Under consideration for publication in Math. Struct. in Comp. Science

Interactive and automated proofs for graph transformations

Martin Strecker [†]

Institut de Recherche en Informatique (IRIT), Université de Toulouse, France

Email: `martin.strecker@irit.fr`

Received 30 May 2013; Revised 3 May 2018

This article explores methods to provide computer support for reasoning about graph transformations. We first define a general framework for representing graphs, graph morphisms and single graph rewriting steps. This setup allows for interactively reasoning about graph transformations. In order to achieve a higher degree of automation, we identify fragments of the graph description language in which we can reduce reasoning about global graph properties to reasoning about local properties, involving only a bounded number of nodes, which can be decided by Boolean satisfiability solving or even by deterministic computation of low complexity.

1. Introduction

1.1. *Setting the stage*

Graph transformations are an interesting conceptual model for describing structural modifications in natural sciences (such as chemistry and biology) and in particular in computer science. Here, they are used for model transformations in the area of model driven engineering, for representing concurrent systems, and for giving a high-level view of pointer manipulating programs. Since these application areas are often safety-critical, being able to reason about graph transformations is of major interest. Computer support for these reasoning tasks conveys a higher degree of reliability than paper-and-pencil proofs and becomes indispensable when the models are complex, a great number of correctness conditions has to be tracked and proofs have to be rechecked frequently after modifications in the design process. The situation becomes particularly delicate if transformations involve non-injective graph matchings, which lead to node aliasing that can entail a combinatorial explosion.

There are essentially two branches of computer supported formal reasoning: model checking and interactive theorem proving. The vast majority of existing approaches for verifying graph transformation systems follows the *model checking* paradigm (see Section 1.3 for a more detailed discussion). It explores which configurations are reached when

[†] This research has been funded by the *Climt* project (ANR-11-BS02-016). The work was completed while the author was at Southern University of Science and Technology, Shenzhen, China.

transformation rules are applied to a given start graph. Typical properties under investigation are therefore whether particular invariants are maintained during graph rewriting, or whether certain configurations are reachable. Model checking is attractive because it offers a high degree of automation and is therefore accessible also to uninitiated users. However, developers of model checkers have to strike a balance between limiting the expressivity of the property language and incurring undecidable or infeasibly complex proof problems. Furthermore, results obtained by model checking typically only hold for individual instances and not for entire classes of graphs or transformation systems.

The present article follows the *interactive theorem proving* approach and proposes, in its first part, a general framework for describing and reasoning about graph transformations in a proof assistant based on a higher-order logic. The question of interest is not primarily how a specific instance evolves under graph transformations, but which properties hold if a transformation is applied to an arbitrary graph, or at least an arbitrary graph satisfying some given preconditions.

In order to be able to manipulate transformations as first-class objects and to inspect them syntactically, we impose some restrictions on the language that formalizes applicability conditions for transformations, and represent this language as an inductive datatype. Apart from that, we have at our disposal the whole expressiveness of higher-order logic for stating properties of graphs and graph transformations and a panoply of methods (such as different forms of induction) for proving them. Section 2 is devoted to a description of this framework. It is versatile, but has the drawback that fully automated sound and complete verification cannot be expected. It is also extensible: we can easily distinguish between different kinds of graph matching morphisms (for example injective ones), and in Section 3, we show how to add typing information to the untyped development of Section 2.

In the second major part of this article, we explore methods for automating the reasoning tasks. The principle is to reduce reasoning about a graph with a potentially unbounded number of nodes (namely the graph a rule is applied to) to reasoning about a graph with a bounded number of nodes (namely the nodes occurring in the rule itself). This only succeeds if the impact of a rule on a graph is local, in a sense made precise below. We concentrate on a particular kind of proof problems, namely reachability problems, and examine two variants that differ by the applicability conditions of the rules. We first describe a set of simplification steps that reduce a proof goal to a Boolean satisfiability problem, for a rather general relational language defined in Section 4.

We then restrict this language still further to conjunctive relational expressions (Section 5). In this case, verification of the preservation of reachability in a graph can be reduced to a simple "calculational" proof requiring the computation of the transitive closure of a finite, concrete set of edges (those occurring in the applicability condition of the transformation rule). This reduction, which looks like a meta-result, is entirely carried out in our framework without appealing to extraneous notions. It is one of the benefits of treating transformations as first-class objects and of reflecting applicability conditions of transformations in an inductive datatype.

To summarize, the contribution of this article is a framework for reasoning about graph transformations in a proof assistant that can be used in two main scenarios:

— when reasoning about the effects a *specific* graph transformation may have on *arbitrary* graphs, thus offering an extension of what can usually be achieved by model checking.

— when reasoning about *arbitrary* graph transformations and of transforming them in such a way that their properties become verifiable automatically or even with low computational complexity.

1.2. *Problem statement*

A major subtopic of this article is to which degree reasoning about a graph transformation can be reduced to reasoning about the shape of the transformation rules. We will illustrate the problem with three examples.

Example 1.1. The first example presents a situation where such a kind of reasoning succeeds. In Figure 1, the upper part shows the transformation rule, consisting of a left hand side with nodes r_1, r_2, r_3 and an edge (r_1, r_3) and a right hand side which is obtained by deleting edge (r_1, r_3) and inserting new edges (r_1, r_2) and (r_2, r_3).

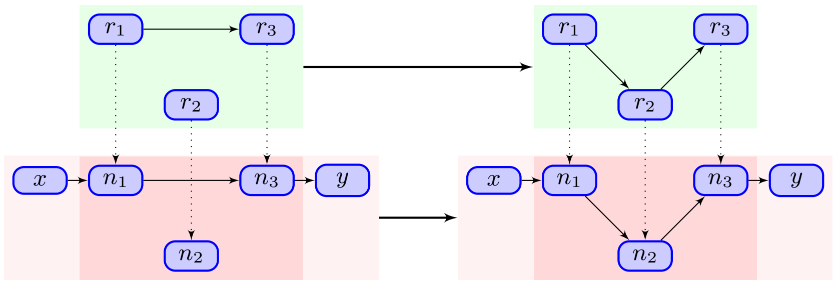

Figure 1: Simple rerouting

The dotted arrows map the rule into a target graph in the lower part. The question is: does this transformation preserve reachability in the graph? More formally, if e is the edge relation of the target graph and e^* its reflexive-transitive closure, if $(x, y) \in e^*$ in the original graph (for arbitrary x, y), is then also $(x, y) \in e^*$ in the transformed graph? We will show (see Theorem 5.4) that we can reduce the problem to simply looking at the transformation rule: $\{(r_1, r_3)\}^* \subseteq \{(r_1, r_2), (r_2, r_3)\}^*$, and thus the property carries over to any graph where the rule is applied.

Example 1.2. We will now see that, in general, this reasoning is incorrect. The example in Figure 2 models an information flow through hierarchically nested components, loosely inspired by an example in Asztalos et al. (2010); Tran and Percebois (2012).

Nodes with bold border are meant to represent "composite" components, the remaining components are "simple", and containment is represented by dashed arrows. There is a data flow relation from simple to composite (e_{sc}) and from composite to simple (e_{cs}) components, and between simple components, which we call e (unmarked edges in the figure). Here, we ask whether the connectivity is preserved by a transformation which

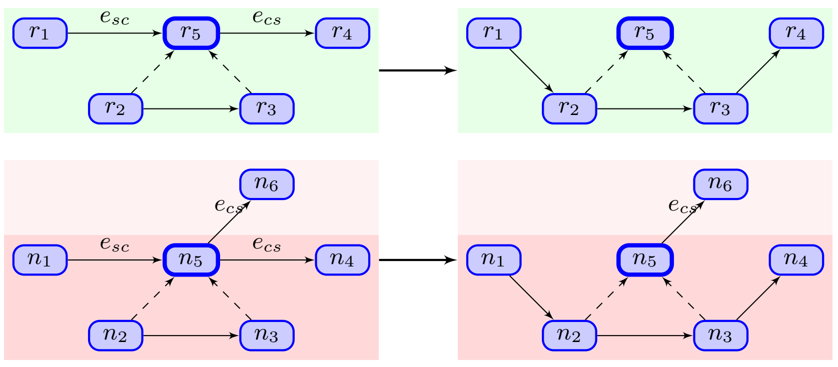

Figure 2: Complex rerouting

routes a flow through nested components r_2, r_3 instead of the containing block r_5. Differently said, is each flow $(x, y) \in (e \cup (e_{cs} O e_{sc}))^*$ preserved by the transformation, where O is relation composition? The rule seems to indicate that this is the case.

When embedding the rule (upper part) into a larger graph (lower part; here, the matching morphism is not shown explicitly), we see that this is not so: data can flow from n_1 to n_6 in the original graph, but not in the transformed graph (remember that the dashed edges are not data flow edges).

Finally, properties of transformations depend crucially on the properties of the morphisms mapping rules into graphs.

Example 1.3. Consider the transformation in Figure 3, where the apparently redundant edge between r_1 and r_3 is deleted. Again, we are interested in preservation of the flow relation. When reasoning purely on the basis of the left and right hand sides of the rule, it seems that the flow relation is unaltered, because $\{(r_1, r_2), (r_2, r_3), (r_1, r_3)\}^* = \{(r_1, r_2), (r_2, r_3)\}^*$.

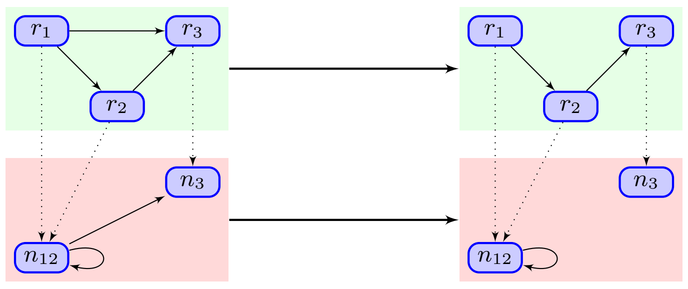

Figure 3: Rewriting under a non-injective morphism

However, a non-injective morphism mapping both node references r_1 and r_2 to node n_{12} and r_3 to n_3 also maps the edge (r_1, r_3) to the edge (n_{12}, n_3), which leads to the deletion of this edge. Therefore, in the resulting graph, the path between n_{12} and n_3 is not preserved.

The methods developed in this article also allow to take such aliasing problems into account, but provide special solutions for injective graph morphisms. We will come back to this situation in Example 5.1.

1.3. *Related work*

Most of the work on verification of graph transformations has so far concentrated on model checking, see Asztalos et al. (2010); Baldan et al. (2008); Ghamarian et al. (2012); Varró (2004) as a non-exhaustive list of representatives of this approach. Often, the purpose is to model concurrent systems as graph transformations and to investigate invariants and reachability problems of these systems. There is however a growing interest in using graph rewriting for model transformations (Arendt et al., 2010) with appropriate verification methods (Varró and Balogh, 2007).

Model checking approaches are rarely applicable for software verification, unless the data structures manipulated by a program are entirely generated by the rules from an initial graph, such as the red-black trees in Baldan et al. (2005), or unless abstraction functions are provided (Zambon and Rensink, 2011).

There is an immense body of work on verification of pointer structures in imperative programs. Static analyses often use specialized logics for expressing shapes of pointer structures (Immerman et al., 2004; Yorsh et al., 2007). 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 Møller and Schwartzbach (2001) or XML processing Hosoya (2011). The global approach developed in this article is not restricted to particular shapes of a graph, but we only offer full automation for specific forms of proof obligations. It remains that the relation between algebraic approaches to graph rewriting (Ehrig et al., 1997) and MSO-definable transductions (Courcelle and Engelfriet, 2011) still has to be explored in detail. MSO for verification of graph *properties* is explored by Courcelle and Durand (2010), but does not address graph *transformations*. Balbiani et al. (2010) develop a modal logic for reasoning about graph programs composed of fine-grained operators for manipulating nodes and redirecting edges.

Exploiting a local environment for reasoning about data structures, as advocated in Section 4 of this article, is adopted in several approaches: McPeak and Necula (2005) 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. There is intense activity around Separation Logic (Reynolds, 2002; Berdine et al., 2005) that constructs complex program properties out of properties of separate memory regions.

Graph rewriting has long been considered in an algebraic tradition as an extension of term rewriting. The tendency to interpret graph structures logically is rather recent (Rensink, 2005; Orejas et al., 2010). The work of Habel et al. (2006) is continued by Pennemann (2008a,b) and Habel and Pennemann (2009), who extract verification conditions from graph transformation programs and feed them into SAT solvers or first-order theorem provers. Rather similar to ours, the approach differs in two respects: it is entirely automatic and does not allow for human intervention for proving "difficult" properties;

and there is no tight coupling between the semantics (expressed in categorical terms in the cited work) and the proof obligation generator, and thus has to rely on a larger trusted code base.

da Costa and Ribeiro (2009, 2012) present a logical model for reasoning about graph transformations that is similar to ours. This approach has been implemented in Event-B (Ribeiro et al., 2010) by coding individual rules as Event-B machines and then profiting from the proof support for this platform to prove the correctness of rules. Further work is needed to see to which point this coding allows for the kind of meta-reasoning that we advocate in this article (Sections 4 and 5).

The present article is a synthesis and extension of work previously published in (Strecker, 2008, 2011). The formalization has been modularized by the introduction of theory contexts ("locales"), which has made it possible to separate structural transformations from typing issues, and to swap easily between different kinds of graph morphisms. The results presented in Section 5 are entirely new.

1.4. *Notation and presentation*

The development described in this article has been carried out in the Isabelle/HOL proof assistant (Nipkow et al., 2002), which combines a functional programming language with a higher-order logic. The functional fragment is similar to the ML family of programming languages. Any diverging notation and additional concepts will be introduced in paragraphs marked as "aside" where needed. We have refrained from using, in an essential manner, specifics of Isabelle that would not be available in similar form in other proof assistants such as Coq or PVS.

The text presented below is an extract of the formal Isabelle development with annotations in LaTeX. Propositions marked as lemma or theorem have been proved in Isabelle, and the formal development is available from the Web page of this article[†]. Proofs provided in this article therefore try to convey an intuition rather than give a complete correctness argument.

2. Graphs and graph transformations

This section introduces a minimalist concept of graphs and a notion of path expressions (Section 2.1) that are syntactic entities representing the applicability conditions of graph transformations (Section 2.2). These notions are sufficient for the further development. We will show later how they can be extended to graphs with typed nodes and edges (Section 3).

[†] `http://www.irit.fr/~Martin.Strecker/Publications/proofs_graph_transformations.html`

2.1. *Graphs and graph expressions*

Graphs are defined classically, as a structure consisting of a set of objects (the *nodes*) and a relation between objects (the *edges*). For the moment, the type of these objects is left abstract.

record *'obj graph* =
  nodes :: *'obj set*
  edges :: (*'obj* ∗ *'obj*) *set*

We limit our considerations to graphs that are structurally well-formed in the sense that the node set is finite and the endpoints of the edges belong to the node set (the *Field* of a relation is the union of its domain and its range). The first requirement is a convenience when reasoning about cardinalities of edge relations; the second requirement is intended to exclude erratic behavior of graph transformations (*utpd* is for "untyped", in contrast to the typed extensions introduced in Section 3).

definition *struct-wf-gr-utpd* :: (*'obj*, *'a*) *graph-scheme* ⇒ *bool* **where**
  struct-wf-gr-utpd gr = (*finite* (*nodes gr*) ∧ (*Field* (*edges gr*)) ⊆ (*nodes gr*))

Aside on Isabelle (records): Access to a component of a record, for example *nodes*, is written in functional notation. Isabelle uses a concept of extensible records that can be enriched with further components, as will be done with typing information in Section 3. To profit from this extensibility, we use *graph-scheme* instead of *graph* in the type of the above function; the additional type parameter is introduced for technical reasons (see (Naraschewski and Wenzel, 1998)) and can be neglected here.

Each graph transformation rule has an applicability condition that describes under which circumstances the rule can be applied; at the same time, it identifies the nodes and edges to be transformed. We have chosen to deeply embed these conditions in the proof assistant, by defining their abstract syntax as a datatype, rather than directly using a predicate in Isabelle's logic. This choice is motivated by two reasons: Firstly, the applicability conditions can refer to an arbitrary number of nodes, but there is no uniform type in simply-typed lambda calculus for an "n-ary predicate" (for variable n), and consequently neither for graph transformations. Secondly, we now have the possibility to characterize fragments of the property language, by means expressible in the proof assistant itself, as for example in the reductions of Section 5.

We first define inductively a type of node set expressions *'nt nodeset* and of path expressions *'et path*. Anticipating the introduction of typed graphs further below, these inductive types are parameterized by node types *'nt* respectively edge types *'et*. Path expressions are inspired by the relation algebra of Tarski (1941) (some of them are not explicitly represented, but can be defined), but go beyond it in that also a transitive closure operator is present.

The constructors of *nodeset* with their informal semantics are the following (the formal semantics will be given below):

— ⌈*Univs*⌉, the set of all nodes.
— ≪n≫, the singleton node set with node reference n. We make a distinction between

a node reference in a node set or path expression (a natural number, as motivated below) and a node (an object of a graph).
— $\lceil$:: *nt*$\rceil$, the set of nodes of node type *nt*.

The constructors of *path* are:

— $\lceil$*Idp*$\rceil$, the identity path (identity relation).
— $\lceil$*Univp*$\rceil$, the universal path (universal relation).
— $\ll n,\ n'\gg$, the edge between node reference n and n'.
— $\lceil$:: *et* ::$\rceil$, the set of edges of edge type *et*.
— $\lceil - \rceil p$, the complement of path p.
— $p\ \lceil^{-1}\rceil$, the converse path (the path p with the edges reversed).
— $p\lceil;\rceil p'$, the composition of paths p and p'.
— $p\lceil + \rceil p'$, the alternative (path p or p').
— $p\lceil\ \hat{}+\rceil$, the transitive closure of path p.

On top of these notions, we build path formulas of type $('nt,\ 'et)$ *path-form* with the constructors:

— $(n\ \lceil\in\rceil\ s)$: node reference n is in node set s.
— $(n \rightsquigarrow\ n'\ \lceil\in\rceil\ p)$: there is a path from node reference n to n' in path p.
— $\lceil\neg\rceil$ *pf*: negation of path formula.
— *pf* $\lceil\wedge\rceil$ *pf*$'$: conjunction of path formulas.
— $\lceil\forall\rceil$ *pf*: universal quantification over the graph nodes.

In the following, we will use $\lceil\ll n\gg\rceil$ as abbreviation for $(n\ \lceil\in\rceil\ \ll n\gg)$, saying that n is a node of the graph under consideration (node sets are relativized to graph nodes), and $\lceil\ll n1,\ n2\gg\rceil$ as abbreviation for $(n1 \rightsquigarrow n2\ \lceil\in\rceil\ \ll n1,\ n2\gg)$, saying that $(n1, n2)$ is an edge. A notation like $\lceil\{1,\ 2,\ 3\}\rceil$ is a shorthand for $\lceil\ll 1\gg\rceil\ \lceil\wedge\rceil\ \lceil\ll 2\gg\rceil\ \lceil\wedge\rceil\ \lceil\ll 3\gg\rceil$ ("$\{1,\ 2,\ 3\}$ is a set of nodes of the graph"). Clearly, we can describe the shape of any finite graph by a conjunction of formulas of the form $\lceil\ll n\gg\rceil$ and $\lceil\ll n1,\ n2\gg\rceil$. We also use connectors and quantifiers such as $\lceil\vee\rceil$ and $\lceil\exists\rceil$ which are defined in the obvious way from the constructors of path formulas.

The formal semantics of *nodeset* and *path* expressions is presented in Figures 4 and 5, respectively. Using Isabelle's locale mechanism (see next paragraph), these expressions are interpreted relative to a fixed graph *gr* and interpretation functions *I-nt* for node types and *I-et* for edge types. The reader can neglect them until the introduction of typed graphs in Section 3. Thus, *gr*, *I-nt* and *I-et* are implicit parameters of the interpretation functions. There is a an explicit parameter I mapping node references (natural numbers) to nodes of a graph (of type $'obj$). With these prerequisite and the informal semantics, the functions *nodeset-interp* and *path-interp* offer few surprises. In Isabelle, the converse of a relation is written with postfix $^{-1}$, relation composition as O and (non-reflexive) transitive closure as postfix $^{+}$. Node and edge sets are always relativized to the node and edge sets of graph *gr*.

Aside on Isabelle (locales): Locales (Ballarin, 2004) are Isabelle's mechanism for structured theory development. Definitions can be carried out relative to previously fixed local constants and under local assumptions that axiomatize a theory. Locales can also be extended with new constants or assumptions, thus giving rise to a locale hierarchy.

```
fun nodeset-interp :: [nat ⇒ 'obj, 'nt nodeset] ⇒ 'obj set where
  nodeset-interp I (⌈Univs⌉) = nodes gr
| nodeset-interp I (≪n≫) = (nodes gr ∩ {I n})
| nodeset-interp I (⌈:: nt⌉) = (nodes gr ∩ I-nt nt)
```

Figure 4: Semantics of node sets

```
fun path-interp :: [nat ⇒ 'obj, 'et path] ⇒ ('obj * 'obj) set where
  path-interp I (⌈Idp⌉) = Id-on (nodes gr)
| path-interp I (⌈Univp⌉) = (edges gr)
| path-interp I (≪n1, n2≫) = (edges gr ∩ {(I n1, I n2)})
| path-interp I (⌈:: et ::⌉) = (edges gr ∩ I-et et)
| path-interp I (⌈−⌉ p) = ((nodes gr) × (nodes gr)) − (path-interp I p)
| path-interp I (p⌈⁻¹⌉) = (path-interp I p)⁻¹
| path-interp I (p ⌈;⌉ p') = (path-interp I p) O (path-interp I p')
| path-interp I (p ⌈+⌉ p') = (path-interp I p) ∪ (path-interp I p')
| path-interp I (p⌈ˆ+⌉) = (path-interp I p)ˆ+
```

Figure 5: Semantics of paths

Locales can later be instantiated, replacing the local constants by specific values. Instantiation generates proof obligations which ascertain that the instances satisfy the locale's assumptions. Locales can make definitions and theorems very compact, but they tend to veil implicit parameter dependencies. Thus, the function definitions in Figures 4, 5, 6 have *gr*, *I-nt* and *I-et* as implicit parameters. In this article, we do not display the locale declarations in Isabelle's syntax, but only describe them verbally.

The function *path-form-interp* (in Figure 6) injects path formulas (which can be understood as meta-level formulas) back into formulas of type *bool* of Isabelle' s object logic and thus provides an interpretation of path formulas. The definition is relatively standard, except for the treatment of the universal quantifier: In order to avoid the typical complications involving bound variables, we use a nameless representation of bound variables with de Bruijn indices (de Bruijn, 1972). In the interpretation function for universal quantification, the currrently introduced variable x is assigned to the node reference 0, and the remaining node references are shifted by one position. For this reason, node references are natural numbers.

The definition of the interpretation functions is entirely constructive: Because the interpretation of quantifiers ranges over the node set of a graph (which we have assumed to be finite), for any computable function I and any path formula *pf* we can effectively determine whether *path-form-interp I pf* holds for a well-structured graph *gr*.

Let us give some further encodings: Equality *n1* ⌈=⌉ *n2* of two nodes *n1* and *n2* is defined by (*n1* ⇝ *n2* ⌈∈⌉ ⌈*Idp*⌉), and inequality ⌈≠⌉ by its negation. The expression *0*⌈≠⌉*1* is not a trivially true formula, but states that node references *0* and *1* refer to different nodes. Even though injectivity of morphisms could thus be coded into appli-

$\textbf{fun}\ \textit{path-form-interp} :: [\textit{nat} \Rightarrow \textit{'obj}, (\textit{'nt}, \textit{'et})\ \textit{path-form}] \Rightarrow \textit{bool}\ \textbf{where}$
$\quad \textit{path-form-interp}\ I\ (n\ \lceil\in\rceil\ s) = (I\ n \in \textit{nodeset-interp}\ I\ s)$
$\mid \textit{path-form-interp}\ I\ (n \rightsquigarrow n'\ \lceil\in\rceil\ p) = ((I\ n,\ I\ n') \in \textit{path-interp}\ I\ p)$
$\mid \textit{path-form-interp}\ I\ (\lceil\neg\rceil\ \textit{pf}) = (\neg\ (\textit{path-form-interp}\ I\ \textit{pf}))$
$\mid \textit{path-form-interp}\ I\ (\textit{pf}\ \lceil\wedge\rceil\ \textit{pf}') = ((\textit{path-form-interp}\ I\ \textit{pf}) \wedge (\textit{path-form-interp}\ I\ \textit{pf}'))$
$\mid \textit{path-form-interp}\ I\ (\lceil\forall\rceil\ \textit{pf}) =$
$\qquad\qquad (\forall\ x.\ x \in \textit{nodes gr} \longrightarrow \textit{path-form-interp}\ ((I \circ (\lambda\ x.\ x - 1))(0{:=}x))\ \textit{pf})$

Figure 6: Semantics of path formulas

cability conditions, we prefer to deal with properties of morphisms separately below, in order to obtain more concise reduction statements (see Section 5).

We now show how to express cardinality constraints in our language. Take the following path formula that says that node n is connected through a path specified by path expression p with at least two distinct other nodes:

$$\lceil\exists\rceil\ \lceil\exists\rceil\ 0\lceil\neq\rceil 1\ \lceil\wedge\rceil\ (n{+}2 \rightsquigarrow 0\ \lceil\in\rceil\ p)\ \lceil\wedge\rceil\ n{+}2\lceil\neq\rceil 0\ \lceil\wedge\rceil\ (n{+}2 \rightsquigarrow 1\ \lceil\in\rceil\ p)\ \lceil\wedge\rceil\ n{+}2\lceil\neq\rceil 1$$

When applying *path-form-interp* I to this path formula, we obtain the object logic formula

$$\begin{aligned}&\exists\, b.\ b \in \textit{nodes gr} \wedge (\exists\, a.\ a \in \textit{nodes gr} \wedge b \neq a \wedge\\ &\quad (I\ n,\ a) \in \textit{path-interp}\ (I'\ a\ b)\ p \wedge (I\ n \in \textit{nodes gr} \longrightarrow a \neq I\ n) \wedge\\ &\quad (I\ n,\ b) \in \textit{path-interp}\ (I'\ a\ b)\ p \wedge (I\ n \in \textit{nodes gr} \longrightarrow b \neq I\ n))\end{aligned}$$

where $(I'\ a\ b)$ is the interpretation that sends node reference 0 to node a and 1 to node b and otherwise behaves like interpretation I.

The path formula just discussed is in fact an expansion of *cardrel-geq* $n\ p\ 2$, where *cardrel-geq* $n\ p\ k$ expresses that node n is related through path p with at least k distinct nodes. It is defined as follows (assuming that p contains no free node references):

$\textbf{definition}\ \textit{cardrel-geq} :: \textit{nat} \Rightarrow \textit{'et path} \Rightarrow \textit{nat} \Rightarrow (\textit{'nt}, \textit{'et})\ \textit{path-form}\ \textbf{where}$
$\quad \textit{cardrel-geq}\ n\ p\ k =$
$\qquad \textit{equantif}\ k\ (\textit{conjs-form}\ [(x\ \lceil\neq\rceil\ y).\ x \leftarrow [0\ ..{<}\ k],\ y \leftarrow [0\ ..{<}\ k],\ (x < y)\ ]\ \ \lceil\wedge\rceil$
$\qquad\qquad\qquad \textit{conjs-form}\ [\ ((n + k \rightsquigarrow\ x\ \lceil\in\rceil\ p)\ \lceil\wedge\rceil\ (n + k\ \lceil\neq\rceil\ x)).\ x \leftarrow [0\ ..{<}\ k]])$

Here, *equantif* k generates an existential quantifier prefix of length k, and *conjs-form* takes the conjunct of a list of path formulas (the latter constructed with a Haskell-like list comprehension).

We have presented the full language of paths and path formulas, which is more expressive than what is needed in the following development and than what can be dealt with by the automated procedures presented in Sections 4 and 5, even though it is useful for specifying properties that can be proved interactively.

2.2. *Graph transformations*

Before introducing the notion of graph transformation, let us first turn to Figure 1 for an informal discussion. The graph in the upper left part, consisting of node references

r_1, r_2, r_3 and an edge between r_1 and r_3, is the application condition, which is particularly simple in this case, but could be any path formula, as seen in the previous section. The transformation pattern is specified by indicating the set of nodes to be generated and deleted, and similarly for the edges. In our example, we might choose to keep the node set the same, delete the edge (r_1, r_3) and replace it by two new edges (r_1, r_2) and (r_2, r_3). Pictorially, this gives the graph in the upper right part of Figure 1.

More formally, we represent a graph transformation by a record consisting of an application condition (a *path-form*), sets of deleted and generated node references and sets of deleted and generated edge references:

```
record ('nt, 'et) graphtrans =
  — applicability condition
  appcond :: ('nt, 'et) path-form
  — mapping of nodes
  ndel  :: nat set            — deleted nodes
  ngen  :: nat set            — generated nodes
  — mapping of edges
  edel  :: (nat ∗ nat) set    — deleted edges
  egen  :: (nat ∗ nat) set    — generated edges
```

This is the most elementary notion of graph transformation which only takes into account structural aspects; typing will be added in Section 3.

Example 2.1. With the abbreviations of page 8, we can now define the transformation of Figure 1:

```
definition rerouting where rerouting =
  (| appcond = ⌈{1, 2, 3}⌉ ⌈∧⌉ ⌈≪1, 3≫⌉ ,
     ndel = {},          ngen = {},
     edel = {(1,3)},     egen = {(1,2), (2,3)}  |)
```

We have to impose some well-formedness restrictions on transformations that will allow us to prove (see Theorem 2.4) that graph transformations preserve the well-formedness of graphs. In particular, we postulate that the node references of deleted nodes are among the free variables of the application conditions of the graph transformation, that only a finite number of nodes is generated, and that these nodes are not among the free variables of the application conditions. There are similar restrictions for the edges (the *Field* of a relation being the union of its domain and range), which are however quite liberal, in view of the expressiveness of the application condition language. Thus, an edge scheduled for deletion (*edel*) is not required to exist in the graph, only its endpoints have to be among the free variables of the application condition. Altogether, structural well-formedness of untyped graph transformations is defined by:

```
definition struct-wf-gt-utpd :: ('nt, 'et, 'a) graphtrans-scheme ⇒ bool where
  struct-wf-gt-utpd gt =
   ((ndel gt) ⊆ (fv-path-form (appcond gt)) ∧
    finite (ngen gt) ∧ fv-path-form (appcond gt) ∩ (ngen gt) = {} ∧
    (egen gt) ∩ (edel gt) = {} ∧
```

$(Field\ (edel\ gt)\ \subseteq\ fv\text{-}path\text{-}form\ (appcond\ gt)) \wedge$
$(Field\ (egen\ gt)\ \subseteq\ (fv\text{-}path\text{-}form\ (appcond\ gt) - (ndel\ gt)) \cup (ngen\ gt)))$

Note that we can easily compute the set of free variables of an application condition (*fv-path-form*) because path formulas are represented as a data type; similar syntactic manipulations would not be possible within Isabelle's object logic.

Referring back to Example 2.1, we obtain the:

Result 2.1. [‡] *struct-wf-gt-utpd rerouting*

Proof. Even though the node and edge set components of a graph transformation could be stated in abstract terms (*i.e.* as predicates), they are most likely given by an enumeration of their elements, as for *rerouting*. In this case, the proof is purely calculational, obtained by expanding the definitions and simplifying the resulting terms. □

Central to defining the application of a transformation to a graph is the concept of morphism, which in our case is a map from node references to objects.

type-synonym *'obj graphmorph* = (*nat* ⇒ *'obj option*)

We recall that the *option* type has the two constructors *None*, which can be conceived as representing an undefined value, and *Some y*, representing a defined value *y*. Maps are therefore convenient for describing partial functions. Isabelle provides some predefined functions on maps: *dom m* is the set of those *x* for which *m x* is defined, *ran m* is the set of all *y* with *m x* = *Some y*. Restriction *m*|‘*A* restricts the domain of *m* to set *A* by sending all elements outside of *A* to *None*, and addition (*m1* ++ *m2*) *x* selects the value of *m2 x* if it is defined, and otherwise *m1 x*.

We can now describe what it means to apply a graph transformation to a graph. We first assume that we have already identified a graph morphism *gm* going from the node references of the applicability condition to the nodes of a graph *gr* to which we apply the graph transformation *gt*. Such a morphism is visualized by the dotted lines in the left part of Figure 1. The application is defined relationally, by specifying what a correct result graph *gr'* should look like. Indeed, as it will soon turn out, the definition contains some non-constructive existential quantifiers (definitions *applicable-transfo* and *apply-transfo-rel* below) and is therefore not executable. Before discussing the full definition, we consider the special case of a transformation which does not generate new nodes:

definition *apply-graphtrans-rel-nogen* ::
  [(*'nt*, *'et*, *'a*) *graphtrans-scheme*, *'obj graphmorph*,
   (*'obj*, *'b*) *graph-scheme*, (*'obj*, *'b*) *graph-scheme*] ⇒ *bool* **where**
 apply-graphtrans-rel-nogen gt gm gr gr' =
  (*let del-nodes* = *ran* (*gm* |‘ (*ndel gt*)) *in*
   let nds = ((*nodes gr*) − *del-nodes*) *in*
   let del-edges = ((*emorph gm*) ‘ (*edel gt*)) *in*

‡ We denote properties of examples as “result” and more general properties as “lemma” or “theorem”, but there is no formal difference between them.

$$\begin{aligned}&\quad \textit{let gen-edges} = ((\textit{emorph gm})\ `\ (\textit{egen gt}))\ \textit{in}\\&(\textit{nodes gr}' = \textit{nds}) \wedge (\textit{edges gr}' = (\textit{restrict-rel}\ (\textit{edges gr})\ \textit{nds} - \textit{del-edges}) \cup \textit{gen-edges}))\end{aligned}$$

The nodes that are deleted in graph *gr'* are the image of the set *ndel gt* under morphism *gm*, the nodes of the result graph are the nodes of the original graph minus the ones that are deleted. Similarly for edges: we calculate the deleted and generated edges as the image of the sets *edel gt* respectively *egen gt*. Here, $f\ `\ S$ is the image of a set S under a function *f*, *i.e.* the set $\{y.\ \exists\ x \in S.\ f\,x = y\}$, and *emorph gm* lifts the graph morphism *gm* from nodes to edges. We restrict the edge relation of the resulting edge set to the set of result nodes, which amounts to deleting dangling edges.

The full definition for transformations of untyped graphs (*apply-graphtrans-rel-utpd* in Figure 7) deals with two additional aspects: Firstly, we want to characterize properties of morphisms and therefore introduce the parameter *morph-prop*, a predicate on morphisms. We can now easily provide locale extensions for arbitrary morphisms (instantiation of *morph-prop* with the constantly *True* function) and injective morphisms (instantiation with *inj-map*), where *inj-map* is defined as $\textit{inj-map}\ m \equiv \textit{inj-on}\ m\ (\textit{dom}\ m)$, saying that m is a function that is injective on its domain.

The second extension concerns the generation of new nodes. The situation is comparable to storage allocation in imperative programs; however, our "memory model" is extremely reduced. The generic object type *'obj* only allows to compare two objects for equality. Unlike in memory models with storage cells arranged in a discrete linear order, we cannot express that we want to allocate a new node in the "next" free cell. We therefore express that there are nodes *gen-nodes* that are not in the current graph and that are the image of *ngen gt* under an extension morphism *gen-gm* of *gm* (definition *graphtrans-gen-nodes*). Even though the existential quantifiers in definition *apply-graphtrans-rel-utpd* are not directly executable, there is a constructive implementation based on an object type of discrete linear orders.

For the rest of this section, we open a locale of graph transformations where we fix a graph *gr*, the interpretation functions *I-nt* and *I-et*, a graph transformation *gt* and a predicate on morphisms, *morph-prop*. Technically, this is achieved by an extension of the locale encountered in Section 2.1. To remain generic, we also fix function *apply-graphtrans-rel* :: [*'obj graphmorph*, (*'obj*, *'a*) *graph-scheme*] ⇒ *bool* that will later be instantiated with typed respectively untyped transformation functions. It expresses that the current graph *gr* is transformed to a successor graph under a graph morphism.

We say that a graph morphism *gm* is applicable relative to a path formula *pf* if *gm* maps the free variables of *pf* into the nodes of *gr*, and the morphism satisfies the path formula (here, (*the* ∘ *gm*) turns the map *gm* into a function). The transformation *gt* is applicable to graph *gr* if there exists an applicable morphism, and *apply-transfo-rel* defines the relation between *gr* and a graph *gr'* resulting from transforming *gr* into *gr'* by the application of some applicable morphism. Here, *SOME x. P x* is Hilbert's choice operator yielding an x satisfying the predicate P. In this case, it selects an arbitrary morphism *gm* among those that satisfy the applicability condition of *gt*.

definition *applicable-gm* :: [*'obj graphmorph*, (*'nt*, *'et*) *path-form*] ⇒ *bool* **where**
  applicable-gm gm pf =

```
definition graphtrans-gen-nodes ::
  [('nt, 'et, 'a) graphtrans-scheme, ('obj, 'b) graph-scheme, 'obj graphmorph, 'obj set]
    ⇒ bool where
  graphtrans-gen-nodes gt gr gen-gm gen-nodes =
  (dom gen-gm = ngen gt ∧ ran gen-gm = gen-nodes ∧ gen-nodes ∩ nodes gr = {})

definition apply-graphtrans-rel-nodes-for-gen ::
  [('nt, 'et, 'a) graphtrans-scheme, 'obj graphmorph,
    ('obj, 'b) graph-scheme, ('obj, 'b) graph-scheme, 'obj graphmorph, 'obj set] ⇒ bool
where
  apply-graphtrans-rel-nodes-for-gen gt gm gr gr' gen-gm gen-nodes =
    (graphtrans-gen-nodes gt gr gen-gm gen-nodes ∧
    (let del-nodes = ran (gm |` (ndel gt)) in
     let nds = ((nodes gr) − del-nodes) ∪ gen-nodes in
     let del-edges = ((emorph gm) ` (edel gt)) in
     let gen-edges = ((emorph (gm ++ gen-gm)) ` (egen gt)) in
    (nodes gr' = nds) ∧
    (edges gr' = (restrict-rel (edges gr) nds − del-edges) ∪ gen-edges)))

definition apply-graphtrans-rel-utpd ::
  [('nt, 'et, 'a) graphtrans-scheme, 'obj graphmorph, 'obj graphmorph ⇒ bool,
    ('obj, 'b) graph-scheme, ('obj, 'b) graph-scheme] ⇒ bool where
  apply-graphtrans-rel-utpd gt gm morph-prop gr gr' =
  (∃ gen-gm gen-nodes.
     morph-prop (gm ++ gen-gm) ∧
     apply-graphtrans-rel-nodes-for-gen gt gm gr gr' gen-gm gen-nodes)
```

Figure 7: Application of graph transformation

```
    ((dom gm = fv-path-form pf) ∧ (ran gm ⊆ nodes gr) ∧
      morph-prop gm ∧ path-form-interp (the ∘ gm) pf)
definition applicable-transfo :: bool where
  applicable-transfo = (∃ gm. applicable-gm gm (appcond gt))
definition apply-transfo-rel :: ('obj, 'a) graph-scheme ⇒ bool where
  apply-transfo-rel gr' = apply-graphtrans-rel (SOME gm. applicable-gm gm (appcond gt)) gr'
```

We now want to show under which conditions graph transformations preserve structural well-formedness of graphs (page 7). Here, we concentrate on untyped transformations and therefore extend the graph transformation locale in the following way: we assume *struct-wf-gr-utpd gr* and *struct-wf-gt-utpd gt*, and its parameter *apply-graphtrans-rel* is instantiated by defining it as an untyped graph transformation:

apply-graphtrans-rel gm gr′ = apply-graphtrans-rel-utpd gt gm morph-prop gr gr′

It is now easy to show the following:[§]

Lemma 2.2. *apply-graphtrans-rel gm gr′* $\Longrightarrow$ *finite* (*nodes gr′*)

For stating the preservation of the second subcondition of structural well-formedness, we need the following separation property of a morphism m by a set S, saying (in its contrapositive form) that $x \in S$ and $y \notin S$ cannot have the same image under m:

definition *morph-sep* **where** *morph-sep m S* $= (\forall\ x\ y.\ x \in S \longrightarrow m\ x = m\ y \longrightarrow y \in S)$

This predicate embodies a weak form of injectivity of morphisms. Indeed, *inj-map m* is equivalent to $\forall\ S \subseteq$ *dom m*. *morph-sep m S*. In a locale of injective morphisms, we therefore directly inherit this property. It may otherwise require a non-trivial proof.

The following lemma is instrumental for proving Theorem 2.4. Please remember that we are in a global context in which both *gr* and *gt* are assumed to be well-formed (*struct-wf-gr* respectively *struct-wf-gt-utpd*).

Lemma 2.3.

graphtrans-gen-nodes gt gr gen-gm gen-nodes $\wedge$
applicable-gm gm (*appcond gt*) $\wedge$ *morph-sep gm* (*ndel gt*)
$\Longrightarrow$ *Field* (*emorph* (*gm* ++ *gen-gm*) ‘ *egen gt*) $\subseteq$ *nodes gr* $-$ *ran* (*gm* |‘ *ndel gt*) $\cup$ *gen-nodes*

Proof. The lemma amounts to showing that nodes belonging to new edges (the image of *egen gt*) are among the old nodes of the graph or the newly generated nodes, but not among the nodes to be deleted. Take n and d to be node references occurring in the application condition of *gt*, with $n \notin$ *ndel gt* and $d \in$ *ndel gt*. Thus, according to *struct-wf-gt-utpd*, node reference n could belong to a new edge, as specified in *egen gt*. However, if the graph morphism maps n and d to the same node, this node *gm n* $=$ *gm d* will be deleted in the target graph, leading to a dangling edge after transformation. To avoid such a situation, we impose the condition *morph-sep gm* (*ndel gt*) that prevents n and d to be mapped to the same node in the target graph. □

We can now show preservation of well-formedness: If applying an applicable graph morphism *gm* produces a graph *gr′*, then this graph is well-formed.

Theorem 2.4.

applicable-gm gm (*appcond gt*) $\wedge$ *morph-sep gm* (*ndel gt*) $\wedge$ *apply-graphtrans-rel gm gr′*
$\Longrightarrow$ *struct-wf-gr-utpd gr′*

Proof. The definition of *struct-wf-gr-utpd* (page 7) requires two properties to be shown. The first one follows from Lemma 2.2. By noting that *Field* $(R \cup S) =$ *Field* $R \cup$ *Field* S and *Field* (*restrict-rel* R A) $\subseteq A$, the second one easily reduces to the property shown by Lemma 2.3. □

[§] For technical reasons, there are two implication symbols in Isabelle, $\Longrightarrow$ and $\longrightarrow$. The difference is not significant for this article.

3. Typed transformations

Our framework is extensible, allowing for more complex notions of graphs. In this section, we show how to integrate typing information. In a similar fashion, one could add node or edge attributes.

There are several ways of defining type information. One of them consists in assigning a unique type to each node and to each edge. We have opted for the inverse. For this, we extend the notion of graph with two more attributes, namely *nodetp* which expresses which nodes belong to a given node type *'nt*, and *edgetp* which expresses which edges belong to a given edge type *'et*.

record $('obj, 'nt, 'et)$ *typed-graph* = *'obj graph* +
  nodetp :: $'nt \Rightarrow 'obj\ set$
  edgetp :: $'et \Rightarrow ('obj * 'obj)\ set$

Consequently, each node can have several types (and similarly for edges), which is useful for modeling the type system of object oriented languages. A minimalist notion of well-formedness of typed graphs extends well-formedness of untyped graphs (page 7) by requiring the typed nodes to be nodes of the underlying graph, and similarly for edges. Refinements of this typing discipline may be imposed for particular domain-specific or modeling languages such as UML or Ecore/EMF.

definition *struct-wf-gr-tpd* :: $('obj, 'nt, 'et, 'a)$ *typed-graph-scheme* $\Rightarrow$ *bool* **where**
  struct-wf-gr-tpd gr =
    (*struct-wf-gr-utpd gr* $\wedge$
    $(\forall\ nt.\ (nodetp\ gr\ nt) \subseteq (nodes\ gr)) \wedge\ (\forall\ et.\ (edgetp\ gr\ et) \subseteq (edges\ gr)))$

Different more evolved notions of subtyping (strict subtyping or with multiple inheritance) can be coded in the language of node sets and path formulas of Section 2.1, where "A is a subtype of B" is coded by a path formula whose expansion is $\forall x. x \in A \longrightarrow x \in B$. This gives a great deal of freedom, but the concept of type soundness of transformations is not native, and enforcing it gives rise to a proof obligation and is not verifiable by type checking as in classical programming languages.

In a similar spirit, typed graph transformations are an extension of untyped graph transformations, allowing to specify the node typing of generated nodes and typing of deleted respectively generated edges:

record $('nt, 'et)$ *typed-graphtrans* = $('nt, 'et)$ *graphtrans* +
  — typing of generated nodes
  ngentp :: $'nt \Rightarrow nat\ set$
  — mapping of edges
  edeltp :: $'et \Rightarrow (nat * nat)\ set$
  egentp :: $'et \Rightarrow (nat * nat)\ set$

In view of Theorem 3.1 to be proved below, we again impose restrictions on the well-formedness of the graph transformation, extending the definition of the untyped case:

definition
  struct-wf-gt-tpd :: $('nt, 'et, 'a)$ *typed-graphtrans-scheme* $\Rightarrow$ *bool* **where**

$$\mathit{struct\text{-}wf\text{-}gt\text{-}tpd}\ gt = (\mathit{struct\text{-}wf\text{-}gt\text{-}utpd}\ gt \wedge (\forall\ et.\ \mathit{egentp}\ gt\ et \subseteq \mathit{egen}\ gt))$$

The definition of application of typed graph transformation is similar in spirit to the definitions in the untyped case in Figure 7: we can modularly define the effect of a graph transformation on node and edge types. We skip the details of the definition.

We now have the necessary ingredients to define a new locale which instantiates the locale of graph transformations of page 13, where node type interpretation *I-nt* is instantiated to (*nodetp gr*), edge type interpretation *I-et* to (*edgetp gr*) and application of graph transformations *apply-graphtrans-rel* is the typed variant sketched above.

Under this interpretation, a graph morphism has to satisfy a given type constraint to be applicable. For example, the application condition $\lceil \ll 1,\ 2 \gg \rceil\ \lceil \wedge \rceil\ \lceil \ll 1,\ 2 \gg\ ::\ r \rceil$ expresses that there is an edge between node references *1* and *2* of type *r* (where *r* is of an appropriately defined edge type).

In this locale, we can prove an analogous result of preservation of well-formedness during graph transformations. It is a “type soundness” result which is not very strong in our case, due to the weak notion of well-typing:

Theorem 3.1.

$$\mathit{applicable\text{-}gm}\ gm\ (\mathit{appcond}\ gt) \wedge \mathit{morph\text{-}sep}\ gm\ (\mathit{ndel}\ gt) \wedge \mathit{apply\text{-}graphtrans\text{-}rel}\ gm\ gr' \Longrightarrow \mathit{struct\text{-}wf\text{-}gr\text{-}tpd}\ gr'$$

The reader should not be misled by the fact that the preconditions appear to be identical to those of Theorem 2.4: this is not so, the locale context is different.

4. Local reasoning

This section investigates methods to make reasoning about graph transformations more automatic. In particular, we will see under which conditions and how we can reduce reasoning about an abstract graph (with an unknown number of nodes) to reasoning about a portion with a bounded number of nodes. These nodes are typically the free variables of the transformation's application condition, or, more precisely, the image of these variables under a graph morphism.

In this article, we concentrate on the preservation of certain properties during graph transformations, of the form $r \subseteq s$ or, more interestingly, $r^+ \subseteq s^+$ or $r^* \subseteq s^*$. The latter two can be used to model *reachability* problems. Typical application scenarios are problems of connectivity in a communication network after a failure of a communication link, or memory leakage in pointer-manipulating programs due to the loss of pointers to a heap area. In this case, r is the edge relation before and s the edge relation after the transformation. Inversely, one can also model *separability*, for example to show the absence of an undesired information flow or the absence of aliasing during pointer manipulation. In this case, s is the edge relation before and r the edge relation after the transformation. In both cases, the procedure is similar and consists in applying the graph decompositions of Section 4.1, and then simplifying the exterior and interior according to the rules presented in Section 4.2. The inductive reasoning inherent to transitivity is "compiled" into the reduction technique, the remaining proof obligation can therefore be handled by Boolean satisfiability solving, even without computing any fixpoints.

Unfortunately, in the general case, even then the complexity remains very high. Arbitrary (non-injective) morphisms may identify any number of node references of the applicability condition and thus invalidate properties that hold for injective morphisms (see Example 1.3). For n node references in the applicability condition, we thus have to examine up to 2^n equivalence classes. In Section 5, we present reduction theorems that can avoid the combinatorial explosion in some situations, by converting a proof problem into a calculational problem, namely checking whether the transitive closures of one concrete graph is contained in that of another graph, which is of complexity $O(n^3)$.

4.1. *Graph decompositions*

Our method consists in splitting a graph into an interior of a node set A, and an exterior of A (the rest of the graph). This node set is typically the image of the free variables of the applicability condition of a given graph transformation. This notion is extended to edge relations r as follows: the edge belongs to the interior if both of its endpoints are in A, and otherwise to the exterior.

definition *interior-rel* $A\ r = r \cap (A \times A)$
definition *exterior-rel* $A\ r = r - (A \times A)$

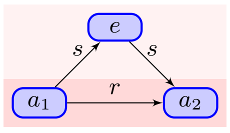

Figure 8: Interior (dark shade) and exterior (light shade) of a relation

An example is depicted in Figure 8, where edge r belongs to the interior of $A = \{a_1, a_2\}$ and the edges labeled s to the exterior.

Simple set-theoretic reasoning allows to prove the following lemmas:

Lemma 4.1. *interior-rel* $A\ r \cup$ *exterior-rel* $A\ r = r$

Lemma 4.2.
$A \subseteq A' \Longrightarrow r \subseteq r' \Longrightarrow$ *interior-rel* $A\ r \subseteq$ *interior-rel* $A'\ r'$
$A' \subseteq A \Longrightarrow r \subseteq r' \Longrightarrow$ *exterior-rel* $A\ r \subseteq$ *exterior-rel* $A'\ r'$

Note the anti-monotonicity in the first argument of *exterior-rel*.

Lemma 4.3. $((\textit{interior-rel}\ A\ r)^* \cup (\textit{exterior-rel}\ A\ r)^*)^* = r^*$

A similar lemma holds for "reflexive" instead of "reflexive-transitive" closure. For the proof of Lemma 4.3, use Lemma 4.1 and the property $(I^* \cup E^*)^* = (I \cup E)^*$ of reflexive-transitive closures.

The following two lemmas are at the heart of the decomposition method we propose.

When applied from right to left, they split up a goal into an exterior and an interior that can then be further simplified with the mechanisms described in Section 4.2.

Lemma 4.4.
$(\mathit{exterior\text{-}rel}\ A\ r \subseteq \mathit{exterior\text{-}rel}\ A\ s \wedge \mathit{interior\text{-}rel}\ A\ r \subseteq \mathit{interior\text{-}rel}\ A\ s) = (r \subseteq s)$

Proof. Expanding $r \subseteq s$ with Lemma 4.1 and then using monotonicity (Lemma 4.2). □

Lemma 4.5.
$(\mathit{exterior\text{-}rel}\ A\ r)^* \subseteq (\mathit{exterior\text{-}rel}\ A\ s)^* \wedge (\mathit{interior\text{-}rel}\ A\ r)^* \subseteq (\mathit{interior\text{-}rel}\ A\ s)^* \Longrightarrow r^* \subseteq s^*$

Proof. Expanding with Lemma 4.3 and then using monotonicity of reflexive-transitive closure. □

A similar lemma holds for transitive closure.

We see that splitting up a simple inclusion $r \subseteq s$ (by applying Lemma 4.4 from right to left) is an equality transformation, but this is not the case for splitting up $r^* \subseteq s^*$. The converse of Lemma 4.5 is not true in general, and indeed, the situation in Figure 8 provides a counterexample: we have $\mathit{interior\text{-}rel}\ A\ r = \{(a_1,\ a_2)\}$ and $\mathit{interior\text{-}rel}\ A\ s = \{\}$ and thus not $(\mathit{interior\text{-}rel}\ A\ r)^* \subseteq (\mathit{interior\text{-}rel}\ A\ s)^*$, even though $r^* \subseteq s^*$. Applying Lemma 4.4 also in the case $r^* \subseteq s^*$ is not useful, because we then cannot further simplify the resulting terms of the form $\mathit{interior\text{-}rel}\ A\ r^*$ in the style of Section 4.2.

However, in certain cases, the converse of Lemma 4.5 holds:

Lemma 4.6.
$\mathit{Field}\ s \subseteq A \wedge r^* \subseteq s^* \Longrightarrow (\mathit{interior\text{-}rel}\ A\ r)^* \subseteq (\mathit{interior\text{-}rel}\ A\ s)^*$
$\mathit{Field}\ r \subseteq A \wedge r^* \subseteq s^* \Longrightarrow (\mathit{exterior\text{-}rel}\ A\ r)^* \subseteq (\mathit{exterior\text{-}rel}\ A\ s)^*$

Proof. Note that $\mathit{Field}\ s \subseteq A \Longrightarrow \mathit{interior\text{-}rel}\ A\ s = s$ and $\mathit{interior\text{-}rel}\ A\ r \subseteq r$, and use transitivity of $\subseteq$. Similarly, the second property follows from $\mathit{Field}\ r \subseteq A \Longrightarrow \mathit{exterior\text{-}rel}\ A\ r = \emptyset$. □

The decomposition suggested by Lemma 4.5 is therefore sound and also complete for a class of graphs where the region A has been chosen large enough to comprise the fields of the relations r and s. In practice, we will choose A to be the largest set of nodes whose existence is ascertained in the actual proof goal.

Example 4.1. Consider the transformation visualized in Figure 9, which has a disjunctive applicability condition. The figure has to be interpreted as follows: arcs with the same number have to be present (uninterrupted black) or must not be present (dashed red) at the same time. Thus, there must be edges (r_1, r_2) and (r_2, r_4) and not (r_1, r_3), or (r_1, r_3) and (r_3, r_4) and not (r_1, r_2). In either case, we add the edge (r_1, r_4). The existing edges are maintained, but that would be difficult to visualize. The textual definition in Figure 10 is more precise.

To see how proofs are carried out in our framework, let us show that this rule does not create any new paths, *i.e.* that the reflexive-transitive closure of the edge relation of

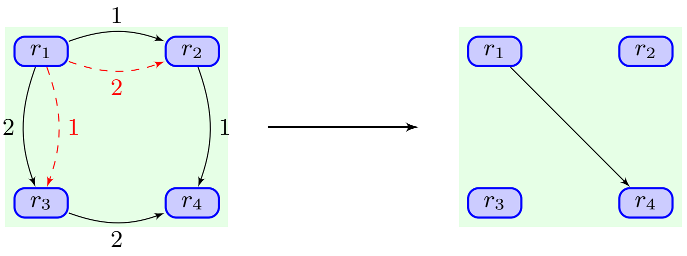

Figure 9: Rewriting rule with disjunctive condition

---

definition *disj-condition* **where** *disj-condition* =
  (| *appcond* = ⌈{*1*, *2*, *3*, *4*}⌉ ⌈∧⌉
        (⌈≪*1*, *2*≫⌉ ⌈∧⌉ ⌈≪*2*, *4*≫⌉ ⌈∧⌉ ⌈¬⌉ ⌈≪*1*, *3*≫⌉)
         ⌈∨⌉ (⌈≪*1*, *3*≫⌉ ⌈∧⌉ ⌈≪*3*, *4*≫⌉ ⌈∧⌉ ⌈¬⌉ ⌈≪*1*, *2*≫⌉),
   ndel = {},   *ngen* = {},
   edel = {},   *egen* = {(*1*,*4*)} |)

Figure 10: Definition of transformation with disjunctive condition

the modified graph is contained in the closure of the original graph's edge relation. We state the proof goal:

$$\textit{applicable-transfo} \wedge \textit{apply-transfo-rel}\ \textit{gr}' \Longrightarrow (\textit{edges}\ \textit{gr}')^* \subseteq (\textit{edges}\ \textit{gr})^*$$

in a locale that is an extension of the graph transformation locale of Section 2.2 and where the constant *gt* has been instantiated to the graph transformation under investigation, in this case *disj-condition*. Note that the locale also fixes an arbitrary graph *gr*, and both *applicable-transfo* and *apply-transfo-rel* implicitly refer to *gr* and *gt*. We could further instantiate the morphism property, *e.g.* to injective morphisms, but we have not done so, therefore *morph-prop* remains uninterpreted. After expansion of definitions and some tidying of the proof state, we get the goal:

$$\begin{array}{l}
[\![n1 \in \textit{nodes}\ gr;\ n2 \in \textit{nodes}\ gr;\ n3 \in \textit{nodes}\ gr;\ n4 \in \textit{nodes}\ gr; \\
\ \textit{morph-prop}\ [4 \mapsto n4,\ 3 \mapsto n3,\ 2 \mapsto n2,\ 1 \mapsto n1]; \\
\ (n1, n2) \in \textit{edges}\ gr \wedge (n2, n4) \in \textit{edges}\ gr \wedge (n1, n3) \notin \textit{edges}\ gr \vee \\
\ (n1, n3) \in \textit{edges}\ gr \wedge (n3, n4) \in \textit{edges}\ gr \wedge (n1, n2) \notin \textit{edges}\ gr; \\
\ \textit{nodes}\ gr' = \textit{nodes}\ gr;\ \textit{edges}\ gr' = \{(n1, n4)\} \cup \textit{edges}\ gr]\!] \\
\Longrightarrow (\{(n1, n4)\} \cup \textit{edges}\ gr)^* \subseteq (\textit{edges}\ gr)^*
\end{array}$$

The preconditions of the goal are contained in [[...]], the conclusion behind ⟹. Application of Lemma 4.5 gives two subgoals, the first of which has the same premisses as above and the conclusion:

$$\begin{array}{l}
(\textit{exterior-rel}\ \{n1, n2, n3, n4\}\ (\{(n1, n4)\} \cup \textit{edges}\ gr))^* \\
\subseteq (\textit{exterior-rel}\ \{n1, n2, n3, n4\}\ (\textit{edges}\ gr))^*
\end{array}$$

and similar for *interior-rel*. The rest of the proof can now proceed along the lines described in the following.

4.2. *Reductions to Boolean satisfiability problems*

The decompositions of Section 4.1 leave behind goals which are of the form *exterior-rel* $A\ r \subseteq$ *exterior-rel* $A\ s$ and *interior-rel* $A\ r \subseteq$ *interior-rel* $A\ s$ or variants with (reflexive) transitive closures. As seen in Example 1.2, we cannot deal with arbitrary relational expressions. In particular, relation composition O poses a problem, because $(x, y) \in (rOs)$ means $\exists z.(x, z) \in r \wedge (z, y) \in s$ and composition thus implicitly involves existential quantification over a node that may lie outside the region under consideration. We therefore limit ourselves to relational expressions r built up inductively according to the following grammar:

$$\begin{array}{rcl} r & ::= & r_b \\ & | & \{\} \\ & | & \mathit{insert}\ (n_1, n_2)\ r \\ & | & r \cup r \\ & | & r \cap r \\ & | & r - r \end{array}$$

Here, n_1, n_2 are variables representing node names, and r_b are basic, non-interpreted relations. In the case of untyped graphs, there is only one such relation, the *edges* relation. In the case of typed graphs, there are as many basic relations as there are edge types. We use $FV(r)$ for the set of free variables occurring in r. Please note that these are expressions of our proof assistant's object logic which are not immediately related to the path expressions and path formulas of Section 2.1.

4.2.1. *Simplification of exterior-rel*

In the following, we show how we can decide inclusions *exterior-rel* $A\ r \subseteq$ *exterior-rel* $A\ s$, by proving them or generating a counter-model.

The simplification mentioned below consists in applying exhaustively the rewrite rules summarized in the following lemma.

Lemma 4.7.
$\mathit{exterior\text{-}rel}\ A\ \{\} = \{\}$
$\mathit{exterior\text{-}rel}\ A\ (\mathit{insert}\ (x,\ y)\ r) = (\{(x,\ y)\} - (A \times A)) \cup\ \mathit{exterior\text{-}rel}\ A\ r$
$\mathit{exterior\text{-}rel}\ A\ (r \cap s) = (\mathit{exterior\text{-}rel}\ A\ r) \cap (\mathit{exterior\text{-}rel}\ A\ s)$
$\mathit{exterior\text{-}rel}\ A\ (r \cup s) = (\mathit{exterior\text{-}rel}\ A\ r) \cup (\mathit{exterior\text{-}rel}\ A\ s)$
$\mathit{exterior\text{-}rel}\ A\ (r - s) = (\mathit{exterior\text{-}rel}\ A\ r) - (\mathit{exterior\text{-}rel}\ A\ s)$

We further apply simple set-theoretic simplifications, in particular simplifications of $\{(x,\ y)\} - (A \times A)$ to $\{\}$ if $x,\ y \in A$, and subsequent elimination of the empty set. Note that A is a concrete enumeration of elements and always permits such a simplification if A has been appropriately chosen with $FV(r) \subseteq A$.

If the original goal was of the form *exterior-rel* $A\ r \subseteq$ *exterior-rel* $A\ s$, after this simplification, we are left with a goal $R \subseteq S$ where R and S are combinations of the operators $\cup, \cap, -$ applied to basic expressions of the form *exterior-rel* $A\ r_b$. This goal can be abstracted to a validity problem of classical two-valued Boolean algebra. For example, $(\textit{exterior-rel}\ A\ r_b) \cup (\textit{exterior-rel}\ A\ s_b) \subseteq (\textit{exterior-rel}\ A\ r_b) - (\textit{exterior-rel}\ A\ s_b)$ is abstracted to $r \vee s \longrightarrow r \wedge \neg\ s$. Such a Boolean formula is either valid (in this case, we can obtain a set-theoretic proof of the set inclusion problem) or admits a counter-model. In this case, we interpret *exterior-rel* $A\ r_b$ by a fixed singleton set (say $\{(x,\ y)\}$ for $x \neq y$ and $x,\ y \notin A$) if the corresponding propositional variables is interpreted as true, and otherwise by the empty set. This is then a counter-model for the inclusion $R \subseteq S$. In our example, the Boolean counter-model r=*false*, s=*true* would give the set-theoretic counter-example $\{\} \cup (\textit{exterior-rel}\ A\ \{(x,\ y)\}) \subseteq \{\} - (\textit{exterior-rel}\ A\ \{(x,\ y)\})$.

A goal of the form $(\textit{exterior-rel}\ A\ r)^+ \subseteq (\textit{exterior-rel}\ A\ s)^+$ is simplified to a goal of the form $R^+ \subseteq S^+$, with R, S as in the above paragraph. We first try to solve the goal $R \subseteq S$. If this goal is provable, then so is the original goal, by monotonicity of transitive closure. If it is not provable, the counter-model constructed according to the above procedure is also a counter-model of $R^+ \subseteq S^+$, because a singleton relation is the same as its transitive closure.

Goals of the form $(\textit{exterior-rel}\ A\ r)^* \subseteq (\textit{exterior-rel}\ A\ s)^*$ are similarly reduced to goals of the form $R^* \subseteq S^*$. We note that we can rewrite $R^* = \textit{Id} \cup R^+$, where *Id* is the identity relation. Simple set-theoretic manipulations then further reduce $\textit{Id} \cup R^+ \subseteq \textit{Id} \cup S^+$ to two subgoals $R^+ \subseteq \textit{Id}$ and $R^+ \subseteq S^+$. The second one is dealt with as above, whereas $R^+ \subseteq \textit{Id}$ holds whenever R is equivalent to the empty set, and otherwise can be refuted by a singleton counter-model as above.

Even though the reduction to Boolean validity makes the problem *co-NP* hard, this is not a limiting factor in practice, because the number of relation symbols is typically small.

4.2.2. *Simplification of interior-rel* In the following, we show how we can decide inclusions $\textit{interior-rel}\ A\ r \subseteq \textit{interior-rel}\ A\ s$ or variants with (reflexive) transitive closure. We have a similar set of rewrite rules as in Section 4.2.1, only the case for *insert* is different:

Lemma 4.8.

$\textit{interior-rel}\ A\ \{\} = \{\}$
$\textit{interior-rel}\ A\ (\textit{insert}\ (x,\ y)\ r) = (\{(x,\ y)\} \cap (A \times A)) \cup (\textit{interior-rel}\ A\ r)$
$\textit{interior-rel}\ A\ (r \cap s) = (\textit{interior-rel}\ A\ r) \cap (\textit{interior-rel}\ A\ s)$
$\textit{interior-rel}\ A\ (r \cup s) = (\textit{interior-rel}\ A\ r) \cup (\textit{interior-rel}\ A\ s)$
$\textit{interior-rel}\ A\ (r - s) = (\textit{interior-rel}\ A\ r) - (\textit{interior-rel}\ A\ s)$

For A with $FV(r) \subseteq A$, $(\{(x,\ y)\} \cap (A \times A))$ reduces to $\{(x,\ y)\}$ if $x \in A$ and $y \in A$, and to the empty set otherwise.

After exhaustive application of these rewrite rules, we are left with a goal of the form $R \subseteq S$ or $R^+ \subseteq S^+$ or $R^* \subseteq S^*$, where R and S are combinations of operators $\cup, \cap, -$ applied to basic expressions which have the form *interior-rel* $A\ r_b$ or $\{(x,\ y)\}$, with x,$y \in A$.

Please note that the set $\{I\ .\ \exists\ r.\ I = \textit{interior-rel}\ A\ r\}$ is just $\textit{Pow}\ (A \times A)$, the powerset of $A \times A$, where A is a finite enumeration of elements. In principle, our method amounts to trying out whether one of the $2^{(|A|^2)}$ possible combinations of $I_b \in Pow\ (A \times A)$, for the basic relations r_b, might provide a solution of the problem $R \subseteq S$. We do not in fact use this naive method, as explained in the following, and thus can perform some intermediate simplifications, but the complexity remains exponential.

We first define the following auxiliary functions:

definition $\mathit{interior\text{-}rel\text{-}elem\text{-}r}\ a\ B\ r = r \cap (\{a\} \times B)$
definition $\mathit{interior\text{-}rel\text{-}elem\text{-}l}\ b\ A\ r = r \cap (A \times \{b\})$

They can be completely eliminated by recursing over the sets B resp. A:

Lemma 4.9.

$\mathit{interior\text{-}rel\text{-}elem\text{-}r}\ a\ \{\}\ r = \{\}$

$\mathit{interior\text{-}rel\text{-}elem\text{-}r}\ a\ (\mathit{insert}\ b\ B)\ r =$
$((\mathit{if}\ (a,\ b) \in r\ \mathit{then}\ \{(a,\ b)\}\ \mathit{else}\ \{\}) \cup \mathit{interior\text{-}rel\text{-}elem\text{-}r}\ a\ B\ r)$

$\mathit{interior\text{-}rel\text{-}elem\text{-}l}\ b\ \{\}\ r = \{\}$

$\mathit{interior\text{-}rel\text{-}elem\text{-}l}\ b\ (\mathit{insert}\ a\ A)\ r =$
$((\mathit{if}\ (a,\ b) \in r\ \mathit{then}\ \{(a,\ b)\}\ \mathit{else}\ \{\}) \cup \mathit{interior\text{-}rel\text{-}elem\text{-}l}\ b\ A\ r)$

This way, we can also eliminate *interior-rel*, by using the following simplifications:

Lemma 4.10.

$\mathit{interior\text{-}rel}\ \{\}\ r = \{\}$

$\mathit{interior\text{-}rel}\ (\mathit{insert}\ a\ A)\ r =$
$(\mathit{interior\text{-}rel}\ A\ r) \cup (\mathit{interior\text{-}rel\text{-}elem\text{-}r}\ a\ (\mathit{insert}\ a\ A)\ r) \cup (\mathit{interior\text{-}rel\text{-}elem\text{-}l}\ a\ (\mathit{insert}\ a\ A)\ r)$

After exhaustive simplification, one has to perform up to $2^{(|A|^2)}$ splits for the cases $(a,\ b) \in r$, to obtain pure set containment problems, *i.e.* combinations of operators $\cup, \cap, -$ and (reflexive) transitive closure, applied to expressions of the form $\{(x,\ y)\}$.

5. Calculational proofs

We now come back to one of the initial questions of our investigation: is it possible to reason about a graph transformation just by looking at the shape of a graph transformation rule (*i.e.* its left and right hand side), without considering mappings into target graphs via morphisms? As explained at the beginning of Section 4, the question is also intimately related to the complexity of reasoning, because having to consider multiple matchings of the rule pattern into the target graphs can induce a combinatorial explosion.

As illustrated by Example 1.3, reasoning only about the shape of the rules can be fallacious for non-injective morphisms.

Example 5.1. We now reconsider this example. The transformation is formally defined in Figure 11. The methods described in Section 4.2 at least permit to identify the problem,

definition *delete-edge* **where**
$\quad$ *delete-edge* =
$\quad (\!|$ *appcond* $= \lceil\{1, 2, 3\}\rceil \lceil\wedge\rceil \lceil\ll 1, 2\gg\rceil \lceil\wedge\rceil \lceil\ll 1, 3\gg\rceil \lceil\wedge\rceil \lceil\ll 2, 3\gg\rceil$,
$\quad\quad$ *ndel* $= \{\}$, *ngen* $= \{\}$,
$\quad\quad$ *edel* $= \{(1,3)\}$, *egen* $= \{\}$ $|\!)$

Figure 11: Deletion of an edge

even if they do not allow to complete the proof. After case splitting, the prover gets stuck on the following subgoal, which corresponds to the configuration depicted in Figure 3:

$$\begin{aligned}
&[\![n1 \in nodes\ gr;\ n3 \in nodes\ gr;\ morph\text{-}prop\ [3 \mapsto n3,\ 2 \mapsto n1,\ 1 \mapsto n1];\\
&\ nodes\ gr' = nodes\ gr;\ edges\ gr' = edges\ gr - \{(n1, n3)\};\\
&\ (n1, n3) \in edges\ gr;\ n1 \neq n3;\ (n1, n1) \in edges\ gr;\ (n3, n1) \in edges\ gr;\\
&\ (n3, n3) \in edges\ gr;\ n2 = n1]\!]\\
&\Longrightarrow \{(n1, n3), (n1, n1), (n3, n1), (n3, n3)\}^+\\
&\qquad \subseteq Id \cup \{(n1, n1), (n3, n1), (n3, n3)\}^+
\end{aligned}$$

Because $(n1, n3)$ is not contained in $Id \cup \{(n1, n1), (n3, n1), (n3, n3)\}^+$, this goal is unsolvable.

We can nevertheless make some progress on this question. The main idea of the following argument is that reflexive-transitive closure remains stable under a context, the C in the following lemma:

Lemma 5.1. $A' \subseteq A \cup C \wedge B \cup C \subseteq B' \wedge A^* \subseteq B^* \Longrightarrow A'^* \subseteq B'^*$

Proof. From $A' \subseteq A \cup C$, one can infer $A'^* \subseteq (A \cup C)^*$ by monotonicity of reflexive-transitive closure. One similarly obtains $(B \cup C)^* \subseteq B'^*$ from $B \cup C \subseteq B'$. From $A^* \subseteq B^*$, one obtains $A^* \cup C^* \subseteq B^* \cup C^*$ and thus also $(A^* \cup C^*)^* \subseteq (B^* \cup C^*)^*$ (using transitivity of reflexive-transitive closure). We further simplify $(A^* \cup C^*)^*$ to $(A \cup C)^*$ (and similarly for $(B^* \cup C^*)^*$), which gives us the desired conclusion by transitivity. □

Roughly speaking, the sets A and B are the images of the left and right side of the rule under the graph morphism, whereas C is the rest of the graph. We will apply the lemma in the inverse direction, reducing a goal of the form $A'^* \subseteq B'^*$ to $A^* \subseteq B^*$. The following lemma shows that this reduction is complete, provided the context C can become empty:

Lemma 5.2. $(A^* \subseteq B^*) = (\forall\, C\ A'\ B'.\ A' \subseteq A \cup C \longrightarrow B \cup C \subseteq B' \longrightarrow A'^* \subseteq B'^*)$

We will now instantiate these lemmas for the case of graph transformations. Let us first state some auxiliary lemmas needed in the proofs of the theorems below.

Lemma 5.3. $A^* \subseteq B^* \Longrightarrow ((emorph\ gm)\ `\ A)^* \subseteq ((emorph\ gm)\ `\ B)^*$

Proof. By induction on reflexive-transitive closure. □

The theorems proved below differ as to whether we are dealing with injective or non-injective morphisms. Theorem 5.4 is embedded in a locale that assumes that gr is a well-formed graph and *gt* a well-formed graph transformation. No assumption is made about properties of the graph morphism. However, a precondition of the theorem is that the transformation does not modify the node set, *i.e.* the sets of nodes to be deleted and generated are empty. This might appear to be a decisive restriction, but this is not so: Let us only remark here (also see Section 6) that any graph transformation can be represented as a sequence of two kinds of transformations, the first of which modifies only the edge set and the second of which only the node set (and is consequently without effect on the edge relation).

Theorem 5.4. (assuming *gr* is a well-formed graph and *gt* a well-formed graph transformation)

$$\begin{aligned}&\mathit{ndel}\ \mathit{gt} = \{\} \wedge \mathit{ngen}\ \mathit{gt} = \{\} \wedge \mathit{applicable\text{-}transfo} \wedge \mathit{apply\text{-}transfo\text{-}rel}\ \mathit{gr'} \wedge\\&(\mathit{edel}\ \mathit{gt})^* \subseteq (\mathit{egen}\ \mathit{gt})^*\\&\Longrightarrow\ (\mathit{edges}\ \mathit{gr})^* \subseteq (\mathit{edges}\ \mathit{gr'})^*\end{aligned}$$

Proof. After expansion of the definitions of *applicable-transfo* and *apply-transfo-rel*, one sees that $\mathit{edges}\ \mathit{gr'} = \mathit{edges}\ \mathit{gr} - \mathit{emorph}\ \mathit{gm}\ `\ \mathit{edel}\ \mathit{gt} \cup \mathit{emorph}\ \mathit{gm}\ `\ \mathit{egen}\ \mathit{gt}$. Let us apply Lemma 5.1 backwards, with $A{=}\mathit{emorph}\ \mathit{gm}\ `\ (\mathit{edel}\ \mathit{gt})$ and $B{=}\mathit{emorph}\ \mathit{gm}\ `\ (\mathit{egen}\ \mathit{gt})$ and $C = (\mathit{edges}\ \mathit{gr}) - (\mathit{emorph}\ \mathit{gm}\ `\ (\mathit{edel}\ \mathit{gt}))$. Since we have $(\mathit{edel}\ \mathit{gt})^* \subseteq (\mathit{egen}\ \mathit{gt})^*$, the precondition $(\mathit{emorph}\ \mathit{gm}\ `\ \mathit{edel}\ \mathit{gt})^* \subseteq (\mathit{emorph}\ \mathit{gm}\ `\ \mathit{egen}\ \mathit{gt})^*$ is an instance of Lemma 5.3, the remaining set inclusions are easy to show. □

Clearly, this theorem is not applicable to Example 1.3, but the correctness of the transformation of Example 1.1 can now be proved.

Result 5.5. (for an arbitrary graph *gr*, where graph transformation *gt* is *rerouting* of Example 1.1)

$$\mathit{applicable\text{-}transfo} \wedge (\mathit{apply\text{-}transfo\text{-}rel}\ \mathit{gr'}) \Longrightarrow (\mathit{edges}\ \mathit{gr})^* \subseteq (\mathit{edges}\ \mathit{gr'})^*$$

Proof. We apply Theorem 5.4. Apart from the trivial preconditions, we have to show $\{(\mathit{1},\ \mathit{3})\}^* \subseteq \{(\mathit{1},\ \mathit{2}),\ (\mathit{2},\ \mathit{3})\}^*$, which is a simple calculational proof that simulates the behavior of a corresponding functional program. Note that we directly deal with the node references occurring in the transformation rule and not with nodes after mapping these references into a graph. □

This result is independent of a particular morphism (and not only valid for an injective morphism as in Figure 1).

In order to deal with situations like Example 1.3, we now develop a reduction that is more expressive, but has more restrictive preconditions. The following Theorem 5.7 is valid for injective morphisms, and intended to be used with a class of graph rewriting rules whose applicability conditions are a conjunction of positive and negative edge relations. We are thus dealing with a classical form of graph rewriting with graph patterns and negative application conditions, as for example in Ehrig et al. (1997). More complex Boolean applicability conditions, in particular disjunction, are not in the scope of this reduction. Technically speaking, the reduction remains applicable even in this case, but one loses completeness.

The function *edge-set-path-form* gathers the edges of a graph that is described by such a restricted form of path formula:

fun *edge-set-path* :: *'et path* ⇒ (*nat* ∗ *nat*) *set* **where**
  edge-set-path (≪*n1*, *n2*≫) = {(*n1*, *n2*)}
| *edge-set-path* - = {}

fun *edge-set-path-form* :: (*'nt*, *'et*) *path-form* ⇒ (*nat* ∗ *nat*) *set* **where**
  edge-set-path-form (*n* ⇝ *n'* ⌈∈⌉ *p*) = {(*n*, *n'*)} ∩ (*edge-set-path p*)
| *edge-set-path-form* (*pf* ⌈∧⌉ *pf'*) = (*edge-set-path-form pf* ∪ *edge-set-path-form pf'*)
| *edge-set-path-form* - = {}

We have the following relation between *edge-set-path-form* and path interpretations:

Lemma 5.6.
path-form-interp (*the* ∘ *gm*) *pf* ⟶ *emorph gm* ‘ *edge-set-path-form pf* ⊆ *edges gr*

Proof. By induction over *pf*. □

The following theorem is embedded in a locale that assumes that graph morphisms are injective.

Theorem 5.7. (assuming *gr* is a well-formed graph, *gt* a well-formed graph transformation, graph morphisms are injective)

ndel gt ={} ∧ *ngen gt* ={} ∧ *applicable-transfo* ∧ (*apply-transfo-rel gr'*) ∧
(*edge-set-path-form* (*appcond gt*) ∪ (*edel gt*))* ⊆
  ((*edge-set-path-form* (*appcond gt*) − *edel gt*) ∪ (*egen gt*))*
⟹ (*edges gr*)* ⊆ (*edges gr'*)*

Proof. Let us note first of all that function image distributes in general over set union (thus f ‘ $(A \cup B) = f$ ‘ $A \cup f$ ‘ B), but over set difference (thus f ‘ $(A - B) = f$ ‘ $A - f$ ‘ B) only if f is injective over a common superset of A and B.

To prove the theorem, we again expand the definitions of the applicability conditions and then apply Lemma 5.1 backwards, with A=*emorph gm* ‘ (*edge-set-path-form* (*appcond gt*) ∪ *edel gt*) and B=*emorph gm* ‘ (*edge-set-path-form* (*appcond gt*) − *edel gt* ∪ *egen gt*) and C=(*edges gr*) − (*emorph gm* ‘ (*edel gt*)). We now have to satisfy instances of the three preconditions of Lemma 5.1.

The first is (*emorph gm* ‘ (*edge-set-path-form* (*appcond gt*) ∪ *edel gt*))* ⊆ (*emorph gm* ‘ (*edge-set-path-form* (*appcond gt*) − *edel gt* ∪ *egen gt*))*, which can be solved with Lemma 5.3.

The second one is *edges gr* ⊆ *emorph gm* ‘ (*edge-set-path-form* (*appcond gt*) ∪ *edel gt*) ∪ (*edges gr* − *emorph gm* ‘ *edel gt*), a trivial inclusion.

The third precondition is *emorph gm* ‘ (*edge-set-path-form* (*appcond gt*) − *edel gt* ∪ *egen gt*) ∪ (*edges gr* − *emorph gm* ‘ *edel gt*) ⊆ *edges gr* − *emorph gm* ‘ *edel gt* ∪ *emorph gm* ‘ *egen gt*. With injectivity of *gm* and thus of *emorph gm*, we may distribute the function image *emorph gm* over union and set difference. Since *appcond gt* is satisfied for the morphism *gm*, we obtain that *emorph gm* ‘ *edge-set-path-form* (*appcond gt*) $\subseteq$ *edges gr* with Lemma 5.6 and thus are left again with a simple set-theoretic inclusion. □

The reader may wonder why the negative application conditions can be neglected in function *edge-set-path-form* and, consequently, in the theorem, but indeed, the non-existence of an edge does not have an impact on the preservation of paths in a graph.

Result 5.8. We take up Example 1.3. In a locale of injective graph morphisms, for graph transformation *delete-edge*, we now show

applicable-transfo $\wedge$ (*apply-transfo-rel gr'*) $\Longrightarrow$ (*edges gr*)* $\subseteq$ (*edges gr'*)*

Application of Theorem 5.7 gives the subgoal $(\{(1, 2), (1, 3), (2, 3)\} \cup \{(1, 3)\})^* \subseteq ((\{(1, 2), (1, 3), (2, 3)\} - \{(1, 3)\}) \cup \{\})^*$, which is again easily provable by calculation.

6. Conclusions

This article has investigated a formalization of graph transformations in a proof assistant, with the purpose of being able to prove properties of specific transformations interactively, but also to derive meta-results involving entire classes of graph transformations, as in Sections 4 and 5, which paves the way for a higher degree of automation.

Several extensions have to be envisaged. The first concerns a combination of individual graph transformation steps to graph transformation programs, with the aim of reasoning about them with a weakest-precondition style like about traditional imperative programs, thus integrating more recent developments, such as Poskitt and Plump (2012), into our previous work (Strecker, 2008). In this context, it might also be interesting to revisit some results that have so far been stated in an algebraic setting, such as confluence of transformations, and to re-interpret them in a logical context, which might allow to take more expressive applicability conditions into account. Inversely, it might be interesting to split up a complex transformation into a semantically equivalent sequence of more elementary steps (such as an edge- and node-manipulating part, as advocated for the theorems of Section 5).

A further question concerns the complexity of graph transformations. The discussion at the end of Section 4 seems to indicate that reasoning about transformations with non-injective morphisms is exponentially more complex than dealing with injective morphisms. However, in special cases, as Theorem 5.4, there is no penalty for non-injective morphisms.

Finally, we would like to generalize the results of Section 5. An essential ingredient of the proofs of Theorems 5.4 and 5.7 is the monotonicity of transitive closure that allows an embedding of the image of a morphism in a larger context to be preserved. We will attempt to generalize this result to (a combination of) other monotone operators.

References

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. URL http://www.mathematik.uni-marburg.de/~swt/Publikationen_Taentzer/Papiere06-09/ABJKT10.pdf.

Mark Asztalos, Laszlo Lengyel, and Tihamer Levendovszky. Towards automated, formal verification of model transformations. *International Conference on Software Testing, Verification, and Validation*, pages 15–24, 2010.

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. ISBN 978-3-642-15927-5. URL http://dx.doi.org/10.1007/978-3-642-15928-2_5.

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).

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. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.155.4078&rep=rep1&type=pdf.

Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-22164-7. URL http://www21.in.tum.de/~ballarin/publications/types2003.pdf.

Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. A decidable fragment of separation logic. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 110–117. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-24058-7. URL http://dx.doi.org/10.1007/978-3-540-30538-5_9.

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. URL http://hal.archives-ouvertes.fr/hal-00522586. 15 pages.

Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic, a language theoretic approach*. Cambridge University Press, 2011. URL http://www.labri.fr/perso/courcell/Book/TheBook.pdf.

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. ISSN 1571-0661. URL http://www.sciencedirect.com/science/article/pii/S1571066109001662. Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008).

Simone André da Costa and Leila Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming*, 77(4):480 – 504, 2012. ISSN 0167-6423. URL http://www.sciencedirect.com/science/article/pii/S016764231000033X. Brazilian Symposium on Formal Methods (SBMF 2008).

Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part II: Single pushout approach and comparison with double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars*, pages 247–312. World Scientific, 1997. ISBN 9810228848.

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. ISSN 1433-2779. URL http://dx.doi.org/10.1007/s10009-011-0186-x.

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. URL http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/mscs-HP09.pdf.

Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations (ICGT), Natal, Brazil*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460, Berlin, September 2006. Springer Verlag. ISBN 3-540-38870-2.

Haruo Hosoya. *XML processing - The Tree-Automata Approach.* Cambridge University Press, 2011.

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. ISBN 978-3-540-23024-3. URL http://www.cs.umass.edu/~immerman/pub/cslPaper.pdf.

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. ISBN 978-3-540-27231-1. URL http://www.cs.berkeley.edu/~necula/Papers/verifier-cav05.pdf.

Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.

Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 349–366. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64987-8. URL http://dx.doi.org/10.1007/BFb0055146.

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. URL http://isabelle.in.tum.de.

Fernando Orejas, Hartmut Ehrig, and Ulrike Prange. Reasoning with graph constraints. *Formal Aspects of Computing*, 22:385–422, 2010. ISSN 0934-5043. URL http://dx.doi.org/10.1007/s00165-009-0116-9.

Karl-Heinz Pennemann. An algorithm for approximating the satisfiability problem of high-level conditions. In *Proc. Graph Transformation for Verification and Concurrency (GT-VC'07)*, volume 213 of *Electronic Notes in Theoretical Computer Science*, pages 75–94. Elsevier, 2008a. URL http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/seeksat.pdf.

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, 2008b. ISBN 978-3-540-87404-1. URL http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/procon.pdf.

Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012. URL http://www.cs.york.ac.uk/plasma/publications/pdf/PoskittPlump.FundInf.12.pdf.

Arend Rensink. The joys of graph transformation. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 9, 2005. URL http://eprints.eemcs.utwente.nl/1443/.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science, Symposium on*, 0:55, 2002. ISSN 1043-6871.

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).

Martin Strecker. Modeling and verifying graph transformations in proof assistants. In Ian Mackie and Detlef Plump, editors, *International Workshop on Computing with Terms and Graphs (TERMGRAPH)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 135–148. Elsevier Science, 2008. URL http://www.irit.fr/~Martin.Strecker/Publications/termgraph07.html.

Martin Strecker. Locality in reasoning about graph transformations. In Dániel Varró Gergely Varró, Andy Schürr, editor, *Pre-proceedings conf. AGTIVE*, Budapest, 2011. URL http://www.irit.fr/~Martin.Strecker/Publications/agtive11.html.

Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):pp. 73–89, 1941. ISSN 00224812. URL http://www.jstor.org/stable/2268577.

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.

Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.

Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214 – 234, 2007. ISSN 0167-

6423. URL http://www.sciencedirect.com/science/article/B6V17-4P47GBW-1/2/3ccc0f0270a5cc6a792aa3320cc65689. Special Issue on Model Transformation.

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. URL http://dx.doi.org/10.1016/j.jlap.2006.12.001.

Eduardo Zambon and Arend Rensink. Using graph transformations and graph abstractions for software verification. *Electronic Communications of the EASST*, 38, August 2011. ISSN 1863-2122. URL http://journal.ub.tu-berlin.de/eceasst/article/view/560.

Index