Inductive Invariant Checking with Partial Negative Application Conditions

Johannes Dyck(✉) and Holger Giese

Hasso Plattner Institute at the University of Potsdam, Potsdam, Germany
{Johannes.Dyck,Holger.Giese}@hpi.de

Abstract. Graph transformation systems are a powerful formal model to capture model transformations or systems with infinite state space, among others. However, this expressive power comes at the cost of rather limited automated analysis capabilities. The general case of unbounded many initial graphs or infinite state spaces is only supported by approaches with rather limited scalability or expressiveness. In this paper we improve an existing approach for the automated verification of inductive invariants for graph transformation systems. By employing partial negative application conditions to represent and check many alternative conditions in a more compact manner, we can check examples with rules and constraints of substantially higher complexity. We also substantially extend the expressive power by supporting more complex negative application conditions and provide higher accuracy by employing advanced implication checks. The improvements are evaluated and compared with another applicable tool by considering three case studies.

1 Introduction

Graph transformation systems are a powerful formal model to capture model transformations, systems with reconfiguration, or systems with infinite state space, among others. However, the expressive power of graph transformation systems comes at the cost of rather limited automated analysis capabilities.

While for graph transformation systems with finite state space of moderate size certain model checkers can be used (e.g., [1,2]), in the general case of unbounded many initial graphs or an infinite state space only support by techniques with rather limited scalability or expressiveness exists.

There is a number of automated approaches that can handle infinite state spaces by means of abstraction [3–6], but they are considerably limited in expressive power as they only support limited forms of negative application conditions at most. Tools only targeting invariants [7,8] also only support limited forms of negative application conditions at most; in some cases additional limitations concerning the graphs of the state space apply (cf. [7]). On the other hand the

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.

F. Parisi-Presicce and B. Westfechtel (Eds.): ICGT 2015, LNCS 9151, pp. 237–253, 2015.
DOI: 10.1007/978-3-319-21145-9_15

SeekSat/ProCon tool [9,10] is able to prove correctness of graph programs with respect to pre- and postconditions specified as nested graph constraints without such limitations, but requires potentially expensive computations.

In this paper we present improvements of our existing approach introduced in [8] for the automated verification of inductive invariants for graph transformation systems. Inductive invariants are properties whose validity before the application of a graph rule implies their validity thereafter. Our general approach involves the construction of a violation of the invariant after application of a graph rule, represented in a symbolic way (target pattern), followed by calculation of the symbolic state before rule application (source pattern). If a violation can then be found in all such source patterns, the rule does not violate the inductive invariant; otherwise, it does and the construction yields a witness. Since inductive invariants are checked with respect to the capability of individual rules to violate or preserve them, this technique avoids the computationally expensive computation of state spaces and can even handle infinite systems.

By employing partial negative application conditions to represent and check many alternative conditions in a more compact manner, our approach is now able to check examples with rules and constraints of substantially higher complexity. Our improvements also provide higher accuracy by employing advanced implication checks and extend expressive power by supporting more complex negative application conditions. While not as expressive as the general concept of nested graph conditions [10], there is a significant number of examples [8,11,12] for which the supported level of expressive power is sufficient. Of those, we employ three case studies concerned with car platooning and model transformations to evaluate our improvements and to compare them with the SeekSat/ProCon tool, demonstrating that our approach shows better scalability for certain cases.

The paper is organized as follows: The formal foundations are introduced in Sect. 2. Our restrictions and important constructions in our algorithms are explained in Sect. 3. Section 4 presents the employed inductive invariant checking scheme with its formal justification. Section 5 presents our evaluation, with Sect. 6 then discussing related work. Finally, Sect. 7 provides a summary and outlook on possible future work. Proofs and additional prerequisites concerning the formal model can be found in the extended version [13].

2 Foundations

This section shortly describes foundations of graph transformation systems we use in our verification approach. For additional definitions, we refer to [13].

The formalism used herein (cf. [14]) considers a *graph* $G = (V, E, s, t)$ to consist of sets of nodes, edges and source and target functions $s, t : E \rightarrow V$. A *graph morphism* $f : G_1 \rightarrow G_2$ consists of two functions mapping nodes and edges, respectively, that preserve source and target functions. In this paper we put special emphasis on *injective morphisms* (or *monomorphisms*), denoted $f : G_1 \hookrightarrow G_2$, and consider *typed graphs*, i.e. graphs typed over a *type graph* TG by a typing morphism $type : G \rightarrow TG$ and *typed graph morphisms* that preserve the typing morphism. We also adopt the concept of *partial monomorphisms*.

Definition 1 (partial monomorphism ([9], adjusted)). *A* partial monomorphism *$p : A \rightharpoonup B$ is a 2-tuple $p = \langle a, b\rangle$ of monomorphisms a, b with $dom(a) = dom(b)$, $dom(p) = codom(a)$, and $codom(p) = codom(b)$. The* interface *of p refers to the common domain of a and b, i.e., $iface(p) = dom(a) = dom(b)$. A partial monomorphism $p = \langle a, b\rangle$ is said to be a total monomorphism b, if a is an isomorphism, i.e. a bijective morphism.*

Thus, the partial monomorphism $p : A \rightharpoonup B$ describes an inclusion of a subgraph A' of A in B. With partial monomorphisms we can define partial application conditions, which, similar to nested application conditions [10], describe conditions on morphisms. *Graph constraints*, on the other hand, describe conditions on graphs.

$$\begin{tikzcd}
A' \arrow[r, hook, "a"'] \arrow[d, hook, "b"'] & A \arrow[dl, harpoon, "p=\langle a,b\rangle"] \\
B &
\end{tikzcd}$$

Definition 2 (partial application condition ([15], extended to partial morphisms)). *A* partial application condition *is inductively defined as follows:*

1. *For every graph P, true is a partial application condition over P.*
2. *For every partial monomorphism $a : P \rightharpoonup C$ with $a = \langle p, c\rangle$ and monomorphisms $p : P' \hookrightarrow P$ and $c : P' \hookrightarrow C$ and every partial application condition ac over C, $\exists(a, ac)$ is a partial application condition over P.*
3. *For partial application conditions ac, ac_i over P with $i \in I$ (for all index sets I), $\neg ac$ and $\bigwedge_{i\in I} ac_i$ are partial application conditions over P.*

$$\begin{tikzcd}
P' \arrow[r, hook, "p"'] \arrow[d, hook, "c"'] & P \arrow[dl, harpoon, "a=\langle p,c\rangle"] \\
ac \triangleright C &
\end{tikzcd}
\qquad
\begin{tikzcd}
P' \arrow[r, hook, "p"'] \arrow[d, hook, "c"'] & P \arrow[dl, harpoon, "a=\langle p,c\rangle"] \arrow[dr, "g"] & \\
ac \triangleright C \arrow[rr, hook, "q \models ac"'] & & G
\end{tikzcd}$$

Satisfiability *of partial application conditions is inductively defined as follows:*

1. *Every morphism satisfies true.*
2. *A morphism $g : P \to G$ satisfies $\exists(a, ac)$ over P with $a : P \rightharpoonup C$ with $a = \langle p, c\rangle$ if there exists an injective $q : C \hookrightarrow G$ such that $q \circ c = g \circ p$ and q satisfies ac.*
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 $(i \in I)$.*

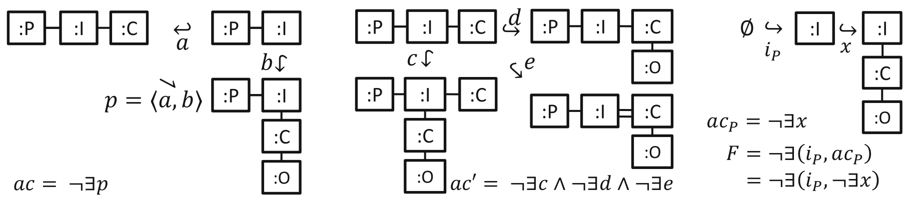
We write $g \models ac$ to denote that the morphism g satisfies ac.

Two application conditions ac and ac' are equivalent*, denoted by $ac \equiv ac'$, if for all morphisms $g : P \to G$, $g \models ac$ if and only if $g \models ac'$.*

If all morphisms involved in a partial application condition are total morphisms we say that it is a total application condition.

$\exists p$ abbreviates $\exists(p, true)$. $\forall(p, ac)$ abbreviates $\neg\exists(p, \neg ac)$.

Definition 3 (graph constraint [10]). *A graph constraint is an application condition over the empty graph $\varnothing$. A graph G then satisfies such a condition if the initial morphism $i_G : \varnothing \hookrightarrow G$ satisfies the condition.*



(a) Partial condition (b) Total condition (c) Graph constraint

Fig. 1. Partial and total conditions and graph constraint

Example 4. Figure 1 shows an example from a software refactoring context (cf. [12]) with node types P, I, C, O standing for Package, Interface, Class, and Operation, respectively. Although equivalent, the partial condition ac in Fig. 1(a) is much more compact—and also less expensive in computation—when compared to the total condition ac′ in Fig. 1(b). Both conditions describe the absence of an implementing class and contained operation for the interface. Further, Fig. 1(c) shows a graph constraint F, which forbids the existence of an interface without an implementing class containing an operation.

Application conditions can also be used in graph rules, which are used to transform graphs. Finally, a *graph transformation system* consists of a number of rules and, in our case of *typed graph transformation systems*, of a type graph.

Definition 5 (rules and transformations [15]**).** *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 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).*

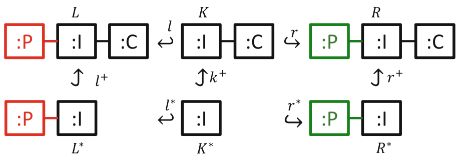
$$\begin{tikzcd}
ac_L \triangleright L \arrow[d, "m \models ac_L"'] \arrow[r, phantom, "(1)" below, shift right=6] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d] \arrow[r, phantom, "(2)" below, shift right=6] & R \triangleleft ac_R \arrow[d, "m' \models ac_R"] \\
G & D \arrow[l, hook', "l'"] \arrow[r, hook, "r'"'] & H
\end{tikzcd}$$

A direct transformation consists of two pushouts (1) and (2) such that $m \models ac_L$ and $m' \models ac_R$. We write $G \Rightarrow_{b,m,m'} H$ and say that $m : L \rightarrow G$ is the match of b in G and $m' : R \rightarrow H$ is the comatch of b in H. We also write $G \Rightarrow_{b,m} H$, $G \Rightarrow_m H$ or $G \Rightarrow H$ to express that there exist m', m or b such that $G \Rightarrow_{b,m,m'} H$.

We also introduce the concept of a *reduced rule*, which basically is a rule without certain elements irrelevant for a specific application via a match once the applicability for that match is ensured. By using reduced rules, we can reduce the effort necessary for verification, as will be shown later.

Definition 6 (reduced rule). *Given a plain rule $b = \langle L \hookleftarrow K \hookrightarrow R\rangle$, we define a* reduced rule *of b as a rule $b^* = \langle L^* \hookleftarrow K^* \hookrightarrow R^*\rangle$ with injective morphisms $r^+ : R^* \hookrightarrow R$, $l^+ : L^* \hookrightarrow L$, and $k^+ : K^* \hookrightarrow K$ such that for* *all graphs G, H and injective morphisms m, m' it holds that $G \Rightarrow_{b,m,m'} H \Leftrightarrow G \Rightarrow_{b^*, m \circ l^+, m' \circ r^+} H$.*

$$\begin{tikzcd}
L \arrow[dd, bend right, "m"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[dd, bend left] & R \arrow[dd, bend left, "m'"] \\
L^* \arrow[u, hook, "l^+"'] & K^* \arrow[l, hook', "l^*"] \arrow[r, hook, "r^*"] \arrow[u, hook, "k^+"] & R^* \arrow[u, hook, "r^+"] \\
G & D \arrow[l, hook'] \arrow[r, hook] & H
\end{tikzcd}$$



Example 7. The figure above shows a plain rule describing the replacement of a package containing an existing interface and class. In general, a corresponding reduced rule (also depicted) can be constructed by choosing K^* as any subgraph of K whose images under l and r include all nodes attached to edges to be deleted or created and then constructing L^* and R^* as the pushout complements of $\langle k^+, l \rangle$ and $\langle k^+, r \rangle$, respectively.

3 Restrictions, Constructions, and Implication

With the foundations established, we will now introduce certain restrictions that apply to our specifications and the main constructions used by our algorithms.

The most important adjustments are concerned with the notion of rules and application conditions. Since most application conditions that will be encountered in this paper have the same structure, we define a special kind of negative application conditions without additional nesting. In comparison to our previous work [8], this is a significant difference in expressive power, as [8] allowed only negative application conditions with each having a node and an edge, at most.

Definition 8 (composed negative application condition). *A* composed negative application condition *is an application condition of the form $ac = \bigwedge_{i \in I} \neg\exists a_i$ for partial monomorphisms a_i of a common domain. An individual condition $\neg\exists a_i$ is called* negative application condition. *A (composed) total negative application condition is a (composed) negative application condition including only total graph morphisms.*

Our properties for verification are described by so-called forbidden patterns:

Definition 9 (pattern). *A* pattern *is a graph constraint of the form $F = \exists(\varnothing \hookrightarrow P, ac_P)$, with P being a graph and ac_P a composed total 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*.*

We also allow graph transformation systems to be equipped with a special variant of composed forbidden pattern called *composed guaranteed pattern*. Such a pattern is a constraint whose validity is guaranteed by some external means or additional knowledge about the system under verification.

While our specification language concerning patterns and application conditions is more limited than the general concept of nested application conditions [10], the level of expressive power we support is sufficient to verify a number of case studies [8,11,12]. On the other hand, the following additional limitations in our approach (except for the second) do not result in a loss of expressive power [10,14,15]:

Morphisms in application conditions (Definition 2) must be injective.
Left application conditions (Definition 5) in rules are required to be composed total negative application conditions.
Right application conditions (Definition 5) in rules are required to be true.
Rule applicability (Definition 5) requires injective matches and comatches.

To conclude the definitions used in our verification approach, we introduce our notion of inductive invariants for graph transformation systems. Informally, all rule applications should preserve the validity of a composed forbidden pattern $\mathcal{F}$. Since the system is assumed to prevent violations of a composed guaranteed pattern $\mathcal{G}$ by other means (e.g., a postprocessing step) or additional knowledge, rule applications leading to such a violation do not need to be considered.

Definition 10 (inductive invariant). *Given a composed forbbidden pattern $\mathcal{F}$ and a composed guaranteed pattern $\mathcal{G}$, a typed graph transformation system $GTS = (TG, B)$ is* preserving $\mathcal{F}$ under $\mathcal{G}$ *if, for each rule b in B, it holds that*

$$\forall G, H((G \Rightarrow_b H) \Longrightarrow ((G \models \mathcal{F} \wedge G \models \mathcal{G}) \Rightarrow (H \models \mathcal{F} \vee H \not\models \mathcal{G}))).$$

A composed forbidden pattern $\mathcal{F}$ preserved by GTS under $\mathcal{G}$ is an inductive invariant *for GTS under $\mathcal{G}$.*

3.1 Constructions

An important part of our algorithm is the transformation of application conditions over morphisms and rules. [15] presents a Shift-construction for a transformation of application conditions over morphisms into equivalent application conditions. For our restricted formal model, we use a marginally adjusted form of the Shift-construction. Its validity is proven in Appendix B in [13].

Construction 11 (Shift-construction, adjusted from [15]). *For each total application condition ac over a graph P and for each morphism $b : P \to P'$, $Shift(b, ac)$ transforms ac via b into a total application condition over P' such that, for each morphism $n : P' \hookrightarrow H$, it holds that $n \circ b \models ac \Leftrightarrow n \models Shift(b, ac)$.*

The Shift-construction is inductively defined as follows:

$$\begin{tikzcd}
P \arrow[r, "b"] \arrow[d, "a"'] \arrow[dr, phantom, "(1)"] & P' \arrow[d, hook, "a'"] \\
C \arrow[r, hook, "b'"'] & C' \\
\triangle_{ac} &
\end{tikzcd}$$

$Shift(b, true) = true$.
$Shift(b, \exists(a, ac)) = \bigvee_{(a',b') \in \mathcal{F}} \exists(a', Shift(b', ac))$ if $\mathcal{F} = \{(a', b') \mid (a', b')$ are jointly surjective, a', b' are injective, and (1) commutes $(b' \circ a = a' \circ b)\} \neq \varnothing$ and false, otherwise.
$Shift(b, \neg ac) = \neg Shift(b, ac)$.
$Shift(b, \bigwedge_{i \in I} ac_i) = \bigwedge_{i \in I} Shift(b, ac_i)$.

While this construction can be employed to equivalently transform total application conditions, the calculation of the respective morphism pairs is computationally expensive. To avoid executing that calculation, we construct partial application conditions instead and establish their equivalence to the result of the Shift-construction in the following construction and lemma. As before, proof of validity and a more detailed version can be found in Appendix B in [13].

Construction 12 (PShift-construction). *For each total application condition ac over P' and for each morphism $p' : P' \hookrightarrow P$, $PShift(p', ac)$ transforms ac via p' into a partial application condition over P such that, for each morphism $n : P \hookrightarrow H$, it holds that $n \circ p' \models ac \Leftrightarrow n \models PShift(p', ac)$.*

The PShift-construction is defined as follows:

$$\begin{tikzcd}
P' \arrow[r, hook, "p'"] \arrow[d, hook, "a"'] & P \arrow[dl, hook, "c=\langle p',a'\rangle"] \\
C & \\
\triangle_{ac} &
\end{tikzcd}$$

$PShift(p', true) = true$.
$PShift(p', \exists(a, ac)) = \exists(c, ac)$ with $c = \langle p', a\rangle$.
$PShift(p', \neg ac) = \neg PShift(p', ac)$.
$PShift(p', \bigwedge_{i\in I} ac_i) = \bigwedge_{i\in I} PShift(p', ac_i)$.

Lemma 13. *For each application condition ac over P and each monomorphism $p' : P \hookrightarrow P'$, we have $Shift(p', ac) \equiv PShift(p', ac)$.*

We also transform application conditions over rules using the L-construction found in [15]. For the formal basis of this construction, we refer to [13].

Construction 14 (L-construction [10,15]). *For each rule $b = \langle L \hookleftarrow K \hookrightarrow R\rangle$ and each total application condition ac over R, $L(b, ac)$ transforms ac via b into a total application condition over L such that, for each direct transformation $G \Rightarrow_{b,m,m'} H$, we have $m \models L(b, ac) \Leftrightarrow m' \models ac$.*

The L-construction is inductively defined:

$$\begin{tikzcd}
L \arrow[d, "a'"'] \arrow[dr, phantom, "(2)"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d] \arrow[dr, phantom, "(1)"] & R \arrow[d, "a"] \\
L(b',ac) \triangleright L' & K' \arrow[l, hook', "l'"] \arrow[r, hook, "r'"'] & R' \triangleleft ac
\end{tikzcd}$$

$L(b, true) = true$.
$L(b, \exists(a, ac)) = \exists(a', L(b', ac))$ (with $b' = \langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ constructed via the pushouts (1) and (2)) if $\langle r, a\rangle$ has a pushout complement (1) and false, otherwise.
$L(b, \neg ac) = \neg L(b, ac)$.
$L(b, \bigwedge_{i\in I} ac_i) = \bigwedge_{i\in I} L(b, ac_i)$.

3.2 Implication

One of the main requirements for our algorithm is the comparison of graph constraints or, more precisely, the notion of implication of patterns.

Definition 15 (implication of patterns). *Let $C = \exists(\varnothing \hookrightarrow P, ac)$ and $C' = \exists(\varnothing \hookrightarrow P', ac')$ with composed partial negative application conditions ac and ac' be two patterns. C'* implies *C ($C' \models C$), if the following condition holds:*

$$\forall G(G \models C' \Rightarrow G \models C).$$

Since a pattern may be fulfilled by an infinite number of graphs, we cannot (in general) check the above condition for all such graphs. Instead, we establish a condition sufficient to imply implication when comparing patterns. Depending on whether the patterns' application conditions ac and ac$'$ are partial, total, or nonexistent (i.e. true), the procedure and its computational effort varies. The following theorem describes the most interesting case with a composed partial (total) negative application in the implying (implied) pattern, respectively.

Theorem 16 (implication of patterns). *Let* $C = \exists(\varnothing \hookrightarrow P, ac)$ *and* $C' = \exists(\varnothing \hookrightarrow P', ac')$ *be patterns with a composed total negative application condition* $ac = \bigwedge_{i\in I} \neg\exists(x_i : P \hookrightarrow X_i)$ *and a composed partial negative application condition* $ac' = \bigwedge_{j\in J} \neg\exists(x'_j : P' \hookrightharpoonup X'_j)$. *Then* $C' \models C$, *if the following conditions are fulfilled:*

$$\begin{tikzcd}
P \arrow[r, hook, "m"'] \arrow[d, hook, "x_i"'] & P' \arrow[l, bend right=10, "m^{-1}"'] \arrow[d, "\langle n'_j, n_j\rangle = x'_j"'] & N_j \arrow[l, hook', "n'_j"] \arrow[dl, hook', "n_j"] \arrow[ll, bend right=30, "m'"'] \\
X_i & X'_j \arrow[l, hook', "y"] &
\end{tikzcd}$$

1. *There exists a monomorphism* $m : P \hookrightarrow P'$ *such that:*
2. *For each* $i \in I$, *there exists a* $j \in J$ *such that* $n'_j(iface(x'_j)) \subseteq m(P)$ *and there exists a monomorphism* $y : X'_j \hookrightarrow X_i$ *such that* $y \circ n_j = x_i \circ m'$, *with* $m' = m^{-1} \circ n'_j$.

For patterns without negative application conditions, the theorem is also applicable as the second condition is trivially true. For cases where the implying pattern's partial negative application conditions do not satisfy the interface condition, a partial expansion of the implied pattern's condition is required, which requires additional computational effort.

In general, all cases can be transformed into a default case by expanding all composed partial negative application conditions into composed total negative application conditions with the Shift-construction. The comparison in that case is explained in Appendix B in [13]. The desired effect of the above theorem is to avoid this computationally expensive default case as often as possible.

This theorem only considers one implying pattern at a time. We also use an *advanced implication check* considering more complex relations between forbidden patterns and negative application conditions, such as implication of a single pattern by multiple patterns. The theory and implementation of such a check for the more general concept of nested conditions have already been introduced by Pennemann et al. in [9]. Hence, we will not discuss our implementation here.

Besides graph constraints we will also encounter application conditions over a rule side, which can be interpreted as graph constraints as follows:

Lemma 17 (reduction to pattern). *Let* $ac = \exists(s : L \hookrightarrow S, ac_S)$ *be an application condition over* L *with* ac_S *being a composed partial negative application condition. For the* reduction to a pattern $ac_\varnothing = \exists(i_S : \varnothing \hookrightarrow S, ac_S)$ *of* ac *we have the following property: For each graph* G *with a monomorphism* $m : L \hookrightarrow G$ *such that* $m \models ac$, *we have* $G \models ac_\varnothing$.

4 Inductive Invariant Checking

Our inductive invariant checking algorithm consists of four basic steps:

(1) From a composed forbidden pattern and a rule set, we create all pairs of individual forbidden patterns and rules to be analyzed on a per-pair basis. (2) We construct *target patterns* for each pair by applying the Shift- and PShift-constructions, such that each target pattern represents a satisfaction of a forbidden pattern after rule application. (3) From each target pattern, we construct a *source pattern* by applying the L-construction such that a source pattern is a representation for graphs before a rule application leads to a forbidden pattern. (4) We analyze source and target pattern pairs (*counterexamples*) for other forbidden or guaranteed patterns, which might invalidate the counterexample.

The first step of splitting a composed forbidden patterns into forbidden patterns for individual analysis is shown to be correct in the following lemma. It also explains the analysis of source and target patterns in step 4.

Lemma 18. *Given a composed forbidden pattern $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$, a composed guaranteed pattern $\mathcal{G} = \bigwedge_{j \in J} \neg G_j$ and a typed graph transformation system $GTS = (TG, B)$, GTS is* preserving *$\mathcal{F}$ under $\mathcal{G}$ if, for each rule b in B it holds that:*

$$\forall G, H((G \Rightarrow_b H) \implies (\exists n(H \models F_n) \Rightarrow \exists k(H \models G_k \vee G \models G_k \vee G \models F_k)))$$

4.1 Step 2: Construction of Target Patterns

The second step in our inductive invariant checking algorithm is the creation of target patterns for each pair of a graph rule and a forbidden pattern such that the forbidden pattern occurs in the target pattern. Target patterns in general represent a set of graphs with a match for the right side of a specific graph rule.

Definition 19 (target pattern). *A* target pattern *over the right side R of a rule b is an application condition of the form $tar = false$ or $tar = \exists(t : R \hookrightarrow T, ac_T)$ with a composed partial negative application condition ac_T over T.*

The set of graphs fulfilling such a target pattern is the set of graphs H with a comatch $m' : R \hookrightarrow H$ such that $m' \models$ tar. For a rule b in B and a forbidden pattern F, we can create target patterns by transforming F over the morphism $i_R : \varnothing \hookrightarrow R$ into an application condition over the right rule side R:

Lemma 20 (creation of target patterns). *Let $b = \langle(L \hookleftarrow K \hookrightarrow R), ac_L, true\rangle$ be a rule and $F = \exists(i_P : \varnothing \hookrightarrow P, ac_P)$ a forbidden pattern with ac_P and ac_L being composed total negative application conditions. Let $b^* = \langle(L^* \hookleftarrow K^* \hookrightarrow R^*)\rangle$ be a reduced rule of the plain rule in b with respective injective morphisms $r^+ : R^* \hookrightarrow R$, $l^+ : L^* \hookrightarrow L$, and $k^+ : K^* \hookrightarrow K$. Then we have:*

1. *$Shift(r^+, Shift(i_{R^*}, \exists i_P)) = \bigvee_{j \in J} \exists t_j$.*

2. $\bigvee_{j\in J} tar_j$ *is a set of target patterns for* $tar_j = \exists(t_j, \mathit{PShift}(t_j^+, \mathit{Shift}(t_k'^*, ac_p)))$.
3. For each graph H *and each monomorphism* $h : R \hookrightarrow H$*, it holds that* $\exists j(j \in J \wedge h \models tar_j) \Leftrightarrow H \models F$.

$$\begin{tikzcd}
\varnothing \arrow[r, hook, "i_{R^*}"'] \arrow[d, hook, "i_P"'] \arrow[dr, phantom, "="] & R^* \arrow[r, hook, "r^+"'] \arrow[d, hook, "t_k^*"] & R \arrow[r, hook, "h"'] \arrow[d, hook, "t_j"] & H \\
P \arrow[r, hook, "t_k'^*"'] & T_k^* \arrow[r, hook, "t_j^+"] & T_j & \\
\triangle_{ac_P} & \triangle_{ac_{T_k^*}} & &
\end{tikzcd}$$

In other words, we shift the exterior application condition ($\exists i_P$ in step (1)) of the forbidden pattern to the right rule side, but its interior composed negative application condition (ac$_P$ in step (2)) to a partial application condition using the reduced rule. Thus, we avoid creating a large number of morphism pairs when shifting the interior application condition to the complete right rule side.

In conclusion, for each morphism $h : R \hookrightarrow H$ the satisfaction of the forbidden pattern F by a graph H is equivalent to the existence of a target pattern tar$_j$ satisfied by h. In other words, for each result of a possible rule application leading to a graph satisfying the forbidden pattern we have constructed a target pattern. Since target patterns (as shown above) are disjunctively combined, we can analyze each target pattern individually and compute its source pattern. By construction, we always have a finite number of target patterns.

4.2 Step 3: Construction of Source Patterns

For each target pattern constructed as described above, we try to generate a source pattern to represent the state before the application of the rule lead to the forbidden pattern. In general, we define source patterns analogously to target patterns as application conditions over the left side of a specific graph rule.

Definition 21 (source pattern). *A* source pattern *over the left side* L *of a rule* b *is an application condition of the form* $src = \mathit{false}$ *or* $src = \exists(s : L \hookrightarrow S, ac_S)$ *with a composed partial negative application condition* ac_S *over* S.

To construct source patterns to our target patterns, each target pattern is transformed into an application condition over the left rule side using the L-construction. Due to the nature of the L-construction, we create at most one source pattern per target pattern transformation.

Lemma 22 (creation of source patterns). *Let* $tar = \exists(t : R \hookrightarrow T, ac_T)$ *be a target pattern specific to a rule* $b = \langle(L \hookleftarrow K \hookrightarrow R), ac_L, \mathit{true}\rangle$ *with a reduced rule* $b^* = \langle(L^* \hookleftarrow K^* \hookrightarrow R^*)\rangle$ *of its plain rule and constructed as described above. Further, let* ac_T *be a composed partial negative application condition* $ac_T = \mathit{PShift}(t^+, ac_T')$ *with* ac_T' *being a composed total negative application condition over* T^**. Then we have:*

1. $L(b, \exists t)$ *is a source pattern and* $L(b, \exists t) = \mathit{false}$ *or* $L(b, \exists t) = \exists s$.

2. For the latter case, $src = \exists(s, PShift(s^+, L(b', ac'_T)))$ is a source pattern, with $b' = \langle S^* \hookleftarrow K' \hookrightarrow T^* \rangle$ being the rule constructed via the pushout complement (1) and the pushout (2) and $s^+ : S^* \hookrightarrow S$ such that $S \Rightarrow_{b', s^+, t^+} T$.
3. For each direct graph transformation $G \Rightarrow_{b,m,m'} H$: $m \models src \Leftrightarrow m' \models tar$.

$$\begin{tikzcd}
 & L \arrow[dl, hook, "s"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] & R \arrow[dr, hook, "t"] & \\
S & L^* \arrow[u, hook, "l^+"] \arrow[d, hook, "s'"] \arrow[dr, phantom, "(2)"] & K^* \arrow[l, hook', "l^*"'] \arrow[r, hook, "r^*"] \arrow[u, hook, "k^+"] \arrow[d, hook] \arrow[dr, phantom, "(1)"] & R^* \arrow[u, hook, "r^+"] \arrow[d, hook, "t'"] & T \\
 & L(b', ac_T) \triangleright S^* \arrow[ul, hook, "s^+"] & K' \arrow[l, hook', "l'"] \arrow[r, hook, "r'"'] & T^* \triangleleft ac_T \arrow[ur, hook, "t^+"'] &
\end{tikzcd}$$

Such a source pattern represents graphs before the application of the rule in question which leads to graphs satisfying the forbidden pattern. To also take left application conditions into account, they need to be transformed via $\mathrm{Shift}(s, \mathrm{ac}_L)$ into conditions over the source pattern. For details, we refer to [13].

In summary, the source and target patterns src and tar represent a correct rule application of a rule b leading to the existence of the forbidden pattern F. To represent all possible rule applications, i.e. all graphs G and H with $G \Rightarrow_b H$, we need to consider all target patterns and their corresponding source patterns.

4.3 Step 4: Analysis of Source Patterns and Counterexamples

Each target pattern and corresponding source pattern specific to a rule and a forbidden pattern specify a counterexample for our inductive invariant, i.e. a situation where a rule application leads to the occurrence of a forbidden pattern F_i. To investigate whether this is indeed a violation of the inductive invariant $\mathcal{F}$ under $\mathcal{G}$, the following three conditions are considered:

1. The target pattern also violates the composed guaranteed pattern.
2. The source pattern violates the composed guaranteed pattern.
3. The source pattern violates the composed forbidden pattern.

Theorem 23 (inductive invariant checking). *Let GTS be a graph transformation system and $\mathcal{F} = \bigwedge_{i \in I} \neg F_i$ and $\mathcal{G} = \bigwedge_{j \in J} \neg G_j$ be a composed forbidden and composed guaranteed pattern. Let, for each rule $b \in B$ and for $i \in I$, $src_{b,i}$ ($tar_{b,i}$) be the set of source (target) patterns constructed from the pair (b, F_i) and $src_{\varnothing,b,i}$ ($tar_{\varnothing,b,i}$) be the set of these source (target) patterns reduced to graph constraints.*

GTS preserves $\mathcal{F}$ under $\mathcal{G}$ if, for all reduced source patterns $src_\varnothing$ created from a pair of a rule and a forbidden pattern (b, F_i) and the corresponding reduced target pattern $tar_\varnothing$, one of the following conditions holds:

1. $\exists k (k \in J \wedge tar_\varnothing \models G_k)$
2. $\exists k (k \in J \wedge src_\varnothing \models G_k)$

3. $\exists k(k \in I \wedge src_{\varnothing} \models F_k)$

This shows that GTS preserves $\mathcal{F}$ under $\mathcal{G}$, if the condition from Theorem 23 holds. In other words, $\mathcal{F}$ is an inductive invariant for GTS under $\mathcal{G}$. The construction of target and source patterns and the verification of this condition by application of Theorem 16 is, in short, the essence of the Invariant Checking algorithm. On the other hand, source and target patterns not discarded by that conditions are counterexamples for the inductive invariant.

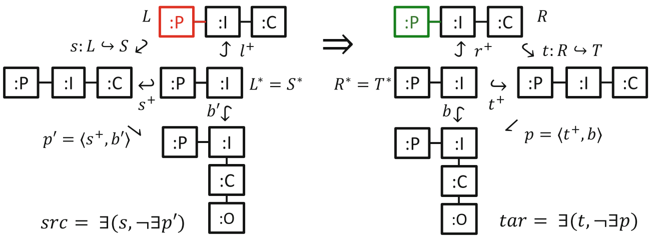


Fig. 2. Source and target pattern pair created from a rule and a forbidden pattern

Example 24. Figure 2 shows a source and target pattern pair src and tar created from the forbidden pattern F and rule in Examples 4 and 7. In tar, the condition $\exists t$ is one amalgamation of F and the right rule side (Lemma 20, step 1); $\neg\exists p$ is the pattern's application condition transformed with PShift over t^+ (Lemma 20, step 2). Since the forbidden pattern can be found in the source pattern ($src_{\varnothing} \models F$), this counterexample is discarded by the analysis in Theorem 23.

Because the implication checks (Theorems 16 and 23) compare only individual patterns and disregard more complex interdependencies and satisfiability of multiple patterns, this algorithm may still produce false negatives (i.e., spurious counterexamples). Our advanced implication check then serves to reduce this number and may also be applied to reduce the number of forbidden patterns to be analyzed by subsuming some of them. Since the general concept has already been introduced by Pennemann et al. in [9], we do not discuss it here. However, our technique is safe in the sense that all violations will be reported.

5 Evaluation and Discussion

To evaluate our results, we employ three case studies: The first example car platooning describes rules and constraints in a car platooning system. It was employed in the context of the SeekSat/ProCon tool [9] and was originally described in [16]. In order to conform to our restrictions it had to be adjusted, resulting in the addition of twelve new constraints. Our second and third case study are a simple and complex example for verification of behavior preservation of model transformations by bisimulation with the simple case initially employed by us in [11] and both examples described in [17]. In the first case (MT - Simple), behavioral equivalence between single lifelines and automata derived by a triple graph grammar (TGG) is proven. In the more complex example (MT - Complex), behavioral equivalence between sequence diagrams with multiple lifelines and networks of automata is proven. In both cases the check involves two inductive invariant checks: one for the TGG generating all possible model pairs and one for the Semantics of any possible pair of models to prove bisimilarity.

The first point of reference for our evaluation is our improved inductive invariant checker in its basic variant (invcheck-total). We also compare variants employing advanced implication checks (invcheck-total/impl), partial negative application conditions (invcheck-partial), and both (invcheck-partial/impl). On the other hand, the former version of our inductive invariant checker [8] only supported a restricted form of negative application conditions for constraints and rules and was thus not expressive enough for the considered case studies.

In addition, we will consider the SeekSat/ProCon tool [9,10], which is able to prove correctness of graph programs with respect to pre- and postconditions specified as nested graph constraints. To verify an inductive invariant ($\mathcal{F}$) of a graph transformation system (GTS) with guaranteed constraints ($\mathcal{G}$), the equivalent check contains a graph program nondeterministically choosing a rule from GTS, the precondition $\{\mathcal{F} \wedge \mathcal{G}\}$ and the postcondition $\{\mathcal{F} \vee \neg\mathcal{G}\}$. While the technique behind SeekSat/ProCon is more expressive than our approach, we use this comparison to demonstrate the relevance of our more specialized tool for the verification of certain cases where that level of expressiveness is not needed.

Besides the evaluation of the case studies as a whole, we also want to study the impact of the complexity of the checking problem by considering the sum of all possible amalgamations between a forbidden pattern and the right side of a rule and the number of total negative application conditions for those amalgamations. To get more fine-grained results, we separated some examples into multiple cases by splitting postconditions $(\bigwedge_{i\in I} F_i) \vee \neg\mathcal{G}$ into less complex i subproblems with postconditions $F_i \vee \neg\mathcal{G}$ or by considering rules in a set separately.

The experiments were executed on a computer with an Intel Core-i7–2640M processor with two cores at 2,8 GHz, 8 GB of main memory and running Eclipse 4.2.2 and Java 8 with a limit of 2 GB on Java heap space. All values were rounded and values under a second were not distingiuished. Timeout refers to a forced timeout issued by the tool (SeekSat/ProCon) or manual abortion (our tool)—for the related cases in our tool after more than two days of calculation. Out of memory means that memory exceeded the Java heap space limit of 2 GB.

Table 1 shows an overview of the verification of our complete examples (marked as complete; in gray) and a more detailed list of subproblems ordered by complexity (marked as subproblem), respectively. All algorithms perform comparably well for the car platooning example, with SeekSat/ProCon performing significantly better for the unadjusted version than our algorithms. However, for the other complete cases our tool terminates while SeekSat/ProCon does not.

Table 1. Complexity of verification problems and results of evaluated algorithms

					without advanced implication check				with advanced implication check			
	Characteristics		SeekSat/ProCon		Invcheck-total		Invcheck-partial		Invcheck-total/impl		Invcheck-partial/impl	
Example	Check	Complexity	time (s)	result	time (s)	result	time (s)	result	time (s)	result	time (s)	result
MT - Simple - Semantics	subproblem	4	20	true	<1	true	<1	true	<1	true	<1	true
MT - Simple - Semantics	subproblem	4	20	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - TGG	subproblem	4	<1	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - TGG	subproblem	4	<1	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - Semantics	subproblem	5	10	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - Semantics	subproblem	5	9	true	<1	true	<1	true	<1	true	<1	true
MT - Simple - Semantics	subproblem	11	40	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - TGG	subproblem	11	<1	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - Semantics	subproblem	12		out of memory		timeout	<1	false negatives		timeout	<1	true
MT - Complex - Semantics	subproblem	17	17	true	<1	false negatives	<1	false negatives	<1	true	<1	true
MT - Complex - TGG	subproblem	20		timeout	<1	true	<1	true	<1	true	<1	true
MT - Simple - Semantics	subproblem	30	20	true	<1	true	<1	true	<1	true	<1	true
MT - Simple - Semantics	subproblem	70	40	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - Semantics	subproblem	72		timeout	<1	false negatives	1	false negatives	1,5	true	1,5	true
MT - Simple - Semantics	subproblem	78	6,5	true	<1	true	<1	true	<1	true	<1	true
MT - Complex - Semantics	subproblem	188		out of memory	1,5	false negatives	2,5	false negatives	<1	true	<1	true
Car Platooning	subproblem	258	<1	true	<1	true	<1	true	<1	true	<1	true
Car Platooning	subproblem	610	<1	true	<1	true	<1	true	<1	true	<1	true
MT - Simple - Semantics	subproblem	807		timeout	<1	true	<1	true	<1	true	<1	true
Car Platooning	complete	947	<1	false	<1	false negatives	<1	false negatives	3	false	3	false
MT - Simple - TGG	subproblem	2778	220	true	1,5	false negatives	1	false negatives	1,5	true	1	true
MT - Simple - TGG	subproblem	2778	226	true	1,25	false negatives	1	false negatives	1,25	true	1	true
MT - Simple - Semantics	complete	3870		timeout	1,5	true	1	true	1,5	true	1	true
MT - Simple - TGG	complete	5556	562	true	2	false negatives	2	false negatives	2,25	true	1,75	true
MT - Complex - Semantics	subproblem	607312		out of memory		timeout	90	false negatives		timeout	<1	true
MT - Complex - Semantics	complete	607500		out of memory		timeout	95	false negatives		timeout	<1	true
MT - Complex - TGG	complete	1817622		timeout		timeout	~100min	true		timeout	~50min	true

It is important to note that the inductive invariant checker without advanced implication checks yields false negatives for certain subproblems. Even more importantly, these false negatives do not occur when using the variant with advanced implication checks. This demonstrates that the improvement in accuracy due to advancement in implication checks is indeed relevant for the case studies.

Further, the results demonstrate that the complex model transformation case study cannot be verified by the inductive invariant checker variants without partial negative application conditions, as these attempts were aborted after more than two days of calculation without a result. In contrast to that, a verification time of 100 min (for the longest case) when employing partial negative application conditions shows a drastic improvement in scalability for the considered more complex cases. The additional use of advanced implication checks does then not only eliminates false negatives, but, for one case, also halves the verification time, showing another notable effect on performance.

While these case studies show both our improvements and the relevance of verification for specifications that conform to our restrictions, the data is not complete and heterogeneous enough to derive claims for the general case. While SeekSat/ProCon's more general approach is also successfully applicable for specifications that are significantly more expressive, our tool has been optimized for a particular class of problems present in the two more complex case studies and their verification only succeeded with our tool.

6 Related Work

As already discussed in Sect. 5, the SeekSat/ProCon tool [9,10] is more general than our approach and thus is in principle capable of addressing the case studies.

However, the limited scalability of the SeekSat/ProCon tool demonstrates that there is still a need for a tool optimized for a particular class of problems that scales up to the presented two more complex case studies.

For all other automated approaches that approach graph transformation systems with infinite state space [3–8, 18], it holds that, in contrast to the approaches considered in the evaluation, they cannot be used for the case studies which require unrestricted negative application conditions: The model checking approach [4] employing abstraction based on the summarization in shape analysis and the model checking approach [3] employing a neighborhood abstraction, but both do not support negative application conditions for the constraints or rules. The tool Uncover [5] supports well-structured graph transformation systems that can only be established for negative application conditions which forbid the existence of edges but not of nodes. The Augur tool [6, 18], which constructs a over-approximation in form of a so-called Petri graph, also considers only graph transformation systems without negative application conditions. Finally, the RAVEN tool [7] can check only invariants for graph transformation systems without negative application conditions whose reachable graphs are accepted by a finite graph automaton. Since two of our case studies describe reachable graphs by TGGs, they cannot be covered by a finite graph automaton.

For additional discussion of related work with respect to the general concept of inductive invariants, we refer to the respective section in [8].

7 Conclusion and Future Work

In this paper, we presented several improvements for the inductive invariant checker for graph transformation systems introduced in [8]. Support for more expressive negative application conditions in constraints and rules was shown to be necessary to address the considered case studies at all. The introduction of partial negative application conditions allowed avoiding the explicit representation of a large number of application conditions, which considerably improved scalability. The addition of advanced implication checks improved the accuracy, so that no false negatives are reported for the case studies.

In addition we demonstrated the outlined improvements by means of three case studies and compared our approach for a restricted class of problems with an existing tool that targets more general problems. For the more complex problems considered, our approach was still able to check them; the other tool was not.

While the results are promising, the evaluation also raises a number of possible future directions such as employing even more partial shifts in our constructions, and experimenting with the parallel execution of alternative strategies.

Acknowledgments. We would like to thank the group of Annegret Habel, in particular the authors of the SeekSat/ProCon tool [9], for allowing us to do the comparison and Leen Lambers for her work on behavior preservation of model transformations.

References

1. Ghamarian, A.H., de Mol, M.J., Rensink, A., Zambon, E., Zimakova, M.V.: Modelling and analysis using GROOVE. Int. J. Softw. Tools Technol. Transf. **14**(1), 15–40 (2012)
2. Schmidt, A., Varró, D.: CheckVML: a tool for model checking visual modeling languages. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 92–95. Springer, Heidelberg (2003)
3. Boneva, I.B., Kreiker, J., Kurban, M.E., Rensink, A., Zambon, E.: Graph abstraction and abstract graph transformations (Amended version). Technical report TR-CTIT-12-26, Centre for Telematics and Information Technology, University of Twente, Enschede (2012)
4. Steenken, D.: Verification of infinite-state graph transformation systems via abstraction. Ph.D. thesis, University of Paderborn (2015)
5. König, B., Stückrath, J.: A general framework for well-structured graph transformation systems. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 467–481. Springer, Heidelberg (2014)
6. König, B., Kozioura, V.: Augur 2 - a new version of a tool for the analysis of graph transformation systems. In: Electronic Notes in Theoretical Computer Science, Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006), vol. 211, pp. 201–210 (2008)
7. Blume, C., Bruggink, H.J.S., Engelke, D., König, B.: Efficient symbolic implementation of graph automata with applications to invariant checking. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 264–278. Springer, Heidelberg (2012)
8. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), Shanghai, China. ACM Press (2006)
9. Pennemann, K.-H.: Development of correct graph transformation systems. Ph.D. thesis, Department of Computing Science, University of Oldenburg (2009)
10. Habel, A., Pennemann, K.-H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**, 1–52 (2009)
11. Giese, H., Lambers, L.: Towards automatic verification of behavior preservation for model transformation via invariant checking. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 249–263. Springer, Heidelberg (2012)
12. Becker, B., Lambers, L., Dyck, J., Birth, S., Giese, H.: Iterative development of consistency-preserving rule-based refactorings. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 123–137. Springer, Heidelberg (2011)
13. Dyck, J., Giese, H.: Inductive invariant checking with partial negative application conditions, 98, Technical report, Hasso Plattner Institute at the University of Potsdam, Germany (2015)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Secaucus (2006)
15. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: M-Adhesive transformation systems with nested application conditions, part 1: parallelism, concurrency and amalgamation. Math. Struct. Comput. Sci. **24**(4) (2014)

16. Hsu, A., Eskafi, F., Sachs, S., Varaiya, P.: The design of platoon maneuver protocols for IVHS. Technical report UCBITS-PRR-91-6, University of California, Berkley (1991)
17. Dyck, J., Giese, H., Lambers, L.: Automatic verification of behavior preservation for model transformation via invariant checking. Technical report, Hasso Plattner Institute at the University of Potsdam, Germany (2015, forthcoming)
18. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)