Towards a Formalisation of Graph Transformations in Proof Assistants

Martin Strecker[1] Mathieu Giorgino[2]

IRIT
Université Paul Sabatier
118 route de Narbonne
F-31062 Toulouse

Abstract

This paper takes first steps towards a formalization of graph transformations in a general setting of interactive theorem provers, which will form the basis for proofs of correctness of graph transformation systems. We present parts of our formalization and take a glimpse at some strategies for simplifying proof obligations.

Keywords: Graph Transformations, Theorem Proving

1 Introduction

Graph transformations have been in use for quite a while. They have a well-established theory, as witnessed for example by algebraic [Bar03] and categorical [CMR+96,EHK+97] approaches. They are well supported by a growing number of tools, some of which [Tae03,KS06,Agr04] aim at a more or less faithful representation of the theory – we will come back to them below. Other approaches, more pragmatic, have surged up recently in the context of "model driven architectures". Here, the idea is to specify a software or hardware artifact graphically, to apply transformations to refactor the model and eventually to generate executable code. Transformation languages such as ATL [BBDV03] and Kermeta [MFV+05] have an algorithmic flavor in that they organize a traversal of a graph, applying user-supplied transformation rules.

In spite of a large body of work on graph transformations, the question of verification of transformations "in general" is far from settled. Graph transformation systems are often perceived as extensions of term rewriting systems, so much of the effort has gone into investigating specific properties such as confluence and termination [Plu99], which does not necessarily allow to determine whether a graph has

[1] Email: strecker@irit.fr
[2] Email: giorgino@irit.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

a certain shape after transformation. These questions may be answered for graph replacement systems having a restricted structure [FM97], for properties expressed in specialized logics such as monadic second order logic [KS93] or type systems [BCE+05].

However, in some circumstances, it is useful to resort to a more general setting, in order to express stronger properties or to overcome limitations of a restricted rule format. This gives us the same kind of advantage a program logic may have over a static analysis for determining the correctness of an imperative program – and it suffers from the same drawbacks, notably a sometimes heavy user intervention to carry out interactive proofs.

The present article describes first steps towards a formal model of graph transformations, with

- a formalization of graph transformations in the proof assistant Isabelle [NPW02]
- the aim to obtain a program logic and support for reasoning about graph transformations

In Section 2, we give an example of a graph transformation, we sketch our formal model of graphs and transformations in Section 3, then describe first attempts at simplifying proof obligations in Section 4 and conclude in Section 5 with a perspective on future work.

2 Graph Transformations

To give an intuition of our approach, we present a simple graph transformation, which duplicates a given graph. The transformation roughly applies the following rules, in the given order:

(i) Mark all the nodes of the original graph.

(ii) For all nodes marked as original, create a duplicate node and an auxiliary *copy* edge between the original and the duplicate.

(iii) For two original nodes related by an edge, create an edge between their duplicates, and mark the edge as duplicated.

(iv) Once all edges are duplicated, reset the *duplicated edge* mark.

(v) Delete the *copy* edges and the node marks of the original graph.

For experimenting with graph transformations and simulating their execution, we use the AGG tool [Tae03]. We also borrow some of their concepts, though in our formalization in Section 3, we depart significantly from the underlying theory. Graphs have to obey a certain typing discipline. In our example, we have node types `Node` and `Orig` (the latter serving as markers) and, among others, edge types `E` (for edges between `Node`s), `Or` (between `Node` and `Orig`) and `Cp` (between an original `Node` and its duplicate).

Figure 1 shows the first rule of our transformation, expressing that if there is a `Node` which has not been marked yet (i.e. there is no `Or` edge leading to that node), then an `Orig` marker is generated and linked to our node. The other rules are similar in spirit, each being composed of a pattern graph (the middle graph, just

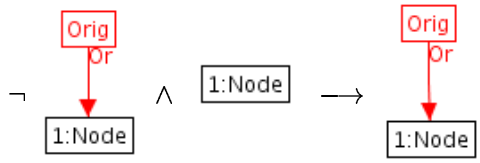

Fig. 1. Mark rule

`Node` in Figure 1), an optional non-applicability condition (the "negated" graph in Figure 1) and the result pattern on the right of the arrow.

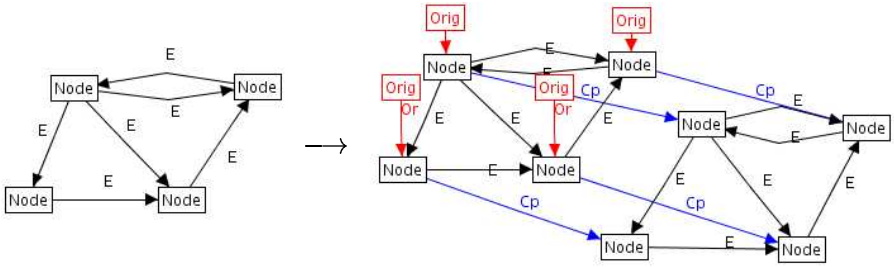

Fig. 2. Duplicating a graph

An example graph and the result of its transformation, just before deletion of the *copy* edges and the markers, is shown in Figure 2.

3 Formal Model

One standard semantics of graph transformations is based on category theory. For our purposes, it has the severe disadvantage of being non-constructive in the sense that often the existence of objects is postulated, but their construction is not effectively being carried out. For example, application of a rule to a graph is described by a pushout construction that yields a graph which is "minimal" in a certain class of graphs. Furthermore, often results are only valid "up to isomorphism of graphs", so reasoning with simple equalities is not possible.

We avoid these notions as far as possible and build our model on simple set-theoretic concepts. A graph consists of a set of nodes (the type *obj* used here is defined as isomorphic to the natural numbers), a set of edges indexed by an edge type $'et$ and a mapping associating a node type $'nt$ to the nodes. Note that our model precludes the existence of two edges of the same type between a pair of nodes.

record $('nt, 'et)\ graph =$
  $nodes :: obj\ set$
  $edges :: 'et \Rightarrow (obj * obj)\ set$
  $nodetp :: obj \Rightarrow 'nt\ option$

Graphs have to conform to some well-formedness constraints: The set of nodes has to be finite, the endpoints of edges have to be among the nodes, and the domain of the node typing *nodetp* has to be equal to the node set. Similarly, there are typing constraints: the edges of the graph have to obey a typing discipline (which essentially defines which edge type is compatible with which node types).

A simple graph transformation without non-applicability condition (NAC) has

to supply the following information: a pattern graph (just the `Node` in Figure 1), the set of nodes and edges to be deleted resp. generated, and a map which types the newly generated nodes:

```
record ('nt, 'et) graphtrans =
  — pattern graph
  patgr :: ('nt, 'et) graph
  — mapping of nodes
  ndel  :: obj set   — deleted nodes
  ngen  :: obj set   — generated nodes
  — mapping of edges
  edel  :: 'et ⇒ (obj * obj) set — deleted edges, indexed by type
  egen  :: 'et ⇒ (obj * obj) set — newly generated edges, indexed by type
  — typing of generated nodes
  ngentp :: obj ⇒ 'nt option
```

How to apply a transformation to a graph? An essential ingredient is a graph morphism, which is just a mapping between objects. Given a graph transformation, a graph morphism and a graph (say *gr*), we can define a function *apply-graphtrans* which hoists the transformation along the morphism. Among others, it deletes the nodes of *gr* which are in the image of the morphism under the set *ndel*.

We omit the definition of *apply-graphtrans* which is quite technical because it has to forestall possible pitfalls (dangling edges) and make precise what it means to create "new" nodes and edges in a graph.

Again, we have to impose soundness constraints both on the graph transformations (deleted nodes are contained in the pattern graph, generated nodes are disjoint from the pattern graph, ...) and on graph morphisms (type preservation of the mapping). As a major result of our formalization, we can prove that under these conditions, the function *apply-graphtrans* preserves well-formedness and well-typing. We can thus statically reduce type soundness of graph rewriting to checking individual transformation rules if we can assume that graph morphisms are sound. For program proofs, this result establishes a global invariant that need not be reproved for each rule application.

Given the definition of *graphtrans*, we can easily add NACs. A transformation rule is henceforth composed of a NAC (for example the leftmost graph of Figure 1) and a *graphtrans*, as before. We say that a transformation rule *trr* is applicable for a graph morphism *gm* in a graph *gr* (predicate *applicable-transfo-gm*) if *gm* is a valid morphism between the pattern of the rule and *gr* and there is no valid extension of *gm* from the NAC into *gr*. Finally, we say that a transformation rule is applicable (*applicable-transfo*) if there exists an applicable graph morphism.

```
ex-valid-gm-extension gm grs grt tp ≡
  (∃ gm'. gm ⊆m gm' ∧ valid-gm (gm' |` (nodes grs)) grs grt tp)

applicable-transfo-gm gm trr gr tp ≡
  (valid-gm gm (patgr (tr-transfo trr)) gr tp ∧ ¬ ex-valid-gm-extension gm (tr-nacs trr) gr tp)

applicable-transfo trr gr tp ≡ ∃ gm. applicable-transfo-gm gm trr gr tp
```

Note that applicability of a transformation rule is now independent of graph morphisms. Similarly, graph transformation programs can be non-deterministic in that a rule may be applied at different positions in a graph. When reasoning about these programs, we will require a correctness property to hold regardless of where the rules are applied.

4 Simplification of Proof Obligations

Simplification of proof obligations comes in two flavors, which we will call "elimination of higher-order quantification" and "problem-specific simplification". This topic is still under investigation, and so we only sketch some ideas.

The formalization presented in Section 3 is higher-order. We frequently talk about graph morphisms, and when considering the applicability of transformations, we have to assume the (non-) existence of morphisms between a source and a target graph or the existence of extensions of morphisms. But in fact, this is not the way we would like to reason. We would prefer to talk about the existence of nodes and edges in the target graph without reference to morphisms. That is what we understand by *elimination of higher-order quantification*.

To see how this might work, assume that we have a morphism gm' which is an extension of a "given" morphism gm, as in the definition of *ex-valid-gm-extension*. We also know the domain of gm', usually from the source graph to which it is applied. We can then apply the following lemma, which gradually reduces the domain of gm', thereby introducing the desired elements b in the target graph, until we end up with the known morphism gm.

lemma *dom-reduce-insert*:
$(dom\ gm' = insert\ a\ A) =$
$(\exists\ b\ gm''.\ gm' = gm''(a \mapsto b) \wedge gm'\ a = Some\ b \wedge dom\ gm'' = A)$

At first glance, it seems that we have not made much progress when applying this lemma in a proof, since the morphism gm' may still occur elsewhere in our goal. However, we can make gm' disappear if it only occurs in predicates that are "point-wise true", such as the welltypedness property of morphisms:

lemma *welltyped-gm-restr-comp*: $a \in dom\ gm \Longrightarrow$
$welltyped\text{-}gm\ (gm\ |`\ (insert\ a\ A))\ tpsrc\ tptgt\ tp =$
$(welltyped\text{-}gm\ (gm\ |`\ A)\ tpsrc\ tptgt\ tp \wedge\ welltyped\text{-}gm\ [a \mapsto the\ (gm\ a)]\ tpsrc\ tptgt\ tp)$

5 Conclusions

We have described a formalization of graph transformations in the Isabelle proof assistant, have demonstrated properties such as type preservation and hinted at how to eliminate higher-order quantification.

In future work, we intend to develop *problem-specific* simplification strategies which are applicable to verification problems expressed in restricted logical fragments, for example cardinalities of edge sets or reachability problems. Here, we hope to profit from some of the analyses for graph transformation systems mentioned in Section 1.

Furthermore, we intend to develop a small programming language which allows for a more fine-grained control of the application of rules (with sequential and parallel composition and possibly nested loops) and provides an annotation language for invariants.

Acknowledgement

This work has greatly benefited from suggestions by Jean-Paul Bodeveix and Mamoun Filali and discussions with Louis Féraud, Reiko Heckel, Ralph Matthes, Marc Pantel, Maxime Rebout, Sergei Soloviev and Gabriele Taentzer.

References

[Agr04] Aditya Agrawal. *A Formal Graph-Transformation Based Language for Model-to-Model Transformations.* PhD thesis, Vanderbilt University, August 2004.

[Bar03] Erik Barendsen. *Term Rewriting Systems*, chapter Term Graph Rewriting. Cambridge University Press, 2003.

[BBDV03] Jean Bézivin, Erwan Breton, Grégoire Dupé, and Patrick Valduriez. The ATL Transformation-based Model Management Framwork. Technical report, IRIN, September 2003.

[BCE+05] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICAH '05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).

[CMR+96] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Loewe. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. Technical Report TR-96-17, Dipartimento di Informatica, March 21 1996.

[EHK+97] 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.

[FM97] P. Fradet and D. Le Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *POPL*, pages 196–205, 1993.

[KS06] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Amsterdam, 2006. Elsevier Science Publ.

[MFV+05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Proc. Model Transformations In Practice Workshop*, 2005.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer Verlag, 2002.

[Plu99] Detlef Plump. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools, chapter Term Graph Rewriting. World Scientific, 1999.

[Tae03] Gabriele Taentzer. AGG: A graph transformation environment for system modeling and validation. In *Proc. Tool Exihibition at Formal Methods 2003*, 2003.