Modeling and Verifying Graph Transformations in Proof Assistants

Martin Strecker[1]

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. Whereas graph rewriting is usually performed by mapping a pattern graph into a source graph by means of a graph morphism and then carrying out operations on the image node and edge set, this article generalises the notion of pattern graph to path expressions, which are formulae in a fragment of first-order logic. We examine the correspondence with traditional graph rewriting and show that this interpretation is beneficial when formally reasoning about model transformations with the aid of proof assistants.

Keywords: Graph Transformations, Theorem Proving

1 Introduction

Graph rewriting examines which structural changes are engendered when applying rewrite rules to a graph. There is no unique approach to graph rewriting - one may cite algebraic [Bar03] and categorical [CMR+97,EHK+97] formalisms.

The discipline has accumulated an impressive amount of results on properties of rewrite systems (such as confluence and termination) resulting from specific rule formats [Plu99]. Recently, there is a growing practical interest in graph rewriting in the context of model driven engineering, where a software or hardware artifact is represented graphically and can be refined or refactored by the application of graph rewriting rules. Several graph rewriting tools are available. They emanate from foundational work and are usually equipped with some analyses of rule properties [Tae03,KS06,Agr04], or take a more pragmatic view (ATL [BBDV03] and Kermeta [MFV+05]).

In spite of a large body of work on graph transformations, the question of verification of transformations "in general" is far from settled. The foundational work

[1] Email: strecker@irit.fr

of [Cou90] aims at a logical characterization of graph transformations, where effective verification of structural properties is not a primary concern. Usually, however, graph transformation systems are 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 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]. There are automated approaches based on model checking [Var04], which however can only handle graphs with an a priori bounded number of elements. [RD06] presents techniques for dealing with specific structural properties such as multiplicities.

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 verification of structural properties will be the main focus of this paper. The work reported here has grown out of an effort to formalise model transformations in interactive proof assistants. A first attempt [SG06], aiming at formalising traditional graph rewriting as sketched above, required complex reasoning about graph morphisms. It has turned out that replacing the pattern graph by formulae over graph structure (which we will call *path formulae* in the following) yields much more manageable proof obligations. At the same time, path formulae are more expressive than pattern graphs and have therefore an interest in their own, independently from concerns about formal verification.

Path formulae can be understood as formulae over a fragment of first order logic (possibly including transitive closure), which are interpreted over graphs. Determining whether a graph satisfies a path formula is decidable, which is indispensable for effectively applying a transformation rule to a given graph. On the downside, validity of path formulae may not be decidable, so that interactive proofs become necessary.

The paper is structured as follows: In Section 2, we informally introduce generalised graph transformations. The formal model is presented in Section 3. In Section 4, we show how we can recover the traditional model of graph rewriting. We take a glimpse at how to reason about graph transformations in a proof assistant in Section 5 before concluding with an outlook on future work.

2 Example Transformations

To set the stage, we describe two toy transformations: a transformation duplicating a graph, and another one implementing a simple garbage collector.

The purpose of the *graph duplication* transformation is to generate a new graph consisting of two exact copies of the original graph. We assume that the original graph has nodes of type `Node`, with edges of type `E` between them. For the purposes of transformation, we need nodes of type `Orig`, supposed to mark the nodes of the original graph during transformation, and edge types `Or` (between `Orig` and `Node`) and `Cp` (between a node and its original).

Duplication proceeds in several steps: First, we mark all nodes of the original graph with `Orig` nodes. We then create a duplicate node for each original, memorising the relation between the original and the clone with a `Cp` edge. We can similarly reproduce the edges of the original graph in the copy. All that remains to be done now is to erase the auxiliary marking.

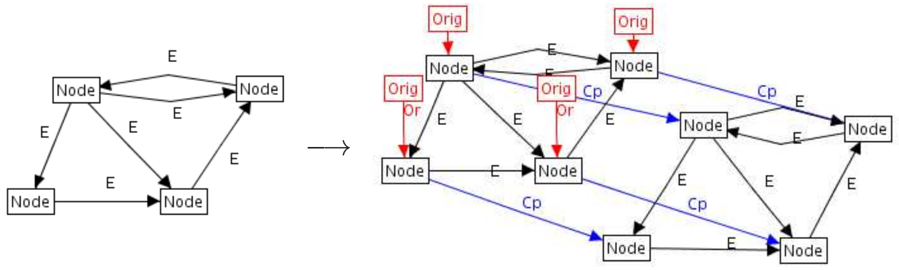

Fig. 1. Duplicating a graph

An example graph and the result of its transformation, just before deletion of the `Cp` edges and the markers, is shown in Figure 1. This is a screen shot of graphs produced by the AGG tool [Tae03], based on a categorical approach, which allows to conveniently model this kind of transformation (a more detailed comparison follows in Section 4).

How do we formalise the marking phase, i.e. the first step of our transformation? For the time being, we use a semi-formal notation that should be intuitively understandable. Precise definitions of graphs and path formulae will be presented in Section 3.1 and Section 3.2, respectively.

In our setting, a transformation rule is composed of two elements: an application condition and an action part. The application condition, a path formula F expressing if and where a rule can be applied, says that the rule can operate on any node n of type `Node` which is not already marked by some node m of type `Orig`:

$$F(n) \equiv Node(n) \wedge \neg \exists m.\ (Orig(m) \wedge m \xrightarrow{Or} n)$$

Thus, typing is expressed by unary predicates ($Node$ and $Orig$), and a binary relation $m \xrightarrow{Or} n$ represents an `Or` edge between m and n.

The action part (not shown here) expresses what we do if F is satisfied for a node n: We generate a new node, say m', having type `Orig`, and we create an `Or`-edge (m', n). We will come back to this example in Section 3.3.

Of course, a single transformation step of this kind will not suffice to mark all nodes of a graph. Rather, we have to iterate the rule until no further application is possible, i.e. until F is false for all nodes of the graph. We will briefly look at this question in Section 5.

The *garbage collector* is an example of a transformation that is not directly expressible in traditional graph rewriting approaches. We assume to have a number of `Root` objects and a number of `Node` objects. `Root` objects are linked to `Node`s through `rn` edges, `Node`s are linked among themselves through `nn` edges. Any `Node` not accessible from a `Root` is considered as garbage.

The predicate $G(n)$ saying that node n is garbage can be written as the path formula

$$G(n) \equiv \neg\exists r\ n'.\ \ r \xrightarrow{rn} n' \land n' \xrightarrow{nn}{}^{*} n$$

where $\xrightarrow{rn}$ is an `rn` edge (and similarly for `nn`), and the "star" is reflexive transitive closure.

$G(n)$ is the application condition of a rule `collect`, whose action part just says that n should be deleted (in doing so, all adjacent edges disappear as well).

In the case of $G(n)$, we have chosen not to make the typing information explicit in the rule itself. In fact, it can be deduced from general typing predicates, expressible as path formulae, that could form the "background theory" of the application. For example, the typing of the `rn` edge is stated as

$$\forall r\ n.\ \ (r \xrightarrow{rn} n) \longrightarrow Root(r) \land Node(n)$$

3 Formal Model

In this section, we formally present the basic notions of our graph rewriting approach, notably graphs, graph transformations and morphisms and some well-formedness conditions we have to impose to ensure consistency of the model. Since our development has been carried out in the Isabelle proof assistant [NPW02], we will use Isabelle's syntax, which we will explain wherever needed.

3.1 Graphs

Our purpose is not to formalize any particular approach to graph rewriting, such as the one based on category theory. Our model is set-theoretic. Roughly, graphs are composed of a finite set of nodes, a finite set of edges and a typing of the nodes.

In order to create new nodes during graph rewriting, we have to have an infinite supply of fresh nodes. We have therefore chosen to take the natural numbers as the base type of our nodes. The edges are sets of pairs of nodes, indexed by an *edge type* $'et$, such as `Cp` and `E` in the introductory example. This precludes having more than one edge of a given edge type between two nodes. However, under this definition, one can more easily use standard relational operators like composition and transitive closure, which comes handy when defining the semantics of path expressions further below. A *node typing* assigns a node type $'nt$ (such as `Root` and `Node`) to each node of the graph. Altogether, this gives the following definition of the type of graphs:

```
record ('nt, 'et) graph =
  nodes :: nat set
  edges :: 'et ⇒ (nat * nat) set
  nodetp :: nat ⇒ 'nt option
```

(An option type *T option* has a distinguished value *None*, representing undefinedness, and defined values *Some t* for *t* and element of *T*.)

In a minimalistic model, node typing is inessential, but it is useful for describing some structural aspects of graphs. However, we have excluded more complex node attributes that would be required for formalising the semantics of an artifact. They could be easily added by providing a mapping in the spirit of *nodetp* from the node set to an attribute domain.

Finiteness of the node set is expressed by a *structural well-formedness* predicate, just as the containment of the endpoints of edges in the node set and well-definedness of node typing:

```
struct-wf-gr :: ('nt, 'et) graph ⇒ bool
struct-wf-gr gr ==
   (finite (nodes gr)) ∧
   (∀ et. (Field (edges gr et)) ⊆ (nodes gr)) ∧
   dom (nodetp gr) = (nodes gr)
```

Here, *dom* is the domain of a mapping, *Field* the union of the domain and range of a relation. Access to a component of a record, such as *nodes*, is written in functional notation.

3.2 Path expressions

The application of graph transformations to a graph is subject to an applicability condition. Traditionally, this applicability condition is given in the form of a *pattern graph* which is mapped, via a graph morphism, into a source graph to which the transformation will be applied.

In a first attempt [SG06], we have faithfully coded this approach, but it has turned out that the formulae resulting from this graph mapping require considerable massaging for being usable any further. We try to circumvent this problem by replacing the pattern graph by a predicate on (source) graphs, which at the same time opens up the possibility of expressing more general properties (we come back to this in Section 4).

However, we have to take care not to use too complex predicates: The least we can expect from a graph rewriting engine is to be able to decide whether a predicate is satisfied for a particular graph and thus, whether a rule is applicable to this graph. Differently said, the model checking problem for the class of predicates should be decidable, even though entailment need not be, see Section 5.

In the following, we present a logic of path formulae, which we have found useful for expressing interesting properties (see the discussion in Section 4). However, there is no intrinsic reason to adopt precisely the language constructors we have selected, and the decidability of the logic, as well as the complexity of model checking, is greatly influenced by this choice. Similar notions can be found in [YRS⁺06,KS93,Ren03].

To have a fine control over the logic of predicates on graphs, we deeply embed it into Isabelle's higher order logic. We start by defining node set expressions (representing sets of nodes) and path expressions (representing endpoints of paths):

```
datatype 'nt nodeset
  = All-set              — set of all nodes of graph
  | Type-set 'nt         — set of all nodes of given type
  | Singleton-set nat    — singleton containing constant

datatype ('nt, 'et) path
  = Empty-pth                              — empty path
  | Edge-pth 'et                           — edge with given edge type
  | InvEdge-pth 'et                        — inverse edge
  | Seq-pth ('nt, 'et) path ('nt, 'et) path  — sequential composition
  | Alt-pth ('nt, 'et) path ('nt, 'et) path  — alternative
  | Clos-pth ('nt, 'et) path                 — transitive closure
```

Based on this, we define path formulae, which are constructed from two base cases (set and path formulae, for node set and path expressions, respectively), and the usual Boolean connectives and quantifiers:

```
datatype ('nt, 'et) path-form
  = S-form 'nt nodeset nat              — set formula
  | P-form ('nt, 'et) path nat nat  — path formula
  | Neg-form ('nt, 'et) path-form     — negation
  | Conj-form ('nt, 'et) path-form ('nt, 'et) path-form     — conjunction
  | All-form ('nt, 'et) path-form    — universal quantification
```

With the above, other connectives and the existential quantifier *Ex-form* can be defined as abbreviation. Universal quantification does not use a named, but rather a positional representation of variables (de Bruijn indices, [dB72]). Thus, variables are not identifiers, but just numbers.

In our informal notation of Section 2, we have written *S-form* (*Type-set* T) n simply as $T(n)$ and *P-form* (*Edge-pth* e) $n\ n'$ as $n \xrightarrow{e} n'$. For instance, the application condition $\neg\exists r\ n'.\ r \xrightarrow{rn} n' \wedge n' \xrightarrow{nn}{}^{*} n$ of the garbage collector example of Section 2 becomes:

```
Neg-form (Ex-form (Ex-form
  (Conj-form
    (P-form (Edge-pth rn) 1 0)
    (P-form (Clos-pth (Edge-pth nn)) 0 2))))
```

The semantics of expressions respectively formulae is defined by means of functions *nodeset-interp*, *path-interp* respectively *path-form-interp* that interpret the expressions respectively formulae under a variable interpretation I : *nat* $\Rightarrow$ *nat* in a graph *gr*.

```
consts
  nodeset-interp :: [nat ⇒ nat, ('nt, 'et) graph, 'nt nodeset] ⇒ nat set
primrec
  nodeset-interp I gr All-set = nodes gr
  nodeset-interp I gr (Type-set t) = {n. nodetp gr n = Some t}
  nodeset-interp I gr (Singleton-set n) = {I n}

consts
  path-interp :: [nat ⇒ nat, ('nt, 'et) graph, ('nt, 'et) path] ⇒ (nat * nat) set

primrec
  path-interp I gr Empty-pth = diag UNIV
  path-interp I gr (Edge-pth e) = edges gr e
  path-interp I gr (InvEdge-pth e) = (edges gr e)^-1
  path-interp I gr (Seq-pth p p') = (path-interp I gr p) O (path-interp I gr p')
  path-interp I gr (Alt-pth p p') = (path-interp I gr p) ∪ (path-interp I gr p')
  path-interp I gr (Clos-pth p) = (path-interp I gr p)^*

consts
  path-form-interp :: [nat ⇒ nat, ('nt, 'et) graph, ('nt, 'et) path-form] ⇒ bool
primrec
  path-form-interp I gr (P-form p n n') = ((I n, I n') ∈ path-interp I gr p)
  path-form-interp I gr (S-form s n) = (I n ∈ nodeset-interp I gr s)
  path-form-interp I gr (Neg-form pf) = (¬ (path-form-interp I gr pf))
  path-form-interp I gr (Conj-form pf pf') =
    ((path-form-interp I gr pf) ∧ (path-form-interp I gr pf'))
  path-form-interp I gr (All-form pf) =
    (∀ x. x ∈ nodes gr ⟶
        path-form-interp ((I o (λ x. x - 1))(0:=x)) gr pf)
```

In the above, *UNIV* is the set of all elements (of the given type), *diag* the diagonal of a set (the relation (e, e)), the converse of a relation R is written $R\hat{}{-1}$, and O is relation composition and $\circ$ function composition. The interpretation of universal quantification is comparable to the "lift" operation for de Bruijn indices:

The current variable x is assigned the index 0, the other indices are shifted by 1.

Model checking of node set and path expressions, i.e. checking that a graph gr satisfies a node set or path expression, reposes on well-known graph algorithms. Universal quantification is relativised to the node set of the graph, which is finite by well-formedness of graphs. Therefore, checking a universal formula only has to examine a finite number of elements.

3.3 Graph Transformations

Roughly speaking, a graph transformation rule should specify under which condition the transformation is applicable, and what to do when applying the transformation at a position in a source graph to obtain a target graph.

The applicability condition is just given by a path formula, as outlined in the previous section. Note that this path formula may contain free variables, for example n in $G(n)$ of Section 2, which can be understood as references to nodes in the source graph. Of course, in its coding as path formula, the free variables are numbers.

It is these numbers that we refer to when specifying the action: we say which nodes are to be deleted respectively freshly generated (*ndel* respectively *ngen*) and which edges are deleted respectively generated (*edel* respectively *egen*). Furthermore, we have to know how to type the newly generated nodes. Altogether, graph transformations have the form:

```
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  :: 'et ⇒ (nat * nat) set   — deleted edges, indexed by type
  egen  :: 'et ⇒ (nat * nat) set   — generated edges, indexed by type
  — typing of generated nodes
  ngentp :: nat ⇒ 'nt option
```

For example, the marking rule of Section 2 can now be expressed by the transformation:

```
  mark :: (nodetp, edgetp) graphtrans
  mark ==
  (| appcond = mark-F 0,
     ndel = {},
     ngen = {1},
     edel = λ et. {},
     egen = (λ et. {})(Or:={(1,0)}),
     ngentp = [1 ↦ Orig]
  |)
```

Here, *mark-F* is the coding of the application condition. The application position of the rule is node 0. No nodes and edges are deleted, a node numbered 1 is generated and an `Or` edge is added between node 1 and 0. (The syntax for update of function f at x with value y is $f(x{:=}y)$.)

For graph transformations to make sense, the references to nodes to be deleted have to be among the references to nodes in the applicability condition (thus, to the free variables of the applicability condition), whereas references to generated nodes should not occur in the applicability condition. We only generate a finite number of nodes in each transformation step, and to all of these nodes we assign a type. Similar constraints hold for deleted and generated edges. To summarise, structural well-formedness of a graph transformation is expressed by the following predicate:

```
struct-wf-gt :: ('nt, 'et) graphtrans ⇒ bool
struct-wf-gt gt ==
  (ndel gt) ⊆ (fv-path-form (appcond gt)) ∧
  finite (ngen gt) ∧ (fv-path-form (appcond gt)) ∩ (ngen gt) = {} ∧
  dom (ngentp gt) = (ngen gt) ∧
  (∀ et. Field (edel gt et) ⊆ (fv-path-form (appcond gt))) ∧
  (∀ et. Field (egen gt et) ⊆ ((fv-path-form (appcond gt)) − (ndel gt)) ∪ (ngen gt))
```

3.4 Applying Graph Transformations

We now come to the application of a graph transformation to a source graph at a particular position. In graph rewriting, matching a pattern graph to a source graph (and thus determining the application position) is traditionally achieved with the aid of a graph morphism. We adopt the same terminology and define

types *graphmorph* = (*nat* ⇒ *nat option*)

with the understanding that the node references occurring in a graph transformation rule are mapped to the nodes in a source graph. For the "garbage collection" example, such a situation is depicted in Figure 2.

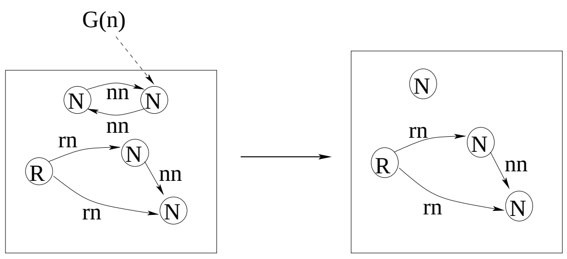

Fig. 2. Application of a graph morphism in a graph

We now have to spell out in detail how the target graph is composed, provided we apply a graph transformation *gt* to a graph *gr* using a morphism *gm*. Quite simply, the nodes to be deleted are just the ones in the image of the morphism under the *ndel*-set.

It is more difficult to express which nodes are generated. The choice could be, non-deterministically, any node set having the same cardinality as the *ngen*-set and having no nodes in common with the nodes of the source graph. We have adopted a deterministic solution: The nodes freshly allocated are numbered $m + 1$ through $m + k$, where m is the maximal number present in the node set of graph *gr* and k is the cardinality of the *ngen*-set. All this is hidden in the definition of *gt-gen-nodes*. However, we only exploit the property that the fresh nodes do not occur in the original graph, and that there is a bijection b between the ngen-nodes and the fresh nodes.

The latter property is needed for determining the type of the generated nodes. How do we compute it, for a fresh node n? We map n back into the graph transformation *gt*, where we can look up its type. Thus, roughly, the type of n is $(ngentp\ gt)(b^{-1}(n))$.

The morphism on nodes induces a morphism on edges. From the *edel*- and *egen*-sets, we can thus determine the edges in the source graph which are candidates for deletion and for insertion. We want to avoid dangling edges that result when nodes are requested to be deleted, but not their adjacent edges. Therefore, the edges that survive are those whose nodes are among the nodes of the target graph. A similar restriction applies to the typing of the target nodes.

With these explanations, the exact definition should be understandable:

```
apply-graphtrans ::
    [('nt, 'et) graphtrans, graphmorph, ('nt, 'et) graph] ⇒ ('nt, 'et) graph
apply-graphtrans gt gm gr ==
  let del-nodes = ran (gm |‘ (ndel gt)) in
  let gen-nodes = gt-gen-nodes gr gt in
  let morph-gen = separ-map (ngen gt) (nodes gr) in
  let morph-c = gm ++ morph-gen in
  let nds = ((nodes gr) − del-nodes) ∪ gen-nodes in
  let del-edges = (λ et. (induced-emorph gm) ‘ (edel gt et)) in
  let gen-edges = (λ et. (induced-emorph morph-c) ‘ (egen gt et)) in
  let tp-ngen = ((ngentp gt) ∘m (inv-m morph-gen)) in
  (| nodes = nds,
    edges = λ et. (restrict-rel ((edges gr et − del-edges et) ∪ gen-edges et) nds),
    nodetp = (restrict-map ((nodetp gr) ++ tp-ngen) nds)
  |)
```

In the above, *f* ‘ *S* is the image of set *S* under function *f*, and *m* |‘ *S* restricts map *m* to *S*. In *m1* ++ *m2*, map *m2* overrides *m1*, and $\circ_m$ is the composition of maps.

3.5 Applicability of Graph Transformations

What we have called "graph morphisms" in Section 3.4 is essential for determining whether a transformation is applicable, and if yes, where to apply it. It should be emphasised again that "graph morphism" is a slight misnomer, because we do not map graphs into graphs, as in traditional graph rewriting. Rather, we want to verify that the applicability condition of a transformation rule is true.

The following predicate states that a graph morphism *gm* satisfies a path formula *pfs* in a graph *grt*:

```
applicable-gm :: [graphmorph, ('nt, 'et) path-form, ('nt, 'et) graph] ⇒ bool
applicable-gm gm pfs grt ==
  (dom gm = fv-path-form pfs) ∧ (ran gm ⊆ nodes grt) ∧
   path-form-interp (the o gm) grt pfs
```

The domain of the graph morphism has to be the set of free variables of the path formula, and its range has to be a subset of the nodes of the graph. Most importantly, the path formula has to be satisfied in the graph when interpreting its free variables by the graph morphism in the given graph. (*the* is the left inverse of *Some*, thus *the* (*Some* x) = x).

In most of our reasoning, we want to abstract away from particular graph morphisms and just say that a transformation is applicable in a graph:

```
applicable-transfo :: [('nt, 'et) graphtrans, ('nt, 'et) graph] ⇒ bool
applicable-transfo gt gr == ∃ gm. applicable-gm gm (appcond gt) gr
```

Now, applying a graph transformation to a graph amounts to selecting an arbitrary graph morphism and applying it to the graph:

```
apply-transfo :: [('nt, 'et) graphtrans, ('nt, 'et) graph] ⇒ ('nt, 'et) graph
apply-transfo gt gr ==
  apply-graphtrans gt (SOME gm. (applicable-gm gm (appcond gt) gr)) gr
```

Here, *SOME* is Hilbert's choice operator which could be replaced by a constructive choice based, for example, on a node ordering.

3.6 Properties of Graph Transformations

We can now state a major result: application of well-formed graph transformations to well-formed graphs yields again well-formed graphs:

$$\textit{struct-wf-gr gr} \wedge \textit{struct-wf-gt gt} \longrightarrow \textit{struct-wf-gr} \ (\textit{apply-graphtrans gt gm gr})$$

This can be construed as a generic invariant of graph transformations that need not be reproved for each transformation rule when reasoning about graph transformation programs (see Section 5). Note that the structural well-formedness of the resulting graph depends on the well-formedness of the graph transformation gt, but is valid for arbitrary graph morphisms gm.

In [SG06], we have shown that for traditional graph rewriting, we can similarly ensure preservation of well-typing. In our current setting, we can express more general typing properties than those examined in [SG06], for example cardinality constraints, so that "typing" in full generality becomes undecidable. We are currently exploring fragments of our path logic that permit sufficiently interesting typing properties to be expressed and preservation of typing to be proved.

4 Correspondence with Graph Rewriting

In the following, we will argue that transformations expressible in traditional graph rewriting approaches can be coded in our system. It is therefore possible to "compile" traditional graph rewriting rules to expressions involving our path formulae. It is then possible to use the techniques described in Section 5 as a verification backend.

In the rules of the AGG system [Tae03], for example, there are positive and negative applicability conditions, and each such condition is a graph that has to occur, respectively must not occur, in the graph where the rule is applied. As seen in Section 2, we can code positively occurring graphs by a conjunction of node set and path constraints, more precisely

- a node set constraint $T(n)$ for every node n of type T in the graph
- a path constraint $n \xrightarrow{e} n'$ for each edge e in the graph.

As mentioned before, we do not allow multiple edges of the same edge type between a pair of nodes. We do not see that as a major drawback – if necessary, edges can be "reified" by introducing a node representing the edge.

For negative applicability conditions, we proceed in an analogous manner, with the difference that the nodes of the graph are asserted not to exist. Thus, for an edge e occurring in a negative applicability graph, we have a path formula $\neg\exists n\, n'. n \xrightarrow{e} n'$.

The GReAT language [AKK+05] includes, among others, cardinality constraints. It is thus possible to specify that a node n must (or must not) have k outgoing e-edges. Cardinality constraints are not present as primitive constructs in our language, but they can be coded by a schema like

$$C_k(n) \equiv \exists x_1 \dots x_k.\ n \xrightarrow{e} x_1 \wedge \dots n \xrightarrow{e} x_k \wedge \textit{distinct}(x_1, \dots x_k)$$

where $\textit{distinct}(x_1, \dots x_k)$ is the conjunction $\neg(x_i = x_j)$, for $i, j \in \{1, \dots, k\}, i \neq j$.

The fact that the graph morphisms between a pattern and a source graph is injective is usually an external notion in traditional graph rewriting. In a similar spirit as the above formula, we can internalise this notion and express that the nodes a rule is applied to are distinct.

5 Reasoning about Graph Transformations

As mentioned in Section 2, it is not sufficient to apply a transformation rule once. Rather, one has to apply a rule repeatedly, or several rules have to be applied in a specific order. Most graph rewriting tools permit to iterate rule application, often by dividing the tool set into "layers". The need for exerting finer control on graph transformations has been recognised, among others, by the developers of the GReAT language, who develop a graphical language including conditional and loop constructs [AKK+05].

We are currently developing a simple language for writing graph transformation programs and reasoning about them. The language is not sufficiently polished to present details, so we just give a sketch and describe how we might treat the "marking" example of Section 2.

The language is composed of statements *stmt*, among which we only mention *Do* and *Loop*. An operational semantics describes how a state is modified by these constructs. We distinguish between success and failure states. In our case, a "state" is just a graph with a "success" or "failure" tag. The meaning of the mentioned constructs is then:

- *Do* b f checks whether condition b is satisfied in the current state s. If this is the case, function f is applied to s to produce a success state s'. Otherwise, s is returned as a failure state.
- *Loop* c applies statement c indefinitely often, until winding up in a failure state, which is the result of the loop.

Let us introduce the following abbreviation:

```
App :: ('nt, 'et) graphtrans ⇒ ('nt, 'et) graph stmt
App gt == Do (λ s. applicable-transfo gt (outcome-val s))
              (λ s. apply-transfo gt (outcome-val s))
```

Here, *outcome-val* discards the success / failure tag of a state. Consequently, *App* applies a graph transformation, if possible, and returns the current state as failure state otherwise.

The marking phase of the introductory example can now be written as the program *Loop* (*App mark*), where we use the definition *mark* of Section 3.3. The entire graph duplication transformation consists of a sequence of such loops, each with a different rule.

The language comes equipped with a Hoare-style program logic. We write $W \vdash \{P\}\ c\ \{Q\}$ to express that statement c establishes the postcondition Q provided the precondition P and some invariant well-formedness conditions W hold. W is typically the predicate *struct-wf-gr* that we have shown to be invariant under application of graph transformations in Section 3.6. Furthermore, the statement c usually contains annotations corresponding to loop invariants.

Suppose we want to show, for our example program, that all nodes of type `Node` are correctly marked, i.e. have exactly one incoming `Or` edge, provided that in the outset, these nodes had zero or one incoming `Or` edges. Let us first define *nset* as the set of nodes in a graph having a given node type:

$$\begin{array}{l} \mathit{nset} :: [(\mathit{'nt}, \mathit{'et})\ \mathit{graph}, \mathit{'nt}] \Rightarrow \mathit{nat\ set} \\ \mathit{nset\ gr\ nt} == \{n \in \mathit{nodes\ gr}.\ (\mathit{nodetp\ gr\ n}) = \mathit{Some\ nt}\} \end{array}$$

We can now state the precondition:

$$\forall x \in \mathit{nset\ gr\ Node}.\ \ \mathit{card}\ ((\mathit{edges\ gr\ Or})^{-1}\ ``\ \{x\}) \leq 1$$

(here, R `` S is the image of a set S under a relation R, and *card* the cardinality of a set). The postcondition is similar, with the inequality replaced by an equality.

The verification condition generator leaves us essentially with two goals: showing that the loop invariant is preserved if the rule *mark* is applicable, and showing that the postcondition is satisfied if the rule is not applicable. We just look at the latter case.

So assume that $\neg$ *applicable-transfo mark gr*. According to the definition of *applicable-transfo*, this is equivalent to $\forall$ *gm*. $\neg$ *applicable-gm gm* (*appcond mark*) *gr*, which contains an annoying second-order quantifier over a graph morphism *gm*.

However, when looking at the definition of *applicable-gm*, we realise that the domain of *gm* is finite. We can therefore eliminate *gm* and instead introduce a first-order quantifier *b*, so that we are eventually left with the hypothesis

$$\begin{array}{l} \forall n.\ n \in \mathit{nodes\ gr} \longrightarrow \mathit{nodetp\ gr\ n} = \mathit{Some\ Node} \\ \longrightarrow (\exists x.\ \mathit{nodetp\ gr\ x} = \mathit{Some\ Orig} \wedge (x, n) \in \mathit{edges\ gr\ Or}) \end{array}$$

which naturally describes the non-applicability of the rule and permits to prove the required cardinality property.

6 Conclusions

In this paper, we have presented first steps towards the verification, in an interactive proof assistant, of structural properties established by graph rewriting systems. At the same time, the path formulae we have introduced give an alternative view on applicability conditions for graph rewriting rules, that may profitably be used in graph rewriting systems.

Our path formulae are very expressive, which has the downside of leading, in general, to undecidable verification problems. As we want to reduce the amount of human proof effort as much as possible, we intend to address this topic in future work, by developing specialized analyses for fragments of our logic. In fact, our path formulae resemble path expressions used in shape analysis for pointer programs [YRS+06,KS93], other subsets have been identified in the context of description logics [GM05]. A detailed comparison of these approaches still has to be done.

As noted before, our current formalization only deals with structural properties. Adding node attributes to the framework presented here is possible, but cumbersome. We are currently working on translating graph transformations to verification environments for pointer programs [Sch05,Fil03]

Acknowledgement

This work has been strongly influenced by suggestions from Jean-Paul Bodeveix and Mamoun Filali and discussions with Louis Féraud, Ralph Matthes, Marc Pantel, Maxime Rebout and Sergei Soloviev. Mathieu Giorgino has elaborated several example transformations.

References

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

[AKK+05] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Journal of Software and System Modeling*, 2005.

[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+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars*, pages 163–246. World Scientific, 1997.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 193–242. Elsevier, 1990.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[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.

[Fil03] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

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

[GM05] Lilia Georgieva and Patrick Maier. Description logics for shape analysis. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM 2005*, pages 321–330, Koblenz, Germany, September 2005. IEEE Computer Society, IEEE.

[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.

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electr. Notes Theor. Comput. Sci*, 157(1):39–59, 2006.

[Ren03] Arend Rensink. Towards model checking graph grammars. In *Proc. Workshop on Automated Verification of Critical Systems (AVoCS)*, 2003.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *LPAR 2005*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 398–414. Springer Verlag, 2005.

[SG06] Martin Strecker and Mathieu Giorgino. Towards a formalisation of graph transformations in proof assistants. In *Proc. AVOCS'06*, September 2006.

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

[Var04] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[YRS+06] Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In Luca Aceto and Anna Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2006.