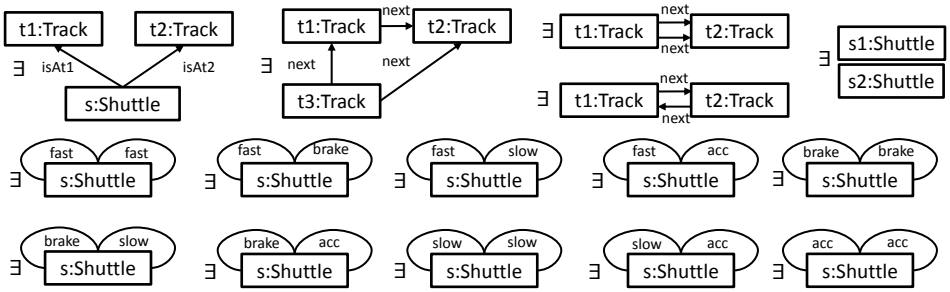
(h) Rule s2a



(i) Rule a2f

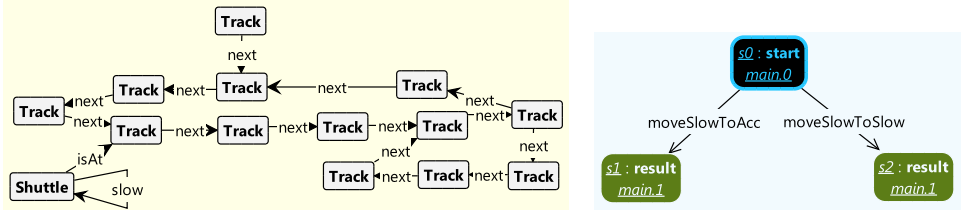


(j) Rule f2f



(k) Assumed forbidden patterns

Figure 7: Rules and properties of variant 2



(a) Initial graph (and track topology) (b) State space

Figure 8: Example topology and generated state space for $k - 1 = 1$

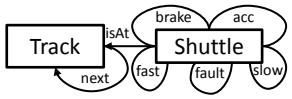
A.3. Variant 3

All elements of variant 3 are depicted in Figure 9, including:

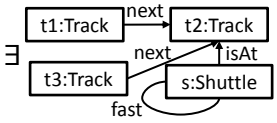
- the type graph: Figure 9(a),
- the forbidden pattern: Figure 9(b),
- the state diagram denoting the shuttle's transitions between modes slow, acc(elerate), fast, and brake: Figure 9(c),
- the rules, realizing movement and mode transitions: Figures 9(d), 9(e), 9(f), 9(g), 9(h), 9(i), 9(j), 9(k), 9(l), and 9(m),
- an additional assumed forbidden pattern H_{16} that models the single fault assumption: Figure 9(n),
- the assumed forbidden patterns (which, negated and conjunctively joined (including $\neg H_{16}$), are a 1-inductive invariant for the system): Figure 9(o).

In comparison to variant 2 (as depicted in Figure 7), the rule a2f has an additional negative application condition to prevent the shuttle from driving in mode fast when there is a switch directly ahead. However, rule f2f does not have such a condition, which will result in the forbidden property not being a k-inductive invariant. The additional rules s2a', a2f', and f2f' model an erroneous execution of the respective rules due to a sensor fault; due to missing negative application conditions those rules do not check for a switch.

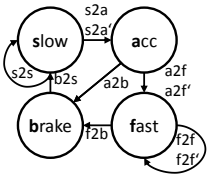
The additional assumed forbidden property (H_{16}) depicted in Figure 9(n) models the assumption that only a single (sensor) fault occurs. Under this assumption (and the other assumed forbidden properties), the forbidden property should be a 2-inductive invariant; however, the missing condition in rule f2f leads to a missing check that is not registered as a sensor fault. Thus, the assumption of a single fault does not exclude a second sensor fault in rule f2f and hence, the property is not a 2-inductive invariant.



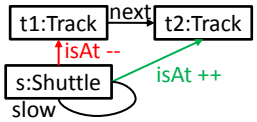
(a) Type graph



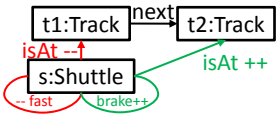
(b) Forbidden graph constraint F



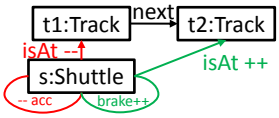
(c) Mode transitions



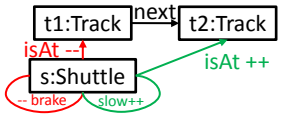
(d) Rule s2s



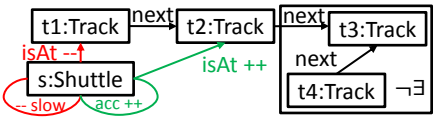
(e) Rule f2b



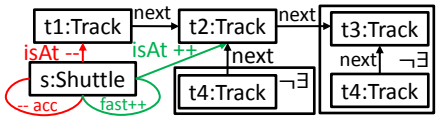
(f) Rule a2b



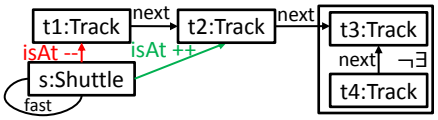
(g) Rule b2s



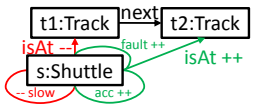
(h) Rule s2a



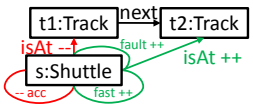
(i) Rule a2f



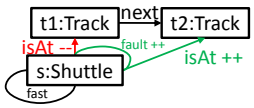
(j) Rule f2f



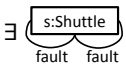
(k) Rule s2a'



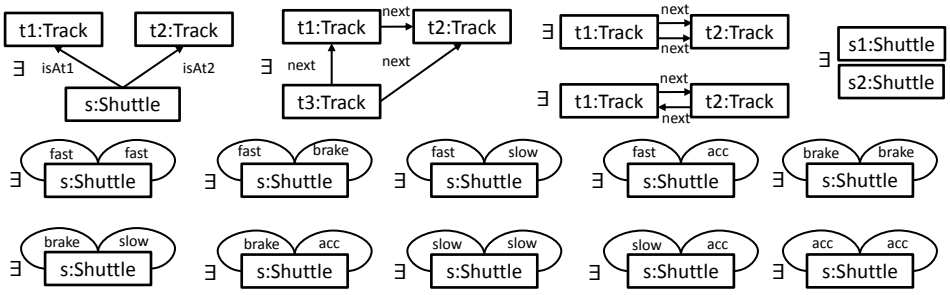
(l) Rule a2f'



(m) Rule f2f'



(n) H_{16}



(o) Assumed forbidden patterns

Figure 9: Rules and properties of variant 3

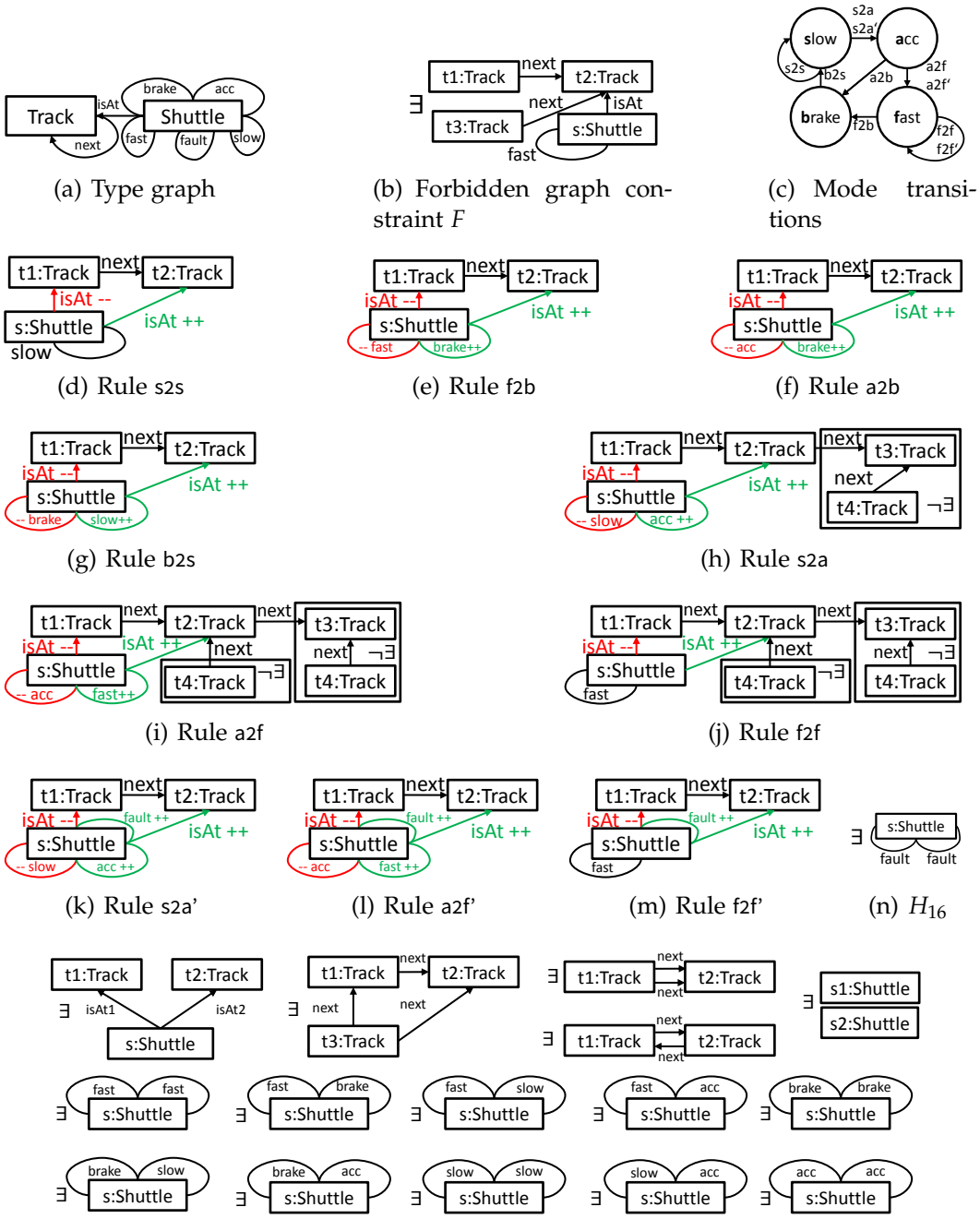
A.4. Variant 4

All elements of variant 4 are depicted in Figure 10, including:

- the type graph: Figure 10(a),
- the forbidden pattern: Figure 10(b),
- the state diagram denoting the shuttle's transitions between modes slow, acc(elerate), fast, and brake: Figure 10(c),
- the rules, realizing movement and mode transitions: Figures 9(d), 10(e), 10(f), 10(g), 10(h), 10(i), 10(j), 10(k), 10(l), and 10(m),
- an additional assumed forbidden pattern H_{16} that models the single fault assumption: Figure 10(n),
- the assumed forbidden patterns (which, negated and conjunctively joined (including $\neg H_{16}$), are a 1-inductive invariant for the system): Figure 9(o).

In comparison to variant 3 (Figure 9), rule f2f has been fixed; the missing negative application condition has been added. Thus, the forbidden property is a 2-inductive invariant under the assumed forbidden properties.

As explained in Section 5, we also used GROOVE to check the absence of the forbidden property in the first $k - 1 = 1$ step(s) from the specific initial topology depicted in Figure 8(a). Then, with the nature of our property as a 2-inductive invariant we can conclude that the forbidden property does not occur (see Lemma 10).



(a) Type graph

(b) Forbidden graph constraint F

(c) Mode transitions

(d) Rule s2s

(e) Rule f2b

(f) Rule a2b

(g) Rule b2s

(h) Rule s2a

(i) Rule a2f

(j) Rule f2f

(k) Rule s2a'

(l) Rule a2f'

(m) Rule f2f'

(n) H_{16}

(o) Assumed forbidden patterns

Figure 10: Rules and properties of variant 4

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
118	978-3-86956-405-0	**Probabilistic timed graph transformation systems**	Maria Maximova, Holger Giese, Christian Krause
117	978-3-86956-401-2	**Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud" 2016**	Stefan Klauck, Fabian Maschler, Karsten Tausche
116	978-3-86956-397-8	**Die Cloud für Schulen in Deutschland : Konzept und Pilotierung der Schul-Cloud**	Jan Renz, Catrina Grella, Nils Karn, Christiane Hagedorn, Christoph Meinel
115	978-3-86956-396-1	**Symbolic model generation for graph properties**	Sven Schneider, Leen Lambers, Fernando Orejas
114	978-3-86956-395-4	**Management Digitaler Identitäten : aktueller Status und zukünftige Trends**	Christian Tietz, Chris Pelchen, Christoph Meinel, Maxim Schnjakin
113	978-3-86956-394-7	**Blockchain : Technologie, Funktionen, Einsatzbereiche**	Tatiana Gayvoronskaya, Christoph Meinel, Maxim Schnjakin
112	978-3-86956-391-6	**Automatic verification of behavior preservation at the transformation level for relational model transformation**	Johannes Dyck, Holger Giese, Leen Lambers
111	978-3-86956-390-9	**Proceedings of the 10th Ph.D. retreat of the HPI research school on service-oriented systems engineering**	Christoph Meinel, Hasso Plattner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Emmanuel Müller
110	978-3-86956-387-9	**Transmorphic : mapping direct manipulation to source code transformations**	Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld
109	978-3-86956-386-2	**Software-Fehlerinjektion**	Lena Feinbube, Daniel Richter, Sebastian Gerstenberg, Patrick Siegler, Angelo Haller, Andreas Polze
108	978-3-86956-377-0	**Improving Hosted Continuous Integration Services**	Christopher Weyand, Jonas Chromik, Lennard Wolf, Steffen Kötte, Konstantin Haase, Tim Felgentreff, Jens Lincke, Robert Hirschfeld
107	978-3-86956-373-2	**Extending a dynamic programming language and runtime environment with access control**	Philipp Tessenow, Tim Felgentreff, Gilad Bracha, Robert Hirschfeld

ISBN 978-3-86956-406-7
ISSN 1613-5652