Proving preservation of transitivity invariants in model transformations

Christian Percebois, Martin Strecker, Hanh Nhi Tran

IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse, France
{Christian.Percebois | Martin.Strecker | Hanh-Nhi.Tran}@irit.fr *

Abstract. This paper develops methods to reason about graph transformations, and in particular to show that transitivity and reachability invariants are preserved during transformations. In our approach, graph transformations consist of a pattern defining an applicability condition, and an operational description of the desired transformation. Whereas previous work was restricted to Boolean combinations of arc expressions as patterns, we extend the approach to patterns containing transitive closure operations, which implicitly denote an unbounded number of nodes. We show how these can be internalized into a finite pattern graph so that model checking techniques can be applied for verification.

Keywords: Model-Driven Engineering; Graph Transformations; Formal Methods; Verification

1 Introduction

1.1 Aims and Contributions

Graph transformations have numerous applications in computer science. They can be used, among others, in Model Driven Engineering (MDE); as compiler transformations; as a high-level view of pointer-manipulating programs; and for modeling security properties of an information system.

Many of these transformations can be considered safety critical and therefore have to satisfy stringent correctness requirements. In some cases, it might be sufficient to carry out an *a posteriori* verification and to check that the outcome of a transformation fulfills all requirements. This is in particular possible if the graph is small enough, and reaching an error state is not problematic. For example, recognizing that a refactoring step in MDE has failed is itself not catastrophic. However, a network management system or an access control system described by a graph transformation should work according to their specification, and detecting a malfunctioning after it has occurred might already be too late.

The work described in this paper adopts a theorem proving approach to demonstrate the correctness of transformations. Our aim is to develop methods

* Part of this research has been supported by the *Climt* project (ANR-11-BS02-016).

allowing to prove that a transformation rule is correct when applied to an arbitrary graph, provided certain applicability conditions are met. Here, "correctness" means that the transformation establishes certain structural conditions, for example cardinality restrictions. Properties that we especially focus upon in this paper are conditions of connectivity respectively separation: does a transformation ensure that two nodes remain connected (respectively unconnected) by a sequence of arcs?

This paper homogenizes and continues the strands developed by the authors in previous papers [24,25], in which we have shown that reasoning about a transformation applied to an arbitrary graph can be essentially reduced to reasoning about a bounded portion of the graph, namely the image of the transformation rule in the target graph. In this paper, we extend this approach and make the following contributions:

- We introduce *transitive closure* patterns in the rules' applicability conditions. These express that the rule can be applied provided that two nodes are connected via the transitive closure of an arc relation. This kind of applicability pattern is interesting in its own right; there are only few graph transformation engines that include it in their transformation language (see Section 1.2).
- We show that this pattern, even though referring to a possibly unbounded number of nodes, can be reduced to a verification on simple arcs, thus allowing for automation also in this case.
- Unfortunately, (positive) transitive closure patterns are not always sufficient as applicability conditions of rules; one sometimes also has to stipulate the non-existence of a connection in the underlying graph. We illustrate this situation with some example. Reasoning about these negative connectivity patterns turns out to be much more difficult, and we cannot give a complete calculus. We present however some reasoning patterns (associated to the graph rewriting rules) that allow a simplification in common situations.

The rest of the paper is structured as follows: After a description of related work (Section 1.2) and some introductory examples (Section 2), we describe the background of graph transformations as we apply them. We then start with a rather technical exposition of the reasoning principles underlying our approach and the way to reduce them to a finite case (Section 3). We conclude with a perspective on future work.

A note on notation: We have used the interactive proof assistant Isabelle [18] to model the transformations and to carry out the proofs described in this paper. As a concession to readability, we describe the transformations in a more traditional notation, as described further in Section 3.1.

1.2 Related work

Verification of graph transformation mostly uses model-checking technology, see [2,5,10,26] for some representatives of this approach. A wide-spread approach is

to model concurrent systems as graph transformations and analyze invariants and reachability problems of these systems. Another emerging interest of this approach is using graph rewriting for model transformations [1] with appropriate verification methods [27].

For software verification, model checking is seldom usable, except if the data structures manipulated by a program are entirely generated by the rules from an initial graph, such as the red-black trees in [4], or unless abstraction functions are provided [29].

We notice an important body of work on verification of pointer structures in imperative programs. Static analyses often use specialized logics for expressing shapes of pointer structures [13,28]. These logics, as well as frameworks based on Monadic Second-Order (MSO) logic, often rely essentially on tree structures with additional pointers, such as the data structure invariants of [17] or the *exact type-checking* in XML processing [12]. Our global approach presented in this article is not restricted to particular shapes of a graph; however, only some specific forms of proof obligations can be fully automatized. In [7], the authors investigate MSO for verification of graph *properties*, but do not address graph *transformations*. [3] develop a modal logic for reasoning about graph programs composed of fine-grained operators for manipulating nodes and redirecting edges.

There are several approaches based on similar ideas as ours on local reasoning about data structures: [16] give a decision procedure for a language that is essentially first-order (and in particular contains no transitive closure), but can deal with relation composition and integrates support for scalar data types.

Traditionally, algebraic approach has been mostly used for graph rewriting in order to reason about graph transformations. Recently, there is a tendency of interpreting graph structures logically [21,19]. The work of Pennemann *et al.* [20,11] extracts verification conditions from graph transformation programs and feeds them into SAT solvers or first-order theorem provers. This approach is entirely automatic and does not allow for human intervention for proving "difficult" theorems. Moreover, there is no tight coupling between the semantics (expressed in categorical terms in the cited work) and the proof obligation generator, and thus there is a dependency on a larger trusted code base.

In [14], the authors propose a new approach to verifying graph transformations written in Core UnCAL against the specified input/output graph structural constraints (schemes) in MSO. They first represent both Core UnCAL transformations and schemes by MSO formulas and then develop an algorithm to reduce the graph transformation verification problem to the validity of MSO over trees. This approach is expressive but its efficiency relies on the algorithm to map the type-annotated Core UnCAL to an MSO-definable graph transduction, and the decision procedure to verify MSO formula.

The verification problem is also addressed by Zarrin Langari and Richard Trefler [15]. The authors can verify invariants that are expressed by CTL by adding proposition graphs to transformation rule graphs. Owing to proposition graphs, the designer can compactly express feature connectivity patterns required during the transformation. The main result of the paper states a satisfaction condition

theorem for a transformation rule which preserve a property P. Rather similar to ours, however, the proposed preservation conditions do not permit verifying the properties of transitivity type which are not present on the right side of transformation rule.

Da Costa and Ribeiro [8,9] present a logical model for reasoning about graph transformations that is similar to ours. They propose an encoding of graphs and rules into relations to enable the use of logic formulas for expressing properties of a graph grammar's reachable states. This approach has been implemented in Event-B [22] by coding individual rules as Event-B machines, such that it is possible to use the Event-B provers to demonstrate properties of a graph grammar. However, this work verifies only the property-preservation of a transformation when applied to a concrete graph, whereas ours permits reasoning about graph transformation applied to an arbitrary graph satisfying some given preconditions.

Another approach for verification and validation of model transformations is presented in [6]. This work translates the definition of graph rewriting rules into an OCL-based representation, which can be combined with constraints defined in the metamodels. Conditions are then split into those to be checked on the LHS before rule application and those on the RHS to be checked after rule application. With this approach, declarative model-to-model transformation rules, compiled into OCL invariants, are interpreted as constraints that a pair of models should satisfy in order to synchronize them. This contrasts with the usual approach of using attribute computations in the rules.

2 Illustrating examples

2.1 Refactoring

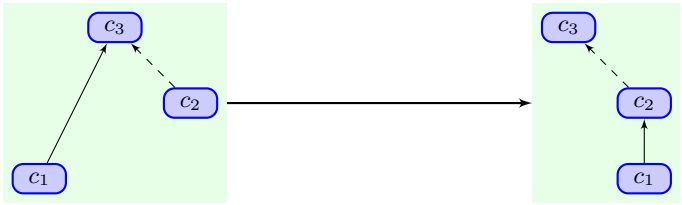

Fig. 1: Refactoring: Changing a class hierarchy

The first example, displayed in Figure 1, describes a refactoring step that might be carried out in an object-oriented programming language. The rule refers explicitly to three classes c_1, c_2 and c_3. The solid arrow $\longrightarrow$ is the direct subclass relation r, the dashed arrow $- \rightarrow$ the inheritance relation r^* (reflexive-transitive closure of subclass relation). The refactoring step consists in moving

c_1 below c_2, by cutting the direct subclass relation between c_1 and c_3 and introducing one between c_1 and c_2. The application context might require that this refactoring only extends, but does not restrict the previous inheritance relation, for example in order to avoid that method calls become undefined. If r is the subclass relation before and r' the subclass relation after refactoring, we can express this preservation property more formally by the requirement that $r^* \subseteq (r')^*$.

The delicate point about this transformation is the pattern $c_2 - \!\!\rightarrow c_3$ in the rule's left-hand side (LHS), because it might refer to an arbitrary number of intermediate nodes that are not explicitly mentioned in the rule. Indeed, the attentive reader will have realized that the rule as given in Figure 1 does not allow the intended correctness statement to be proved. We will come back to this point later, show which additional condition is required to repair the rule and how we can reduce reasoning about the transitive closure relation r^* in the rule's LHS to reasoning about the direct edge relation r.

2.2 Access Control

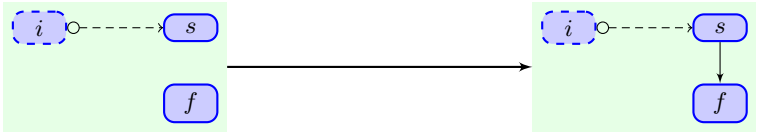

Fig. 2: Access control

The example displayed in Figure 2 depicts a scenario of access control: The node named s is a server that can host several files. We are about to add a new file f to the server. The server is not accessible to intruder i. We want to make sure that, after the transformation, the file f does not become accessible to i if it was not accessible before. Several remarks are in order here:

- "Accessibility" is again a transitive relation and is as such depicted by a dashed line in our figure (and its negation by a small circle). The immediate access relation (such as: file by server) is displayed by a solid arrow.
- Saying that the server is not accessible to a particular intruder I is not sufficient – we would rather like to express that it is not accessible to *any* intruder. This distinction is difficult to visualize. We have used a dashed box in Figure 2, but this notation is *ad hoc*. We are convinced that a textual notation is more precise – see Section 3.1, where we give the full description.
- It should be noted that in this case, we do not talk about the preservation of a separability relation, such as $(r')^* \subseteq r^*$, because this is clearly not the case (f becomes accessible from s). However, a restricted variant is provable, of the form $(r')^* \rhd I \subseteq r^* \rhd I$ ("reachachable from set I"), where I is the set of all intruders.

Transfo	$Refactoring(c_1, c_2, c_3)$	**Transfo**	$Refactoring(c_1, c_2, c_3)$
Appcond	$\ll c_1, c_3 \gg \wedge (c_2 \rightsquigarrow c_3)$	**Appcond**	$\ll c_1, c_3 \gg \wedge (c_2 \rightsquigarrow c_3)$ $\wedge \neg (c_2 \rightsquigarrow c_1)$
Action	*delete-edges:* $\ll c_1, c_3 \gg$ *add-edges:* $\ll c_1, c_2 \gg$	**Action**	*delete-edges:* $\ll c_1, c_3 \gg$ *add-edges:* $\ll c_1, c_2 \gg$
	(a) original		(b) corrected

Fig. 3: Definition of refactoring rule

3 Reductions

3.1 Graphs and Graph Transformation

In its simplest form, a graph *gr* is a datatype with two functions *nodes* (yielding the set of the nodes of the graph) and *edges* (yielding the set of edges of the graph). An edge is just an ordered pair of nodes. The node set of a graph is assumed to be finite (and, consequently, is the edge set).

A graph transformation rule is characterized by the following elements:

- A *name*, followed by a list of *parameters* that designate nodes of the graph that the rule is applied to.
- An *applicability condition*, having as only free variables the rule's parameters. This condition is a *path formula* whose structure will be defined more precisely below.
- An *action* describing which nodes and edges are to be deleted respectively added during the transformation.

We could introduce more complex notions of graph, for example graphs with typed nodes and edges, but we refrain from doing it here, because these concerns are orthogonal to the questions dealt with in the following.

The example transformation of Figure 1 is defined in Figure 3a as a transformation applicable to three nodes c_1, c_2 and c_3. The applicability condition is that there is an edge between c_1 and c_3 (written as $\ll c_1, c_3 \gg$) and a path between c_2 and c_3 (written as $c_2 \rightsquigarrow c_3$). The action is to delete the arc between c_1 and c_3 and to add one between c_1 and c_2.

A more precise definition of *path expression* pe and *path formula* pf is as follows:

$$
\begin{array}{lll}
pe ::= \ll n_1, n_2 \gg & & \text{– edge between nodes } n_1 \text{ and } n_2 \\
\quad \mid \; n_1 \rightsquigarrow n_2 & & \text{– path between nodes } n_1 \text{ and } n_2 \\
\\
pf ::= pe & & \text{– elementary path formula} \\
\quad \mid \; \neg pf & & \\
\quad \mid \; pf \wedge pf & & \\
\quad \mid \; \forall n.pf & &
\end{array}
$$

The use of quantification is illustrated in Figure 4. The rule has only two free variables, s and f. The precondition expresses that there is no intruder i that can access node s.

Transfo	$AC(s, f)$
Appcond	$\forall i.\neg(i \rightsquigarrow s)$
Action	*add-edges:* $\ll s, f \gg$

Fig. 4: Definition of access control rule

Before we can sketch what it means to apply a rule to a target graph gr, we need the notion of *morphism* which maps the variables of the rule (thus, c_1, c_2 and c_3 in the example of Figure 3) to nodes of the target graph. The morphism of Figure 5 is the mapping $[c_1 \mapsto n_1, c_2 \mapsto n_2, c_3 \mapsto n_3]$. Quite naturally, there are nodes of the graph (such as n_4) that are not in the image of the morphism.

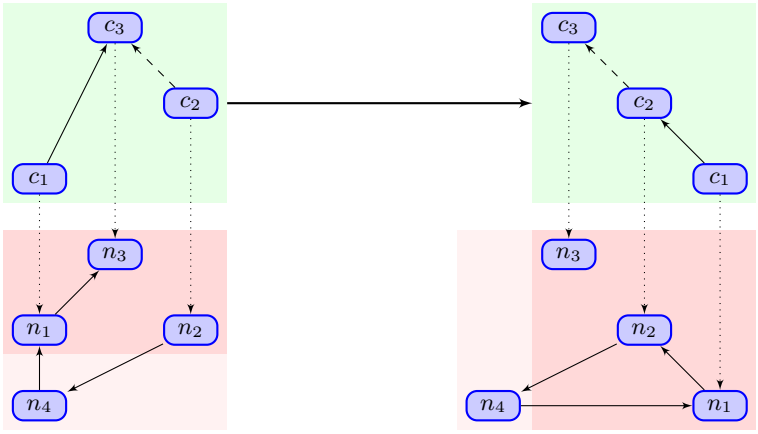

Fig. 5: Refactoring: Application of the rule

Given a graph gr, a graph morphism gm and a path formula pf, the predicate *path-form-interp* defines what it means for pf to be satisfied under gm in gr. The definition is the standard definition of an interpretation in logic, and we omit it here.

The application *apply-graphtrans-rel* of a graph transformation under a morphism performs the modifications specified in the "action" part of the transformation rule gt, by adding respectively deleting nodes and edges. The precise definition is technically more complex (see [23] for details) because it has to take deletion of dangling edges into account.

With these preliminaries, we can define *apply-transfo-rel*, the relation between a graph gr and the graph gr' resulting from applying the graph transformation gt to gr.

$$\frac{\exists gm.\mathit{path\text{-}form\text{-}interp}\ gr\ gm\ (\mathit{appcond}\ gt)\ \ \wedge\ \ \mathit{apply\text{-}graphtrans\text{-}rel}\ gt\ gr\ gr'}{\mathit{apply\text{-}transfo\text{-}rel}\ gt\ gr\ gr'}$$

Please note that this definition is entirely descriptive and not executable, because it imposes no choice as to which morphism gm (among several applicable morphisms) is selected.

The properties that we want to prove are properties of preservation of reachability or non-reachability (separation), of the following style:

- global preservation, of the form $(edges\ gr)^* \subseteq (edges\ gr')^*$ (reachability) or $(edges\ gr')^* \subseteq (edges\ gr)^*$ (separation).
- relativized preservation of reachability from a set: $(edges\ gr)^* \triangleright A \subseteq (edges\ gr')^* \triangleright A$, where $R \triangleright A = \{b.\exists a \in A.(a,b) \in R\}$ is the image of set A under relation R; and similarily for relativized separation.

3.2 Local Reasoning

A fundamental question underlying our approach is: is it possible to reason about a graph transformation by just taking into account the nodes appearing in the rule itself, without having to consider other nodes that might exist in the graph where the rule is applied?

In our previous work [24], we have shown that this is so, provided we restrict the applicability conditions to path formulae that are essentially Boolean combinations of simple edge relations. We will briefly recapitulate the approach in Section 3.4.

The situation becomes more complex in the presence of transitive closure, as in the relation $c_2 - \rightarrow c_3$ in Figure 5. As one can see in the example, the nodes in the image of the graph morphims, namely n_2 and n_3, are connected by a path running through node n_4, which is outside the image of the rule in the graph (the dark-shaded area in the lower part of Figure 5). And indeed, application of the rule leads to an incorrect result, in the sense that n_1 is no more connected to n_3 after application of the rule, contrary to the intention of the rule.

Specifically for this example, a solution is to forbid a path between c_2 and c_1, as expressed by the following modification of the applicability condition (see Figure 3b for the corrected version of the rule):

$$\mathbf{Appcond} \ll c_1, c_3 \gg \wedge (c_2 \rightsquigarrow c_3) \wedge \neg (c_2 \rightsquigarrow c_1)$$

Unfortunately, the negated path condition $\neg(c_2 \rightsquigarrow c_1)$ is more difficult to deal with than the positive one, $c_2 \rightsquigarrow c_3$. In the following, we will outline

– how to eliminate positive path conditions (see Section 3.3): the idea is to replace a path, such as $c_2 \rightsquigarrow c_3$, by an edge, such as $\ll c_2, c_3 \gg$, and thus to reduce reasoning about paths to reasoning about edges only.
– where negative path conditions come into play during elimination of positive path conditions.
– how to reason about graph transformations after these reductions (see Section 3.4).

We recall that we are mainly interested in problems of preservation of *reachability* of the form $(edges\ gr)^* \subseteq (edges\ gr')^*$. Slightly rewritten, this is the problem of showing $(x, y) \in (edges\ gr)^* \Rightarrow (x, y) \in (edges\ gr')^*$, for arbitrary x, y.

The problem of reachability from a set: $(edges\ gr)^* \triangleright A \subseteq (edges\ gr')^* \triangleright A$ is essentially the same, with an additional assumption $x \in A$.

Lastly, the problems of preservation of *separation* are symmetric and can be handled with identical methods. For these reasons, we only concentrate on the first kind of problem, namely $(x, y) \in (edges\ gr)^* \Rightarrow (x, y) \in (edges\ gr')^*$. The graph gr' does not appear as such, but as a function of gr.

To simplify the discussion of examples and avoid complicated case distinctions, we furthermore make the (otherwise inessential) assumption that graph morphisms are injective.

Example of Figure 3: In this example, we have to show $(x, y) \in \{(n_1, n_2)\} \cup (edges\ gr - \{(n_1, n_3)\})^*$, under the preconditions that $(x, y) \in (edges\ gr)^*$ and $(n_1, n_3) \in edges\ gr$ and $(n_2, n_3) \in (edges\ gr)^*$. After "repairing" the rule with the strengthened applicability condition, we furthermore have $(n_2, n_1) \notin (edges\ gr)^*$. □

3.3 Materialization of paths

The first step in our simplification procedure consists in replacing paths in our applicability conditions by edges. The following property justifies this step:

Lemma 1 (Path materialization).

$$(a, b) \in R^* \Longrightarrow (\{(a, b)\} \cup R)^* = R^*$$

and similarly for transitive closure $(.)^+$ *instead of reflexive-transitive closure* $(.)^*$.

Proof. We show the property for transitive closure; reflexive-transitive closure is similar, but slightly more involved. One direction of this equation is trivial, by monontonicity of transitive closure. The direction $(\{(a, b)\} \cup R)^+ \subseteq R^+$ can be seen by expanding $(v, w) \in (\{(a, b)\} \cup R)^+$ into $(v, w) \in R^+ \vee (v = a \vee (v, a) \in R^+) \wedge (b = w \vee (b, w) \in R^+)$ and then showing $(v, w) \in R^+$ by case distinction. □

The lemma expresses that a path $a \rightsquigarrow b$ known to exist in a graph (thus: $(a, b) \in R^+$) can be "materialized" by adding the edge $\ll a, b \gg$ (thus: $(\{(a, b)\} \cup R)$, and then taking the closure) without changing the path relation.

In Lemma 1, we have dealt with the addition of a new edge; we need a related lemma for removal of an edge, as the edge $\ll c_1, c_3 \gg$ in the graph of Figure 3.

Lemma 2 (Deletion of unreachable edge).

$$(v, a) \notin R^* \Longrightarrow ((v, w) \in (R - \{(a, b)\}))^* \leftrightarrow ((v, w) \in R^*)$$

It expresses that if a node a is not reachable from a node v in a graph, then any edge (a, b) starting from a can be removed without influencing the reachability from v.

Proof. Again, with monotonicity, the left-to-right direction is trivial.

As to the other direction: define the set $reach\ v\ R$ of nodes reachable from node v under relation R. It is now an easy inductive proof to show that

$$(v, w) \in R^* \Longrightarrow (v, w) \in (R \cap (reach\ v\ R) \times (reach\ v\ R))^*$$

Let us now assume that $(v, w) \in R^*$ and use this implication to show that $(v, w) \in (R \cap (reach\ v\ R) \times (reach\ v\ R))^*$. It follows that $((v, w) \in (R - \{(a, b)\}))^*$, because $a \notin reach\ v\ R$ and therefore $(R \cap (reach\ v\ R) \times (reach\ v\ R))^* \subseteq (R - \{(a, b)\})^*$. □

Lemma 1 and Lemma 2 are used as conditional rewrite rules in the process of materialization. The starting point is to show that $(\{(a, b)\} \cup R)^* = R^*$, if there is a path $a \rightsquigarrow b$ in the applicability condition of a rule. During simplification, we may obtain subgoals of the form $(x, y) \in R^*$, which may be simplified by

- recursive use of Lemma 1,
- recursive use of Lemma 2,
- monotonicity rules of the form $(x, y) \in R^* \Longrightarrow (x, y) \in S^*$, for $R \subseteq S$.

To ensure termination, we do not try to simplify or to prove goals of the form $(x, y) \notin R^*$. Rather, these negative path conditions have to be directly given as hypotheses. We emphasize that this is a heuristic process which is sound (we do not derive wrong conclusions) but not complete (our proof may fail even for valid proof goals).

With these observations, we can make some progress on our example.

Example of Figure 3: Under the preconditions enumerated before, we have to show $(x, y) \in (\{(n_1, n_2)\} \cup (edges\ gr - \{(n_1, n_3)\}))^*$.

We now materialized the path (n_2, n_3), thus showing that this goal is equivalent to $(x, y) \in (\{(n_2, n_3), (n_1, n_2)\} \cup (edges\ gr - \{(n_1, n_3)\}))^*$; this proof goal can then be tackled with the methods of Section 3.4.

Indeed,

- $(\{(n_2, n_3), (n_1, n_2)\} \cup (edges\ gr - \{(n_1, n_3)\}))^* = (\{(n_1, n_2)\} \cup (edges\ gr - \{(n_1, n_3)\}))^*$ (by Lemma 1) because $(n_2, n_3) \in (\{(n_1, n_2)\} \cup (edges\ gr - \{(n_1, n_3)\}))^*$
- $(n_2, n_3) \in (\{(n_1, n_2)\} \cup (edges\ gr - \{(n_1, n_3)\}))^*$ because $(n_2, n_3) \in (edges\ gr - \{(n_1, n_3)\})^*$ (by monotonicity of reflexive-transitive closure)
- $(n_2, n_3) \in (edges\ gr - \{(n_1, n_3)\})^* \leftrightarrow (n_2, n_3) \in (edges\ gr)$ (by Lemma 2) because $(n_2, n_1) \notin (edges\ gr)^*$ (by assumption).

□

3.4 Graph Decomposition

To make the paper self-contained, we mention the main steps of the further simplification procedure. A more technical development with details and justifications can be found in [23]. After the manipulations of Section 3.3, we are left with a goal of the form

$$(x, y) \in R^* \Longrightarrow (x, y) \in (R')^*$$

where R is the edge relation *edges gr* of the original graph and R' is the edge relation *edges gr'* of the transformed graph, possibly after addition of some edges that materialize paths.

When reasoning about a graph transformation, we do not know the nodes and edges that exist in the graph the rule is applied to. However, if the rules only have preconditions that are a Boolean combination of edge relations, it is sufficient to split the graph into an *interior* (the subgraph which lies entirely within the image of the rule's free variables under the graph morphism; the dark-shaded part in Figure 1) and an *exterior* (the rest of the graph; light-shaded in Figure 1). The exterior of the graph can henceforth be disregarded; it is sufficient to verify the desired property on the interior of the graph, which can be done by a Boolean satisfiability check or, in the simplest case, by a symbolic computation. *Example of Figure 3:* In the example proof, the remaining goal is to show

$$(x, y) \in (\mathit{edges}\ \mathit{gr})^* \Longrightarrow (x, y) \in (\{(n_2, n_3), (n_1, n_2)\} \cup (\mathit{edges}\ \mathit{gr} - \{(n_1, n_3)\}))^*$$

To get rid of the abstract set *edges gr*, we perform the above-mentioned split, which leaves us with the goal

$$\{(n_1, n_3)\}^* \subseteq \{(n_2, n_3), (n_1, n_2)\}^*$$

which can be verified by a simple symbolic computation. □

4 Conclusions

The present paper tries to make several points: Expressive transformation patterns (such as transitive closure) that go beyond what is commonly used in graph rewriting systems are useful in some application domains, and they are amenable to a formal analysis. In this sense, we have presented simplification strategies that reduce reasoning about paths to reasoning about edges (Section 3.3). These strategies can be understood as preprocessing steps carried out before verification procedures applicable to more restricted graph transformations (Section 3.4).

The simplification method we have presented is sound, but not complete. Apart from that, our approach is currently geared towards particular preservation properties (reachability and separation). Dealing with transitive closure is a difficult problem that quickly becomes undecidable [13]. We therefore think that our heuristic approach is a good compromise that we try to extend to other

common reasoning patterns. Apart from that, we will also investigate more systematic sound and complete procedures, but for weaker logics.

As witnessed by our examples, it is difficult to get rules right; in particular, this means that some preconditions covering unsuspected special cases are usually missing. Another interesting line of research is therefore to help developers of rules find the right applicability patterns for transformations that are supposed to satisfy particular correctness conditions.

References

1. Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Proceedings of MoDELS'10*, volume 6394 of *LNCS*. Springer, 2010.
2. Márk Asztalos, Péter Ekler, László Lengyel, Tihamer Levendovszky, Gergely Mezei, and Tamás Mészáros. Automated verification by declarative description of graph rewriting-based model transformations. *ECEASST*, 42, 2011.
3. Philippe Balbiani, Rachid Echahed, and Andreas Herzig. A dynamic logic for termgraph rewriting. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Graph Transformations*, volume 6372 of *Lecture Notes in Computer Science*, pages 59–74. Springer Berlin / Heidelberg, 2010.
4. Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COncurrent Systems with dynaMIC Allocated Heaps, COSMICAH '05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).
5. Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206:869–907, 2008.
6. Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
7. Bruno Courcelle and Irène Durand, A. Verifying monadic second order graph properties with tree automata. In Christophe Rhodes, editor, *Proceedings of the 3rd European Lisp Symposium*, pages 7–21, Lisboa, France, May 2010. 15 pages.
8. Simone André da Costa and Leila Ribeiro. Formal verification of graph grammars using mathematical induction. *Electronic Notes in Theoretical Computer Science*, 240(0):43 – 60, 2009. Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008).
9. Simone André da Costa and Leila Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming*, 77(4):480 – 504, 2012. Brazilian Symposium on Formal Methods (SBMF 2008).
10. Amir Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)*, 14:15–40, 2012.
11. Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(02):245–296, 2009.
12. Haruo Hosoya. *XML processing - The Tree-Automata Approach*. Cambridge University Press, 2011.

13. Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin / Heidelberg, 2004.
14. Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Graph-transformation verification using monadic second-order logic. In *Proceeding of the 13th International ACM SIGPLAN Symposium on Symposium on Principles and Practice of Declarative Programming*. ACM Press, July 2011.
15. Zarrin Langari and Richard Trefler. Application of graph transformation in verification of dynamic systems. In *Proceeding of the 7th International Conference on Integrated Formal Methods*, 2009.
16. Scott McPeak and George Necula. Data structure specifications via local equality axioms. In Kousha Etessami and Sriram Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 476–490. Springer Berlin / Heidelberg, 2005.
17. Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.
18. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002.
19. Fernando Orejas, Hartmut Ehrig, and Ulrike Prange. Reasoning with graph constraints. *Formal Aspects of Computing*, 22:385–422, 2010.
20. Karl-Heinz Pennemann. Resolution-like theorem proving for high-level conditions. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 289–304. Springer Berlin / Heidelberg, 2008.
21. Arend Rensink. The joys of graph transformation. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 9, 2005.
22. Leila Ribeiro, Fernando Luís Dotti, Simone André da Costa, and Fabiane Cristine Dillenburg. Towards theorem proving graph grammars using Event-B. *ECEASST*, 30, 2010. Proc. of International Colloquium on Graph and Model Transformation (GraMoT).
23. Martin Strecker. Interactive and automated proofs for graph transformations. Technical report, IRIT/ Université de Toulouse, 2012. http://www.irit.fr/~Martin.Strecker/Publications/proofs_graph_transformations.html.
24. Martin Strecker. Locality in reasoning about graph transformations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 7233 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin Heidelberg, 2012.
25. Hanh Nhi Tran and Christian Percebois. Towards a rule-level verification framework for property-preserving graph transformations. In *Proceeding of the IEEE ICST Workshop on Verification and Validation of Model Transformations*, April 2012.
26. Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.
27. Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214 – 234, 2007. Special Issue on Model Transformation.

28. Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. *J. Log. Algebr. Program*, 73(1-2):111–142, 2007.
29. Eduardo Zambon and Arend Rensink. Using graph transformations and graph abstractions for software verification. *Electronic Communications of the EASST*, 38, August 2011.