GCM 2010

The Third International Workshop on Graph Computation Models

Proceedings

Enschede, The Netherlands, October 2010

Editors:
Rachid Echahed, Annegret Habel and Mohamed Mosbah

Contents

Preface

This volume contains the proceedings of the Third International Workshop on *Graph Computation Models (GCM 2010*[1]*)*. The workshop took place in Enschede, The Netherlands, on October 2nd, 2010, as part of the fifth edition of the International Conference on Graph Transformation (ICGT 2010).

The aim of GCM[2] workshop series is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation techniques. It promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications, and implementations of graph computation models and related areas. Previous editions of GCM series were held in Natal, Brazil (GCM2006) and in Leicester, UK (GCM2008).

These proceedings contain 12 accepted papers. All submissions were subject to careful refereeing. The topics of accepted papers range over a wide spectrum, including theoretical aspects of graph transformation, proof methods, semantics as well as application issues of graph computation models. Selected papers from these proceedings will be published as an issue of the international journal *Electronic Communications of the EASST*.

We would like to thank all who contributed to the success of GCM 2010, especially the Program Committee and the additional reviewers for their valuable contributions to the selection process as well as the contributing authors. We would like also to express our gratitude to all members of the Joint ICGT/SPIN Conference Organizing Committee for their help in organizing GCM 2010 at Enschede, The Netherlands.

August, 2010 Rachid Echahed, Annegret Habel and Mohamed Mosbah
Program co-chairs of GCM 2010

[1] GCM2010 web site: http://gcm2010.imag.fr
[2] GCM web site : http://gcm-events.org

Program committee of GCM 2010

Frank Drewes	Umea University, Sweden
Rachid Echahed	LIG Lab., Grenoble, France (co-chair)
Emmanuel Godard	University of Provence Aix-Marseille, France
Stefan Gruner	University of Pretoria, South Africa
Annegret Habel	University of Oldenburg, Germany (co-chair)
Dirk Janssens	University of Antwerp, Belgium
Hans-Jörg Kreowski	University of Bremen, Germany
Mohamed Mosbah	University of Bordeaux 1, France (co-chair)
Detlef Plump	University of York, UK

Additional Reviewers

Berthold Hoffmann
Barbara König
Alexander Paar
Christopher Poskitt
Caroline von Totth
Bruce Watson
Zhilin Wu

Construction of Pushout Complements in the Category of Hypergraphs

Marvin Heumüller[1], Salil Joshi[2], Barbara König[1], and Jan Stückrath[1]

[1] Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, Germany
[2] Indian Institute of Technology, Delhi, India

Abstract. We describe a concrete construction of all pushout complements for two given morphisms $f\colon A \to B$, $m\colon B \to D$ in the category of hypergraphs, valid also for the case where f, m are non-injective. To our knowledge such a construction has not been discussed before in the literature. It is based on the generation of suitable equivalence relations. We also give a combinatorial interpretation and show how well-known coefficients from combinatorics, such as the Bell numbers, can be recovered.

1 Introduction

Pushout complements are an integral part of double-pushout rewriting [2, 4, 5]: they implement the deletion of elements, whereas the creation of new elements is implemented via a pushout. Hence the construction of pushout complements is needed for many tools based on double-pushout graph rewriting. Most of the time the left leg of a rule is considered to be injective and thus the construction of pushout complements is greatly simplified compared to the general case, where both morphisms might be non-injective. A thorough study of the expressiveness of injective and non-injective rules and matches can be found in [6].

In [7] we considered a backwards analysis technique for graph transformation systems where rewriting steps have to be applied backwards. That is we are interested in *all* predecessors of a given graph, which is a common scenario in verification techniques. In this setting pushout complements have to be constructed for the right leg of a rule and in many applications this morphism is *not* injective, especially in cases where graph nodes and edges are fused by rewriting. (In [7] we considered in fact single-pushout rewriting [3] with pushouts in the category of partial morphisms. The problem of computing such pushout complements can be reduced to the construction of pushout complements for total morphisms, hence the construction given in this paper can also be adapted to the scenario in [7].)

Taking pushout complements for morphisms which are non-injective means—intuitively—to "unmerge" or split nodes in all possible ways, which can lead to a combinatorial explosion and serious efficiency problems.

In the literature the general case has so far received little attention. In the 70s the papers introducing and studying the notion of pushout complement [4, 5, 9]

restricted to cases where either a vertical or a horizontal morphism is injective. Furthermore there are some investigations into taking pushout complements in more general categories [1, 8], but they usually assume that the first morphism is a mono or consider only the minimal pushout complement. Since a construction of general pushout complements does not seem to be available in the literature, we specified this construction ourselves and found it surprisingly complex. Hence we believe that it is of general interest.

We will in the following define the construction which computes all pushout complements for two given morphisms $f\colon A \to B$, $m\colon B \to D$. This is done by defining an auxiliary graph $A \oplus \tilde{D}$ which is the disjoint union of A and a disjoint collection of all nodes and edges of D, which are not in the image of m. Then we enumerate all equivalences on $A \oplus \tilde{D}$ satisfying certain conditions and factor through these equivalences. In this way we obtain all pushout complements and our main theorem proves this fact. Furthermore—since the enumeration of all equivalences on $A \oplus \tilde{D}$ is very costly and there are serious issues with efficiency—we consider optimizations. Finally we show how some coefficients from combinatorics, such as Bell numbers or Stirling number of the second kind arise as the number of pushout complements for certain pairs of arrows. This also shows that there can be a combinatorial explosion in the number of constructed pushout complements.

2 Preliminaries

We first define the usual notions of hypergraph and hypergraph morphism.

Definition 1 (Hypergraph). *Let Λ be a finite set of labels and a function $ar\colon \Lambda \to \mathbb{N}_0$ that assigns an* arity *to each label.*

*A (Λ-)*hypergraph *is a tuple (V_G, E_G, c_G, l_G) where V_G is a finite set of nodes, E_G is a finite set of edges, $c_G\colon E_G \to V_G^*$ is a connection function and $l_G\colon E_G \to \Lambda$ is the labelling function for edges. We require that $|c_G(e)| = ar(l_G(e))$ for each edge $e \in E_G$.*

Definition 2 (Hypergraph morphism). *Let G, G' be (Λ-)hypergraphs. A* hypergraph morphism *(or simply* morphism*) $\varphi\colon G \to G'$ consists of a pair of functions $(\varphi_V : V_G \to V_{G'}, \varphi_E : E_G \to E_{G'})$ such that for every $e \in E_G$ it holds that $l_G(e) = l_{G'}(\varphi_E(e))$ and $\varphi_V(c_G(e)) = c_{G'}(\varphi_E(e))$.*

In the following we will simply use *graph* to denote a hypergraph.

We will work extensively with equivalence relations and one required operation is equivalence closure that turns an arbitrary relation into an equivalence.

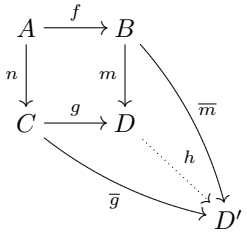
Definition 3 (Equivalence closure). *Let A be a set and $\mathcal{R}$ be a relation $\mathcal{R} \subseteq A \times A$. The equivalence closure $\overline{\mathcal{R}}$ of $\mathcal{R}$ is the smallest equivalence containing $\mathcal{R}$.*

In the following equivalence closure is mainly used if $\mathcal{R}$ is the union of two equivalences $\equiv_1, \equiv_2$ on A, i.e., $\mathcal{R} = \equiv_1 \cup \equiv_2$. In this case $\overline{\mathcal{R}}$ is simply the

transitive closure of $\equiv_1 \cup \equiv_2$ and can be written as

$$\overline{\mathcal{R}} = \{(x,y) \in A \times A \mid \exists x_1, y_1, \ldots, x_n, y_n : \\ x = x_1 \equiv_1 y_1 \equiv_2 x_2 \equiv_1 \cdots \equiv_1 y_{n-1} \equiv_2 x_n \equiv_1 y_n = y\}$$

Definition 4 (Pushout). *Let A, B, C be graphs with graph morphisms $f\colon A \to B$ and $n\colon A \to C$.*



The graph D together with $g\colon C \to D$ and $m\colon B \to D$ is a pushout of f, n if the following conditions are satisfied:

(1) $m \circ f = g \circ n$.
(2) For all $\overline{m}\colon B \to D'$, $\overline{g}\colon C \to D'$ satisfying $\overline{m} \circ f = \overline{g} \circ n$ there exists a unique morphism $h\colon D \to D'$ such that $h \circ m = \overline{m}$ and $h \circ g = \overline{g}$.

There is a well-known construction of pushouts [5] in the category of hypergraphs, where pushouts are obtained by taking the disjoint union of B and C and factoring through an equivalence obtained from the morphisms f, n.

Proposition 1 (Pushout via equivalence classes). *Let A, B, C be graphs with graph morphisms $f\colon A \to B$, $n\colon A \to C$. We call $A = (V_A, E_A, c_A, l_A)$ the interface. We also assume that all node and edge sets are disjoint.*[3]

Let $\equiv$ be the equivalence closure of the relation $\tilde{\equiv}$ on $V_B \cup E_B \cup V_C \cup E_C$ which is defined as $f(x) \tilde{\equiv} n(x)$ for all $x \in V_A \cup E_A$.

The gluing of B, C over A (written as $D = (B \oplus C)/\equiv$) is defined as $D = (V, E, c, l)$ with:

- $V = (V_B \cup V_C)/\equiv$,
- $E = (E_B \cup E_C)/\equiv$,
- $c\colon E \to V^*$ *where* $c([e]_\equiv) = [v_1]_\equiv \ldots [v_k]_\equiv$ *and* $v_1 \ldots v_k = \begin{cases} c_B(e) \text{ if } e \in E_B \\ c_C(e) \text{ if } e \in E_C \end{cases}$
- $l\colon E \to \Lambda$ *where* $l([e]_\equiv) = \begin{cases} l_B(e) \text{ if } e \in E_B \\ l_C(e) \text{ if } e \in E_C \end{cases}$

The resulting morphisms are $m\colon B \to D$, $g\colon C \to D$ with:

$$g(x) = [x]_\equiv \qquad\qquad m(x) = [x]_\equiv$$

Then D together with the morphisms g, m is the pushout of f, n.

[3] Disjointness can be achieved easily by renaming.

Definition 5 (Pushout complement). *Given morphisms $f\colon A \to B$, $m\colon B \to D$ a* pushout complement *of f, m is a graph C and a pair of morphisms $n\colon A \to C$, $g\colon C \to D$ such that g, m form the pushout of f, n. We say that two pushout complements C_i with $n_i\colon A \to C_i$, $g_i\colon C_i \to D$ for $i = 1, 2$ are* isomorphic *if there exists an isomorphism $j\colon C_1 \to C_2$ with $j \circ n_1 = n_2$ and $g_2 \circ j = g_1$.*

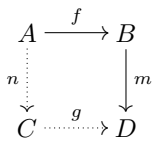
There is a well-known characterization of the existence of pushout complements (see for instance Proposition 3.3.4 of [2]).

Proposition 2 (Existence of pushout complements). *A pushout complement of f, m exists if and only if the following two conditions are satisfied:*

- Identification condition: *for all $x, y \in V_B \cup E_B$ with $m(x) = m(y)$ there exist $x', y' \in V_A \cup E_A$ with $f(x') = x$, $f(y') = y$.*
- Dangling condition: *for every node $v \in V_B$ where $m(v)$ is attached to an edge $e \in E_D$ which is not in the range of m, there exists a node $v' \in V_A$ with $f(v') = v$.*

3 Construction of pushout complements

In this section we will give a concrete construction for pushout complements, i.e., given morphisms $f\colon A \to B$ and $m\colon B \to D$, we construct *all* pairs of morphisms $n\colon A \to C$, $g\colon C \to D$ (up to isomorphism) such that the resulting square is a pushout.



We use the following abbreviations: since it is often not necessary to distinguish between edges and nodes of a graph, we will use $x \in A$ as shorthand for ($x \in E_A$ or $x \in V_A$) and $f(x)$ as shorthand for $f_V(x)$ if $x \in V_A$ and $f_E(x)$ if $x \in E_A$ respectively.

Construction 1 (Pushout complements)

(1) Construct a graph $\tilde{D}$ as follows:
- *For every node $v \in V_D$ that is not in the range of m, add a copy of v to $\tilde{D}$. The copy of v will be denoted by v'.*
- *For every edge $e \in E_D$ that is not in the range of m, add a copy of e, attached to fresh nodes, to $\tilde{D}$. (This is done also if some of the nodes attached to e are in the range of m.) The copy of e will be denoted by e' and the fresh nodes by (e', i), $i \in \{1, \ldots, ar(l_D(e))\}$.*

This means that $\tilde{D}$ is a collection of disjoint nodes and edges.

(2) Now construct $A \oplus \tilde{D}$, the disjoint union of A and $\tilde{D}$, with morphisms $n'\colon A \to A \oplus \tilde{D}$, $g'\colon A \oplus \tilde{D} \to D$ as follows:

- n' *is the canonical embedding of* A *into* $A \oplus \tilde{D}$.
- *For an item* x *of* $A \oplus \tilde{D}$ *we define* $g'(x) = m(f(x))$ *if* x *is contained in* A. *If* $x = y'$ *for some item* y *of* D *we define* $g'(x) = y$. *Finally if* $x = (e', i)$ *for some edge* e *of* D *we have* $g'((e', i)) = [c_D(e)]_i$.[4] *(See Step (1) of this construction where items of the form* y' *were created.)*

Clearly $g' \circ n' = m \circ f$.

(3) Define two equivalences on the items of $A \oplus \tilde{D}$:
- $x \equiv_{g'} y$ *if and only if* $g'(x) = g'(y)$.
- $x \equiv_f y$ *if either* $x = y$ *or* x, y *are both items of* A *and* $f(x) = f(y)$.

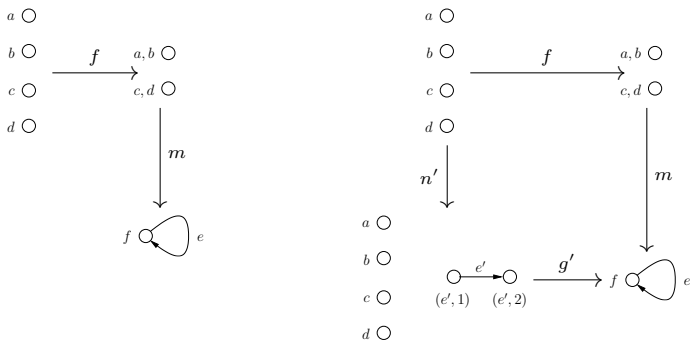
It can easily be seen that $\equiv_f$ *is a refinement of* $\equiv_{g'}$, *i.e.,* $x \equiv_f y$ *implies* $x \equiv_{g'} y$.

(4) Now consider all equivalences $\equiv'$ *on* $A \oplus \tilde{D}$ *such that* $\equiv_{g'}$ *is the equivalence closure of* $\equiv_f \cup \equiv'$. *Furthermore whenever* $e_1 \equiv' e_2$ *for two edges* e_1, e_2, *we require that* $[c_{A \oplus \tilde{D}}(e_1)]_i \equiv' [c_{A \oplus \tilde{D}}(e_2)]_i$ *for all* $1 \leq i \leq ar(l_G(e_1)) = ar(l_G(e_2))$. *For each such equivalence* $\equiv'$ *construct the graph* $C = (A \oplus \tilde{D})/\equiv'$ *with morphisms* $n\colon A \to C$, $g\colon C \to D$ *as specified below:*

$$n(x) = [n'(x)]_{\equiv'} \qquad g([x]_{\equiv'}) = g'(x)$$

Note that g *is well-defined since* $\equiv'$ *refines* $\equiv_{g'}$.

Example 1. Consider for instance the situation below on the left. We have a single binary edge, which is unlabeled (labels do not play a role for this example).



On nodes we have the equivalences $\equiv_{g'}$, $\equiv_f$, represented by their equivalence classes:

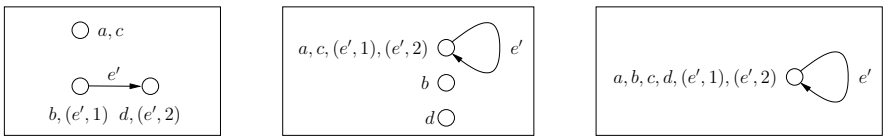
- $\equiv_{g'}$: $\{a, b, c, d, (e', 1), (e', 2)\}$
- $\equiv_f$: $\{a, b\}$, $\{c, d\}$, $\{(e', 1)\}$, $\{(e', 2)\}$

Now there are many equivalences $\equiv'$, which are possible. First, we have to relate at least one node from $\{a, b\}$ to one node from $\{c, d\}$. Furthermore we have to relate each of the two nodes $(e', 1)$, $(e', 2)$ to an equivalences class containing one of a, b, c, d. For instance the following three equivalences $\equiv'$ are all permissible:

[4] For a sequence s we denote by $[s]_i$ the i-th element of s.

– $\{a, c\}, \{b, (e', 1)\}, \{d, (e', 2), \}$
– $\{a, c, (e', 1), (e', 2)\}, \{b\}, \{d\}$
– $\{a, b, c, d, (e', 1), (e', 2)\}$

This results in the following three graphs:



But there are many more possibilities. In order to enumerate them more systematically we consider all 15 equivalences on the set $\{a, b, c, d\}$, given by equivalence classes. The ones that do not satisfy the requirement above are crossed out.

$\{a, b, c, d\}$ $\quad$ $\{a\}, \{b, c, d\}$ $\quad$ $\{b\}, \{a, c, d\}$ $\quad$ $\{c\}, \{a, b, d\}$ $\quad$ $\{d\}, \{a, b, c\}$
~~$\{a, b\}, \{c, d\}$~~ $\quad$ $\{a, c\}, \{b, d\}$ $\quad$ $\{a, d\}, \{b, c\}$ $\quad$ ~~$\{a, b\}, \{c\}, \{d\}$~~
$\{a, c\}, \{b\}, \{d\}$ $\quad$ $\{a, d\}, \{b\}, \{c\}$ $\quad$ $\{b, c\}, \{a\}, \{d\}$ $\quad$ $\{b, d\}, \{a\}, \{c\}$
~~$\{c, d\}, \{a\}, \{b\}$~~ $\quad$ ~~$\{a\}, \{b\}, \{c\}, \{d\}$~~

Now for k equivalence classes there are k^2 possibilities to associate $(e', 1)$ and $(e', 2)$ to these equivalence classes. Hence in total there are $1 + 6 \cdot 2^2 + 4 \cdot 3^2 = 61$ equivalences. Some of them result in isomorphic graphs, however they are all non-isomorphic in the sense of Definition 5 (see also Proposition 4).

We now show that every graph C constructed as specified in Construction 1 is a pushout complement and that all pushout complements can be obtained in this way.

Proposition 3. *Assume that $f\colon A \to B$, $m\colon B \to D$ are given and that the conditions of Proposition 2 are satisfied. Then every equivalence relation $\equiv'$ created by Construction 1 generates a pushout complement.*

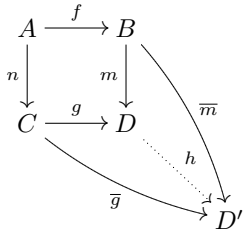
Proof. Assume that $\equiv'$ is one of the equivalences of Construction 1 and that C and n, g have been obtained by factoring $A \oplus \tilde{D}$ through this equivalence.

As a first step we show that $m \circ f = g \circ n$, i.e., the resulting square commutes: because n' is the canonical embedding of A into $A \oplus \tilde{D}$ (and therefore injective) and $g'(x)$ is defined as $m(f(x))$ if $x \in A$, $m(f(x)) = g'(n'(x))$ holds. Furthermore by definition of n, g we have:

$$m(f(x)) = g'(n'(x)) = g([n'(x)]_{\equiv'}) = g(n(x))$$

Now we show that C is indeed a pushout complement by verifying that the conditions of Definition 4 are satisfied: we have to prove that for every other commuting pair of morphisms $\overline{g}\colon C \to D'$, $\overline{m}\colon B \to D'$ there is a unique

morphism $h\colon D \to D'$ such that $h \circ g = \overline{g}$ and $h \circ m = \overline{m}$.



We define the required morphism h as follows:

$$h(x) = \begin{cases} \overline{g}(\tilde{x}) \text{ if } \exists \tilde{x} \in C : g(\tilde{x}) = x \\ \overline{m}(\tilde{x}) \text{ if } \exists \tilde{x} \in B : m(\tilde{x}) = x \end{cases}$$

It remains to be shown that h is a well-defined morphism, and that it is the unique morphism such that the triangles commute.

Commutativity. By definition $h(m(x)) = \overline{m}(x)$ and $h(g(x)) = \overline{g}(x)$ hold.

Uniqueness. Let h' be another morphism with $h' \circ g = \overline{g}$ and $h' \circ m = \overline{m}$. Due to the definition of g each element of D has a preimage either under g or m.

(1) if $x = g(x')$ then $h'(x) = h'(g(x')) = \overline{g}(x') = h(g(x')) = h(x)$
(2) if $x = m(x')$ then $h'(x) = h'(m(x')) = \overline{m}(x') = h(m(x')) = h(x)$

Well-definedness. As seen before h is defined for all elements of D. To show well-definedness it is therefore only necessary to prove that different $\tilde{x}$ having the same image under g or m also have the same image under $\overline{g}$ or $\overline{m}$.

Every element of C is an equivalence class of $\equiv'$. Therefore, let $x = [x']_{\equiv'}$ and $y = [y']_{\equiv'}$. In the following we do not strictly distinguish between an element of A and its image under n' because n' is a canonical embedding. Hence for $x' \in A \oplus \tilde{D}$ the property $x' \in A$ holds if and only if x' has a preimage under n'.

The *first property* we show is that $g(x) = g(y) \Rightarrow \overline{g}(x) = \overline{g}(y)$ holds for all $x, y \in C$. For $x \neq y$ there are two cases which have to be considered:

(1) $x', y' \in A$, i.e., we assume that the equivalence classes x, y have representatives in A (which also implies $n(x') = x$ and $n(y') = y$). We distinguish further subcases:
   (a) Case $f(x') = f(y')$

$$\begin{array}{ccccc} f(x') = f(y') & \Rightarrow & \overline{m}(f(x')) = \overline{m}(f(y')) & \Rightarrow \\ \overline{g}(n(x')) = \overline{g}(n(y')) & \Rightarrow & \overline{g}(x) = \overline{g}(y) \end{array}$$

   (b) Case $f(x') \neq f(y') \Rightarrow x' \not\equiv_f y'$ because $x' \neq y'$.
   $x' \equiv_{g'} y'$ because of $g'(x') = g([x']_{\equiv'}) = g(x) = g(y) = g([y']_{\equiv'}) = g'(y')$.
   Due to this equivalence there are $x_1, y_1, \ldots, x_n, y_n \in A$ such that $x' \equiv_f$

x_1, $x_i \equiv' y_i$, $y_i \equiv_f x_{i+1}$ and $y_n \equiv_f y'$ for $1 \leq i < n$. Using the definition of n and the fact that x_i and y_i are elements of A it can be shown that the equivalence $x_i \equiv' y_i$ implies $n(x_i) = [n'(x_i)]_{\equiv'} = [n'(y_i)]_{\equiv'} = n(y_i)$. These properties lead to the following equality

$$\overline{m}(f(x_i)) = \overline{g}(n(x_i)) = \overline{g}(n(y_i)) = \overline{m}(f(y_i)) = \overline{m}(f(x_{i+1}))$$

for every i. Together with the equalities $\overline{g}(n(x')) = \overline{m}(f(x')) = \overline{m}(f(x_1))$ and $\overline{g}(n(y_n)) = \overline{m}(f(y_n)) = \overline{m}(f(y')) = \overline{g}(n(y'))$ it follows that $\overline{g}(x) = \overline{g}(y)$.

(2) x contains no elements of A (implying $x' \notin A$)
Because x contains no elements of A, it also has no preimage under n. As already shown $g([x']_{\equiv'}) = g([y']_{\equiv'})$ implies $x' \equiv_{g'} y'$. Because of this equivalence there are $x_1, y_1, \ldots, x_n, y_n \in A$ satisfying $x' \equiv_f x_1$, $x_i \equiv' y_i$, $y_i \equiv_f x_{i+1}$, $y_n \equiv_f y'$ for $1 \leq i < n$. Due to the definition of $\equiv_f$ it holds that $x' = x_1$ because x' is not in A. Also y_1 can not be an item of A because otherwise $[x']_{\equiv'}$ would contain items of A. This property can be extended to $y_i = x_{i+1}$ and $y_n = y'$, which leads to $x_i \equiv' x_{i+1}$. Because of $x' = x_1$ and $x_n \equiv' y'$, x' and y' are equivalent according to $\equiv'$ and hence x and y must be equal. This clearly implies $g(x) = g(y) \Rightarrow \overline{g}(x) = \overline{g}(y)$.

The *second property* needed for well-definedness is $m(x) = m(y) \Rightarrow \overline{m}(x) = \overline{m}(y)$. The identification condition (see Proposition 2) states that because of $m(x) = m(y)$ there are $x', y' \in A$ such that $f(x') = x$ and $f(y') = y$. Using this and the first property the desired equality can easily be shown by:

$$\begin{array}{lllllll} m(x) = m(y) & \Rightarrow & m(f(x')) = m(f(y')) & \Rightarrow & g(n(x')) = g(n(y')) & \Rightarrow \\ \overline{g}(n(x')) = \overline{g}(n(y')) & \Rightarrow & \overline{m}(f(x')) = \overline{m}(f(y')) & \Rightarrow & \overline{m}(x) = \overline{m}(y) & \end{array}$$

The *last property* to show is $g(x) = m(y) \Rightarrow \overline{g}(x) = \overline{m}(y)$. We first show that $g(x) = m(y)$ implies that there is a y' with $f(y') = y$: the only items of D which are in the range of both g and m are the images of elements of A and nodes in the range of m which are attached to edges which are not in the range of m. However, due to the dangling condition (see Proposition 2) such nodes must have a preimage in A. Together with the first property this implies:

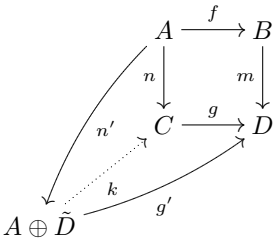
$$\begin{array}{lllllll} g(x) = m(y) & \Rightarrow & g(x) = m(f(y')) & \Rightarrow & g(x) = g(n(y')) & \Rightarrow \\ \overline{g}(x) = \overline{g}(n(y')) & \Rightarrow & \overline{g}(x) = \overline{m}(f(y')) & \Rightarrow & \overline{g}(x) = \overline{m}(y) & \end{array}$$

Morphism. Finally it is straightforward to prove that h satisfies indeed the morphism properties. For instance in order to show that $h(c_D(e)) = c_{D'}(h(e))$ for an edge $e \in D$ we have to distinguish two cases: if there exists an edge $\tilde{e} \in C$ with $g(\tilde{e}) = e$, then—since g is a morphism—we have $g(c_C(\tilde{e})) = c_D(e)$. Hence $h(c_D(e)) = h(g(c_C(\tilde{e}))) = \overline{g}(c_C(\tilde{e})) = c_{D'}(\overline{g}(\tilde{e})) = c_{D'}(h(e))$ by definition of h. The case $\tilde{e} \in B$ with $m(\tilde{e}) = e$ is analogous.

This proves that every diagram formed by an equivalence generated in the given construction is a pushout diagram. □

Proposition 4. *Assume that $f\colon A \to B$, $m\colon B \to D$ are given. Then every pushout complement $n\colon A \to C$, $g\colon C \to D$ of f, m can be obtained using Construction 1. Furthermore two isomorphic pushout complements give rise to the same equivalence $\equiv'$.*

Proof. Now assume that C with morphisms n, g is a pushout complement of f, m. We will show that there is an equivalence $\equiv'$, as specified by Construction 1, such that C is obtained by factoring $A \oplus \tilde{D}$ through this equivalence.



For the given pushout of f,n we will define a surjective morphism $k\colon A \oplus \tilde{D} \to C$ (see diagram above). Our next step is then to define an equivalence relation $\equiv'$ where $x, y \in A \oplus \tilde{D}$ are equivalent if and only if $k(x) = k(y)$. The factorization of $A \oplus \tilde{D}$ through $\equiv'$ then results in C and it has to be shown that the equivalence relation $\equiv'$ is one of the equivalence relations obtained by the presented construction.

Let $\equiv$ be the equivalence closure of the relation $\hat{\equiv}$ where $f(a) \hat{\equiv} n(a)$ for all $a \in A$. Due to the construction of pushouts using equivalence classes we can assume without loss of generality that $D = (B \oplus C)/\equiv$ (see Proposition 1). Furthermore for $b \in B$ we have $m(b) = [b]_{\equiv}$ and for $c \in C$ we have $g(c) = [c]_{\equiv}$.

We define k as follows: if $x \in A$, then $k(x) = n(x)$. If x is of the form y' for some item y of D, then — since y is not in the image of m — there must be a $c \in C$ with $g(c) = y$. In this case we define $k(x) = c$. If x is of the form (e', i) for some edge e of D, then $k(x) = [c_C(k(e))]_i$.

Well-definedness. Problems with well-definedness may arise only in the second case of the definition of k, where x is of the form y' for some item y of D. In this case y is not in the range of m due to the construction of $A \oplus \tilde{D}$. Therefore y as an equivalence class does not contain elements of B. Because of the definition of $\equiv$ every equivalence class containing elements of either B or C (but not both) only contains one element, hence y contains exactly one element c of C. Because $g(c) = [c]_{\equiv} = y$ the preimage of y under g is unique and therefore $k(x)$ is well-defined in this case.

Morphism. Note that k is obviously a morphism on the elements of A. Furthermore $\tilde{D}$ is a disjoint collection of nodes and edges and the third case in the definition of k ensures that it is indeed a valid morphism.

Surjectivity. We now show that k is surjective. Let therefore $c \in C$ be any element of C and we distinguish the following two cases:

(1) $\exists y \in A\colon n(y) = c$: By definition $k(y) = n(y) = c$.
(2) $\nexists y \in A\colon n(y) = c$: Without a preimage under n the equivalence class $[c]_{\equiv}$ contains only c because c is not equivalent to any element of B according to $\equiv$. Therefore $[c]_{\equiv}$ is not in the range of m since otherwise the equivalence class would contain elements of B. Because of the definition of k there is a $y' \in \tilde{D}$ with $g'(y') = y = [c]_{\equiv} = g(c)$, hence $k(x) = c$.

Commutativity. We have to show that both triangles commute:

(1) We first check that $k(n'(x)) = n(x)$ for any $x \in A$:
As already seen $n'(x) = x$ if $x \in A$. Using the definition of k we obtain $k(n'(x)) = k(x) = n(x)$.
(2) Now we show that $g(k(x)) = g'(x)$ for any $x \in A$. There are two cases:
   (a) $x \in A$: Using $k(x) = n(x)$ if $x \in A$ and $m \circ f = g' \circ n'$ due to the definition of g' and n' it can be shown that:
   $g(k(x)) = g(n(x)) = m(f(x)) = g'(n'(x)) = g'(x)$
   (b) $x \in \tilde{D}$: In this case $k(x) = c$ and $g(c) = g'(x)$, therefore $g(k(x)) = g(c) = g'(x)$.

The equivalence $\equiv'$ is generated. We will now show that $\equiv'$ is generated by the given construction. Specifically we have to show that the equivalence closure of $\equiv' \cup \equiv_f$ is $\equiv_{g'}$, i.e., that $\overline{\equiv' \cup \equiv_f} = \equiv_{g'}$.

– $\overline{\equiv' \cup \equiv_f} \subseteq \equiv_{g'}$:
The equivalence $\equiv_f$ is clearly a subset of $\equiv_{g'}$ because $g'(x) = m(f(x))$ if $x \in A$. Having the same image under f therefore implies having the same image under g'.
The equivalence $\equiv'$ is also a subset of $\equiv_{g'}$ because of:

$$x \equiv' y \Rightarrow k(x) = k(y) \Rightarrow g'(x) = g(k(x)) = g(k(y)) = g'(y)$$

– $\overline{\equiv' \cup \equiv_f} \supseteq \equiv_{g'}$:
Let x, y be elements of $A \oplus \tilde{D}$ with $x \equiv_{g'} y$, hence $g'(x) = g'(y)$. As shown above the equivalence classes $g'(x)$ and $g'(y)$ of $\equiv$ contain $k(x)$ and $k(y)$ respectively, therefore $k(x) \equiv k(y)$. Hence there are $c_0, b_1, c_1, \ldots b_m, c_m$ such that $b_i \mathrel{\tilde{\equiv}} c_i$ for $1 \leq i \leq m$ and $b_{j+1} \mathrel{\tilde{\equiv}} c_j$ for $0 \leq j < m$ with $k(x) = c_0$ and $k(y) = c_m$. Using the definition of $\tilde{\equiv}$ leads to the following properties:

$$\begin{aligned} b_i \mathrel{\tilde{\equiv}} c_i \quad &\Rightarrow \quad \exists a_i \in A\colon f(a_i) = b_i \wedge n(a_i) = c_i \\ b_{i+1} \mathrel{\tilde{\equiv}} c_i \quad &\Rightarrow \quad \exists a'_i \in A\colon f(a'_i) = b_{i+1} \wedge n(a'_i) = c_i \end{aligned}$$

It can be inferred that a_{i+1} and a'_i have the same image under f, hence $a_{i+1} \equiv_f a'_i$, and that a_i and a'_i have the same image under n, hence $a_i \equiv' a'_i$. This leads to $x \equiv' a'_0 \equiv_f a_1 \equiv' a'_1 \equiv_f \cdots \equiv' a'_{m-1} \equiv_f a_m \equiv' y$, hence $x \overline{\equiv' \cup \equiv_f} y$.

This proves that every pushout complement can be obtained by using the given construction.

Isomorphism of pushout complements. It is left to show that, given two isomorphic pushout complements $n_i\colon A \to C_i$, $g_i\colon C_i \to D$ with $i = 1, 2$ and an isomorphism $j\colon C_1 \to C_2$ with $j \circ n_1 = n_2$, $g_2 \circ j = g_1$, the corresponding equivalences $\equiv'$ are the same. For this it is sufficient to show that j commutes with the morphisms k_1, k_2, where $k_i\colon A \oplus \tilde{D} \to C_i$ and k_1, k_2 are constructed analogously to the morphism k above. That is, we have to show that $j \circ k_1 = k_2$. Then k_1, k_2 give rise to the same equivalence $\equiv'$.

We distinguish the following cases (as in the definition of k): if $x \in A$, then $j(k_1(x)) = j(n_1(x)) = n_2(x) = k_2(x)$. If x is of the form y' for some item y of D, then we define $k_i(x) = c_i$ for c_i with $g_i(c_i) = y$. Since $g_2(j(c_1)) = g_1(c_1) = y$ we obtain $c_2 = j(c_1)$. Hence $j(k_1(x)) = j(c_1) = c_2 = k_2(x)$. Finally, if x is of the form (e', ℓ) for some edge e of D, then $k_i(x) = [c_C(k_i(e))]_\ell$ and so $j(k_1(x)) = j([c_C(k_1(e))]_\ell) = [c_C(j(k_1(e)))]_\ell = [c_C(k_2(e))]_\ell = k_2(x)$. This completes the proof. □

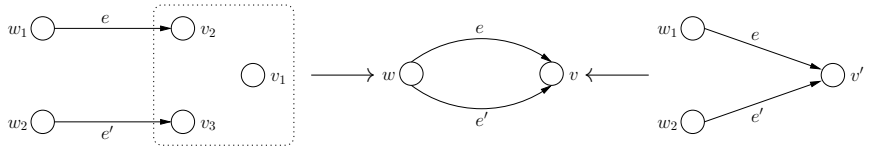
The fact that two isomorphic pushout complements give rise to the same equivalence means that the number of generated (valid) equivalences is exactly the number of different pushout complements. However, if we consider only isomorphisms on C—without requiring commutativity of the triangles consisting of morphisms j, n_1, n_2 and j, g_1, g_2 (in the terminology of Definition 5)—there will usually be fewer different pushout complements. The examples in Section 5 are chosen in such a way that both interpretations give rise to the same number.

4 Optimizations

In the given construction there exist several possibilities for optimization. These lie in the construction of $A \oplus \tilde{D}$ and in the method used to enumerate all possible equivalences $\equiv'$.

4.1 Possible Simplifications

In Step (1) of Construction 1 the graph $\tilde{D}$ is constructed by inserting all nodes and edges of D which are not in the range of m. Additionally for every edge e of D for every node connected with e a new node is inserted. This ensures that every node attached to e is also in $\tilde{D}$. However, if e is connected to a node x not in the range of m, another copy of this node has been added earlier to $\tilde{D}$. Both are equivalent with respect to $\equiv_{g'}$ but not with respect to $\equiv_f$ since they do not have a preimage under n'. Therefore these two copies have to be equivalent according to every possible equivalence $\equiv'$. Hence the first copy was superfluous and it was unnecessary to create it in the first place.



The previous diagram shows an example graph $\tilde{D}$ generated by the given construction if the middle graph is D and only w is in the range of m, but not in the range of $m \circ f$. In the left graph v_1, v_2 and v_3 are all copies of v in the middle graph and all have to be in the same $\equiv'$-class. The construction would therefore still be correct if the right graph is generated instead of the left graph.

In general it is only necessary to add one node to $A \oplus \tilde{D}$ for every node not in the range of m and for every node in the range of m as many nodes as there are edges not in the range of m connected with the node. This improvement can help to manage the combinatorial explosion when determining all possible equivalences $\equiv'$.

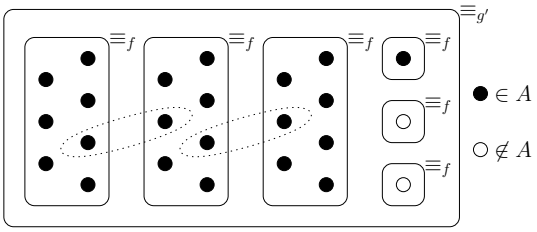
4.2 Enumerating Equivalences

A problem not addressed earlier is how to generate all permissible equivalences $\equiv'$. The straightforward way would be to enumerate all possible equivalences over $A \oplus \tilde{D}$ and to store every equivalence satisfying $\overline{\equiv' \cup \equiv_f} = \equiv_{g'}$. This method is however not recommended because of combinatorial explosion. Furthermore many of these equivalences will not satisfy the required conditions. In the following we explain how the generation of equivalences could be handled more efficiently.

If f is injective there is only one permissible equivalence $\equiv'$. This is true since in this case g must necessarily also be injective and hence $\equiv'$ equals $\equiv_{g'}$.

A non-injective morphism f produces several permissible equivalences $\equiv'$. In this case it is sufficient to look at each equivalence class of $\equiv_{g'}$ separately. We further distinguish between equivalence classes which contain elements of A and those which do not. In either case every equivalence class of $\equiv_f$ is entirely contained in exactly one equivalence class of $\equiv_{g'}$ due to the definition of g'.

If an equivalence class c of $\equiv_{g'}$ contains no elements of A, every equivalence class of $\equiv_f$ contained in c only contains one element. Therefore c must also be an equivalence class of $\equiv'$, i.e., all elements of c must be merged.



If an equivalence class c of $\equiv_{g'}$ contains elements of A, the equivalence classes of $\equiv_f$ in c contain either only elements of A or no elements of A (see figure above). Only equivalence classes of $\equiv_f$ containing elements of A can consist of more than one element. Elements already equivalent according to $\equiv_f$ do not have to be equated via $\equiv'$ because they will anyway be equivalent after the equivalence closure. It is however necessary to add relations between elements in such

a way that the resulting structure connects all equivalence classes to each other, possibly indirectly. (One such possibility connecting the three leftmost equivalence classes is indicated by the dashed ovals in the figure above.) Therefore, in order to calculate all permissible equivalences $\equiv'$ for all elements of c, we first enumerate all equivalences over elements contained in equivalence classes of $\equiv_f$ with more than two elements, but keep only those that induce connectivity. We then distribute the remaining elements (contained in equivalence classes of $\equiv_f$ with only one element) to the resulting equivalence classes in every possible way. The results are all equivalences $\equiv'$ restricted to elements of c. If we perform these steps for all other equivalence classes of $\equiv_{g'}$, a complete equivalence $\equiv'$ can be obtained by taking arbitrary combinations of such (restricted) equivalences $\equiv'$ for each class c.

5 Combinatorial Interpretation

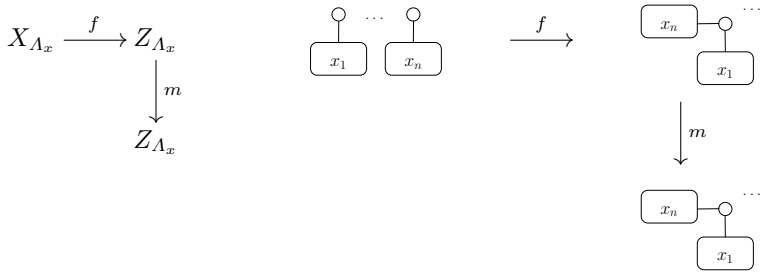
Some coefficients from combinatorics arise naturally as the number of pushout complements for a (parameterized) pair of arrows. We now present some examples, all of them for hypergraphs with unary edges only.

5.1 Bell Numbers

The n-th Bell number B_n is the number of equivalence relations on the set $\{1, \ldots, n\}$. The first Bell numbers (starting with B_1) are: 1, 2, 5, 15, 52, 203, 877, 4140, ... (see the On-Line Encyclopedia of Integer Sequences which can be queried at `http://www.research.att.com/~njas/sequences/`).

Now take $\Lambda_x = \{x_1, \ldots, x_n\}$ as a label set. Assume that X_{Λ_x} is the graph with n nodes, where to each node we attach a unary hyperedge and each hyperedge has a different label. Furthermore Z_{Λ_x} is the graph with one node to which n hyperedges are attached, where each hyperedge has a different label.

We consider the unique morphism $f\colon X_{\Lambda_x} \to Z_{\Lambda_x}$ and the identity $m = id_{Z_{\Lambda_x}}\colon Z_{\Lambda_x} \to Z_{\Lambda_x}$. Then—if we apply our construction—the graph $A \oplus \tilde{D}$ will consist only of $A = X_{\Lambda_x}$ and *all* equivalences $\equiv'$ on the nodes of X_{Λ_x} are admissible (for the edges each edge must be in a separate equivalence class). Hence there are B_n different pushout complements up to isomorphism.

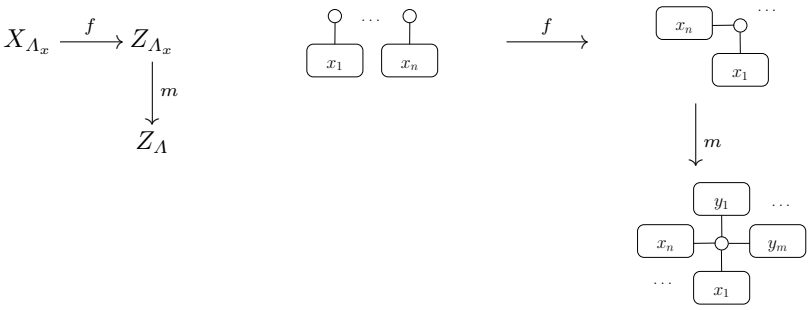


5.2 Stirling Numbers of the Second Kind

The Stirling number of the second kind $S_{n,k}$ is the number of equivalence relations with k equivalence classes on the set $\{1, \ldots, n\}$. It holds that $B_n = \sum_{k=1}^{n} S_{n,k}$.

The Stirling numbers satisfy the following recursive equation: $S_{n,k} = S_{n-1,k-1} + k \cdot S_{n-1,k}$, which is based on a case distinction according to the element n: either n is in an equivalence class of its own and the remaining $n-1$ elements have to be grouped in $k-1$ equivalence classes; or the remaining $n-1$ elements have to be grouped in k equivalence classes and there are k possibilities to assign n to one of these classes. Our implemented method for enumerating equivalences follows the same pattern.

Now we set $\Lambda_x = \{x_1, \ldots, x_n\}$, $\Lambda_y = \{y_1, \ldots, y_m\}$ and $\Lambda = \Lambda_x \cup \Lambda_y$. We take the unique morphism $f\colon X_{\Lambda_x} \to Z_{\Lambda_x}$ and the unique morphism $m\colon Z_{\Lambda_x} \to Z_{\Lambda}$.



Then $A \oplus \tilde{D}$ is the disjoint union of X_{Λ_x} and separate copies of m edges which are labelled $y_1, \ldots, y_m$. Now we take all permissible equivalences on the nodes of the copy of X_{Λ_x}. Assume that we have k equivalence classes. Then there are k^m possibilities to distribute the m nodes of the separate edges over the equivalence classes. Hence the total number of pushout complements is

$$\sum_{k=1}^{n} S_{n,k} \cdot k^m$$

Note that for the special case of $m = 0$ we obtain again the Bell numbers. Another special case is $n = 2$, for which we obtain $S_{2,0} \cdot 0^m + S_{2,1} \cdot 1^m + S_{2,2} \cdot 2^m = 1 + 2^m$ pushout complements.

6 Conclusion

We have shown how to construct pushout complements in the category of hypergraphs in the general case when both given morphisms might be non-injective.

Such a construction is necessary for performing backwards analysis and computing the set of predecessors of a given graph. We have implemented this construction (in a tool that performs backwards search in well-structured transition systems, based on [7]) and we presented the optimizations that we used in the implementation.

Concerning combinatorics it would be interesting to have a general formula that directly computes the number of pushout complement for an arbitrary pair f, m of morphisms. However, the computation seems to be quite involved.

It is unclear to us whether the construction could be transferred to a more categorical setting, similar to [1]. However, our main intention was to obtain an efficient implementation.

Acknowledgements: We would like to thank Benjamin Braatz for our discussions on this topic.

References

1. Benjamin Braatz, Ulrike Prange, and Thomas Soboll. How to delete categorically – two pushout complement constructions. Unpublished, 2009.
2. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation—part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, chapter 3. World Scientific, 1997.
3. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation—part II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 4. World Scientific, 1997.
4. H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: An algebraic approach. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 167–180, 1973.
5. Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. 1st International Workshop on Graph Grammars*, pages 1–69. Springer-Verlag, 1979. LNCS 73.
6. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
7. Salil Joshi and Barbara König. Applying the graph minor theorem to the verification of graph transformation systems. In *Proc. of CAV '08*, pages 214–226. Springer, 2008. LNCS 5123.
8. Yasuo Kawahara. Pushout-complements and basic concepts of grammars in toposes. *Theoretical Computer Science*, 77:267–289, 1990.
9. Barry K. Rosen. Deriving graphs from graphs by applying a production. *Acta Informatica*, 4:337–357, 1975.

Production Networks as Communities of Autonomous Units and Their Stability

Sergey Dashkovskiy, Hans-Jörg Kreowski, Sabine Kuske, Andrii Mironchenko, Lars Naujok, Caroline von Totth *

University of Bremen
{kreo,kuske,caro}@informatik.uni-bremen.de
{dsn,andmir,larsnaujok}@math.uni-bremen.de

Abstract. In this paper, a discrete variant of production networks is considered. Besides the mathematical models in terms of matrices and vectors, production networks are modeled as communities of autonomous units in a rule-based, graph-transformational and visual manner. Moreover, a sufficient criterion for the stability of a production network is given where stability means that there exist suitable storage capacities at the production sites that never flow over.

1 Introduction

In this paper, we consider a discrete variant of production networks (see, e.g., [1]) inspired by the work in [2–4] on continuous production networks and their stability. For a certain scenario it has been shown that the application of local autonomous control methods on integrated production and transport processes improves the handling of internal and external dynamics. A production network in this scenario consists of production sites, which are represented as nodes, and of links between sites, which are represented as directed edges. There is a particular input site with a continuous inflow. The production at each site runs continuously at some rates that are bounded by the maximum production rates and subject to suitable constraints. The processed product of each site is continuously distributed to the direct neighbors for further processing according to fixed distribution rates. Moreover, there is an output site with a continuous output which is computed in some suitable way. A production network is called stable if the quantity of products at each site is bounded all the time. In [4] conditions were derived by mathematical systems theory, which guarantee stability of the network. The calculation of these conditions is based on the work [5–7].

In the present paper, the continuity is replaced by stepwise input, production, transportation, and output. To take into account dynamic changes of the input, the input flow is not assumed to be constant. Moreover, the production rates

* The authors would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

are not determined uniquely, but may vary within certain bounds (Section 2). The discrete production networks are modeled in a graph-transformational way as communities of autonomous units [8–11] in Section 3. These rule-based and visual models allow one to use graph-transformational tools like GrGen.NET to simulate production networks in such a way that production processes are not only statistically analyzed, but also visualized displaying their smooth running or the overflow of bottlenecks (Section 4). Each production site becomes a unit that can act independently of the other sites within certain bounds. This allows one to use decentralized decision criteria for the choice of the production and distribution rates; however, this aspect will not be further addressed in the paper, being a subject of future work.

If the input rate is constant and the production rates are chosen exhaustively, meaning that the current quantities are processed completely up to the maximum production rates in each step, then the production network becomes deterministic with a unique production process. In this case, the distribution rates and the input rate induce a system of linear equations. If this linear system is solvable, then the production network turns out to be stable, as shown in Section 5. As this result applies to the graph-transformational model of production systems, the investigation introduces a new kind of analytical problems to the area of graph transformation that may be of interest beyond the topic of this paper. The question of stability is of similar interest in the discrete case as in the continuous one, because the quantities left at the production sites may grow beyond any bound so that their storage can overflow eventually.

Summarizing, the paper is structured in the following way. The discrete variant of production networks and processes is introduced in Section 2. The visual and rule-based models of production networks are specified in Section 3 in form of communities of autonomous units. Section 4 describes an implementation of our production networks in the graph transformation engine GrGen.NET that gives some first ideas of the potentials of the visual modeling. In Section 5, a sufficient condition for the stability of deterministic production networks is given based on the solvability of a system of linear equations which is induced by the distribution rates and the input quantity. Section 6 concludes the paper.

2 Production Networks and Production Processes

In this section, the notion of production networks and their processes is introduced where the input, processing, flow and output of material are not continuous, but happen step-by-step. A production network consists of production sites, which are represented as nodes, and of transportation channels between sites, which are represented as directed edges. There is one input site and one output site. In each state, the present material at each site is given as a quantity. A production step changes these quantities by distributing the production rate of each site to its direct neighbors. Moreover, the input site gets some input in each step and a part of the production rate of the output site is put out. There may be a maximum input rate, which can be chosen as ∞ if no bound is assumed.

The same applies to the production rates. The distribution at a site is done according to a distribution vector, the entries of which specify which fraction of the production rate is moved to which neighbor site. The distribution vectors of all sites form a distribution matrix. A production process starts with the initial site quantities and records the changes of site quantities depending on the input, the production rates and the distribution matrix in each step.

Let $\mathbb{N}$ denote the set of natural numbers, $\mathbb{N}_{>0}$ denote the set $\mathbb{N}\backslash\{0\}$ and let $[k]$ denote the subset $\{1,\ldots,k\}$ of $\mathbb{N}$. The set of real numbers is denoted by $\mathbb{R}$; we use $\mathbb{R}_+$ to describe the set of non-negative real numbers with 0. Moreover, $\langle X, Y\rangle$ denotes the set of mappings from a set X to a set Y.

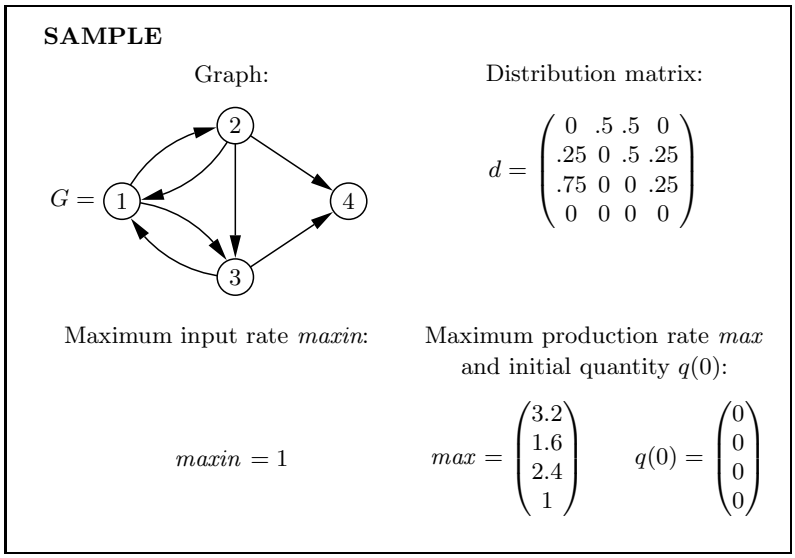


Fig. 1. An example of a production network

Definition 1 (Production Network).
A production network *PN consists of*

- *a simple directed graph* $G = ([n+1], E)$ *with* $E \subseteq [n+1] \times [n+1]$, *the* input site *1 and the* output site $n+1$,
- *an* initial site quantity $q(0)\colon [n+1] \to \mathbb{R}_+$,
- *a* maximum input rate $maxin \in \mathbb{R}_+ \cup \{\infty\}$,
- *a* maximum production rate $max\colon [n+1] \to \mathbb{R}_+ \cup \{\infty\}$, *and*
- *a* distribution matrix $d\colon [n+1]\times[n+1] \to \mathbb{R}_+$ *with* $\sum_{j\in[n+1]} d(i,j) = 1$ *for* $i \in [n]$, $\sum_{j\in[n+1]} d(n+1,j) \leq 1$ *and* $d(i,j) = 0$ *for all* $(i,j) \in [n+1]\times[n+1] - E$.

2.1 Example

A sample production network is given by the components of Figure 1.

A production network specifies stepwise production processes which go on forever. If one assumes that the input rates and the production rates can be randomly chosen (within certain limits) and that the output is always the part of the production rate of the output site which is not distributed to other sites, then one gets the following production processes.

Definition 2 (Production Process).
A production process *pp (in PN) consists of*

- *an infinite sequence of* input rates $in\colon \mathbb{N}_{>0} \to \mathbb{R}_+$,
- *an infinite sequence of production rates* $p\colon \mathbb{N}_{>0} \to \langle [n+1], \mathbb{R}_+ \rangle$,
- *an infinite sequence of output rates* $out\colon \mathbb{N}_{>0} \to \mathbb{R}_+$, *and*
- *an infinite sequence of site quantities* $q\colon \mathbb{N}_{>0} \to \langle [n+1], \mathbb{R}_+ \rangle$

subject to the following production process conditions *for all* $k \in \mathbb{N}_{>0}$:

$$in(k) \leq maxin, \tag{1}$$

$$p(k) \leq \min(q(k), max), \tag{2}$$

$$q(k)(j) = \begin{cases} in(k) + y & \text{if } j = 1 \\ y & \text{otherwise} \end{cases} \tag{3}$$

$$\text{where } y = \left(\sum_{i \in [n+1]} d(i,j) \cdot p(k)(i) \right) + q(k-1)(j) - p(k)(j),$$

$$out(k) = \left(1 - \sum_{j \in [n+1]} d(n+1, j) \right) \cdot p(k)(n+1). \tag{4}$$

The first condition makes sure that no input rate exceeds the maximum input rate. The second condition requires that the processed quantity in every step is a part of the present quantity. The third condition describes the present site quantity after every step as the site quantity before the step diminished by the production rate and expanded by the quantities moved from other sites. The latter quantities are given by the fractions of the production rates due to the distribution factors. The input site gains the input rate in addition. The last condition fixes the output in each step as the part of the production rate of the output site that is not distributed to other sites.

To get shorter formulas, we use the following notational conventions for $k \in \mathbb{N}, l \in \mathbb{N}_{>0}$ and $i, j \in [n+1]$.

1. $in_l = in(l); p_{il} = p(l)(i); d_{ij} = d(i,j); out_l = out(l); q_{il} = q(l)(i); max_i = max(i)$,
2. $In_k = \sum_{j \in [k]} in_j, Out_k = \sum_{j \in [k]} out_j$, $Q_k = \sum_{i \in [n+1]} q_{ik}$ where the convention $[0] = \emptyset$ and $\sum_{j \in \emptyset} = 0$ for $k = 0$ is used.

The definition of production processes has some immediate consequences:

(1) If in and p are given in some way, then q and out are uniquely determined and can be computed due to the required equations.
(2) As a particular case, one may consider constant input rates, e.g., $in_k = maxin$ for all $k \in \mathbb{N}_{>0}$.
(3) As a particular case, one may consider exhaustive production rates, e.g., $p_{ik} = \min(q_{i(k-1)}, max_i)$ for all $i \in [n+1]$ and $k \in \mathbb{N}_{>0}$.
(4) Production networks with constant input rate and exhaustive production rates have a unique production process. They are further discussed in Section 5.

2.2 Example

The production network **SAMPLE** in Figure 1 may run with constant maximum input rate and exhaustive production rates. Then there is a unique production process with output rates computed due to the respective constraints. It is not difficult to show that the site quantities of this production process never exceed the maximum production rates, so that the production rates always coincide with the site quantities.

2.3 Lossfreeness

The production process conditions (see Definition 2) guarantee that no input material gets lost, because the whole processed material is moved to other sites and put out in every step. This is formally stated in the following result.

Theorem 1. $Q_k = Q_0 + In_k - Out_k \;\; for\ all \;\; k \in \mathbb{N}$

As this is easily proved by induction, the proof is omitted.

3 Production Networks as Communities of Autonomous Units

In this section, production networks are modeled as communities of autonomous units in such a way that the production processes of a network correspond to the runs of the respective community. At first sight, the production community may look more complicated than the mathematical model. But it is worth noting that the rule applications in a running step correspond directly to the multiplications and additions that define a process step in the mathematical model. Moreover, the rule-based model provides an explicit description of parallel computations. The main difference between both models is that the rule-based version provides a visual level.

In the following we use edge-labeled directed graphs, the double pushout approach (see [12]), for rule application, and communities of autonomous units as defined in [11] for the structuring of rules.

3.1 The Community C(*PN*)

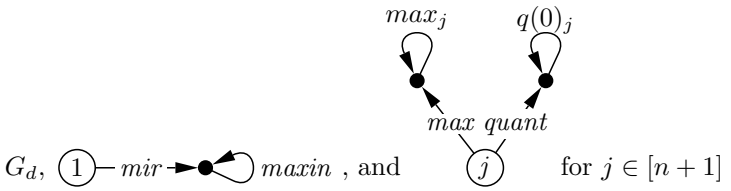
A production network PN is transformed into the community of autonomous units in Figure 2. There is an autonomous unit *j-prod* for each production site $j \in [n+1]$ and an extra *input* unit.

C(PN)	
aut:	*input*, *j-prod*, for $j \in [n+1]$
init:	$env(PN)$
control:	(*input* || *1-prod* || ... || *n+1-prod*)$^\infty$

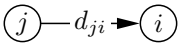
Fig. 2. The community C(PN) models the production network PN

3.2 Initial environment

The initial environment graph $env(PN)$ integrates all the information of PN. The graph $env(PN)$ consists of the subgraphs



where G_d is obtained from G by replacing each $(j,i) \in E$ by



and the subgraphs share the nodes in $[n+1]$. Moreover, we assume that each node $j \in [n+1]$ is attached with a loop labeled by j. In drawings, the loop is omitted and its label is placed inside the node. This ensures that the node j can only be mapped to itself by a graph morphism.

In summary, the node representing the input site is labeled with the number 1; the output site is labeled with n+1. Each site j has an initial quantity q_{j0} of material, indicated by a pointer edge with the label *quant*, and a maximum production rate, denoted by a pointer with the label *max*.

The maximum input rate *maxin* is attached to the input site by a special pointer labeled *mir*. The connecting edges between sites are labeled with the distribution rates.

Additionally, if the initial quantities q_{j0} are replaced by some quantities q_j for $j \in [n+1]$, then the environment graph is denoted by $env(PN)(q)$.

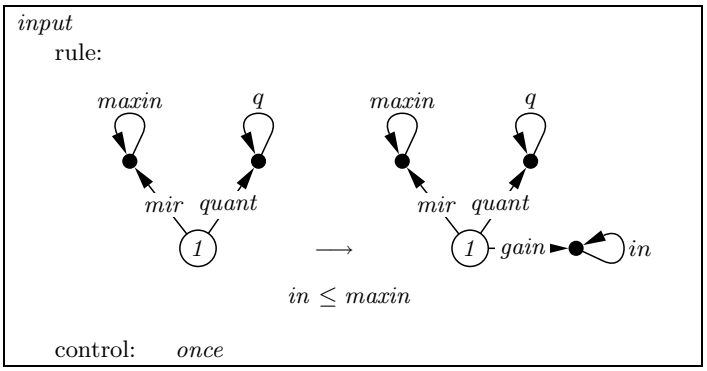


Fig. 3. The unit *input*

3.3 Autonomous unit *input*

The unit *input* in Figure 3 has only one rule, which is applied exactly once in every execution step of the community. An input value *in* is chosen by some mechanism (e.g., this may be some input function like *sine* or some stochastic function or even a constant), and a *gain* pointer is added to the site, with the value of *in* attached to it by a loop edge. The choice of *in* is restricted only insofar that its value may not exceed *maxin*.

3.4 Autonomous units *j-prod*

The parallel execution of the *j-prod* units (Figure 4) together with *input* model one production step of the network.

The tasks of the *j-prod* units are twofold: On one hand, they manage for each site the production and the distribution of material to neighbor sites. On the other hand, the fact that the distribution runs in parallel for all sites at once makes some cleanup actions necessary in preparation for the next step.

Production and distribution. The first rule of a *j-prod* unit, *produce*, chooses a production rate for site j much in the same manner as the *input* unit, by some mechanism. A *prod* pointer is added to j with the new production rate attached to it. Here, too, the choice of p is restricted by whichever is the smaller of two upper bounds: the quantity q of material present at the site and the maximal production rate max_j.

The second rule in *j-prod*, *transport*(i), is a parametric one and it is applied in one parallel step to each neighbor i of the site j. This rule moves the fraction $d_{ji} \cdot p$ of the current production rate of material at the site j to a neighboring site i, where d_{ji} is the distribution value inscribed on the edge from j to i. Rather than adding this value directly to the quantity of material already present at i, the transported value is instead attached to a *gain* pointer for the following reasons.

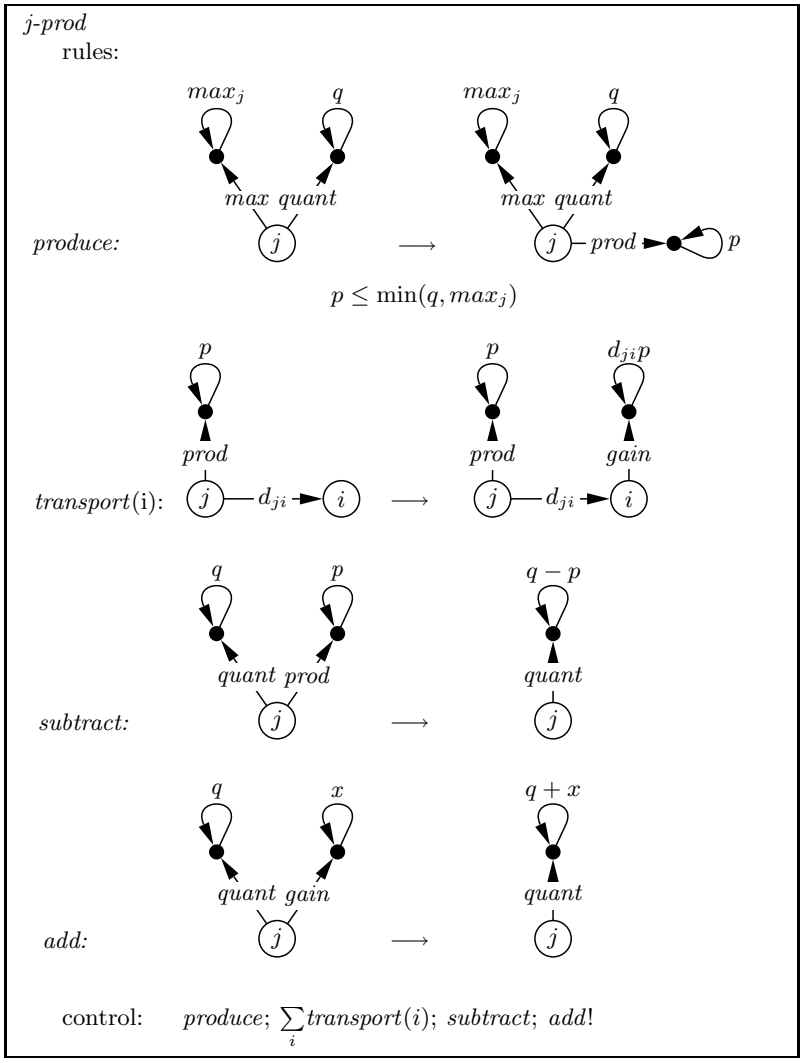


Fig. 4. The unit *j-prod*

Cleanup and update. Depending on the configuration of the network, one production site j may receive input from many neighbor sites. To avoid conflicts generated by concurrent access to the quantity value q of j, each source generates a *gain* edge at the target site, labeled with the appropriate value. Now in order to make a next step in the production process possible, a cleanup of sorts needs to take place; this is the task of the rules *subtract* and *add*, which restore the original pointer configuration and update all values to reflect the changes that have taken place in the current production step. First, the *subtract* rule removes the amount p_j of material which is distributed by j in the current production step from the quantity q_j of material present at j, resulting in an intermediate quantity value $\bar{q_j}$. Implicitly, this behavior also models the output of material from the network: the difference between p_{n+1} and the amount of material the site $n+1$ sends to other sites simply disappears from the network. If, in particular, the output site has no outgoing edges, then its whole production rate leaves the network in every running step. Additionally, *subtract* also removes the *prod* pointer at each site, so that a new value for p_j can be entered into the network in the next production step. The rule *subtract* is applied exactly once, after the transport has been completed and before the application of *add*. Now, the values in the *gain* edges at each site plus the remaining quantity $\bar{q_j}$ at the site have to be consolidated into one single quantity value $q'_j = \sum d_{ij}p_i + \bar{q_j}$. This is done in the *add* rule by picking a random gain edge and adding its value to the existing quantity: the exclamation mark in the control expression *add*! requires that *add* must be applied as long as possible, i.e., until no *gain* edge remains.

3.5 Control condition

The control condition of $C(PN)$ prescribes to execute the *input* and all of the *j-prod* units in parallel (denoted by $\|$) and to iterate this ad infinitum (denoted by ∞).

Summarizing, the following observation relates a running step in the community with a process step in the mathematical model.

Observation. A running step of the community $C(PN)$ has the form $env(PN)(q) \Longrightarrow env(PN)(q')$ where q' is obtained from q by

$$q'_j = in \cdot \delta_{1j} + \sum d_{ij}p_i + q_j - p_j$$

with $\delta_{11} = 1$ and $\delta_{1j} = 0$ for $j > 1$.

As a consequence of this observation, we get the following result.

Theorem 2. *Each production process pp in PN with the sequence of site quantities* $q : \mathbb{N} \to \langle [n+1], \mathbb{R}_+ \rangle$ *corresponds to an infinite run of the community* $C(PN)$ *with the steps* $env(PN)(q_k) \Longrightarrow env(PN)(q_{k+1})$ *for all* $k \in \mathbb{N}$ *and conversely.*

This shows that $C(PN)$ models PN correctly.

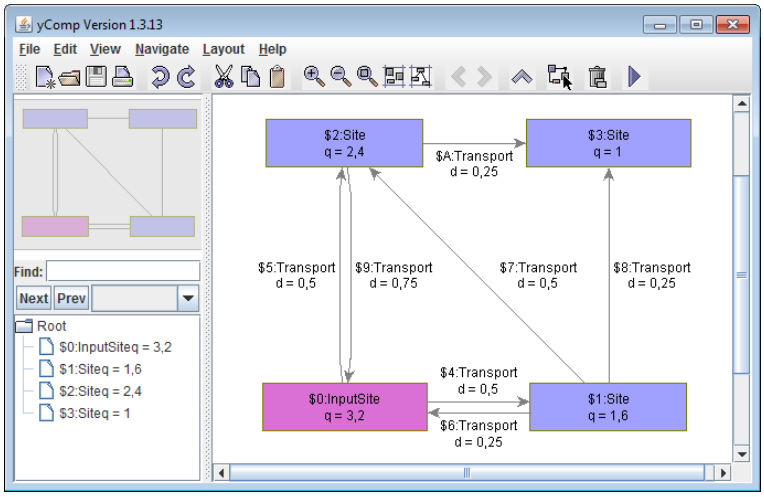


Fig. 5. Community C(SAMPLE) in GrGen.NET: all sites have reached the saturation point (i.e., the maximal production rate) after 217 steps

4 Visual Simulation

In order to simulate runs on our sample production network as well as larger production networks, we have implemented the general production community from Section 3 using the graph transformation engine GrGen.NET (see [13]).

The GrGen.NET graph model is based on typed, attributed, directed multigraphs with inheritance. The base types at the core of this model are Node and Edge, and the primitive attribute data types int, float, double, string, boolean and object, the latter denoting a .NET object.

We made use of the subpattern matching capability of GrGen.NET, using the iterated subpattern in order to simulate parallel rule application. GrGen.NET also does not provide autonomous units; however, it allows to structure rule application by embedding imperative calls to other rules into the declarative right-hand-side of a rule. Furthermore, such calls may be controlled using, for example, regular expressions. We made use of this feature to emulate autonomous units very closely to our original specification.

The simulation runs very fast, with our example network **SAMPLE** completing 217 steps and reaching the maximal production rate at all four sites in less than 1 millisecond (GrGen gives the time as 0 ms) on an Intel Core i5 M520 CPU with 2.40 GHz and 6 GB of RAM, having found 4340 matches and performed 4340 graph rule applications in that time.

In order to test run times on larger networks (Figure 6), we have written an additional graph grammar which creates random production networks for simulation purposes. A graph with 402 nodes is generated in 655 ms; 3000 production steps are completed after another 21840 ms (i.e., some 21 seconds), with over 4 million matches found and rewrite steps executed in that time.

The simulation is valuable as a visual way to model and debug production networks or detect flaws in existing ones, altering them until they are stable. Additionally, the declarative nature of graph transformation rules makes the modeling less error-prone, and the production process model easily scalable, e.g., by introducing different material types, variable inflow and other extensions.

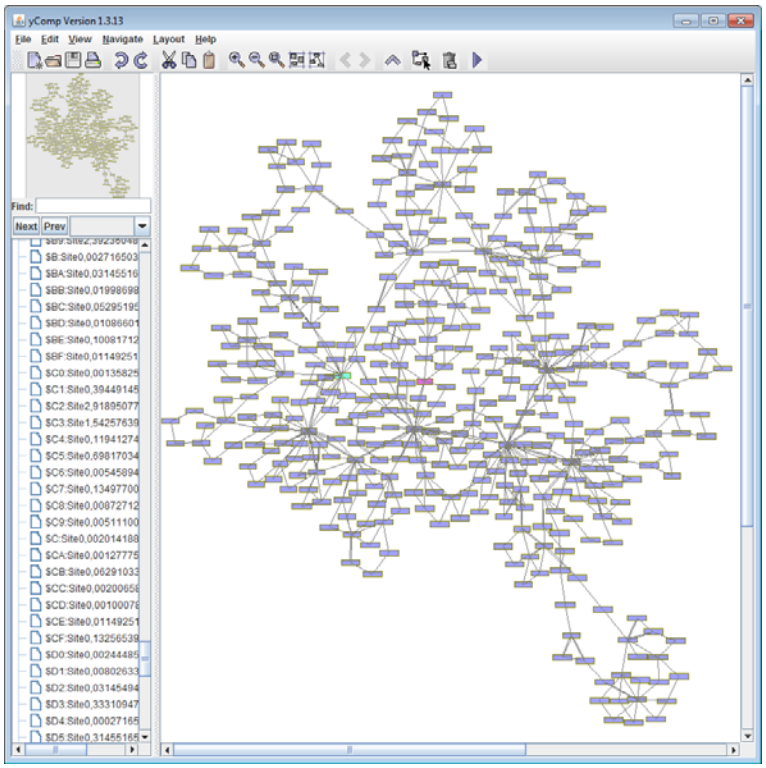


Fig. 6. A network with 400 nodes in GrGen.NET after 3000 production steps

5 Deterministic Production Networks and Stability

In practice, a site in a production network has only a bounded storage capacity so that the question of stability becomes important. A production network is stable if the site quantities of each production process do not exceed a fixed bound. It will be shown in this section that deterministic production processes, which have a constant input rate and exhaustive production rates, are stable if a certain system of linear equations is solvable.

A production network PN is *stable* if an upper bound vector $m : [n+1] \to \mathbb{R}_+$ exists such that the following holds for each production process pp with the site quantity sequence $q : \mathbb{N} \to \langle\, [n+1], \mathbb{R}_+ \rangle$: $q_{ik} \leq m_i$ for all $i \in [n+1]$ and $k \in \mathbb{N}$.

A production network PN is *deterministic* if the input rate is constant, i.e., $in_k = in$ for all $k \in \mathbb{N}_{>0}$ and for some $in \in \mathbb{R}_+$, and if the production rates are exhaustive, i.e., $p_{ik} = \min(q_{i(k-1)}, max_i)$ for all $k \in \mathbb{N}_{>0}$ and $i \in [n+1]$.

Let PN be a deterministic production network, and let its unique production process have – in addition – a constant site quantity, meaning that there is a quantity vector $m\colon [n+1] \to \mathbb{R}_+$ with $q_k = m$ for all $k \in \mathbb{N}$. Let moreover the vector m be smaller than or equal to the maximum production rate, i.e. $m \leq max$. Consequently, the production rates are also equal to m :

$$p_{ik} = \min(q_{i(k-1)}, max_i) = \min(m_i, max_i) = m_i.$$

And with a constant production rate, the outputs become constant:

$$out_k = (1 - \sum_{i \in [n+1]} d_{(n+1)i}) p_{(n+1)k} = (1 - \sum_{i \in [n+1]} d_{(n+1)i}) m_{n+1}.$$

Such a network is obviously stable with an upper bound being the site quantities (or more). Moreover, the production process condition 3 in Definition 2 holds, yielding the following equality for the quantities of m:

$$\begin{aligned} m_j = q_{jk} &= in \cdot \delta_{1j} + \sum_{i \in [n+1]} d_{ij} p_{ik} + q_{j(k-1)} - p_{jk} \\ &= in \cdot \delta_{1j} + \sum_{i \in [n+1]} d_{ij} m_i + m_j - m_j = in \cdot \delta_{1j} + \sum_{i \in [n+1]} d_{ij} m_i \end{aligned}$$

for all $j \in [n+1]$ where $\delta_{11} = 1$ and $\delta_{1j} = 0$ for $j \geq 2$.

If one subtracts the latter sum and denotes the transposed distribution matrix by d^t, then one gets

$$(E - d^t)m = in \cdot e_1$$

where E is the identity matrix and e_1 the first unit vector.

In other words, a deterministic production network with a constant site quantity m implies that the system of linear equations

$$(E - d^t)x = in \cdot e_1$$

has m as a solution.

Interestingly enough, the considerations work also the other way round meaning that each solution of the system of linear equations given by the constant input quantity and the constant distribution matrix gives rise to stable production networks, provided that the initial quantity is bounded by the solution and the maximum production rate equals the solution or is greater.

5.1 Example

Solving the linear system $(E - d^t_{\mathbf{SAMPLE}})m = in \cdot e_1$ for the deterministic production network **SAMPLE** from Section 2.2 results in the maximal production rate vector $m = max = \begin{pmatrix} 3.2 \\ 1.6 \\ 2.4 \\ 1 \end{pmatrix}$.

Now we state our second main result, which guarantees stability of production networks under a sufficient condition.

Theorem 3. *Let PN be a deterministic production network and* $m\colon [n+1] \to \mathbb{R}_+$ *be a solution of the system of linear equations*

$$(E - d^t)x = in \cdot e_1$$

with $m \leq max$ *and* $q_0 \leq m$*. Then PN is stable.*

Proof. We show by induction that the sequence of site quantities of the unique production process of *PN* is bounded by m, i.e. $q_k \leq m$ for all $k \in \mathbb{N}$.

Base: $q_0 \leq m$ by assumption.

Step:

$$\begin{aligned}
q_{j(k+1)} &= in \cdot \delta_{1j} + \sum_{i \in [n+1]} d_{ij} p_{i(k+1)} + q_{jk} - p_{j(k+1)} && (1)\\
&= in \cdot \delta_{1j} + \sum_{i \in [n+1]} d_{ij} \min(q_{ik}, max_i) + q_{jk} - \min(q_{jk}, max_j) && (2)\\
&= in \cdot \delta_{1j} + \sum_{i \in [n+1]} d_{ij} q_{ik} + q_{jk} - q_{jk} && (3)\\
&\leq in \cdot \delta_{1j} + \sum_{i \in [n+1]} d_{ij} m_i && (4)\\
&= m_j && (5)
\end{aligned}$$

where equality 1 is the site quantity condition, equality 2 uses the exhaustiveness, equality 3 follows from the induction hypothesis $q_k \leq m$ and the assumption $m \leq max$, the inequality 4 is again the induction hypothesis, and equality 5 uses that m solves $(E - d^t)x = in \cdot e_1$.

With the boundedness of all site quantities, the sum of them over all sites is also bounded.

6 Conclusion

In this paper, we have introduced and investigated a variant of production networks with step-by-step production processes. The first main result shows that production networks can be transformed into communities of autonomous units such that production processes correspond to infinite runs of the modeling communities. The second main result yields a sufficient criterion for the stability of deterministic production networks. As this is the very first attempt to relate production networks and autonomous units, future research should shed more light on the significance of this approach including the following topics:

1. The stability results may be improved by enlarging the class of production networks for which sufficient conditions yield stability.
2. One may also look for necessary conditions or even proper characterizations.
3. So far, we have considered only two ways to choose the input rates and the productions: randomly on one hand and deterministically on the other. An interesting question is which other control conditions for the input unit and the production units will do to make proper use of their autonomy.
4. To improve the behavior of a production network one may allow variable distribution rates so that further circumstances like waiting time can be considered.
5. To make the model more flexible, one may enhance the notion of production networks by relaxing and modifying various assumptions like the following:
   - There may be more than one input site and one output site.
   - There may be an explicit control of the output rates.
   - There may be different kinds of materials and information flows through the network rather than a single homogeneous matter.
   - There may be particular time conditions for production and transportation at each site rather than the homogeneous step assumption.

   We expect that modifications like these will not be difficult to get.
6. Another possible modification would be to assume that the produced and distributed material consists of a number of atomic items such that only integer division is possible. In this case, the graph-transformational model may be particularly suitable as the atomic items could be represented by atomic graph components explicitly.
7. In some applications, it may not be realistic to assume that the underlying network is invariant, but it may grow or shrink due to economic circumstances. Again, the graph-transformational model may help to dynamize the structure of the production networks because the local insertion and removal of nodes and edges is just what happens if rules are applied.

References

1. Hans-Peter Wiendahl and Stefan Lutz. Production in Networks. *Annals of the CIRP- Manufacturing Technology*, 51(2):1–14, 2002.

2. Bernd Scholz-Reiter, Michael Görges, Thomas Jagalski, and Afshin Mehrsai. Modelling and Analysis of Autonomously Controlled Production Networks. In *Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 09). Moscow, Russia*, pages 850–855, 2009.
3. Bernd Scholz-Reiter, Afshin Mehrsai, and Michael Görges. Handling the Dynamics in Logistics - Adoption of Dynamic Behavior and Reduction of Dynamic Effects. *Asian International Journal of Science and Technology in Production and Manufacturing Engineering (AIJSTPME)*, 2(3):99–110, 2009.
4. Sergey Dashkovskiy, Michael Görges, and Lars Naujok. Local Input to State Stability of Production Networks. 2009. To appear in Proceedings of the Second International Conference, LDIC 2009, Bremen, Germany, August 2009.
5. Sergey Dashkovskiy, Björn S. Rüffer, and Fabian R. Wirth. Small gain theorems for large scale systems and construction of ISS Lyapunov functions. *SIAM Journal on Control and Optimization*, 48(6):4089–4118, 2010.
6. Sergey Dashkovskiy, Björn S. Rüffer, and Fabian R. Wirth. Numerical verification of local input-to-state stability for large networks. In *Proceedings of the 46th IEEE Conference on Decision and Control, New Orleans, LA, USA, Dec. 12-14, 2007*, pages 4471–4476, 2007.
7. Sergey Dashkovskiy and Björn S. Rüffer. Local ISS of large-scale interconnections and estimates for stability regions. *Systems and Control Letters*, 59(3–4):241–247, 2010.
8. Karsten Hölscher, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous units and their semantics — the sequential case. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Proc. 3rd Intl. Conference on Graph Transformations (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2006.
9. Hans-Jörg Kreowski and Sabine Kuske. Autonomous units and their semantics - the parallel case. In J.L. Fiadeiro and P.Y. Schobbens, editors, *Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006*, volume 4408 of *Lecture Notes in Computer Science*, pages 56–73, 2007.
10. Karsten Hölscher, Renate Klempien-Hinrichs, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous Units: Basic Concepts and Semantic Foundation. In Michael Hülsmann and Katja Windt, editors, *Understanding Autonomous Cooperation and Control in Logistics – The Impact on Management, Information and Communication and Material Flow*, pages 103–120. Springer, 2007.
11. Karsten Hölscher, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous Units to Model Interacting Sequential and Parallel Processes. *Fundamenta Informaticae*, 92(3):233–257, 2009.
12. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.
13. Rubino Geiß and Moritz Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, volume NN of *LNCS*. Springer, 2008. http://www.springerlink.com/content/105633/.

On Decidability of Bigraphical Sorting

Giorgio Bacci[1⋆] Davide Grohmann[2⋆⋆]

[1] Dept. of Mathematics and Computer Science, University of Udine
giorgio.bacci@uniud.it
[2] Programming, Logic and Semantics Group, IT University of Copenhagen
davg@itu.dk

Abstract. Bigraphs are a general framework for mobile, concurrent, and communicating systems. They have been shown to be suitable for representing several process calculi formalisms, but despite their expressive power, in many cases some disciplines on the structure of bigraphs are needed to faithfully encode the computational model at hand. *Sortings* have been proposed as an abstract technique to discipline bigraphs.
In this paper, we study the decidability problem of bigraphical sorting: to decide whether a bigraph belongs to some sorted bigraph category. Whilst the general problem is undecidable, we propose a decidable subclass of bigraphical sortings, named *match predicate sortings*, which are expressive enough to capture homomorphic sortings and local bigraphs.

1 Introduction

Bigraphical Reactive Systems (BRSs) [15] have been proposed as a promising meta-model for ubiquitous and mobile systems. The states of a BRS are *bigraphs*. Like an ordinary graph, a bigraph has nodes and edges connecting nodes, but unlike an ordinary graph, the nodes can be nested inside one another, hence they allow to represent both locality relationship between entities and (channel) connections. The dynamics of agents are represented by a set of rewrite rules on this semi-structured data.

Notably, Bigraphs and BRSs have been used for representing many domain-specific calculi and models: programming languages, calculi for concurrency and mobility, context-aware systems and web-services [11,12,3,9,5].

Process models (for example CCS, π-calculus, Ambient calculus) define processes syntactically, and in many cases it turns out that the encoding of processes into bigraphs is not exact, because bigraphs have too many degrees of freedom. This problem is usually overcome introducing "specialized versions" of bigraphs. For example, *binding bigraphs* [11] have been introduced to encode scoping for binding inputs in π-calculus: they allow for restricting name scope to a specific portion of a bigraph's locations. Many other variants have been proposed in literature, and almost all of them uses *sorting* techniques in achieving this. Example of sortings are *homomorphic* [15] and *many-one sortings* [12].

⋆ Work funded by MIUR PRIN project "SisteR", prot. 20088HXMYN.
⋆⋆ Work partially funded by CosmoBiz project supported by ITU of Copenhagen and the Danish Research Council for Technology and Production. Grant no. 274-06-0415.

The main drawback in using sorting techniques is that each time a sorting is introduced, the theory of bigraphs must be redefined anew. Recently, Debois and coauthors [4] generalized the ad hoc constructions providing the definition of a class of *sorted categories* for which the behavioral theory of pure bigraphs is preserved, the so called *predicate sortings*. Intuitively predicate sortings rule out bigraphs that do not satisfy the predicate P.

Although using predicate sortings provides a general construction which sustain the behavioural theory of bigraphs, this technique has a main drawback: the systematic construction makes sorted categories very difficult to handle, due to the fact that their objects are defined as pairs of sets of morphisms from the original category, closed by prefix and suffix composition, and most of the difficulties arise when one wants to implement effectively such construction.

In order to overcome these difficulties, but at the same time keeping the technique as general as possible (we do not want to define sortings by hand), we propose to look at predicate sortings from a different point of view. The sorted category will not be constructed at all, but we will use sortings as a way of (automatically) checking if a morphism of the original category has a "counterpart" in the sorted category and, more importantly, if compositions in the non-sorted category are admissible in its sorted variant.

The aim of this paper is to investigate the decidability of the proposed checking procedure for sortings. It turns out that this procedure is undecidable in general, even if we restrict only on bigraphical sortings. For this reason we propose a decidable subclass of bigraphical predicate sortings, named *match predicate sortings*, for which there exists an effective algorithm to check if a bigraph "belongs" to the sorted category.

The key idea relies on the factorization theorem [4], which characterizes decomposable predicates as those that disallow factorisation by a given set of morphisms. Intuitively, given a predicate P such that $P(f)$ holds, there must exists a set Φ of morphism such that f cannot be decomposed in a form $f = g \circ \psi \circ h$ where ψ is in Φ. Intuiticely, P disallows occurrences of ill-formed morphisms, which are exactly those in Φ. In bigraphs it turns out to be much more intuitive to identify occurrences as matches. In this way, (some) decomposable predicates can be expressed as a set of ill-patterns which cannot be matched in well-sorted bigraphical morphisms. This choice permits to apply the bigraphical matching procedure to check whether a (pure) bigraph has a counterpart in the match sorted category, hence this class is decidable. Notably, this sub-class of sortings captures a good variety of sortings proposed in the literature, for example homomorphic sortings [15] and local bigraphs [14,16].

Synopsis The paper is structured as follows. In Sections 2 and 3 we recall the theory of bigraphs and (bigraphical) sortings. In Section 4 we prove the undecidability of the (bigraphical) sorting problem in the general setting. After that, in Section 5 the class of match predicate sortings is introduced and analyzed for decidability. In Section 6 we show how to express two relevant sortings introduced in literature as match predicate sortings. Finally, conclusions and ideas for further developments are in Section 7.

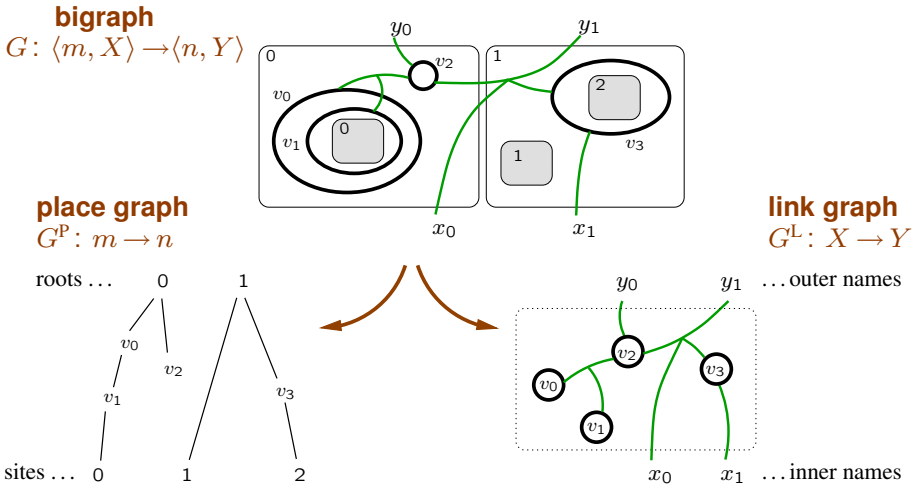


Fig. 1. Bigraph = Place graph + Link graph (picture taken from [15]).

2 Bigraphs

In this section we recall Milner's *bigraphs* [15]. Intuitively, a bigraph represents an open system: it has an inner and an outer interface to "interact" with subsystems and the surrounding environment (see Figure 1). The *width* of the outer interface describes the *roots*, that is, the various locations containing the system components; the width of the inner interface describes the *sites*, that is, the holes where other bigraphs can be inserted. On the other hand, the *names* in the interfaces describe the free links, that is, end points where links from the inner parameters or/and the external environment can be pasted, creating new links among nodes. We refer the reader to [15] for a longer description of bigraphs.

In this paper, we use the following terminology and notation. Natural numbers are frequently treated as finite ordinals, that is, $m = \{0, 1, \ldots, m-1\}$. We write $S \uplus T$ for the union of sets assumed to be disjoint. For two functions f and g with disjoint domains S and T we write $f \uplus g$ for the function with domain $S \uplus T$ such that $(f \uplus g) \restriction S = f$ and $(f \uplus g) \restriction T = g$. We write id_S for the identity function on the set S. In defining bigraphs we assume that names, node-identifiers and edge-identifiers are drawn from three infinite sets, respectively $\mathcal{X}$, $\mathcal{V}$ and $\mathcal{E}$, disjoint from each other.

The bigraphical category is defined over a *signature* $\mathcal{K}$ of controls with arity function $ar\colon \mathcal{K} \to \mathbb{N}$ which identifies the *ports* $ar(k)$ of a control $k \in \mathcal{K}$.

Definition 1 (Interface). *An* interface *is a pair* $\langle m, X\rangle$ *where* m *is a finite ordinal (named* width*) and* X *is a finite set of names.*

Definition 2 (Bigraphs). *A* bigraph $G\colon \langle m, X\rangle \to \langle n, Y\rangle$ *compounds a* place graph G^P *and a* link graph G^L *which describe the nesting of nodes and the*

(hyper-)links among nodes, respectively.

$$G = (V, E, ctrl, G^P, G^L) \colon \langle m, X\rangle \to \langle n, Y\rangle \qquad \textit{(bigraph)}$$
$$G^P = (V, ctrl, prnt) \colon m \to n \qquad \textit{(place graph)}$$
$$G^L = (V, E, ctrl, link) \colon X \to Y \qquad \textit{(link graph)}$$

where V, E are (finite) sets of nodes and edges respectively; $ctrl \colon V \to \mathcal{K}$ is the control map*, assigning a control to each node; $prnt \colon m \uplus V \to V \uplus n$ is the (acyclic)* parent map*; $link \colon X \uplus P \to E \uplus Y$ is the* link map*, where $P = \sum_{v \in V} ar(ctrl(v))$ is the set of ports.*

Definition 3 (Bigraph category). *The category of bigraphs over a signature $\mathcal{K}$, denoted as* **Big**$(\mathcal{K})$*, has interfaces as objects and bigraphs as morphisms.*

Given two bigraphs $G \colon \langle m, X\rangle \to \langle n, Y\rangle$, $H \colon \langle n, Y\rangle \to \langle k, Z\rangle$, the composition $H \circ G \colon \langle m, X\rangle \to \langle k, Z\rangle$ is defined by composing their place and link graphs:

$$H^P \circ G^P = (V, ctrl, (id_{V_G} \uplus prnt_H) \circ (prnt_G \uplus id_{V_H})) \colon m \to k$$
$$H^L \circ G^L = (V, E, ctrl, (id_{E_G} \uplus link_H) \circ (link_G \uplus id_{P_H})) \colon X \to Z,$$

where $V = V_G \uplus V_H$, $ctrl = ctrl_G \uplus ctrl_H$, and $E = E_G \uplus E_H$.

An important operation on bigraphs, is the *tensor product*. Intuitively, this operator puts "side by side" two bigraphs, i.e., given $G \colon \langle m, X\rangle \to \langle n, Y\rangle$ and $H \colon \langle m', X'\rangle \to \langle n', Y'\rangle$, their tensor product is $G \otimes H \colon \langle m + m', X \uplus X'\rangle \to \langle n + n', Y \uplus Y'\rangle$ defined when name sets X, X' and Y, Y' are pairwise disjoint.

As shown in [15], all bigraphs can be constructed by composition and tensor product from a set of *elementary bigraphs:*

- $1 \colon \langle 0, \emptyset\rangle \to \langle 1, \emptyset\rangle$ is the barren (i.e., empty) root.
- $merge_n \colon \langle n, \emptyset\rangle \to \langle 1, \emptyset\rangle$ merges n roots into a single one.
- $\gamma_{m,n} \colon \langle m + n, \emptyset\rangle \to \langle n + m, \emptyset\rangle$ is a *symmetry*, that swaps the first m roots with the following n roots.
- $/x \colon \langle 0, \{x\}\rangle \to \langle 0, \emptyset\rangle$ is a *closure*, that is it maps x to an edge.
- $y/X \colon \langle 0, X\rangle \to \langle 0, \{y\}\rangle$ substitutes the names in X with y, i.e., it maps the whole set X to y. Notice that X can be the empty set, i this way y is linked to nothing and it is said to be *idle*.
- $K_{\vec{x}} \colon \langle 1, \emptyset\rangle \to \langle 1, \{x_1, \ldots, x_n\}\rangle$ is a control which may contain other bigraphs, and it has ports linked to the name in $\vec{x} = x_1, \ldots, x_n$.

A bigraph is said a *renaming* if it is of the form $x_1/\{y_1\} \otimes \cdots \otimes x_n/\{y_n\}$ (abbreviated to $\vec{x}/\vec{y}$, where $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_n$); a *permutation* if it is formed by composition and tensor product of symmetries; a *prime* when it has no inner names and its outer width is 1, a *discrete* when its link map is a bijection. Two useful variants of tensor product can be defined using tensor and composition: the *parallel product*, denoted as $G \parallel H$, merges shared outer names of G and H, the *merge product* written $G \mid H$, which moreover merges all roots in a single one is defined as $merge_n \circ (G \parallel H)$ (with n outer width of $G \parallel H$).

3 Sortings

When one is adopting the bigraphical framework for defining algebraic models or programming languages, it turns out that such framework is too general and one has to discipline it with some constraints to fit precisely the problem at hand. To this end, general and powerful techniques, named *sortings*, have been developed by Debois and coauthors in [4].

Definition 4 (Sortings). *A* sorting *of a category* $\mathbf{C}$ *is a functor* $F : \mathbf{X} \to \mathbf{C}$*, that is faithful and surjective on objects. We call* $\mathbf{X}$ sorted category*.*

Intuitively, a sorting functor F defines $\mathbf{X}$ by refining the category $\mathbf{C}$. The objects (i.e., interfaces) of $\mathbf{X}$ carry more information than the original ones, thus morphism (i.e., system) composition turns out to be finer-grained. This yields back a category $\mathbf{X}$ where morphisms are more informative than those in $\mathbf{C}$ in the sense that, some compositions in $\mathbf{C}$ no longer hold in $\mathbf{X}$.

Due to the very general nature of sorting refinements needed by each particular application, it could be tricky to construct a sorting directly using the definition above. Moreover, each time a sorting is adopted the behavioral theory of bigraphs must be redeveloped. Debois observed that most sortings in the literature on bigraphs are actually means of banning particular morphisms from the original category. In each of these cases, it is possible to identify a predicate on the morphisms of the category in question, which holds precisely at the morphisms in the image of the sorting functor. Unfortunately a predicate on morphisms might not give rise to a subcategory, indeed we might have composable morphisms which individually satisfy the predicate, but whose composite does not. Debois proved that for the construction of such a sorted category it is sufficient that the predicate is decomposable.

Definition 5 (Decomposable predicate). *A* predicate P *on morphisms of a category is* decomposable *iff* P *holds on identities and* $P(f \circ g) \Rightarrow P(f) \wedge P(g)$*.*

Notably, the class of decomposable predicates can be characterized as those morphisms that disallow factorization by a given set of morphisms.

Theorem 1 (Factorization, [4, Proposition 14]). *A predicate* P *on morphisms of a category* $\mathbf{C}$ *is* decomposable *iff there exists a set of morphisms* Φ *such that* $P(f)$ *holds iff for any* g, ψ, h *we have* $f = g \circ \psi \circ h$ *implies* $\psi \notin \Phi$*.*

In [4] it is also given a method to systematically construct a well-behaved sorting for any decomposable predicate.

Definition 6 (Predicate Sorting). *Let* $\mathbf{C}$ *be a category and let* P *be a decomposable predicate on the morphisms of* $\mathbf{C}$*. The* predicate sorting $\mathcal{S}_P : \mathbf{X} \to \mathbf{C}$ *is defined as follows. The category* $\mathbf{X}$ *has pairs* (X, Y) *as objects, where, for some object* C *of* $\mathbf{C}$*,* X *is a set of* $\mathbf{C}$*-morphisms with codomain* C *and* Y *is a set of* $\mathbf{C}$*-morphisms with domain* C*, subject to the following conditions.*

$$
\begin{array}{lll}
id_C \in X & f \in X \cup Y \Rightarrow P(f) & g \circ f \in X \Rightarrow g \in X \\
id_C \in Y & f \in X,\ g \in Y \Rightarrow P(g \circ f) & g \circ f \in Y \Rightarrow f \in Y\,.
\end{array}
$$

There is a morphism $f : (X, Y) \to (U, V)$ whenever the following holds.

$$f \in Y \cap U \qquad x \in X \Rightarrow f \circ x \in U \qquad v \in V \Rightarrow v \circ f \in Y\,.$$

Proposition 1. *Let P be a decomposable predicate on a category $\mathbf{C}$. The image of the predicate sorting $\mathcal{S}_P$ is precisely the set of morphisms satisfying P.*

All the above definitions and results apply naturally to the bigraph category.

4 Undecidability of bigraphical sortings

In this section, we focus our attention on the undecidability issues of predicate sortings and in particular for the case of bigraphical sortings.

Looking at the Definition 6, it is obvious that an exhaustive construction of a predicate sorted category is unfeasible (one must quantify on all morphisms to construct an object of the sorted category). Instead, what can be done is not to establish the decidability of the construction of the category, but looking at the problem of checking if a given morphism f from the base category has a pre-image in the sorted one. The key idea behind this approach is to "simulate" the existence of the sorted category, and checking that each morphism that comes into play is actually well-sorted.

When a predicate sorting $\mathcal{S}_P \colon \mathbf{X} \to \mathbf{C}$ is used, the existence of the pre-image $x = \mathcal{S}_P(f)$ in the sorted category $\mathbf{X}$ is garanteed whenever $P(f)$ holds by Proposition 1. Unfortunately, decidability cannot be assumed neither for general predicate P nor for decomposable predicates, even if we restrict to consider only decomposable predicates over bigraphical morphisms.

Let us define a decomposable predicate over bigraphs which will be proved to be undecidable by a reduction from the Post Correspondence Problem (PCP) [17]. We use α, β, γ for words in $\Sigma = \{a, b\}^*$, and ϵ for the empty word. An instance of PCP is a set of pairs of words $\{(\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)\}$ over the two-letter alphabet $\{a, b\}$ (that is, $\alpha_i, \beta_i \in \Sigma$). The question is whether there exists a sequence $i_0, i_1, \ldots, i_k$ ($1 \leq i_j \leq n$ for all $0 \leq j \leq k$) such that $\alpha_{i_0} \cdot \ldots \cdot \alpha_{i_k} = \beta_{i_0} \cdot \ldots \cdot \beta_{i_k}$, where $\cdot$ denotes word concatenation.

Let $\mathcal{K} = \{\mathsf{list}\colon 0, \mathsf{pair}\colon 0, \mathsf{a}\colon 0, \mathsf{b}\colon 0\}$ be a bigraphical signature of 0-arity controls. Any word $\gamma \in \Sigma$ can be represented in $\mathbf{Big}(\mathcal{K})$ by means of the encoding $\mathit{wrd}\colon \Sigma \to \mathbf{Big}(\mathcal{K})$, pairs of words by $\mathit{pair}\colon \Sigma \times \Sigma \to \mathbf{Big}(\mathcal{K})$ and n-length lists of word pairs by $\mathit{lst}_n \colon (\Sigma \times \Sigma)^n \to \mathbf{Big}(\mathcal{K})$ defined as follows:

$$\mathit{wrd}(\epsilon) = 1, \qquad \mathit{wrd}(a \cdot \gamma) = \mathsf{a} \circ \mathit{wrd}(\gamma), \qquad \mathit{wrd}(b \cdot \gamma) = \mathsf{b} \circ \mathit{wrd}(\gamma),$$
$$\mathit{pair}(\alpha, \beta) = \mathsf{pair} \circ (\mathit{wrd}(\alpha) \mid \mathit{wrd}(\beta)),$$
$$\mathit{lst}_n((\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)) = \mathsf{list} \circ (\mathit{pair}(\alpha_1, \beta_1) \mid \cdots \mid \mathit{pair}(\alpha_n, \beta_n)).$$

Proposition 2. *Let Φ_{PCP} be the following a set of morphisms in $\mathbf{Big}(\mathcal{K})$:*

$$\Phi_{PCP} = \{\mathit{lst}_n((\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)) \mid (\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n) \in \mathit{PCP}\}$$

then the set U below is a decomposable and undecidable predicate over $\mathbf{Big}(\mathcal{K})$.

$$U = \{f \text{ morphism of } \mathbf{Big}(\mathcal{K}) \mid \forall g, \phi, h.\ f = g \circ \phi \circ h \Rightarrow \phi \notin \Phi_{PCP}\}$$

Proof. By Theorem 1, U is a decomposable predicate. Let us prove its undecidability. By contradiction, assume U to be decidable, hence the characteristic function P_U for U, defined as $P(u) = 1$ if $u \in U$, $P(u) = 0$ otherwise, must be computable. This obviously contradicts the fact that PCP is undecidable, since an algorithm for P_U will decides also PCP because for $u = lst_n((\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n))$,

$$P_U(u) = 0 \iff u \notin U \iff ((\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)) \in \text{PCP}$$

Notice that u has an obvious occurrence of a morphism $\psi \in \Phi_{PCP}$, that is, $\psi = u$ since $u = id \circ u \circ id$. □

Theorem 2. *Let $\mathcal{S}_P \colon \mathbf{X} \to \mathbf{C}$ be a predicate sorting over a decomposable predicate P. The problem of checking if a given morphism f in $\mathbf{C}$ has a pre-image $x = \mathcal{S}_P(f)$ in the sorted category $\mathbf{X}$ is undecidable.*

Proof. It follows immediately from Propositions 1 and 2. □

As a corollary, checking the existence of a sorted pre-image is undecidable.

Corollary 1. *Let $\mathcal{S} \colon \mathbf{X} \to \mathbf{C}$ be sorting. The problem of checking if a given morphism f in $\mathbf{C}$ has a pre-image $x = \mathcal{S}(f)$ in $\mathbf{X}$ is undecidable.*

5 Match predicate sortings

In this section, we introduce a characterization of a decidable class of bigraphical sortings, which turns out to be a proper subclass of the predicate sortings.

Following the observations that drove the definition of predicate sortings of Debois and coauthors, that is, that most sortings in literature are actually defined for banning particular ill-formed patterns, we propose to identify the notion of pattern occurrence with that of pattern match occurrence. A bigraph G has a match within a bigraph H if and only if $H = F \circ (G \otimes id_X) \circ E$ for some name set X and bigraphs F, E. The problem of finding a match of a bigraph into another was investigated in [2] where and inductive characterization of this problem was proposed and a resolutive algorithm is given.

This scenario suggests the definition of a family of decomposable predicates based on the bigraphical matching problem.

Definition 7 (Match predicate). *Let $\mathcal{R}$ be a recursive set of bigraphs. We say that $P_{\mathcal{R}}$ is a* match predicate *with respect to the set $\mathcal{R}$, if for every bigraph G, $P_{\mathcal{R}}(G)$ holds iff every $R \in \mathcal{R}$ does not have a match in G.*

Proposition 3. *Any match predicate is a decomposable predicate.*

Proof. Let $\mathcal{R}$ be a set of redexes and P its match predicate. Now, suppose by absurdity that P is not decomposable. So there exist two bigraphs such that $P(G \circ H)$ holds and one between $P(G)$ or $P(H)$ does not. Suppose $P(H)$ does not hold (the other case is analogous), this means that there exist C, D such that $H = (id \otimes C) \circ (id \otimes R) \circ D$, for some $R \in \mathcal{R}$. Therefore $G \circ H = (G \circ (id \otimes C)) \circ (id \otimes R) \circ D$ is a match of R in $G \circ H$, an absurd by hypothesis. □

Hence, the class of match predicates is a *proper subclass* of decomposable predicates, and it is decidable by means of the matching algorithm.

Proposition 4 (Decidability). *Any match predicate is decidable.*

Proof (Sketch). Let $G \colon \langle m, X \rangle \to \langle n, Y \rangle$ be a bigraph, we give a decidable decision procedure for $P_{\mathcal{R}}(G)$. Since $\mathcal{R}$ is a generic recursive set, it could be infinite in general, hence it is not be possible to check for all elements of $\mathcal{R}$ whether they have a match within the given bigraph G. Note, however, that G is finite, that is, it has finite node and edge sets, and finite interfaces as well. Let $\mathcal{K}_G$ be the set of controls used by nodes in G, and S the set of bigraphs using only controls taken from $\mathcal{K}_G$ and such that they have at most $|V_G|$ nodes, $|E_G|$ edges, $|X|$ inner names, $|X| + |P|$ outer names (where P is the set of ports, hence depends on the chosen set of nodes and controls), and m, n inner and outer width.

Since the set of nodes V_G is finite, $\mathcal{K}_G$ must be finite ($|\mathcal{K}_G| \leq |V_G|$); by a similar argument also S is finite. The set S contains all the possible bigraphs such that they have a match in G (the proof is by contradiction and omitted due to lack of space), by the finiteness of S and by the hypothesis that $\mathcal{R}$ is recursive, the set $\mathcal{R} \cap S$ is computable and finite. It is not difficult to prove that $P_{\mathcal{R}}(G)$ holds iff R does not have a match in G, for all $R \in \mathcal{R} \cap S$, hence, since the bigraphical matching problem is decidable [8], $P_{\mathcal{R}}$ is decidable as well. □

Now we specialize the Factorization Theorem 1 in the following sense: given a recursive set of bigraphs M, the set Φ of unwanted bigraphs is defined from M as $\Phi = \{m \otimes id_X \mid m \in M \wedge X \text{ is a set of names}\}$.

Theorem 3 (Factorization). *A predicate P is a* match predicate *iff there exists a recursive set of morphisms M such that $P(f)$ holds iff for any g, ψ, h and any set of names X we have $f = g \circ (\psi \otimes id_X) \circ h$ implies $\psi \notin M$.*

Proof. Direct consequence of Proposition 3 and Theorem 1. □

In this way, deciding if a bigraph G is well-sorted is reduced to decide if no $m \in M$ has a match into G. Moreover, we can use the Proposition 3 in combination with the Definition 6 to define the bigraphical sorted category. We call this class of sortings *match predicate sortings*. Consequence of the decidability of any match predicate is that the set M is *recursive*, indeed it is essential to not contradict Proposition 4. Although not forbidden, M is supposed to contain no identities, otherwise (almost) all bigraphs are sorted out, resulting in a useless sorting strategy. Finally, our match predicate sortings work up-to tensor product with identities, i.e., the unwanted bigraphical structures are "homset independent", indeed a match can be found in any context, so we cannot force the decomposition to work only with some particular interfaces.

6 Sortings in literature and their decidability

In order to investigate the expressive power of our decidable class of sortings, we analyze some sortings introduced in literature. We focus our attention on the sortings shown in [7, Table 6.1], which are replaceable by a predicate sorting. In particular, we consider *homomorphic sortings* [15] and the bigraph's variant known as *local bigraphs* [14,16]. In this section we show that each decomposable predicate used in [7] can be characterized as a match predicate of Definition 7. Notice that the construction of the sorted category is left unchanged because match predicates are decomposable by Proposition 3.

6.1 Homomorphic sortings and CCS

Firstly, we recall homomorphic sortings as given in [15] with the variants of [7][3]. In order to exemplify the use of the match predicate sortings, we also recall the encoding of CCS [13] into bigraphs proposed by Milner in [15] which adopts a particular homorphic sorting.

We start giving the definition of place-sorted bigraph.

Definition 8 (Place-sorted interface). *Let Θ be a set of sorts. An interface $I = \langle m, X \rangle$ is* Θ-place-sorted *if it is enriched by ascribing a sort to each place $i \in m$. If I is place-sorted, we denote its underlying unsorted interface by $U(I)$.*

We denote by $\mathbf{Big}(\mathcal{K}, \Theta)$ the category in which the objects are place-sorted interfaces, and each morphism $G : I \to J$ is a bigraph $G : U(I) \to U(J)$.

Such definition refines only the objects of the bigraph category, the next one is cutting down some morphisms (i.e., bigraphs).

Definition 9 (Place-sorting). *A* place-sorting *is a triple $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$, where Φ is a condition on the place graph of Θ-sorted bigraphs over $\mathcal{K}$. The condition Φ must be satisfied by identities and preserved by composition and tensor.*

A bigraph in $\mathbf{Big}(\mathcal{K}, \Theta)$ is Σ-place-sorted if it satisfies Φ. The Σ-sorted bigraphs form a sub-category of $\mathbf{Big}(\mathcal{K}, \Theta)$ denoted by $\mathbf{Big}(\Sigma)$.

Notably, Milner in [15, Proposition 10.3] shows that U can be extended to a functor $U : \mathbf{Big}(\Sigma) \to \mathbf{Big}(\mathcal{K}, \Theta)$ which is surjective on objects and faithful, and hence a sorting by Definition 4.

Due to the very general nature of place-sorting, Milner defines a particular class of such sortings, named *homomorphic sortings*.

Definition 10 (Homomorphic sorting). *A place-sorting $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$ is an* homomorphic sorting *if the condition Φ assigns a sort $\theta \in \Theta$ to each control in $\mathcal{K}$ by means of a surjective function sort : $\mathcal{K} \to \Theta$ and it also defines a parent map $prnt_\Theta : \Theta \to \Theta$ over sorts. (We impose that Θ has a least two elements[4].)*

In a bigraph G, via its control map, the sort assignment to $\mathcal{K}$ determines a sort for each node. The Φ requires that, for each site or node w in G with sort θ:

[3] The variants are quite technical and do not change the resulting sorted categories.

[4] Otherwise the homomorphic sorting sorts out no bigraph, hence it is useless.

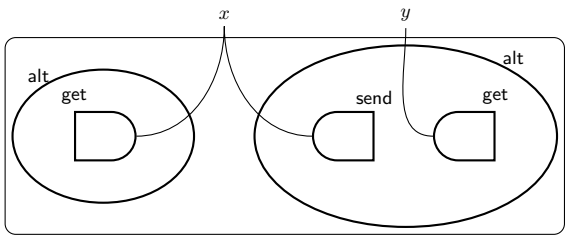


Fig. 2. A bigraph encoding the CSS process $x.\mathbf{0} \mid (\bar{x}.\mathbf{0} + y.\mathbf{0})$.

1. *if* $prnt_G(w)$ *is a node then its sort is* $prnt_\Theta(\theta)$*;*
2. *if* $prnt_G(w)$ *is a root then its sort is* θ*.*

As already mentioned, the translation of (finite) CCS in bigraphs provides a interesting non-trivial example of homomorphic sorting. Suppose a universal set of name $\mathcal{N}$ and let $x, y, z, \ldots$ range over $\mathcal{N}$, whilst P, Q range over processes and A over summations. The CCS syntax is

$$\begin{array}{rcl} P & ::= & (\nu x)P \mid P \mid P \mid \ A \\ A & ::= & \ \mathbf{0} \ \ \mid \ x.P \ \mid \bar{x}.P \mid A + A\,. \end{array}$$

Intuitively $\mathbf{0}$ denotes termination. $(\nu x)P$ means that the name x is restricted in P. $x.P$ and $\bar{x}.P$ are the input and output actions respectively. Finally, $\mid$ is the parallel composition and $+$ the non-deterministic choice. The set of free names is composed by all names of the process not under the scope of a ν. The processes are taken up-to the following structural equivalence ($\equiv$) [15]: $\equiv$ contains the α-equivalence on processes, $\mid$ and $+$ are commutative and associative under $\equiv$, and the following rules hold

$$\begin{array}{c} A + 0 \equiv 0 \qquad (\nu x)(A + \alpha.P) \equiv A + \alpha.(\nu x)P \quad \text{if } \alpha \in \{y, \bar{y}\} \text{ and } x \neq y \\ (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad\qquad (\nu x)P \equiv P \quad \text{if } x \notin fn(P) \\ (\nu x)(P \mid Q) \equiv P \mid (\nu x)Q \quad \text{if } x \notin fn(P)\,. \end{array}$$

Notice that $P \mid \mathbf{0} \not\equiv P$, but we can prove that they are bisimilar.

Now we introduce the homorphic sorting for encoding CCS into bigraphs.

$$\Sigma_{CCS} = (\{\mathsf{a}, \mathsf{p}\}, \{\mathsf{alt} : 0, \mathsf{get} : 1, \mathsf{send} : 1\}, \Phi)$$

where a, p are types representing "summations" and "processes". The control alt encodes a summation and the controls get and send denote input and output actions. The condition Φ assigns the type a to alt and p to both get and send. Φ also imposes an alternation of controls of type a and p in the place graphs, i.e., $prnt_\Theta(\mathsf{a}) = \mathsf{p}$ and $prnt_\Theta(\mathsf{p}) = \mathsf{a}$. Both composition and tensor preserves Φ.

The translation of a CCS process into a bigraphs is defined as follows. We map processes into ground homset having a single root typed with p, i.e., $\epsilon \to \langle \mathsf{p}, X \rangle$, and analogously summations into $\epsilon \to \langle \mathsf{a}, X \rangle$. In order to do this we define two

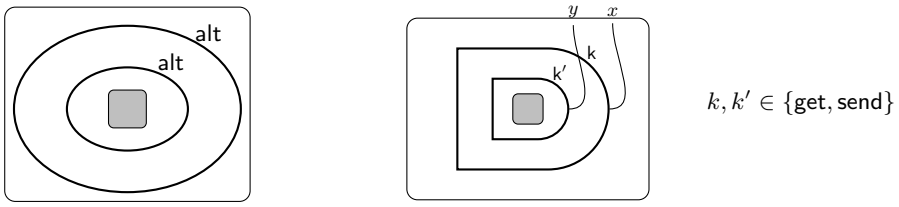


Fig. 3. The ill-bigraphs used to define a match predicate sorting for the CCS.

translation operators $\mathcal{P}_X[\cdot]$ and $\mathcal{A}_X[\cdot]$ each indexed on a (finite) set of names X. They are defined by mutual recursion:

$$\begin{array}{ll} \mathcal{P}_X[(\nu x)P] = /y \circ \mathcal{P}_{X \uplus \{y\}}[P\{y/x\}] & \mathcal{A}_X[\mathbf{0}] = X \mid 1 \\ \mathcal{P}_X[P \mid Q] = \mathcal{P}_X[P] \mid \mathcal{P}_X[Q] & \mathcal{A}_X[a.P] = (\mathsf{get}_x \mid \mathsf{id}_X) \circ \mathcal{P}_X[P] \ (x \in X) \\ \mathcal{P}_X[A] = (\mathsf{alt} \mid \mathsf{id}_X) \circ \mathcal{A}_X[A] & \mathcal{A}_X[\bar{a}.P] = (\mathsf{send}_x \mid \mathsf{id}_X) \circ \mathcal{P}_X[P] \ (x \in X) \\ & \mathcal{A}_X[A + B] = \mathcal{A}_X[A] \mid \mathcal{A}_X[B]\,. \end{array}$$

As an example consider the CSS process $x.\mathbf{0} \mid (\bar{x}.\mathbf{0} + y.\mathbf{0})$, its translation in a bigraph is depicted in Figure 2.

In order to construct a predicate from a homomorphic sorting, it is sufficient to restrict the condition of Φ to consider just 1. and dropping 2., indeed we can focus only on constraining the internal components (i.e., nodes) of the bigraph. The roots belong to the interfaces, and those are refined automatically by the predicate sortings (see Definition 6).

The predicate constructed by Debois in [7] is the following.

Definition 11. *Let $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$ be a homomorphic sorting, and let $prnt_\Theta$ be the parent maps on sorts defined by Φ. The predicate P_Σ holds on a bigraph G iff whenever the control of a node v in G has sort $\theta \in \Theta$ and $w = prnt_G(v)$ is a node, then the control of w has sort $prnt_\Theta(\theta)$.*

On such definition, it is easy to yield a "negative" version of the predicate by means of the Factorization Theorem 1: the set of unwanted bigraphs Φ can be constructed by complementing the condition 1.. This observation also suggests a way of deriving a match predicate by means of our Factorization Theorem 3.

Definition 12. *Let $\Sigma = \{\mathcal{K}, \Theta, \Phi\}$ be a homomorphic sorting, and let $prnt_\Theta$ be the parent maps on sorts defined by Φ. The match predicate $P(M_\Sigma)$ for Σ can be defined on the set of bigraphs M_Σ below and by using the Theorem 3.*

$$M_\Sigma \triangleq \{(K_{\vec{x}} \otimes \mathsf{id}_{\vec{y}}) \circ H_{\vec{y}} \mid K, H \in \mathcal{K} \wedge prnt_\Theta(sort(H)) \neq sort(K)\} \qquad (1)$$

It is trivial to prove that the meaning of the two predicates coincides, indeed if a match exists condition 1. is violated, otherwise it does not hold. It is important for decidability to notice that the set M_Σ defined in equation (1) is finite.

In the case of CCS, the set of undesired graph $M_{\Sigma_{CCS}}$ must contain the bigraphs which has ill-nested controls. In particular, we should forbid the nesting

of a a-typed control into another a-typed one (analogously for the type p). In other words this means that we do not allow the nesting of two consecutive alt nodes or two send and/or get nodes. Formally, the set of ill-formed bigraphs for the match predicate sorting are defined below and depicted in Figure 3.

$$M_{\Sigma_{CCS}} = \left\{ \begin{array}{l} \mathsf{alt} \circ \mathsf{alt}, (\mathsf{get}_x \otimes \mathsf{id}_{\{y\}}) \circ \mathsf{get}_y, (\mathsf{get}_x \otimes \mathsf{id}_{\{y\}}) \circ \mathsf{send}_y, \\ (\mathsf{send}_x \otimes \mathsf{id}_{\{y\}}) \circ \mathsf{get}_y, (\mathsf{send}_x \otimes \mathsf{id}_{\{y\}}) \circ \mathsf{send}_y \end{array} \right\}.$$

The following result follows directly from the above considerations.

Theorem 4. *Homomorphic sortings correspond exactly to match predicate sortings over the predicate M_Σ.*

Proof. It follows from the characterization given in [7, Section 6.3] and by the fact that $P_\Sigma = M_\Sigma$. □

Remarkably, homomorphic sortings are of particular interest in the setting of bigraphical encoding of process algebras, in fact, they have been employed in the encoding of π-calculus variants [18] (cfr. Jensen [10]), but also in the definition of variants pure bigraphs, e.g. kind bigraphs [6].

Corollary 2. *Homomorphic sortings are decidable.*

Proof. Direct consequence of Theorem 4 and Proposition 4. □

6.2 Local bigraphs

In this section first we recall Milner's *local bigraphs* [14,16] and then we discuss how to use a match predicate sorting on (pure) bigraphs to catch local bigraphs.

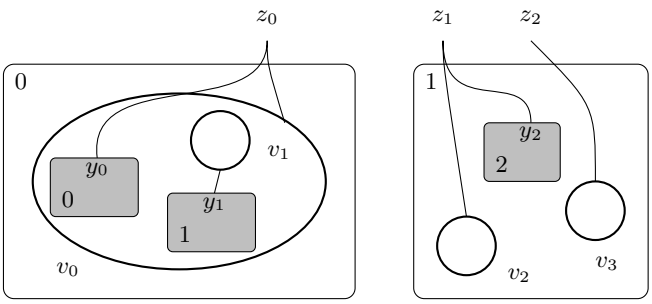
Intuitively, a local bigraph is like a standard (pure) bigraph but it has names which are deeply connected with placing, i.e., there is a precise scoping rule: linking must respect the nesting of nodes. An example of a local bigraph is shown in Figure 4, Note that inner names are "localized" on the bigraph's sites and outer names on the roots.

Let $\mathcal{K}$ be a *binding signature* of controls, and $ar : \mathcal{K} \to \mathbb{N} \times \mathbb{N}$ be the arity function. The arity pair (h, k) (often written as $h \to k$) consists of the *binding arity* h and the *free arity* k, indexing respectively the binding ports and the free ports of a control.

Definition 13. *A* local interface *is a list $(X_0, \ldots, X_{n-1})$, where n is the* width *and X_is are disjoint sets of names. X_i represents the names located at i.*

Definition 14. *A* local bigraph $G : (\vec{X}) \to (\vec{Y})$ *is defined as a (pure) bigraph $G^u : \langle |\vec{X}|, \bigcup \vec{X} \rangle \to \langle |\vec{Y}|, \bigcup \vec{Y} \rangle$ satisfying certain locality conditions. Let π_1 and π_2 be the canonical projections of pair components, let $P = \sum_{v \in V} \pi_1(ar(ctrl(v)))$ be the set of ports, and let $B = \sum_{v \in V} \pi_2(ar(ctrl(v)))$ be the set of bindings (associated to all nodes), the* link map *is $link : X \uplus P \to E \uplus B \uplus Y$.*

The locality conditions are the following:



$G : (\{y_0\}, \{y_1\}, \{y_2\}) \to (\{z_0\}, \{z_1, z_2\})$

Fig. 4. An example of a local bigraph.

1. *if a link is bound, then its inner names and ports must lie within the node that binds it;*
2. *if a link is free, with outer name x, then x must be located in every region that contains any inner name or port of the link.*

Definition 15. *The category* $\mathbf{Lbg}(\mathcal{K})$ *of local bigraphs over a binding signature* $\mathcal{K}$ *has local interfaces as objects, and local bigraphs as morphisms. Composition and tensor product are defined analogously as for (pure) bigraphs.*

Local bigraphs have been introduced in literature by Milner for encoding the λ-calculus [1] into bigraphs and investigating confluence properties for bigraphical reactive systems [16].

In [7] it is given the predicate that follows, and it is proven that predicate sortings can be replaced with the category of local bigraphs.

Definition 16 ([7, Definition 6.22]). *Let* Σ *be a binding signature. Define* P_Σ *to be the predicate on the morphisms of* $\mathbf{Big}(U(\Sigma))$ *given by* $P_\Sigma(f)$ *iff in* f

"all ports linked to a binding port of a node v lie under v".

It is straightforward to prove that such predicate is decomposable, but we want to characterize it as a matching predicate, that is, provide a recursive set of unwanted redex patterns. Fortunately the predicate P_Σ is very simple to falsify, indeed it is enough to find a match of a redex with the form in (2) below (see also Figure 5). We denote a node as $K_{(\vec{x})\vec{y}}$, which means that the node has control K, its free ports are linked to the outer names in $\vec{y}$, and the inner names $\vec{x}$ are linked to its binding ports.

$$(K_{(\vec{x}w\vec{y})\vec{z}} \parallel \mathsf{id}_{\langle 1,\vec{b}\vec{c}\rangle}) \circ (\mathsf{id}_{\langle 1,\vec{x}w\vec{y}\rangle} \parallel N_{(\vec{a})\vec{b}w\vec{c}}) \tag{2}$$

for all controls $K, N \in \Sigma$. Intuitively, if a bigraph has a match for a redex with the form in (2), that match is a counterexample for P_Σ (the binding port targeted by w of the K-node has as peer the port linked to w of the another node N which is not beneath the K-node).

Now we can define the binding match predicate.

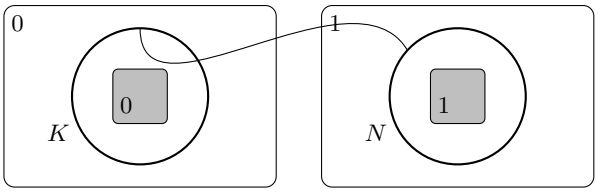


Fig. 5. A (simplified) example of ill-formed bigraph.

Definition 17. *Let Σ be a binding signature and let $\mathcal{R}_{bind}$ be the following set of bigraphs:*

$$\mathcal{R}_{bind} = \{(K_{(\vec{x}w\vec{y})\vec{z}} \parallel \mathsf{id}_{\langle 1,\vec{b}\vec{c}\rangle}) \circ (\mathsf{id}_{\langle 1,\vec{x}w\vec{y}\rangle} \parallel N_{(\vec{a})\vec{b}w\vec{c}}) \mid K, M \in \Sigma\}$$

Then define M_Σ as the match predicate defined on the (recursive) set $\mathcal{R}_{bind}$.

Theorem 5. *The category of local bigraphs corresponds to the one obtained by applying the match predicate sorting on M_Σ over the morphisms of* **Big**.

Proof. It follows immediately from the characterization given in [7, Section 6.4] and by the fact that $P_\Sigma = M_\Sigma$. $P_\Sigma \subseteq M_\Sigma$ can be proved noticing that any match of a redex from R_Σ in a bigraph f of $\mathbf{Big}(U(\Sigma))$ is a counterexample for $P_\Sigma(f)$; whereas $M_\Sigma \subseteq P_\Sigma$ follows immediately from the fact that R_Σ has a redex for any pair of controls in Σ and for any binding port. □

7 Conclusions

In this paper, we have investigated the decidability problem of (bigraphical) sortings. In particular, we have shown the undecidability of Debois and coauthors' predicate sortings and then we have identified a proper sub-class of them, named *match predicate sortings*, which turned out to be decidable. For the match predicate sortings, we have proposed a characterization that induces the definition of a decision procedure to check if a given morphism in the unsorted category has a pre-image into the sorted one, which holds independently from the chosen predicate. The procedure is based on the bigraphical matching problem, for which a decision algorithm was proposed in [2].

Notably, our match predicate sortings preserve many interesting properties of predicate sortings, such as the possibility of describing unwanted bigraphs by means of BiLog formulae. Moreover, we have shown that the match predicate sortings are powerful enough to capture two important bigraphical sortings proposed in literature: *homomorphic sorting* and *local bigraphs*.

As possible future developments, we plan to investigate if other decidable classes of sortings exist and if there are other (possibly) more efficient algorithms to decide if a bigraph belongs to a sortings (remark that the matching problem for bigraphs is NP-complete). Finally, another interesting future work is the analysis of the problem in a more general setting, not focusing only on bigraphs.

References

1. H. Barendregt. *The lambda calculus: its syntax and its semantics*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.
2. L. Birkedal, T. C. Damgaard, A. J. Glenstrup, and R. Milner. Matching of bigraphs. *Electr. Notes Theor. Comput. Sci.*, 175(4):3–19, 2007.
3. L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, and H. Niss. Bigraphical models of context-aware systems. In L. Aceto and A. Ingólfsdóttir, editors, *Proc. FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2006.
4. L. Birkedal, S. Debois, and T. T. Hildebrandt. Sortings for reactive systems. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2006.
5. M. Bundgaard, A. J. Glenstrup, T. T. Hildebrandt, E. Højsgaard, and H. Niss. Formalizing higher-order mobile embedded business processes with binding bigraphs. In D. Lea and G. Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.
6. S. Ó. Conchúir. Kind bigraphs. *Electr. Notes Theor. Comput. Sci.*, 225:361–377, 2009.
7. S. Debois. *Sortings and Bigraphs*. PhD thesis, IT University of Copenhagen, 2008. `http://www.itu.dk/people/debois/pubs/thesis.pdf`.
8. A. Glenstrup, T. Damgaard, L. Birkedal, and E. Højsgaard. An implementation of bigraph matching. *IT University of Copenhagen*, 2007. `http://www.itu.dk/~tcd/docs/implBigraphMatching.pdf`.
9. D. Grohmann and M. Miculan. Reactive systems over directed bigraphs. In L. Caires and V. T. Vasconcelos, editors, *Proc. CONCUR 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2007.
10. O. H. Jensen. *Mobile Processes in Bigraphs*. PhD thesis, University of Aalborg, 2008. To appear.
11. O. H. Jensen and R. Milner. Bigraphs and transitions. In *Proc. POPL*, pages 38–49, 2003.
12. J. J. Leifer and R. Milner. Transition systems, link graphs and petri nets. *Mathematical Structures in Computer Science*, 16(6):989–1047, 2006.
13. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
14. R. Milner. Bigraphs whose names have multiple locality. Technical Report 603, University of Cambridge, CL, Sept. 2004.
15. R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation*, 204(1):60–122, 2006.
16. R. Milner. Local bigraphs and confluence: Two conjectures. In *Proc. EXPRESS 2006*, volume 175(3) of *Electronic Notes in Theoretical Computer Science*, pages 65–73. Elsevier, 2007.
17. E. L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944.
18. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

Generating Instance Graphs from Class Diagrams with Adaptive Star Grammars

Berthold Hoffmann[1] and Mark Minas[2]

[1] Universität Bremen and DFKI Bremen, Germany
[2] Universität der Bundeswehr München, Germany

Abstract. In model-driven software engineering, class diagrams are used to define the structure of object-oriented software and valid object configurations, i.e., what objects may occur in a program and how they are related. Object configurations are essentially graphs, so that class diagrams define graph languages. Class diagrams are declarative, i.e., it is quite easy to check whether a graph is an instance of a class diagram. Graph grammars, on the other hand, define a graph language by derivation and are thus well suited for constructing instance graphs. This paper describes how a class diagram can be translated into a graph grammar that defines the same graph language as the original class diagram. Such a graph grammar may then be used for, e.g., automatically generating valid object configurations as test cases. In contrast to earlier attempts, the presented approach allows to translate class diagrams with arbitrary multiplicities, unique and non-unique associations, composition associations, and class generalization. This is made possible by using adaptive star grammars, a special kind of graph grammars.

1 Introduction

The model-driven design of software relies on the precise specification of models, often with diagrams in the *uniform modeling language*, UML. Class diagrams are the sub-language of UML for specifying the structure of object-oriented classes and possible configurations of objects that are instances of these classes. Configurations primarily consist of sets of class instances and links between them. Links are instances of the class diagram's associations. Configurations can be considered as graphs, called *instance graphs* in the following: each object corresponds to a node, and each link corresponds to a directed edge where source and target are used to distinguish the two roles of the (binary) link. The set of all object configurations that are compatible with a class diagram, therefore, corresponds to a language of graphs, i.e., a class diagram specifies a graph language.

It is easy to check whether a given object diagram is compatible with a class diagram. In contrast, constructing sample instance graphs for a class diagram is rather difficult. This, however, is important to generate test cases for a software model. In [6], graph transformation rules have been used to do this. Here we use an adaptive star grammar [3–5] for this purpose. The approach described in [6] supports only restricted meta models (e.g., class diagrams with constraints): it

does not distinguish *unique* from *non-unique* associations, considers only very restricted association multiplicities, and does not cover composite associations. Adaptive star grammars, as shown in this paper, allow for a fairly straightforward treatment of all of these concepts. Moreover, adaptive star grammars do not need additional control mechanisms like negative application conditions or prioritizing some rules over others by rule layers as it is necessary in [6].

The rest of the paper is structured as follows. We briefly introduce adaptive star grammars in the next section, before we recall some properties of UML class diagrams in Section 3. In Section 4, the main part, the rules generating instance graphs of class diagrams are defined and explained. The conclusions (Section 5) mention some related and future work.

2 Adaptive Star Grammars

Graph grammars generalize the idea of Chomsky grammars to graphs: A set of rules defines how the graphs of the language can be derived by applying them to a given initial graph.

Node replacement and hyperedge replacement [7] have been studied most thoroughly as grammars for deriving graph languages. Their rules remove a nonterminal node, and attach a replacement graph to its neighbor nodes. The sort (i.e., the label) and the direction of the edge connecting a neighbor to the nonterminal determine completely how the neighbor is attached to the replacement graph; in hyperedge replacement, the number of neighbors is even fixed for every nonterminal. Both formalisms specify context-free compositions of graphs in the sense of [2]; however, they fail to define even simple languages, such as the class of all graphs. *Adaptive star grammars* [3] overcome these limitations by means of a cloning mechanism that makes its rules more powerful.

Let us briefly define the concepts needed; for more detailed definitions, see [5] and for a more thorough discussion [3].

Graphs. Let the set C of *sorts* be the disjoint union of finite disjoint sets $\dot{C}$ and $\bar{C}$ for labeling nodes and edges, respectively. We distinguish a subset $N \subseteq \dot{C}$ of *nonterminal names*.

A *graph* $G = \langle \dot{G}, \bar{G}, s_G, t_G, \dot{\ell}_G, \bar{\ell}_G \rangle$ consists of finite sets $\dot{G}$ and $\bar{G}$ of *nodes* and *edges*, respectively., *source* and *target functions* $s_G, t_G \colon \bar{G} \to \dot{G}$, and functions $\dot{\ell}_G \colon \dot{G} \to \dot{C}$ and $\bar{\ell}_G \colon \bar{G} \to \bar{C}$ assigning a sort to each node and edge, respectively. A node $x \in \dot{G}$ is called *nonterminal* if $\ell_G(x) \in N$, and *terminal* otherwise.

For a node x in G, the subgraph consisting of x and its adjacent nodes and incident edges is denoted by $G(x)$. A *border node of* x is a node in $\dot{G}(x) \setminus \{x\}$. Note that graphs may have several edges with the same source, target, and label; such edges are called *parallel*. In the following, we assume that the reader is familiar with common graph terminology, such as subgraph, disjoint union and isomorphism.

Rules and Replacement. Adaptive star grammars are based upon a simple kind of graph transformation that replaces a subgraph $G(x)$ by another graph.

For this, define a *rule* $r = \langle y, R\rangle$ to be a pair consisting of a graph R with a distinguished node $y \in \dot{R}$. We call $R(y)$ and $R \setminus \{y\}$ the left- and right-hand side of r, respectively.

Let G be a graph with a node $x \in \dot{G}$ such that $G(x) \cong_g R(y)$ for some isomorphism g. Then the graph $H = G[x \,/_g\, r]$ is obtained from the disjoint union of G and R by identifying $R(y)$ with $G(x)$ according to the isomorphism g and removing x and its incident edges.

Multiple Nodes. To make graphs (and rules) adaptive, we distinguish a subset $\ddot{G} \subseteq \dot{G}$ of terminal nodes in a graph G as *multiple nodes*, similar to the set nodes of Progress [12]. A multiple node x represents any number of ordinary nodes, which are called *clones of* x. (The nodes $\dot{G} \setminus \ddot{G}$ are called *singular*.) In figures, a multiple node is distinguished by drawing it with double lines (see Example 1 below). A graph that does not contain any multiple node is said to be *singular*. Note that nonterminal nodes are always singular.

Cloning. Let G be a graph. A function $\varrho\colon \ddot{G} \to \mathbb{N}$ is a *replicator for* G. The graph G^{ϱ} is obtained from G by *cloning* each node $x \in \ddot{G}$ according to ϱ, by replicating x and its incident edges $\varrho(x)$ times. If $\varrho(x) = 0$, then x and its incident edges are simply deleted.

Adaptive Star Grammars. Call a graph *simple* if it contains neither adjacent nonterminal nodes nor indistinguishable edges, i.e., parallel edges. A graph of the form $G(x)$ is a *star* if it contains neither loops nor parallel edges, such that x is nonterminal and its border nodes are terminal. An *adaptive star rule over* C is a rule $r = \langle y, R\rangle$, where R is a simple graph with sorts in C, and $R(y)$ is a star. A *clone of* r is a rule $\langle y, R'\rangle$ such that R' is simple, and there is a replicator ϱ for R such that R' can be obtained from R^{ϱ} by identifying some of the border nodes of y with each other (where, of course, only nodes of the same sort can be identified).

An *adaptive star grammar* (ASG) is a system $\Gamma = \langle C, \mathcal{P}, Z\rangle$, $\mathcal{P}$ is a finite set of adaptive star rules, and Z is the initial star, which has no multiple border nodes. (All sorts are taken from C.) Given a graph G, we write $G \Longrightarrow_{\mathcal{P}} H$ if $H = G[x \,/\, r]$ for some node $x \in G$ and a clone r of an adaptive star rule in $\mathcal{P}$. The *adaptive star language* generated by Γ is the set of all terminal graphs G that can be derived from Z:

$$\mathcal{L}(\Gamma) = \{G \mid Z \Longrightarrow_{\mathcal{P}}^{+} G \text{ and } \dot{\ell}_G(x) \in \dot{C} \setminus N \text{ for all } x \in \dot{G}\},$$

where $\Longrightarrow_{\mathcal{P}}^{+}$ denotes the transitive closure of $\Longrightarrow_{\mathcal{P}}$.

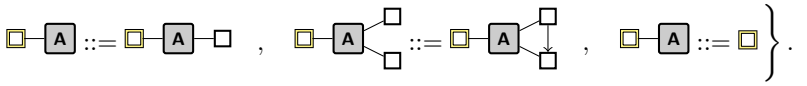
Late Cloning. In the terminology of [3], these definitions make use of *early cloning*, where neither the graphs in derivations, nor the clones of rules contain multiple nodes. However, it requires to "guess" in advance how many clones of a multiple node must be made, which is not always adequate. Especially for constructing derivations, it is better to do cloning as late as possible. In the following, we will use *late cloning*, a corresponding way of constructing derivations, which has been considered in [3] as well.

A *late replicator* $\ddot{\varrho}\colon \ddot{G} \to \mathbb{N} \times \mathbb{N}$ sends multiple nodes to pairs of numbers that represent the numbers of singular and multiple nodes that shall be made of a multiple node, respectively. The graph $G^{\ddot{\varrho}}$ is an adaptive graph wherein, for every multiple node $x \in \ddot{G}$ with $\ddot{\varrho}(x) = (n, m)$, x has $n + m$ clones, whereof m are designated as multiple.

In order to apply an adaptive rule $r = \langle y, R\rangle$ to an adaptive graph G with late cloning, a combined late replicator $\ddot{\varrho}\colon G(x) \cup R(y) \to \mathbb{N} \times \mathbb{N}$ is used to clone both G and r so that $G(x)^{\varrho} \cong R'(x)$ where R' is simple and obtained from R^{ϱ} by identifying some of its border nodes. A step using late cloning is denoted as $G \Rightarrow_{\mathcal{P}} H$ again; H is obtained by applying R' to G^{ϱ}. Like early cloning, late cloning does not restrict the language of singular graphs generated by an adaptive star grammar [3]. [3]

Example 1 (The Language of Unlabeled Graphs). As an example, consider the adaptive star grammar Γ which is given by $\Gamma = (C, \mathcal{P}, Z)$, where $\dot{C} = \{\square, \mathbf{A}\}$, $\bar{C} = \{\text{-}\}$, $Z =$ 

, and

$\mathcal{P} =$ 

A rule $\langle y, R\rangle$ is drawn as $R(y) ::= R \setminus \{x\}$. The grammar derives arbitrary graphs without loops over the "invisible" sorts $\square$ and -, that is, the class of finite unlabeled graphs. Starting with Z, the first rule makes it possible to add an arbitrary number of border nodes. The second rule adds edges between border nodes, and the third removes the nonterminal node.

Let us briefly discuss a major difference between the definition of adaptive star grammars used here (that has already been introduced in [5]) and the one in [3]. Rules can be applied to subgraphs $G(x)$ that are not stars, but contain parallel edges. Note that, although the left-hand side $R(y)$ of an adaptive star rule $r = \langle y, R\rangle$ is a star, cloning it prior to application may involve taking a quotient that identifies border nodes of y in R^{ϱ} with each other. This may sound alarming, as it was shown in [4] that quite a similar type of adaptive star grammars can generate all recursively enumerable languages. However, note that we restrict this ability to the case where the resulting right-hand side R' is simple. In particular, R' must not contain indistinguishable edges. As a consequence, adaptive star grammars of the sort defined above can be simulated by ordinary ones (i.e., those in [3]) by using subsets of $\bar{C}$ to label edges in stars and turning every rule into a finite number of rules (corresponding to the allowed quotients).

3 Class Diagrams

Class diagrams are a well-known graphical language of the UML. The focus of this paper is on the specification of graphs without attributes. Hence, we ignore

[3] In that paper, we consider conditions under which a late replicator is *minimal*; this is not necessary here.

method and attribute specifications within classes. They can be easily added if required. Moreover, we ignore associations with cardinality greater than 2 and newer concepts of class diagrams like association sub-setting or redefinition.

The class diagrams used in this paper consist of *classes* and (binary) *associations* between them. *Concrete* classes are distinguished from *abstract* classes. Each association has a name, and its two end-points carry *role names*. For simplicity, however, we assume an implicit ordering on the role names, draw associations as directed edges, and omit role names.[4] Each end-point of an association is equipped with a *multiplicity* of the form $u..v$ where $u \in \mathbb{N}_0$ is the lower bound and $v \in \mathbb{N} \cup \{*\}$ the upper bound such that $u \leqslant v$ if $v \in \mathbb{N}$. As usual, $*$ stands for "infinity". An association may be declared *unique* (the default) or *non-unique*, indicated by the annotation '{non-unique}'. We distinguish *regular* associations (the default) from *composite* associations. The latter have a black diamond at one of their end-points. The multiplicity at this end-point is either 0..1 or 1..1. The class at the end-point with the diamond is called *composite*, the class at the other end-point is called its *part*. Edges representing composite associations are always directed towards the part class. A class may be composite and part of the same composite association at the same time. However, no object can be part of two objects, and no object is (directly or indirectly) part of itself. Finally, class diagrams may contain *generalization* arrows, which point from sub-classes to super-classes. Each class may have an arbitrary number of sub-classes and super-classes. However, generalization arrows are not allowed to form cycles. We usually extend the notion of sub-classes and super-classes to all classes that are reachable by chains of generalization arrows (including chains of length 0).

A class diagram specifies graphs by the mechanism of instantiation: A node is an *instance* of a class iff it is labeled with the class name. An instance of a class C is called *member* of C and also of each super-class of C. An edge e is an *instance* of an association from a class C_1 to a class C_2 if e is labeled with the association label, and if the source and target nodes are members of C_1 and C_2, respectively. Moreover, each association defines a *multiplicity constraint*: Let a be an association from class C_1 to class C_2 with multiplicities $u..v$ at its source end-point and multiplicity $r..s$ at its target end-point. The multiplicity constraint of a is satisfied iff no member of C_1 has less than r or more than s outgoing a-instances as edges and if no member of C_2 has less than u or more than v incoming a-instances as edges.

The graph language specified by a class diagram consists of all graphs that satisfy the following conditions:

C_1: Each node is an instance of a concrete class, and each edge is an instance of an association.
C_2: The multiplicity constraints of all associations are satisfied.
C_3: No two instances of any unique association are parallel edges.
C_4: The subgraph induced by composite edges (i.e., instances of composite associations) must be a collection of trees.

[4] These directed edges should not be confused with navigation arrows, which solely represent implementation issues in object-oriented programming.

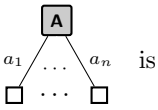
Such graphs are called *instance graphs* in the following. This definition follows the UML specification when nodes are considered as objects and edges as links [11].

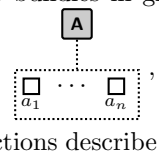
4 An Adaptive Star Grammar for Instance Graphs

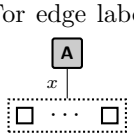
In the following, we assume an arbitrary, but fixed class diagram. We first describe how this class diagram can be translated into an ASG defining the same graph language. The set of terminal node sorts, hence, must consist of the names of all concrete classes. The set of edge sorts must contain all association names, but also additional edge sorts that will be used for labeling edges between nonterminal and terminal nodes. These edge sorts, but also nonterminal nodes sorts, will be introduced as needed in the following.

The idea of translating a class diagram into an adaptive star grammar defining the same graph language is to start from a graph (called *actual initial graph*, AIG, in the following) that closely resembles the class diagram. We add an initial rule that derives the AIG from the initial star that consists of just a single node with a unique nonterminal label. For each concrete class, the AIG contains a multiple node labeled with the class name. Cloning these nodes in a derivation creates the instances of the corresponding classes. The AIG's other nodes are nonterminal nodes representing the associations of the class diagram: Each *regular* association is represented by a single node with a unique nonterminal label. It is connected to all multiple nodes of those concrete classes that participate in the association, possibly by being a subclass of a class at an endpoint of the association. A set of rules is responsible for eventually creating all edges representing instances of the association, i.e., links between objects. The rules make sure that conditions C_2 and C_3 (see Sect. 3) are satisfied. The construction is more complicated for *composite* associations since it must make sure that condition C_4 is not violated. Actually, each connected subgraph of the class diagram consisting of composite associations and generalizations only has to be represented by a nonterminal node with a unique label. Rules must be defined that create all possible trees consistent with the class diagram.

These constructions are described in the following. But first, we introduce some notation that makes drawing of rules easier: In rules, corresponding nodes of the left- and right-hand sides are associated by their positions in the drawing. Terminal nodes in rules are always unlabeled; on the left-hand side of a rule, they match terminal nodes with arbitrary labels. Because none of the following rules introduces new terminal nodes on the right-hand side (new terminal nodes are rather created by cloning), there is no need for labeling them. Finally, we use a simplified notation for bundles in graphs: For edge labels $a_1, \ldots, a_n$, bundle



is drawn as


, and as


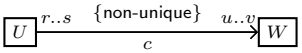
if $a_1 = \cdots = a_n = x$.

The next two sub-sections describe the construction for regular associations, the first one for associations declared "non-unique", the second for "unique as-

sociations". The presented construction must be repeated for each regular association within the class diagram. Afterwards, we then discuss the construction for composition associations, and finally present an example.

4.1 Non-Unique Regular Associations

Consider an association

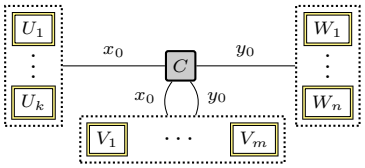


where $r, s, u, v \in \mathbb{N}_0, r \leqslant s, u \leqslant v$

within the class diagram. Classes U and W may be arbitrary classes, U and W may even reference the same class, or U may be a sub-class of W or vice versa. We assume the upper bounds s and v to be finite, i.e., different from "$*$". However, the following construction can be extended easily to the case $s = *$ and/or $v = *$ as we will discuss briefly at the end of this subsection.

Let $\mathbb{U}$ and $\mathbb{W}$ be the sets of concrete sub-classes of U and W, respectively. Note that $\mathbb{U}$ and $\mathbb{W}$ contain U and W, respectively, if they are concrete classes. Note also that $\mathbb{U} \cap \mathbb{W}$ contains all concrete classes being sub-classes of both U and W. Each instance of a class in $\mathbb{U} \setminus \mathbb{W}$ has $u..v$ outgoing, but no incoming c-edges, each instance of a class in $\mathbb{W} \setminus \mathbb{U}$ has $r..s$ incoming, but no outgoing c-edges, and each instance of a class in $\mathbb{W} \cap \mathbb{U}$ has $u..v$ outgoing as well as $r..s$ incoming c-edges. Since the association is non-unique, parallel c-edges are permitted. The following construction assures these properties.

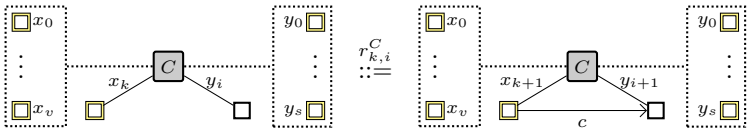
Let $\{U_1, \ldots, U_k\} = \mathbb{U} \setminus \mathbb{W}$, $\{W_1, \ldots, W_n\} = \mathbb{W} \setminus \mathbb{U}$, and $\{V_1, \ldots, V_m\} = \mathbb{U} \cap \mathbb{W}$. Note that each of these sets may be empty. Let C be a nonterminal label that is unique for this c-association, and let $\{x_0, x_1, \ldots, x_v\} \cup \{y_0, y_1, \ldots, y_s\}$ be a set of pairwise distinct edge labels. The AIG must contain the following graph as a subgraph:



The meaning of the edges labeled x_k and y_i is the following: An x_k-edge connects C with a terminal node that has k outgoing c-edges already, and an y_i-edge connects C with a terminal node that has i incoming c-edges already. Note that terminal nodes with labels in $\mathbb{W} \cap \mathbb{U}$ may have incoming as well as outgoing c-edges; they may even have c-loops where each loop counts as an incoming and as an outgoing edge. The number of incoming and outgoing edges of terminal nodes with labels in $\mathbb{W} \cap \mathbb{U}$ is indicated by two edges labeled x_k and y_i, respectively.

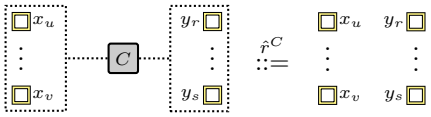
We may add another c-edge between two appropriate nodes where the source node must have $k < v$ outgoing c-edges and the target node $i < s$ incoming c-edges. This is the task of the following set of productions $r^C_{k,i}$, $0 \leqslant k < v$

and $0 \leqslant i < s$, that add a new c-edge between two nodes and increment the "counters" realized by edge labels.



Note that such a rule can also be applied in situations where nodes are connected with two parallel edges labeled x_k any y_i. The corresponding border nodes have to be identified in those cases. A c-loop is added if the two singular border nodes get identified.

We can stop adding c-edges if there is no x_k-edge and no y_i-edge with $k < u$ and $i < r$; we can then remove the nonterminal node C representing the association. This is realized by the following final rule $\hat{r}^C$

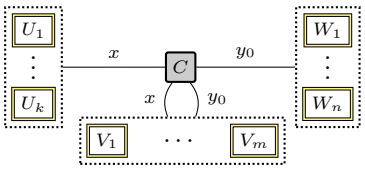


The presented construction uses the fact that v and s are finite numbers and different from *. The construction, however, is extended easily to the case $v = *$ and/or $s = *$. Let us assume just $v = *$. The new meaning of an x_u-edge connecting C with a terminal node is that the terminal node has *at least* u outgoing c-edges already. And we change rule $r^C_{u,i}$, such that it no longer creates a new x_{u+1}-edge, but an x_u-edge again. It is clear that this construction now creates any number of outgoing c-edges, but at least u, at appropriate nodes.

4.2 Unique Regular Associations

We now consider the slightly different situation with a *unique* instead of a *non-unique* association; the other aspects remain unchanged.

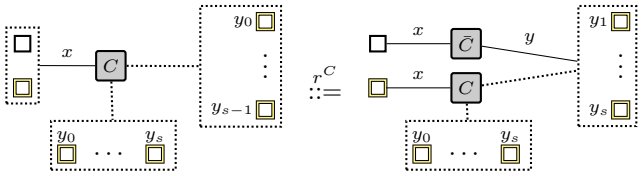
Let C and $\bar{C}$ be two nonterminal labels that are unique for this c-association, and let $\{x, y_0, y_1, \ldots, y_s\}$ be a set of pairwise distinct edge labels. The AIG must contain the following graph as a subgraph:



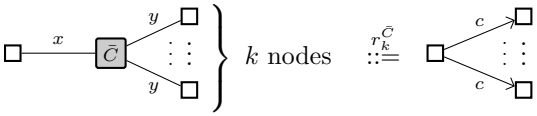
The meaning of y_i-edges is the same as in the previous section, i.e., a terminal node connected to C with a y_i edge means that the terminal node has i incoming c-edges already. However, we do not count outgoing c-edges. Instead, an x-edge

between a terminal node and C means that the terminal node has not yet any outgoing c-edge.

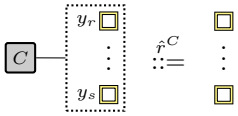
Since parallel c-edges are prohibited, we cannot create c-edges independently from each other like in the previous subsection. Instead, we use rules that create all outgoing c-edges of any singular node in a single derivation step; an x-edge visits those terminal nodes that receive a bundle of outgoing c-edges to pairwise distinct nodes visited by y_i-edges. We define a rule r^C that selects one singular node visited by an x-edge and some of the terminal nodes visited by y_i-edges. The latter nodes will receive new incoming c-edges. Rule r^C, therefore, increments the "counters" realized by y_i-edges:



The rule adds a new nonterminal node labeled with $\bar{C}$. The following set of rules $r^{\bar{C}}_k$, $u \leqslant k \leqslant v$, adds a bundle of c edges from the node visited by the x-edge to the k nodes visited by the y-edges.



Finally, we can remove the C-node as soon as all terminal nodes connected to C with an x-edge have been processed by rule r^C. This is the task of rule $\hat{r}^C$:



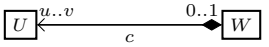
Like in the previous subsection, these rules use the fact that the upper multiplicity bounds v and s are finite and different from *. The case $s = *$ can be realized similarly to the discussion at the end of the previous subsection. As an example, see Sect. 4.4. The case $v = *$ is even simpler: The set of rules $r^{\bar{C}}_k$ gets replaced by a single rule $r^{\bar{C}}$ that looks like $r^{\bar{C}}_{u+1}$; however, one of the border nodes visited by an y-edge is turned into a multiple node. Hence, this production can add bundles with an arbitrary number of c-edges, but at least u.

4.3 Composite Associations

With the constructions of the last two subsections, we can translate each class diagram whose associations are all regular into an adaptive star grammar defining the same graph language. Composite associations are special since subgraphs induced by composite edges must be collections of trees (condition C_4 in Sect. 3).

The ASG, hence, must be able to create all such trees. In order to translate this part of a class diagram into an ASG, we have to find those classes whose instances can belong to the same tree. This is done by finding the largest connected subgraphs of the class diagram that are made up by composite associations and generalizations only. The instance nodes of the classes that belong to the same connected subgraph may, but need not, belong to the same tree. This can be represented in the ASG in the following way: We add a new nonterminal node to the AIG for each of these connected subgraphs and connect this nonterminal node to each multiple node representing a concrete class within the subgraph. Then we define star rules that create all possible trees consistent with the class diagram.

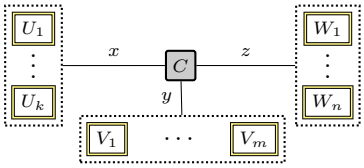
We demonstrate this procedure for just a single composite association as shown here. However, the construction can be generalized easily to more composite associations in the same connected subgraph.



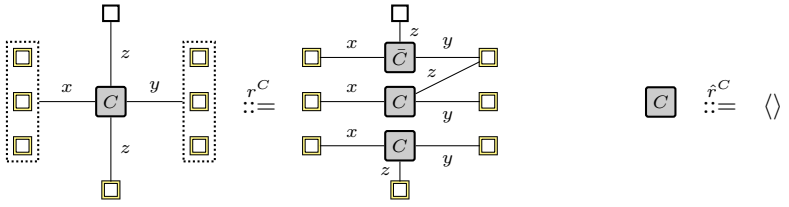
where $u, v \in \mathbb{N}_0, u \leqslant v$

Again, classes U and W may be arbitrary classes, U and W may even reference the same class, or U may be a sub-class of W or vice versa.

Let the sets $\mathbb{U}$, $\mathbb{W}$, $\{U_1, \ldots, U_k\} = \mathbb{U} \setminus \mathbb{W}$, $\{W_1, \ldots, W_n\} = \mathbb{W} \setminus \mathbb{U}$, and $\{V_1, \ldots, V_m\} = \mathbb{U} \cap \mathbb{W}$ be defined as in the beginning of Sect. 4.1. Let $C, \bar{C}$ be two nonterminal labels that are unique for this c-association, and let x, y, z be distinct edge labels. The AIG must contain the following graph as a subgraph:



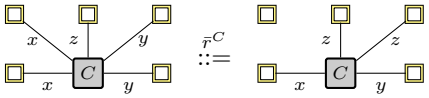
The nonterminal node C stands for all trees with instances of classes in $\mathbb{U} \cup \mathbb{W}$ as nodes connected by c-edges; z-edges visit those terminal nodes (roots) that will "receive" $u..v$ outgoing, but no incoming c-edges. Nodes visited by y-edges (inner nodes) will "receive" $u..v$ outgoing and possibly one incoming c-edge. Finally, nodes visited by x-edges (leaves) will "receive" at most one incoming, but no outgoing c-edge. These properties are assured by the following rules.



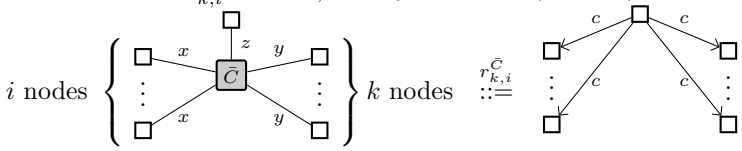
The recursive rule r^C selects a singular root (visited by a z-edge) and recursively proceeds with the rest of the roots by adding a new C-node connected to the

multiple node representing the rest of the roots. Moreover, r^C creates a $\bar{C}$-node that will be derived by a rule $r^{\bar{C}}_{k,i}$ (see below) to a bundle of c-edges from the root to its children. Those children that are inner nodes become roots of sub-trees, indicated by the other created C-node. The recursion stops as soon as all nodes have been processed, represented by rule $\hat{r}^C$, which replaces a C-node by the empty graph $\langle\rangle$.

The rules r^C and $\hat{r}^C$ make sure that each member of U receives an incoming c-edge. However, this is actually not required by the association. Rule $\bar{r}^C$, therefore, allows to turn any inner node into a root and any leaf into a singleton tree node. This rule must be dropped if the multiplicity of association c is 1..1 instead of 0..1 at the composite end-point.



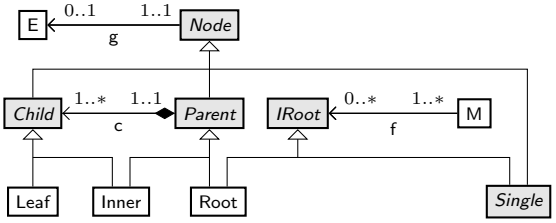
Finally, $\bar{C}$-nodes must be derived to bundles of $u..v$ many c-edges. This is the task of the set of rules $r^{\bar{C}}_{k,i}$ for all $i, k \in \mathbb{N}_0$ so that $u \leqslant i + k \leqslant v$:



Note that this construction can be easily extended to the situation where $v = *$. The set of rules $r^{\bar{C}}_{k,i}$ must be replaced by the set of rules $r'^{\bar{C}}_{k}$ such that $0 \leqslant k \leqslant u$; $r'^{\bar{C}}_{k}$ looks like $r^{\bar{C}}_{k+1,u-k+1}$, but two of the border nodes, one being visited by an x-edge, the other with an y-edge, are turned into multiple nodes. Hence, these rules can add bundles with an arbitrary number of c-edges, but at least u.

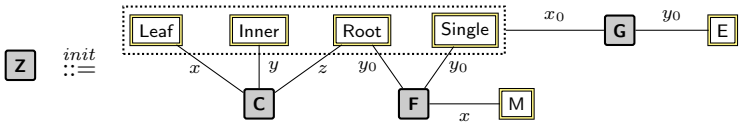
4.4 Example

Consider the following class diagram specifying trees where each tree node may be connected to an E-object. "Manager"-objects of class M are connected to root nodes:



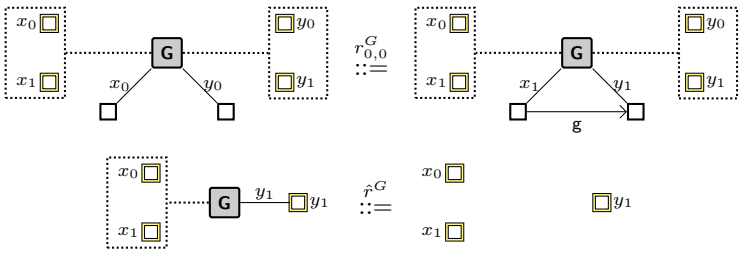
We assume that the association f is unique, and that g is not (which is actually irrelevant for an m-to-1 association). The association c is a composite association.

Then the initial rule for the initial star **Z**, deriving to the AIG, looks as follows:



The rules for the composition c are r^C and $\hat{r}^C$ as in Sect. 4.3, but without $\bar{r}^C$ since the lower bound of the association at the composite end-point is 1. Because the upper bound at the part end-point is *, we need the following rules for $\bar{C}$:

The rules for association g are constructed as shown in Sect. 4.1 with $r = s = v = 1$ and $u = 0$:



The rules for association f are constructed as described in Sect. 4.2 with $r = 1, u = 0, s = v = *$. Because of $s = *$, rule r^F differs from the one shown in Sect. 4.2: an y_1-edge means that the connected terminal node has at least one incoming f-edge. Moreover, as discussed at the end of Sect. 4.2, there is only a single rule $r^{\bar{F}}$ creating $0..*$ f-edges from an $\bar{F}$-node:

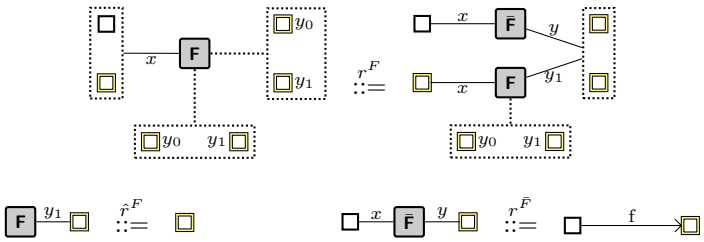


Fig. 1 shows a sample derivation of an instance graph using these rules. Note that node labels have been abbreviated by initial letters of class names. The derivation arrows $d_1, \ldots, d_9$ do not represent single derivation steps, but have the following meaning: d_1 clones the multiple R-node to a singular as well as

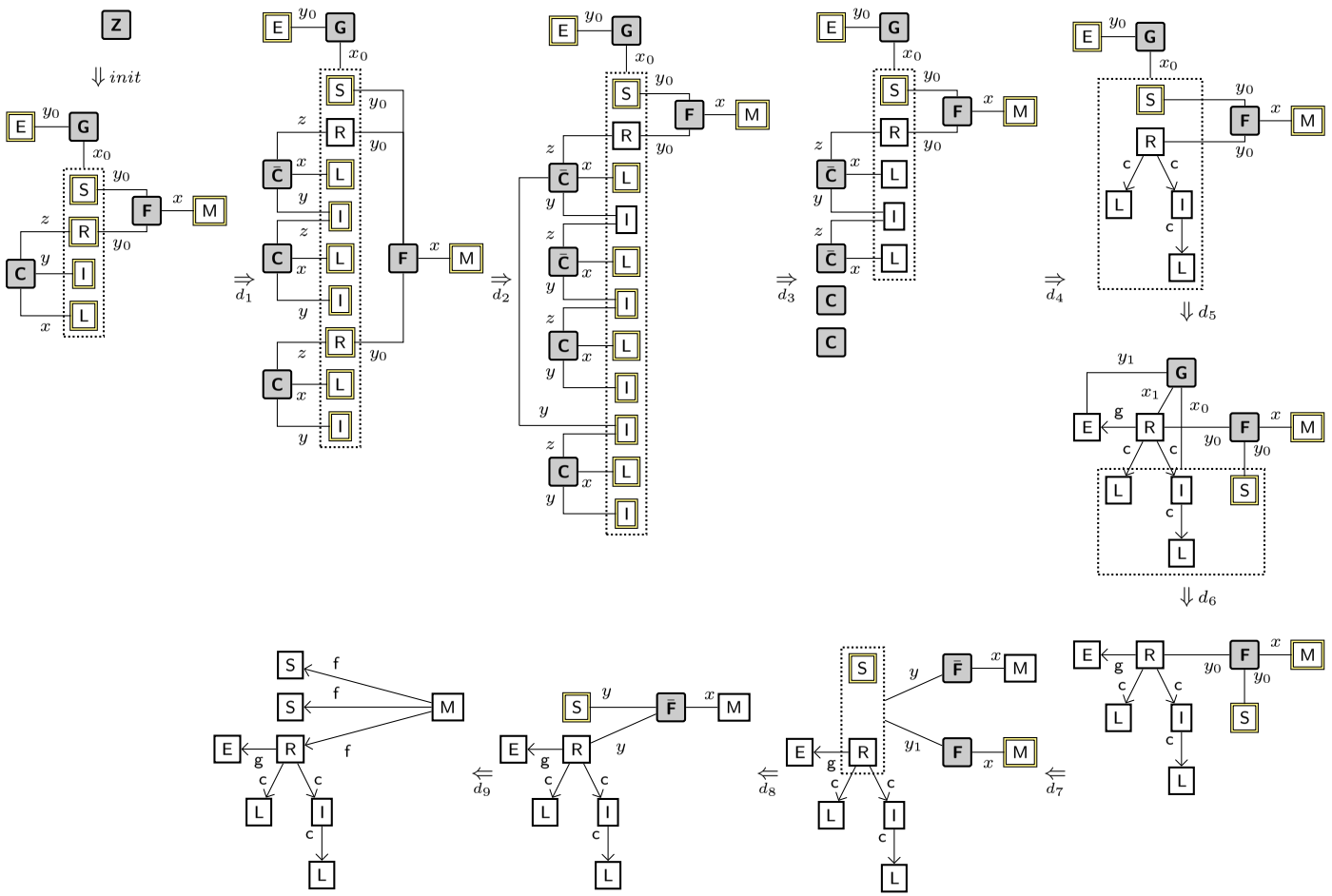


Fig. 1. A sample derivation of an instance graph (to be read clockwise)

another multiple one, clones the I- and L-nodes three times each, and applies rule r^C. d_2 clones the topmost I-node to a singular and another multiple one, deletes the R-, L-, and I-nodes at the bottom (by replicating them 0 times), and applies r^C again after cloning the remaining I- and L-nodes three times each. d_3 deletes the 6 lowermost multiple nodes and clones the remaining two multiple L-nodes to singular ones. d_4 applies $\hat{r}^C$ twice, deleting the isolated C-nodes, and then $r_0'^{\bar{C}}$ twice after adapting the rule to the context of the corresponding $\bar{C}$-node. d_5 applies $r^G_{0,0}$, and d_6 applies $\hat{r}^G$, deleting the G-node. d_7 applies r^F; both the R- and the S-node get visited by y_1-edges. d_8 deletes the multiple M-node by replicating it 0 times, and applies $\hat{r}^F$, deleting the F-node. Finally, d_9 clones the S-node twice and applies r^F.

5 Conclusions

We have described how class diagrams can be translated into an adaptive star grammar that defines the same language of instance graphs as the class diagram. The translation process works for all class diagrams using generalizations, containing unique or non-unique associations and also composite associations. Associations can have arbitrary multiplicities. We have shown the translation process explicitly for regular associations with arbitrary multiplicities, and we have outlined the translation and demonstrated the situation with a single composition association.

The presented approach closes the gap between class diagrams as a declarative approach for defining instances and graph grammars providing a constructive definition. That way, techniques available for graph grammars become available for class diagrams, too. Creating test cases automatically by simply creating sample derivations is just one example.

The only closely related work, to the best of our knowledge, is by K. Ehrig, J. M. Küster, and G. Taentzer [6], extending the work by R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer [1] considerably. In [6], they translate meta models into a graph grammar that defines the same language of instance graphs. However, they do not consider composite associations, they restrict multiplicities only to very simple cases, and they do not distinguish unique from non-unique associations. Moreover, their generated graph grammars must make heavy use of negative application conditions in the form of negative context graphs, and they must prioritize some rules over others in the form of layered graph grammars. This complicates the grammar and, what is more important, reasoning about the grammar and the generated graph language. On the other hand, they are able to cover some (basic) meta model constraints by translating them into application conditions. So far, we have not considered yet how meta model constraints can be translated along with the class diagram into an adaptive star grammar, possibly with application and graph conditions. A related, but not general approach has been described in previous work [10] where rules must not be applied if their application conditions are not satisfied. In future work, we intend to make use of generalized results on context conditions and their relation

to logical graph properties and constraints by A. Habel and K.-H. Pennemann [8] as well as A. Habel and H. Radke [9].

Furthermore, we will extend the presented construction to newer concepts of class diagrams like association sub-setting and redefinition in future work. Instantiating such associations in instance graphs primarily means selecting the correct association depending on the classes of the connected nodes.

References

1. R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In *Proc. Fundamental Approaches to Software Engineering (FASE'04)*, LNCS 2984, pp. 214–228. Springer, 2004.
2. B. Courcelle. An axiomatic definition of context-free rewriting and its application to NLC rewriting. *Theoretical Computer Science*, 55:141–181, 1987.
3. F. Drewes, B. Hoffmann, D. Janssens, and M. Minas. Adaptive star grammars and their languages. Theoretical Computer Science (2010), doi:10.1016/j.tcs.2010.04.038.
4. F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. V. Eetvelde. Adaptive star grammars. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *3rd Int'l Conf. on Graph Transformation (ICGT'06)*, LNCS 4178, pp. 77–91. Springer, 2006.
5. F. Drewes, B. Hoffmann, and M. Minas. Adaptive star grammars for graph models. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *4th Int'l Conf. on Graph Transformation (ICGT'08)*, LNCS 5214, pp. 201–216. Springer, 2008.
6. K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Software and System Modeling*, 8(4):479–500, 2009.
7. J. Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3: Beyond Words, ch. 3, pp. 125–213. Springer, 1999.
8. A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.
9. A. Habel and H. Radke. Expressiveness of graph conditions with variables. In H. Ehrig and C. Ermel, editors, *Int'l Colloquium on Graph and Model Transformation (GraMoT'10)*, 2010. To appear in Electr. Comm. of the EASST.
10. B. Hoffmann and M. Minas. Defining models – meta models versus graph grammars. In *Proc. 6th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'10), Paphos, Cyprus.* Appears in Electr. Comm. of the EASST, vol. 29, 2010.
11. *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2.* OMG Document Number: formal/2009-02-04.
12. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, ch. 13, pp. 487–550. World Scientific, Singapore, 1999.

Formalizing Models with Abstract Attribute Constraints ⋆ ⋆⋆

Márk Asztalos, Péter Ekler, László Lengyel, Tihamér Levendovszky, and Tamás Mészáros

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
{asztalos, peter.ekler, lengyel, tihamer, mesztam}@aut.bme.hu

Abstract. Offline verification of model processing programs has become a fundamental issue in model-based software development. Offline means that only the definitions of the program and the concerned modeling languages are taken into account. Therefore, the results of the analysis hold for every possible input model. Although the offline analysis is very complex, it must be performed only once. In our work, we concentrate on the verification of graph rewriting-based model transformations. Previous work has presented an automated framework for the verification of model transformations. As a part of our framework, we propose a formalism to describe models with abstract attribute constraints in this work. The operations of our framework are based on this mathematical background. On a case study of refactoring mobile-centric social network models, we demonstrate that our framework is able to verify important properties automatically by the declarative description of model transformations, which is based on the contributed formalism for models.

1 Introduction

With the increasing need of reliable systems, the verification of model processing programs has become a fundamental issue in model-based software engineering, where verification covers the analysis of non-functional properties of the model processing programs (e.g. termination) and the validation of their outputs.

Graph rewriting-based model transformation is a frequently used technique for defining programs that work on models formalized as graphs. In our terminology, a *model transformation* is the definition of a model processing program that is based on graph rewriting systems [6] and is specified by a set of rewriting rules (based on the double-pushout approach) as well as an additional control structure that explicitly defines the execution order of the rules.

⋆ This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

⋆⋆ This work is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BUTE" project. This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

When they are feasible, *offline* verification methods are extremely useful in industrial applications. A verification technique is called *offline* if only the definition of the program and the specification of the languages that describe the models to be transformed are used during the analysis process. Therefore, the results of the offline analysis are general in the sense that they are independent from the concrete input models. Although the offline analysis is very complex (e.g. the termination of a graph transformation is undecidable in general [10]), it must be performed only once. Several techniques can be found for the verification of graph rewriting-based model transformations, however, these methods usually lack generalization possibilities, since the analysis is performed manually or the methods can be applied only to a certain transformation class or to the analysis of a certain type of property only. Therefore, there is an increasing need for automated verification methods and tools (for a more detailed discussion, see [2]). The goal of our research is to provide an offline, automated verification framework for the analysis of graph rewriting-based model transformations. This paper outlines the concept of our verification approach, and we introduce a formalism for describing metamodels, models, model patterns with attribute constraints. The verification of the model transformation is based on the analysis of a declarative formal description of the programs. This paper provides the essential mathematical background for this declarative description by formalizing patterns that are used to specify the left-hand side and right-hand side of the graph rewriting rules. The complete presentation of the declarative description of the transformations would exceed the limits of this paper, but we demonstrate informally on a case study how our formalism makes this description possible.

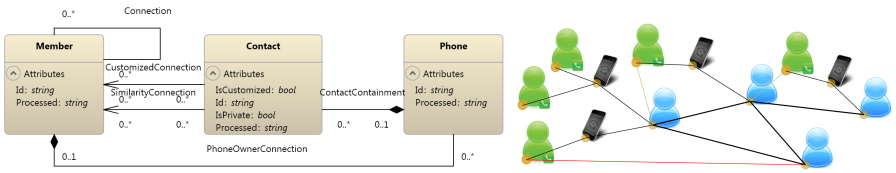
Section 2 presents a case study of refactoring mobile-centric social network models, which will be used to demonstrate our verification concepts. Section 3 introduces the main concepts of our verification methods informally. After the mathematical background is presented in Section 4, we provide the contribution of this paper in Section 5 and demonstrate its applicability on the case study in Section 6. Finally, we describe the related work in Section 7 and summarize our results in Section 8.

2 Case Study

Phone books in the mobile devices represent social relationships that can be integrated into social networks. *PhoneBookMark* is a phone book-centric social network implementation by Nokia Siemens Networks [7]. We took part in the project and, before the public introduction, it was available for a group of general users: it had 420 registered members with more than 72000 private contacts.

Visual Modeling and Transformation System (VMTS) [14] is a metamodeling and model transformation framework. In VMTS, we have created a domain-specific modeling environment *PhoneBookMark* . The entities of its metamodel are presented in Figure 1a: a *member* is a user of the social network, a *phone* is a mobile device of a member, which can contain phone book entries, a *contact* corresponds to a phone book entry of a phone. Relations between the entities have also been defined: each member can own several phones (*PhoneOwnerConnection*), each phone can contain several contacts (*ContactContainment*).

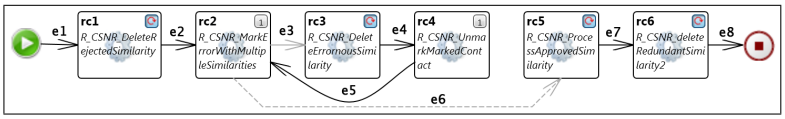
A contact can be connected to a member with a *CustomizedConnection* or a *SimilarityConnection* edge. A *CustomizedConnection*, or shortly customization edge, means that the current entry corresponds to the member of the social network. Whenever the owner member of the entry connects to the social network, the data can be synchronized. *PhoneBookMark* provides a semi-automatic similarity detecting and resolving mechanism, which detects similarities between phone book contacts and the members of the network. Similarity means that the algorithm suggest to the user that the contact and the member represent the same person. In this case, a *SimilarityConnection*, or shortly, a similarity edge is created between the contact and the appropriate member, later, the user has to decide the acceptance of this relation. For this purpose, *ApprovalState* attribute has been defined for similarity edges, whose value can be *approved*, *rejected*, or, the default value, *ignored*, which means that the user has not made a decision yet. During the refactoring of a model, approved edges will be converted to customization edges and rejected edges will be deleted from the model. In VMTS, the domain-specific environment for *PhoneBookMark* includes the metamodel and a concrete syntax for the instance models. A sample model is presented in Figure 1b. The entities can be easily distinguished by their icons and colors. Similarity edges are denoted by red, customization edges by goldenrod colors.



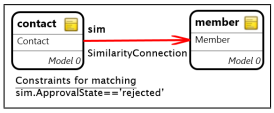
(a) *PhoneBookMark* Metamodel (b) Sample Instance Model

Fig. 1: *PhoneBookMark* Domain in VMTS

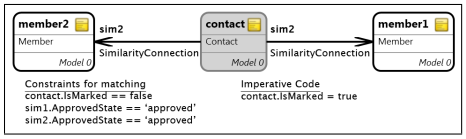
In VMTS, the model transformations are based on graph rewriting and are defined with the use of two modeling languages: the Visual Control Flow Language (VCFL) and the Visual Transformation Definition Language (VTDL) [1]. The activity diagram-like VCFL models controls the execution order of the rewriting rules, while the rewriting rules are described with VTDL models. The application of the rules is based on the double pushout approach [6]. We have implemented a model transformation (*Similarity Handling Transformation*) that refactors *PhoneBookMark* models, it processes the rejected, approved, and ignored similarity edges. The control flow graph of the transformation, as implemented in VMTS, is presented in Figure 2a. The dashed, gray control flow edges are followed if the application of the source rules was unsuccessful, which happens when no matches of the left-hand side can be found. The solid, gray edges are followed if the application of the previous rule was successful, while solid black edges are always followed. Rules with a circle in the top right bottom are executed exhaustively, which means that the rules are applied repeatedly, until they cannot be applied any more. Figure 2 contains the definition of the rules of the transformation. In VMTS, the left-hand side and right-hand side of the rules are merged, elements that are deleted by the rule are red, newly created



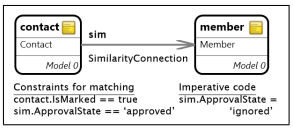
(a) Transformation Control Flow



(b) Rule *rc1*

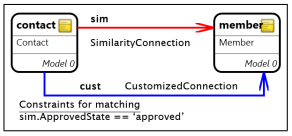


(c) Rule *rc2*

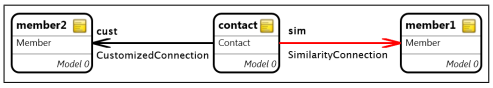


(d) Rule *rc3*

(e) Rule *rc4*



(f) Rule *rc5*



(g) Rule *rc6*

Fig. 2: Similarity Handling Transformation

are blue and the attributes of gray elements are modified. In Figure 2, we show the definition of the rules in VMTS along with the attached constraints and imperative code for the modification of the attributes.

In the following, two edges are called *forking* if they have the same source node. The requirements of this transformation are as follows: (i) Inconsistent states, when the model has forking approved edges (approved similarity edges that have the same source contact), should be identified. In this case, all forking approved edges should be modified to be in ignored state. (ii) All rejected similarity edges should be deleted. (iii) In a consistent state, all approved similarity edges should be transformed to a customization edge. (iv) Whenever a customization edge c is present, all similarity edges that are forking with c should be deleted. We will use the Similarity Handling Transformation to present our verification techniques in the following sections.

3 Offline Verification Methods

In this section, we provide the informal outline of our verification approach based on our previous work [3] [2] [4]. Recall that we restrict ourselves to the analysis of model processing programs based on graph rewriting systems, which are defined by a set of rewriting rules and an additional control structure. We mentioned that such a program is referred to as *model transformation*. We assume that the control structure is a directed control flow graph, which meets the following conditions. Start node and end nodes are used to mark the starting point and possible end points of the transformations, other nodes of the transformation are the rewriting rules. The flow edges of the control flow graph define the execution order. Also, branches can be defined. We assume that the output is always the modified input model, but this is not a restriction of the generality of the model

transformations, since assuming that we have multiple input and multiple output models, we can always compose their union and treat them as a single model.

Assume that we have a language (MCDL [2]) that is able to express the verifiable properties of the output models of the transformations. For example, given the Similarity Handling Transformation presented above, we may need to express the following properties of the output models (the properties themselves are in italic): **(v1)** *After the application of the transformation, no approved similarity edge should be present in the model.* Each approved edge should be transformed to a customization edge, or should be deleted if there are more than one approved similarity edge from the same contact. **(v2)** *After the application of the transformation, no rejected similarity edge should be present in the model.* All rejected similarity edges should be deleted. **(v3)** *After the application of the transformation, it is forbidden that a contact has a similarity and a customization edge at a time.* In this case, the similarity edge should have been deleted. **(v4)** *After the application of the transformation, it is forbidden that a contact has two customization edges at a time, provided that before that transformation started this pattern was also forbidden.* This would result in an inconsistent state.

Model Condition Inference Logic (MCIL) [2] [4] is an inference logic that is able to analyze logical implications such as $\phi_1 \Rightarrow \phi_2$, where Greek letters denote MCDL formulae. The result of the analysis can be the proof, the refutation, or the result that the implication is undecidable. For example, given an MCDL formula φ stating that *after the application of the transformation, the approval state of all similarity edges in the model will be ignored*, we can derive the first two verifiable formulae presented above from φ, because if all similarity edges are in ignored state, it implies that there are neither approved nor rejected edges. For MCIL, we have defined several extensible deduction rules.

In the following, we detail the main concept of our verification approach that is based on the components MCDL and MCIL. Given a model transformation, assume that we are able to assign MCDL formulae to each control flow edge of the transformation such that given a flow edge f, the formula ϕ_f assigned to f is a property that is satisfied by the model under transformation at its current state when the execution of the transformation reaches f. If we have only one end node in the control flow and it has only one incoming edge, and the formula ϕ_{final} is assigned to this edge, ϕ_{final} will be satisfied by all possible output models of the transformation. During the analysis of a model transformation, the goal of our methods is to produce these *assignments*. Its main benefit is as follows: given a property of the output models that should be validated, which is described as an MCDL expression ϕ_{ver}, if we can prove $\phi_{final} \Rightarrow \phi_{ver}$, then the property is validated. Another benefit is that we assign formulae to all flow edges, which helps locating the problematic points while debugging.

Obviously, the main question is how to produce the assignments. The start node of the transformation has one outgoing edge. The formula assigned to this edge is called the initial formula. It is known, since this is the condition that must be satisfied by all possible input models. Assume that we have a rule in the transformation with several incoming edges and one outgoing edge, and we have

already assigned formulae to the incoming edges. In other words, we know some properties of the model under transformation when the execution reaches a rule. We have the formal, declarative definition of the rule, therefore, by its definition, we may derive certain properties that will be true after the application of the rule. These properties described in MCDL can be assigned to the outgoing edge. This method is called the *propagation of formulae through a rule*, which is a very complex task itself, it depends on the MCDL formulae, and the definition of the rule. The goal of our methods is to collect the most information in the formulae that are assigned to the edges. However, if nothing can be derived, it will not imply the failure of our algorithm, only that the assigned formulae will not contain relevant information, therefore, the verifiable properties could not be derived from them by MCIL.

The contribution of this paper is a formalism to define patterns of models and describe metamodels and models. The declarative description of the transformations, which makes the creation of the assignment possible by automatic algorithms, and the MCDL language are based on this formalism as we demonstrate it in Section 6.

4 Mathematical Background

This section summarizes the mathematical background of typed graphs [5, 6].

Definition 1 (grap and graph morphism). *A graph $G = (N, E, s, t)$ consists of a set N of nodes, E of edges and two functions $s, t : E \to N$, the source and target functions. The elements L_G of a graph G are its nodes and edges, i.e. $L_G = N_G \cup E_G$. Given graphs $G_1 = (N_1, E_1, s_1, t_1)$ and $G_2 = (N_2, E_2, s_2, t_2)$, a graph morphism $f : G_1 \to G_2$, $f = (f_N, f_E)$ consists of two functions $f_N : N_1 \to N_2$ and $f_E : E_1 \to E_2$ that preserve the source and target functions, i.e. $f_N \circ s_1 = s_2 \circ f_E$ and $f_N \circ t_1 = t_2 \circ f_E$.*

Definition 2 (type graph with inheritance and inheritance clan). *A type graph with inheritance is a double (G_T, I) consisting of a type graph $G_T = (N, E, s, t)$ (with a set N of nodes, a set E of edges, a source and a target function $s, t : E \to N$), and inheritance graph I sharing the same set of nodes N. Given a type graph with inheritance $T = (G, I)$ For each node n in N ($N \equiv N_I \equiv N_G$), the inheritance clan is defined by $clan_I(n) = \{n' \in N | \exists$ path $n' \to^* n$ in $I\}$ where path of length 0 is included, i.e. $n \in clan_I(n)$.*

Given a node n in a type graph, $clan_I(n)$ is the set of nodes that are inherited from n, moreover, $clan_I(n)$ also contains n. Instance of a type graph, (a graph typed over a concrete type graph), is defined by the instance graph itself and a special type morphism that assigns an element of the type graph to each element of the instance graph. The type morphism should take inheritance into account and is called clan morphism.

Definition 3 (clan morphism, instance graph and typed morphism). *Given a type graph with inheritance $T = (G, I)$, and a graph H, a clan morphism $\tau : H \to T$ consists of two functions $\tau_N : N_H \to N_G$, $\tau_E : E_H \to E_G$ such that:*

(i) $\forall e \in E_H : \tau_N \circ s_H(e) \in clan_I(s_G \circ \tau_E(e))$*, and (ii)* $\forall e \in E_H : \tau_N \circ t_H(e) \in clan_I(t_G \circ \tau_E(e))$*. Given a type graph (with inheritance)* T*, a double* (G, τ) *of a graph* G *along with a clan morphism* $\tau : G \to T$ *is called an instance of* T*.* G *is said to be typed over* T*. Given type graph* T *and two instance graphs* (G_1, τ_1)*,* (G_2, τ_2)*, a graph morphism* $f : G_1 \to G_2$ *typed over* T *is a graph morphism, such that* $\tau_2 \circ f = \tau_1$*.*

5 Verification Framework

In this section, we introduce a formalism to provide the mathematical background of our verification framework. The main components what we want to formalize are: (i) *metamodels*: types of entities and relations, with the names of the attributes; (ii) *models*: entities and relations typed over a metamodel and attribute values assigned to each possible attribute; (iii) *patterns*: a model pattern that contains model elements and abstract *attribute constraints*; (iv) *matches*: formal mapping between patterns or between patterns and models.

The key concept in our approach is the handling of attributes and attribute constraints. We handle attribute types and values in an abstract form. For example, when we formalize the *PhoneBookMark* metamodel, we have an element *Contact* with attributes Id and $IsPrivate$, but we do not define the types and possible values of the attributes, only their names. The concept is similar to the way in which we refer to the elements of a set. Therefore, we say that we define only the *interface* of the metamodel. Attribute constraints are also handled by their interface without explicitly defining their language. An attribute constraint over a pattern is a logical function with parameters that refer to the attributes of the pattern elements. If the values of the attributes are known, the function can be evaluated. For example, given an element c in a model, let the type of c be *Contact*. If we defined that the Id attribute would be a natural number, we could write an attribute constraint $c.Id > 5$. However, in our case, without explicitly defining the types of the attributes, we need to handle this constraint in a more abstract level. The constraint will be a function f such that f has one parameter, and returns a logical value. Moreover, we need to specify that the single parameter of f is mapped to the Id attribute of element c. Given another constraint such as $c.Id < 4$, a constraint logic may derive that the two previous constraint can never be true at the same time, or they are *conflicting*. The conflict can be treated as an abstract relation between the constraints, and instead of relying on a concrete constraint logic we work with abstract attribute constraints and such relations. The main benefit of this abstraction is that our framework can be easily implemented and integrated into any tool, where complex languages are used for the specification of attributes and constraints, for example C# in the case of VMTS.

5.1 Metamodel Interfaces and Abstract Attribute Constraints

We formalize the interface of metamodels by specifying the types of the entities (nodes) and relations (edges) along with the attributes. Recall that we do not deal with types of the attributes, i.e. we only define their *names*.

Definition 4 (metamodel interface). *A metamodel interface $\mathfrak{M}$ is a triple $(T, \mathcal{A}, \sigma)$, where T is a type graph (with inheritance), $\mathcal{A}$ is a set of attributes that are defined on the elements of the type graphs, and $\sigma : L_T \rightarrow 2^{\mathcal{A}}$ is the attribute assignment function that assigns a set of attributes to each element in the type graph. Because of the inheritance of attributes, the following condition must hold: $\forall n, n' : n, n' \in N_T, n' \in clan_I(n) \Rightarrow \sigma(n) \subseteq \sigma(n')$.*

An abstract attribute constraint over a model consists of a function, which evaluates the value of the constraint. The return value of such a function is the logical value *true* or $false$. The function takes the values of the attributes as parameters, therefore, we need additional functions that map each parameter to a certain element of the model and to a certain attribute of the element. We can evaluate constraints by assigning values to the referenced attributes.

In the following definitions, the set of natural numbers from a to b is denoted by $[a, b]$. Moreover, to facilitate the formalization, assume that we have a set $\mathcal{V}$ that contains all possible attribute values.

Definition 5 (abstract attribute constraint). *Given a metamodel interface $\mathfrak{M} = (T, \mathcal{A}, \sigma)$ and an instance graph G typed over T by clan morphism τ. An abstract attribute constraint c over G is defined by the triple $(\varepsilon, \varrho, \omega)$, where: (i) $\varepsilon : \mathcal{V}^n \rightarrow \{true, false\}$, $n > 0$, true and $false$ denote the logical constants; (ii) n is the number of the parameters of function ε, called the arity of c and is denoted by $n = |c|$; (iii) $\varrho : [1, n] \rightarrow L_G$; (iv) $\omega : [1, n] \rightarrow \mathcal{A}$, such that $\forall i \in [1, n] \Rightarrow \omega(i) \in \sigma(\tau \circ \varrho(i))$.*

Definition 6 (attribute value assignment). *Given a metamodel interface $\mathfrak{M} = (T, \mathcal{A}, \sigma)$, and a graph G typed over T by clan morphism τ. An attribute assignment is a function $v : L_G \times \mathcal{A} \rightarrow \mathcal{V}$, where $\mathcal{V}$ denotes the set of all possible attribute values. An attribute value assignment v is called complete if $\forall l, a : l \in L_G, a \in \sigma(l) \Rightarrow \exists v(l, a)$, otherwise, it is called partial. An assignment is called empty, denoted by $v_\emptyset$, if it does not assign a value to any of the attributes. Given an abstract attribute constraint c over G, we say that v is complete with respect to c if $\forall i \in [1, |c|] \Rightarrow \exists v(\varrho(i), \omega(i))$.*

Definition 7 (evaluation of a constraint). *Given a constraint $c = (\varepsilon, \varrho, \omega)$ over a graph G, and an attribute value assignment v that is complete with respect to c, the evaluation of c is the evaluation of the function ε as follows: $\varepsilon(v(\omega(1)), v(\omega(2)), ...v(\omega(n)))$. We say that G satisfies c with respect to v (i.e. $G \vDash_v c$) if the return value of the evaluation is true, otherwise c is not satisfied ($G \nvDash_v c$).*

Definition 8 (attribute value assignment extension). *Given a graph G typed over a metamodel $\mathfrak{M}$, and a partial value assignment v. We say that another value assignment v' over G is an extension of v (denoted by $v' \rhd v$) if $\forall l, a : l \in L_G, a \in \sigma(l), \exists v(l, a) \Rightarrow \exists v'(l, a) \wedge v'(l, a) = v(l, a)$.*

Although we handle attribute constraints in an abstract level, we may need to explicitly specify that an attribute has a certain value, which can be expressed by a special abstract attribute constraint.

Definition 9 (equality constraint). *Given a metamodel interface $\mathfrak{M}$, a graph G typed over $T_{\mathfrak{M}}$ by the clan morphism τ, an equality constraint is an abstract attribute constraint specified by the triple (l, a, ν), where $l \in L_G$, $a \in \sigma(\tau(l))$, and $\nu \in \mathcal{V}$. The abstract attribute constraint $(\varepsilon, \varrho, \omega)$ is as follows: (i) $n = 1$, (ii) $\varrho(1) = l$, (iii) $\omega(1) = a$, (iv) $\varepsilon : \mathcal{V} \to \{true, false\}$, such that $\varepsilon(\nu) = true$, $\forall \nu' \neq \nu \Rightarrow \varepsilon(\nu') = false$.*

The relations *conflict* and *derivation* of constraints are defined as follows:

Definition 10 (relations of constraints). *Given two sets C_1, C_2 of abstract attribute constraints over the same graph G, and a value assignment v over G.*

- *C_1 and C_2 are **conflicting** (or are in conflict) with respect to v, denoted by $C_1 \otimes_{v'} C_2$, if $\nexists v' \rhd v$ such that v' is complete, where $G \vDash_{v'} C_1$ and $G \vDash_{v'} C_2$.*
- *C_2 is **derivable** from C_1 with respect to v, denoted by $C_1 \vdash_v C_2$, if $\forall v' \rhd v$ such that v' is complete, and $G \vDash_{v'} C_1 \Rightarrow G \vDash_{v'} C_2$.*

The opposite of the previous two relations are as follows:

- *C_1 and C_2 are **not conflicting** with respect to v, if $\exists v' \rhd v$ such that v' is complete and $G \vDash_{v'} C_1, G \vDash_{v'} C_2$.*
- *C_2 is **not derivable** from C_1 with respect to v, if $\exists v' \rhd v$ such that v' is complete and $G \vDash_{v'} C_1$, but $G \nvDash_{v'} C_2$.*

Remark 1. In the previous definitions, if v is the empty assignment, we simply ignore the term 'with respect to v', and we say that C_1 and C_2 are (not) in conflict, or C_2 is (not) derivable from C_1.

Obviously, given two constraint sets C_1 and C_2, they are *conflicting*, or *not conflicting*, no other options are possible. However, it depends on the constraint logic if the relation can be determined. It may happen that none of these relations can be proved. Since we assume that we do not know the concrete functions in the constraints, we will use these relations during the analysis, for this purpose, we introduce two functions and assume that both of them are available as global functions in our system:

- The function $conflicting(C_1, C_2, G, v)$ takes two sets C_1, C_2 of constraints, a graph G, on which the constraints are defined, and a (possibly empty) value assignment v. Given any possible parameters, this function returns a value either *true*, $false$, or *unknown*. The value *true* means that it can be proved that C_1 and C_2 are conflicting, $false$ means that it can be proved that C_1 and C_2 are not conflicting, and *unkown* means that neither can be proved, i.e. the system does not have enough information.
- The function $derivable(C_1, C_2, G, v)$ takes two sets C_1, C_2 of constraints, a graph G, on which the constraints are defined, and a (possibly empty) value assignment v. Given any possible parameters, this function returns a value either *true*, $false$, or *unknown*. The value *true* means that C_2 can be proved to be derivable from C_1, $false$ means that it can be proved that C_2 is not derivable from C_1, and *unknown* means that neither can be proved, i.e. the system does not have enough information

The assumption that we have these two functions seems to be very restrictive, since in complex attribute constraint description languages, it is really hard to determine the relation between arbitrary constraints. However, the implementation of the previous functions always have the possibility to return the value *unknown*. For example, assume that we have an instance graph of the *PhoneBookMark* metamodel, with a single *Contact* element c. The constraints $c.Id > 5$, $c.Id < 5$ can be proved to be in conflict if the implementation of the function $conflicting$ contains some constraint logic based on intervals of natural numbers. However, it is also possible that $conflicting$ is not implemented in that way in our system and it cannot prove this relation. In this case the return value will be *unknown*. The analysis of the transformations and the verification of the properties of the models are based on the relations of certain constraint sets in several cases. If the relation between the sets can be determined by the current $conflicting$ and *derivable* functions, we can obtain more information and derive the proof of more properties. However, if the relations between the sets are unknown, the system may not be able to prove certain properties. Hence, the usability and efficiency of our framework is largely depends on the capabilities of the current implementation of the previous two functions.

5.2 Patterns of Models

In order to handle type inheritance in matches, we introduce the definition of weakly typed morphisms. One can easily show that typed graphs along with weakly typed morphisms form a category, but its formal presentation would exceed the limits of this paper.

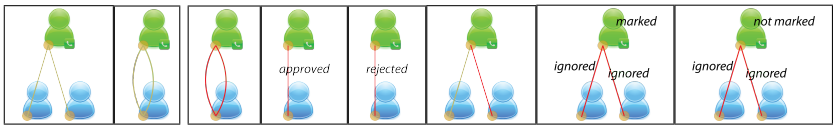
Definition 11 (weakly typed morphism). *Given typed graphs (G_1, τ_1) and (G_2, τ_2) typed over a type graph with inheritance T. Let $\tau_1 = (\tau_1^N, \tau_1^E)$ and $\tau_2 = (\tau_2^N, \tau_2^E)$ be the clan morphisms of G_1 and G_2. A weakly typed morphism $m : G_1 \to G_2$ is a graph morphism such that: (i) $\forall n \in N_{G_1} : \tau_2^N \circ m(n) \in clan_I(\tau_1^N(n))$, (ii) $\forall e \in E_{G_1} : \tau_2^E \circ m(e) = \tau_1^E(e)$.*

Remark 2. Given a type graph T, the category $\mathbf{GraphsWM_T}$ of graphs typed over T with weakly typed morphisms consists of instance graphs of T as objects and weakly typed morphisms as morphisms.

Patterns are instances of metamodel interfaces. A pattern defines the elements of a model part along with additional attribute constraints.

Definition 12 (pattern). *A pattern $P = (G, C)$ of a metamodel interface $\mathfrak{M}$ is an instance graph G of $\mathfrak{M}$ and a set C of abstract attribute constraints defined over G.*

Remark 3. The constraints of a pattern P are also denoted by $C = \mathcal{C}@P$. We say that the elements of a pattern (denoted by L_P) are the elements of the graph of the pattern, i.e. $L_P = L_G$.



(a) P_1 (b) P_2 (c) P_3 (d) P_5 (e) P_6 (f) P_7 (g) P_12 (h) P_13

Fig. 3: Sample Patterns

We have presented several patterns in Figure 3. We use the concrete syntax to present the elements of the patterns. For example pattern P_5 contains a contact, a member, and a similarity edge between them, and we define a constraint which states that the similarity edge should be approved, i.e. the value of the *ApprovalState* attribute must be *approved*. We will use the presented patterns in the next section.

Definition 13 (satisfaction of constraints by patterns). *Given a pattern $P = (G, C)$, an additional set of constraints C' on the graph of P and an attribute value assignment v.*

- *We say that P **satisfies** C' with respect to v (denoted by $P \models_v C'$) if (i) $\exists v' \rhd v$ such that v' is complete with respect to C and $G \models_{v'} C$, and (ii) $\forall v' \rhd v$, where v' is complete w.r.t. C and C': $G \models_{v'} C \Rightarrow G \models_{v'} C'$.*
- *We say that P **does not satisfy** C' with respect to v (denoted by $P \not\models_v C'$), if $\nexists v' \rhd v$, where v' is complete w.r.t. C_P and C', $G \models_{v'} C$ and $G \models_{v'} C'$.*
- *We say that C' is **satisfiable** by P with respect to v if $\exists v' \rhd v$ that v' is complete with respect to C and $G \models_{v'} C$, and $G \models_{v'} C'$.*
- *Otherwise, **the satisfaction** of C' **is not decidable**, denoted by $P \models^? C'$.*

If v is the empty attribute value assignment, we simply say that P satisfies C' ($P \models C'$), or P does not satisfy C' ($P \not\models C'$), etc.

Given a pattern, the satisfaction of other constraints may be decidable by the relation of the constraint sets even when we do not have a complete attribute value assignment.

Proposition 1. *Given pattern $P = (G, C)$, an attribute value assignment v, and an additional constraint set C' over G, and we assume that $\exists v : G \models_v C$.*

- *$derivable(C, C', G, v) = true \Rightarrow P \models_v C'$*
- *$conflicting(C, C', G, v) = true \Rightarrow P \not\models_v C'$*
- *$conflicting(C, C', G, v) = false \Rightarrow C'$ is satisfiable by P*

We need to define the mapping of attribute constraints by morphisms. A morphism maps elements of a pattern into elements of another one. Therefore, in the case of weakly typed morphisms, the constraints on the first graph can be mapped as well.

Definition 14 (mapped constraints). *Given a metamodel interface $\mathfrak{M} =$, two patterns $P_1 = (G_1, C_1)$, $P_2 = (G_2, C_2)$ of $\mathfrak{M}$, and a weakly typed morphism $m : G_1 \to G_2$. For each $c = (\varepsilon, \varrho, \omega) \in C_1$, the mapping $c' = m(c)$ is a constraint on P_2, which exists if $\forall i \in [1, |c|] : \exists m(\varrho(i))$. The mapped constraint c' is an abstract attribute constraint $(\varepsilon', \varrho', \omega')$ on P_2 such that $\varepsilon' = \varepsilon$, $\varrho' = m \circ \varrho$, and $\omega' = \omega$. We can also map a whole set C_1 of constraints, denoted by $m(C_1)$.*

Metamodel interfaces extend the definition of typed graphs with inheritance and patterns extend the definition of instance graphs, therefore, we can define morphisms between patterns as well. A *pattern morphism* is a weakly typed graph morphism between the graphs of the patterns. A pattern morphism describes a possible match of one pattern in another.It is not certain that the match always exists, because the attribute constraints of the second pattern and the mapped constraints of the first pattern may be in conflict. For this reason, we introduce the definition of weak and strong matches that contains more strict conditions on the morphisms.

Definition 15 (pattern morphism). *Given patterns* $P_1 = (G_1, C_1)$, $P_2 = (G_2, C_2)$ *of the metamodel interface* $\mathfrak{M}$, *an injective, total, weakly typed morphism* $p : G_1 \to G_2$ *is called a pattern morphism and is denoted by* $p : P_1 \to P_2$ *if* $conflicting(C_2, p(C_1), G_2) \neq true$. *We say that* p *is a weak match if* $conflicting(\mathcal{C}@P_2, p(\mathcal{C}@P_1), G_2) = false$. p *is a strong match if* $derivable(\mathcal{C}@P_2, p(\mathcal{C}@P_1), G_2) = true$.

Models can be formalized as a special patterns, where we know the values of all attributes. Such a pattern is called a *model pattern*.

Definition 16 (model pattern). *A pattern* P *is called a model pattern, if: (i)* $\forall l, a : l \in L_P, a \in \sigma(l) \Rightarrow \exists$ *a unique* $c \in \mathcal{C}@P$ *such that* c *is an equality constraint on* l *and* a, *(ii) no other attribute constraints are in* $\mathcal{C}@P$.

The equality attribute constraints in a model pattern clearly define the values of the attributes, therefore, a value assignment can be derived. Moreover, it is easy to prove that this assignment is unique and complete.

Definition 17 (attribute value assignment of model patterns). *A value assignment* v_M *by a model pattern* M *is defined as follows:* $\forall$ *equality constraint* $c = (l, a, \nu) \in \mathcal{C}@M \Rightarrow v_M(l, a) = \nu$, *and* $\exists v_M(l, a) \Leftrightarrow \exists c = (l, a, \nu) \in \mathcal{C}@M$.

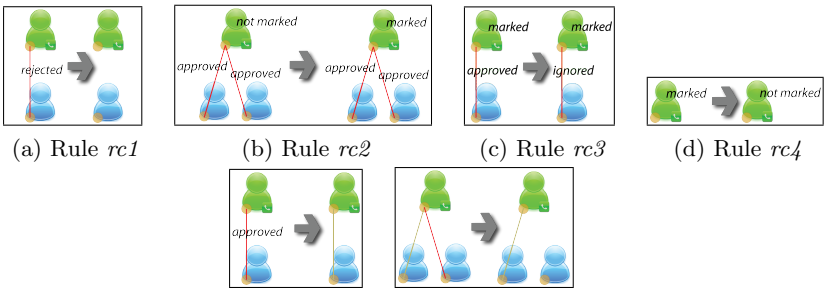
Proposition 2. *Given a model pattern* M, *and a value assignment* v_M *by* M, v_M *is complete and unique.*

An important benefit of the model patterns is that the satisfiability of any constraints on the model pattern can be determined, because we know the values of all attributes (we have v_M, which is complete). This also results that for all possible constraints, their relation (e.g. if they are in conflict) can be also determined with respect to v_M, which results that given a pattern, all matches can be enumerated. The proofs of these statements are trivial by the definitions, exploiting that the only possible extension of v_M is v_M itself.

6 Verification of the Similarity Handling Transformation

In the previous section, we have provided the essential mathematical background to formalize metamodels, patterns, and models with abstract attribute constraints and to formalize matches between patterns. Based on this formalism, model

transformations can be described in a declarative way. The formal presentation of this description would exceed the limits of this paper, but informally, a model transformation is described as a set of rules and an additional directed control flow graph. Each rule is defined by its left-hand side and right-hand side that are *patterns*. For example, the declarative description of the rules of the Similarity Handling Transformation are presented in Figure 4. In the following, we use rc_{rule}^{side} to denote a pattern of a specific rule. For example rc_2^L is the pattern in the left-hand side of $rc2$ and rc_6^R is the pattern in the right-hand side of $rc6$. In these rules, all constraints are equality constraints, therefore, the relations of the constraints can be easily determined, i.e. the functions $conflicting$ and $derivable$ will never return *unknown* in this case study.



(a) Rule *rc1* (b) Rule *rc2* (c) Rule *rc3* (d) Rule *rc4*

(e) Rule *rc5* (f) Rule *rc6*

Fig. 4: Formal Description of the Rules of Transformation Similarity Refactoring

VMTS contains the implementation of a verification framework based on the concepts presented in Section 3. This framework can automatically perform the verification of the Similarity Handling Transformation, which is presented in the following to demonstrate the usefulness of MCDL. For the verification, we need to provide the initial conditions: (i1) there exist no forking customization edges, (i2) forking similarity edges cannot have the same target, and (i3) initially all contacts are *not marked* (the value of the *IsMarked* attribute is *false*). The initial conditions and the verifiable properties presented informally in Section 3 are formally specified in Table 1, let $\phi^{init} = \phi_1^{init} \wedge \phi_2^{init} \wedge \phi_3^{init}$. Given the initial conditions, the discovery algorithm traverses the control flow of the transformation and assigns a formula to each control flow edge. Table 2 shows the discovered formulae. The final formula can be derived, which is $\phi^{final} = \phi_{e8}$ in our case. MCIL can derive all four verifiable properties from the final formula, therefore, all properties can be verified. We point out that VMTS also provides the derivation steps of the inference.

Table 1: MCDL Conditions on *PhoneBookMark* Models

Initial Conditions			Verifiable Properties			
$\phi_1^{init} = \nexists P_1 \wedge \nexists P_2$	$\phi_2^{init} = \nexists P_3$	$\phi_3^{init} = \nexists rc_4^L$	$\phi_1^{ver} = \nexists P_5$	$\phi_2^{ver} = \nexists P_6$	$\phi_3^{ver} = \nexists P_7$	$\phi_4^{ver} = \nexists P_1$

7 Related Work

In this section, we present the work related to the formalism that constitutes the contribution of this paper. For a detailed discussion on offline verification

Table 2: Assignments of MCDL Formulae

Edge	Discovered Formula	Edge	Discovered Formula
e1	$\phi_{e1} = \phi^{init}$	*e5*	$\phi_{e5} = \phi_1^{init} \wedge \phi_2^{init} \wedge \nexists P_6 \wedge \phi_3^{init} \wedge \exists P_1 3$
e2	$\phi_{e2} = \phi^{init} \wedge \nexists P_6$	*e6*	$\phi_{e6} = \phi^{init} \wedge \nexists P_6 \wedge \nexists rc_2^L$
e3	$\phi_{e3} = \phi_1^{init} \wedge \phi_2^{init} \wedge \nexists P_6 \wedge \exists rc_2^R$	*e7*	$\phi_{e7} = \phi_{e6} \wedge \nexists rc_5^L$
e4	$\phi_{e4} = \phi_1^{init} \wedge \phi_2^{init} \wedge \nexists P_6 \wedge \nexists rc_3^L \wedge \exists P_1 2$	*e8*	$\phi_{e8} = \phi_{e7} \wedge \nexists rc_6^L$

methods for graph rewriting-based model transformations in general, see the related work section in our previous paper [2].

Our formalism is based on typed graphs with inheritance as presented in [6]. To use typed attributed graphs, we need to specify the data type algebra. Our abstraction of the attribute constraints makes it possible to work with attributes and constraints without explicitly stating anything about the type of attributes.

[13] presents an approach to formally describe certain parts of graph transformations in order to reason about the transformations in a proof assistant. However, this approach is limited to the structural aspects of graph rewriting.

In [12], [11], a verification method for graph rewriting-based model transformations are presented, and a formal representation is provided for the description of model transformations as declarative relations in Prolog style. The specification of the transformations is not based on rules-based graph grammars, but uses a textual description based on a relational, declarative calculus. The presented representation can be directly translated into representations for theorem provers. One of the key differences between this approach and that presented in our paper is the handling of attributes. In [12], attribute values are explicitly defined in the declarative description of the rules, while our approach can contain arbitrary constraints by the definition of abstract attribute constraints.

[9] presents a notation of structural transformations, namely programs with interface, that are a generalization of programs over transformation rules. The presented formalism makes it possible to analyze and verify the programs. Nested conditions that can express the verifiable properties are also defined. Similarly to the concept presented in [8], [9] assumes that the language of attribute constraints is defined by a data signature and algebra. In our approach, we do not rely on these formalisms, the definition of abstract attribute constraints make it possible to work with any type of imperative and constraint description language (e.g. C# in VMTS) during the implementation of the verification system.

8 Conclusions

The verification of model processing programs is a fundamental issue. This paper concentrates on the verification of graph rewriting-based model transformations. Moreover, we focus on the automated, offline verification of such programs. We have implemented a verification framework in our modeling tool VMTS, and in this paper, we have outlined the concept of the verification. This paper provides the essential mathematical background to describe model transformations formally and declaratively based on the contribution of this paper, the definition of patterns with abstract attribute constraints. The declarative description makes it possible to perform reasoning about the properties of the transformation in an

offline way. The use of abstract attribute constraints makes it easier to realize our verification methods in real-world model transformation frameworks with complex languages to specify attribute constraints and write imperative code. In a case study of refactoring social network models, we presented the outline of the transformation verification based on the formalism presented in this paper. In future work, we plan to test our methods framework in industrial case studies.

References

1. Angyal, L., Asztalos, M., Lengyel, L., Levendovszky, T., Madari, I., Mezei, G., Mészáros, T., Siroki, L., Vajk, T.: Towards a fast, efficient and customizable domain-specific modeling framework. In: Software Engineering. IASTED (2009), Innsbruck, Austria
2. Asztalos, M., Lengyel, L., Levendovszky, T.: Towards Automated, Formal Verification of Model Transformations. In: ICST. Paris, France (2010)
3. Asztalos, M., Lengyel, L., Levendovszky, T.: A formalism for describing modeling transformations for verification. In: MoDeVVa. pp. 1–10. ACM, NY, USA (2009)
4. Asztalos, M., Madari, I., Vajk, T., Lengyel, L., Levendovszky, T.: Formal verification of model transformations: an automated framework. In: ICCC-CONTI. pp. 493–498. Timisoara, Romania (May 2010)
5. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In: FASE. pp. 214–228 (2004)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science. An EATCS Series, vol. XIV. Springer (2006)
7. Ekler, P., Lukovszki, T.: Experiences with phonebook-centric social networks. In: CCNC. Las Vegas, USA (2010)
8. Orejas, F.: Attributed graph constraints. In: ICGT. pp. 274–288. Springer-Verlag, Berlin, Heidelberg (2008)
9. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Department of Computing Science, University of Oldenburg (2009)
10. Plump, D.: Termination of graph rewriting is undecidable. Fundam. Inf. 33(2), 201–209 (1998)
11. Schätz, B.: Verification of model transformations. In: Pre-Proc. of GT-VMT. pp. 129–142. ECEASST, Paphos, Cyprus (March 2010)
12. Schätz, B.: Formalization and Rule-Based Transformation of EMF Ecore-Based Models. SLE, Toulouse, France, September 29-30, 2008. Revised Selected Papers pp. 227–244 (2009)
13. Strecker, M.: Modeling and verifying graph transformations in proof assistants. Electr. Notes Theor. Comput. Sci. 203(1), 135–148 (2008)
14. VMTS website: `http://vmts.aut.bme.hu/`

Incremental update of constraint-compliant policy rules

Paolo Bottoni[1] and Andrew Fish[2] and Francesco Parisi Presicce[1]

[1] Department of Computer Science, "Sapienza" University of Rome, Italy
(bottoni,parisi)@di.uniroma1.it
[2] School of Computing, Engineering and Mathematics, University of Brighton, UK
Andrew.Fish@brighton.ac.uk

Abstract. Organizations typically define policies to describe (positive or negative) requirements about strategic objectives. Examples are policies relative to the security of information systems in general or to the control of access to organizations' resources. Often, the form used to specify policies is in terms of general constraints (what and why) to be enforced, with procedures given as rules (how and when). The consistency of the system (procedures causing a transformation from valid states to valid states) can be compromised and rules can violate some constraints when constraints are updated due to changing requirements. In this paper, we propose a systematic way to update rules as a consequence of modifications of constraints, by incrementally modifying one or more of the components of a rule: the left-hand side, the right-hand side, or its application conditions.

1 Introduction

Graph constraints and graph transformations provide a formal model for defining and managing strategic policies adopted by organisations with respect to security or access control issues. Policies can be specified with constraints, while management procedures consistent with these policies are given via rules. In both cases, the models are given in the framework of typed attributed graphs, with special attributes defining the relevant properties, while rules exploit application conditions to enforce consistency. By consistency, we mean that procedures can only cause transformations from valid states to valid states. As a consequence, the class of graphs generated, starting from a valid graph, by using the procedure is a subset of the class of graphs satisfying the constraints.

Usually, policies and procedures are not fixed, but can evolve independently, following the introduction, revision or deletion of constraints and rules, respectively. However, at each moment the current configuration of constraints must be satisfied by the current collection of rules. Two cases can be considered in which the consistency of the system can be compromised: introduction of new rules – which must be adapted to the current constraints – and definition of new constraints, again requiring rule adaptation. The identification of the modifications needed to adapt rules to constraints has been the subject of several studies [8,

5]. However, these usually require the complete analysis of each rule after each modification, without the possibility to adopt an incremental approach, taking into account the relations between the previous sets of constraints and rules.

In this paper, we consider a class of constraint modifications for which such an incremental approach is feasible, to propose a systematic way to update rules as a consequence of modifications of constraints. We show how this can be achieved for rules which were directly derived from the original constraints, thus identifying the part of the rules affected by the modification.

Paper organisation. After presenting related work in Section 2, we provide background notions on graph constraints and graph transformations in Section 3. Section 4 introduces the running example for the paper and Section 5 presents the notion of security policy and the derivation of an incremental procedure for realising a policy. Section 6 discusses modifications to the rules derived from the procedure. Finally, Section 7 draws conclusions and points to future work.

2 Related work

In [8], the notion of security policy framework was defined, given by a type graph, a set of (named) graph transformation rules, and two sets of simple positive and negative constraints, expressed as morphisms from a premise to a conclusion. A framework was considered to be (positive/negative) coherent if all the graphs derivable from the rules and consistent with the type graph respected all the (positive/negative) constraints. A number of constructions were given for repairing possible violations, through the modification of rules by adding application conditions, in situations where different policy frameworks were merged. Based on this approach, a UML-based language was defined in which to specify access control policies [9]. The language exploits graph transformations to give a semantics and a verification method for such specifications. This line of research has also been exploited in [12] to define basic rules for controlling access in workflows. Rules allow the addition/removal of tasks and roles, plus the execution of tasks, and application conditions are defined based on existing constraints.

In [1], we have considered the construction of transformation units ensuring coherence with a simple form of nested constraints, admitting a single alternation of universal and existential quantifiers. The units were defined starting with the simple addition of an element, and producing repair rules to restore violated constraints. The procedures presented here can be exploited in transformation units, but each rule is guaranteed to not violate any constraint.

Habel and Pennemann [5] have extensively treated the problem of making rules compatible with nested constraints by the addition of positive and negative application conditions. They unify theories about application conditions [2] and nested graph conditions [11], lifting them to high-level transformations. An existing rule is transformed so as to make it constraint preserving or constraint guaranteeing, but no direct construction of rules from constraints is given.

Orejas is investigating a new approach relating attributed graph constraints with attribute evaluation (for a summary see [10]). In the restricted form of attributed evaluation adopted here, the approach should produce the same results.

Some works have studied the problem of constructing instances of metamodels, specialised by additional constraints. In [7] constraints are given in a logical language, while in [4] they are given in OCL and tested after rule derivation from the metamodel. Following the definition of [7], our approach leads to a soundness preserving construction, rather than a completeness preserving one.

While [8] studied the modifications required by the introduction of new constraints, or of merging systems of constraints, the consequences of modifying existing constraints, to weaken or strengthen them, were not analysed, nor does this problem seem to have been treated by other authors.

3 Background

We set our study in the context of attributed typed graphs. A graph $G = (V, E, s, t)$ consists of a set of *nodes* $V = V(G)$, a set of *edges* $E = E(G)$, and *source* and *target* functions, $s, t : E \to V$. In a *type graph* $TG = (V_T, E_T, s^T, t^T)$, V_T and E_T are sets of node and edge types, while the functions $s^T : E_T \to V_T$ and $t^T : E_T \to V_T$ define source and target node types for each edge type. A typed graph on $TG = (V_T, E_T, s^T, t^T)$ is a graph $G = (V, E, s, t)$ equipped with a graph morphism $type : G \to TG$, composed of two functions $type_V : V \to V_T$ and $type_E : E \to E_T$, preserving the *source* s^T and the *target* t^T functions, i.e. $type_V(s(e)) = s^T(type_E(e))$ and $type_V(t(e)) = t^T(type_E(e))$. To introduce attributes, we follow [2]. For simplicity, we consider attributes only on nodes, the generalisation to edges being straightforward. Intuitively, we distinguish between *graph* and *value* nodes, called V_G and V_D, respectively. Graph edges E_G are equivalent to those for graphs, while an attribute edge in the set E_A defines the assignment of a value to an attribute of a node. Hence, in an attributed typed graph $G = (V, E, s, t)$, we have: $V = V_G \cup V_D$, with $V_G \cap V_D = \emptyset$; $E = E_G \cup E_A$, with $E_G \cap E_A = \emptyset$; $s = s_G \cup s_A$, with $s_G : E_G \to V_G$ and $s_A : E_A \to V_G$; and $t = t_G \cup t_A$, with $t_G : E_G \to V_G$ and $t_A : E_A \to V_D$. Analogously, the type graph TG has distinct sets V_T^G and V_T^D of graph and value nodes respectively, as well as E_T^G and E_T^A for graph and attribute edges. Given $t \in V_T^G$, nodes of type t are associated with a specific subset $A(t) \subset E_T^A$ of edge types, corresponding to the set of attribute names for t. Formally: $\forall t \in V_T^G[\exists! A(t) \subset E_T^A[\forall n \in V_G[type_V(n) = t \implies \{type_E(e) \mid e \in E_A \ \wedge \ s_A(e) = n\} \subset A(t)]]]$[3]. V_D is taken to be the disjoint union of the set of sorts in a *data signature* $DSIG$.

An *atomic constraint* is a morphism[4] of typed attributed graphs $ac : X \to Y$. X is called the *premise* and Y the *conclusion* of the constraint ac. A graph G *satisfies* ac, noted $G \vDash ac$, if for each match $m : X \to G$ there exists a morphism $y : Y \to G$ s.t. $y \circ ac = m$. If $ac : X \to Y$ is an atomic constraint, $\neg ac$ is an atomic constraint. We define $G \vDash \neg ac$ iff $G \nvDash ac$. In the particular case of the

[3] Some attributes may not be present for some node, indicating "don't care" situations.
[4] All morphisms considered in the paper, except typing ones, are injective.

atomic constraint $\neg i_X : X \to X$, where i_X is the identity on X, we call X a *negative atomic constraint*, and we represent it by simply showing X. We call $M(c) = \{G \mid G \vDash c\}$ the set of *models* for c.

We use graph transformations according to the Double-PushOut (DPO) approach [3]. Rules are defined by three graphs: the left- and right-hand sides (L and R), and the interface graph K, containing the elements preserved by the rule application. Two injective morphisms $l\colon K \to L$ and $r\colon K \to R$ model the embedding of K in L and R. The left of Figure 1 shows a DPO direct derivation diagram, modeling the application of a rule[5] $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ to a host graph G to produce a target graph H. First, the pushout complement D is evaluated. This is the unique graph for which morphisms $K \to D \to G$ exist s.t. square (1) is a pushout (i.e. G is the union of L and D through their common elements in K). In particular, D contains the elements of G which are not in the image of the elements in $L \setminus K$. The pushout (2) is then computed, adding to D new elements to form H, viz. the elements in $R \setminus K$. The right of Figure 1 shows that an atomic constraint can be associated with a rule in the form of an *application condition* AC, of the form $\{x_i\colon L \to X_i, \{y_{ij}\colon X_i \to Y_{ij}\}_{j\in J_i}\}_{i\in I}$, for a match $m\colon L \to G$ of the LHS of a rule. An AC is satisfied by m if, for each $n_i\colon X_i \to G$ s.t. $n_i \circ x_i = m$, there exists some $o_{ij}\colon Y_{ij} \to G$ s.t. $o_{ij} \circ y_{ij} = n_i$. A *negative application condition* (NAC) derives from a negative application constraint: for the rule to be applicable, X_i must not be present.

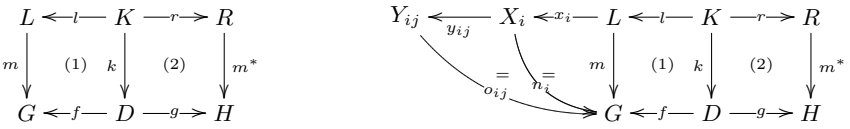


Fig. 1. DPO Direct Derivation Diagram for simple rules (left) and with AC (right).

DPO rules for typed graphs are lifted to attributed typed graphs by considering algebras on some signature including the sorts for V_D. Since morphisms can only identify values in V_D present in L and R, the modification of the value of an attribute a for a given node n from v_1 to v_2 is represented by removing an edge e_1 of type a from n to v_1 (i.e. e appears in L but not in K), and creating an edge e_2 of the same type a from n to v_2 (i.e. e_2 appears in R but not in K).

4 A scenario

We present a simple scenario to illustrate the type of problems considered, here, introducing the notation exploited in the paper, where rectangular boxes represent instances of nodes in V_G and ovals represent values in V_D, while attribute

[5] Where no ambiguity arises, we will omit explicit mention of morphisms l and r.

edges are distinguished from graph edges by the arrow end, the direction of edges in E_G being implied by the type graph in Section 5.

Consider an organisation in which the document authentication policy requires each authenticated document D to present signatures from two managers, one of level $Lv1$ and one of level $Lv2$, $Lv2$ being higher than $Lv1$ in the organisation hierarchy. Moreover, a general policy requires that documents signed off by $Lv2$ managers must have also been signed off by a $Lv1$ manager. Figure 2 illustrates the two resulting constraints.

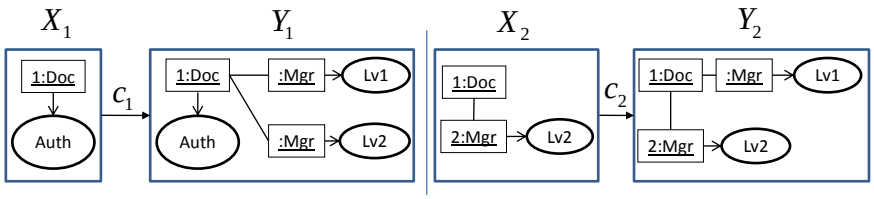


Fig. 2. The two original constraints for document authentication.

The organisation adopts a procedure for authentication which is presented in Figure 3. First an $Lv1$ manager will provide a reference for the document, and then an $Lv2$ manager will authorise it. The rules are equipped with NACs to avoid that the same manager reviews the same document twice. Note that a procedure which requires a simultaneous authorisation by both managers would also be coherent with the constraints in Figure 2.

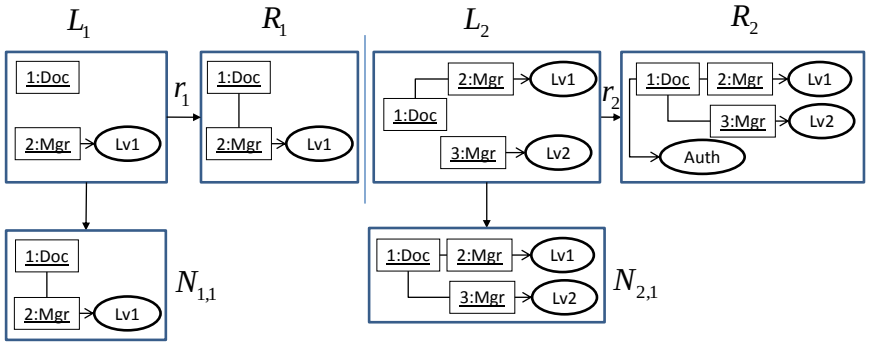


Fig. 3. The two original rules for document authentication.

Realising that the current implementation of the policy does not prevent duplication of work on one document, the organisation decides to forbid a document from being authorised twice, or referenced by two or more managers of the

same level. Such conditions could be added as NACs to the rules, but a decision is made for these to be defined as the negative constraints, shown in Figure 4 as forbidden graphs, $\neg i_{X_3} : X_3 \to X_3$, $\neg i_{X_4} : X_4 \to X_4$, and $\neg i_{X_5} : X_5 \to X_5$, respectively, becoming part of the general policy for document management.

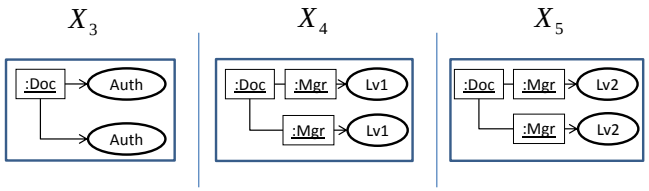


Fig. 4. Negative atomic constraints for the authentication policy.

Rules r_1 and r_2 of Figure 3 would not be coherent with the current configuration of constraints and must be amended by adding NACs to them. In particular, rule r_1 must be upgraded with the NAC $N_{1,2}$ requiring that no other manager of level $Lv1$ has already pre-approved the document, while rule r_2 is upgraded using $N_{2,2}$ to check that no authorisation already exists, besides requiring that no other $Lv2$ manager has approved the document, using $N_{2,1}$. Figure 5 shows the new versions of the two rules.

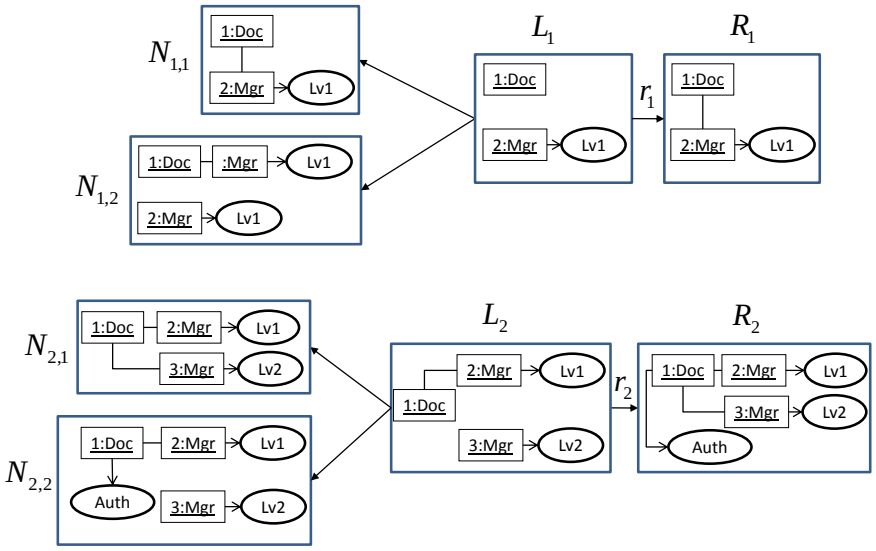


Fig. 5. The modified procedure with NACs.

In a second moment, a general revision of the authorisation policies is issued, now requiring that two $Lv1$ managers must approve a document, while maintaining the request for $Lv2$ authentication, represented by the constraint in Figure 6, which subsumes $c_1 : X_1 \to Y_1$. The constraint defined by the forbidden graph X_4 is also dropped.

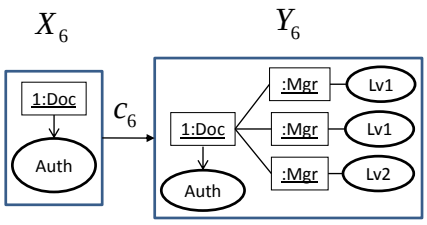


Fig. 6. Revising authorisation constraints.

This modification has impact on both rules r_1 and r_2 in Figure 5. Indeed, condition $N_{1,2}$ for r_1, which was derived from the forbidden graph X_4, now dropped, would prevent the satisfaction of constraint $c_6 : X_6 \to Y_6$. Moreover, the right-hand side of rule r_2 would not be coherent with c_6, as it might lead to approval of documents without the signature of two $Lv1$ managers. While the solution to the first problem is simple (remove $N_{1,2}$), the second problem can be addressed in different modes: by requiring concurrent approval by the additional $Lv1$ manager and the $Lv2$ one, by sequentialising the approvals of the manager according to the level (i.e. simultaneous approval of both $Lv1$ managers and subsequent $Lv2$ approval) or by considering a sequential process, where the second $Lv1$ and the $Lv2$ approval can occur in any order. Note that since $c_2 : X_2 \to Y_2$ still holds, it remains impossible to have a procedure which leads to $Lv2$ approval without prior or concurrent $Lv1$ approval.

5 Constraint-compliant policy rules

We define policies on type graphs which are instances of the metamodel MM in Figure 7 (left). Here, `Document`, `Personnel` and `Resource` are meta-types for graph node types, defining the structural elements involved in a policy, while `State` and `Level` are metatypes for attribute types. We call *structural* nodes those nodes whose type is an instance of `Document`, `Personnel` or `Resource`.

Let the *structural part* of a constraint $c : X \to Y$ be the constraint resulting by projecting X and Y onto the subgraphs formed only with structural nodes.

Figure 7 (right) presents the type graph exploited in the paper, where type *Doc* is an instance of `Document`, *Mgr*, representing managers, is an instance of `Personnel`, and *Printer* is a `Resource` type. The attribute types denote the authorisation state for a document and different types of levels associated with structural elements.

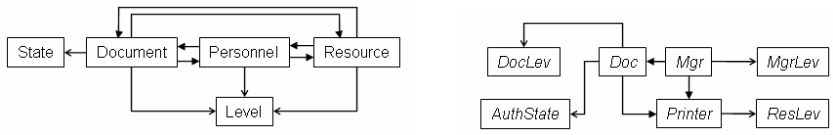


Fig. 7. Metamodel for security policies (left) and type graph for the scenario (right).

Policies employ a class of constraints where Y is a connected graph, presenting additional edges with respect to X only if their sources or targets are in $Y \setminus X$ and possibly adding edges only to a single element in X. The resulting class of constraints is still significant for practical cases, and ensures local control on constraint enforcing. Procedures are considered to be formed only with rules which are increasing, with $L = K$, (or decreasing, with $K = R$) in the structural part, and which may use only equality or inequality in conditions on attributes.

We introduce here the notion of incremental procedure realising a policy, deriving rules from positive constraints according to the following definitions.

Given a positive constraint $c : X \to Y$, we call *completion* of c the graph difference between the premise X and the conclusion Y. This is evaluated as the pushout complement Q' in Figure 8 along the minimal graph X' for which the pushout complement exists. The graph X' is computed as the discrete graph formed by the structural nodes in X which are attached to edges in $E(Y) \setminus E(X)$.

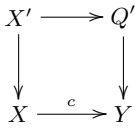


Fig. 8. Construction of the completion for a constraint.

We build an *incremental procedure* $P(c)$ *realising* c as follows:

Let $\mathcal{Q} = \{Q_0, \ldots, Q_k\}$ and $\mathcal{H} = \{h_i^j : Q_i \to Q_j\}$ be the sets of all graphs and associated morphisms s.t.:

- $Q_0 = X'$ and $Q_k = Q'$;
- for each $i = 1, \ldots, k$, there exist injective morphisms[6] $X' \overset{h_0^i}{\to} Q_i \overset{h_i^k}{\to} Q'$;
- for each $i, j = 1, \ldots, k$, $h_0^j = h_i^j \circ h_0^i$ and $h_i^k = h_j^k \circ h_i^j$;
- Q' is the colimit of the diagram obtained with $\mathcal{Q}$ and $\mathcal{H}$;
- attribute edges are added only together with their source structural nodes[7].

[6] In order to emphasise $X' = Q_0$ and $Q' = Q_k$ we will abuse notation in the examples, writing $h_i^{Q'}$ for $h_i^k : Q_i \to Q'$ and $h_{X'}^j$ for $h_0^j : X' \to Q_j$.

[7] As value nodes are assumed to be always existing, we leave them understood when not associated with structural elements.

$\mathcal{Q}$ and $\mathcal{H}$ are uniquely determined by X' and Q'. Then $P(c)$ is obtained by first producing the set of rules $P_Q(c) = \{p : L = K \to R\}$, with $L = Q_i$ and $R = Q_j$, for each $h_i^j : Q_i \to Q_j \in \mathcal{H}$, s.t.

1. given three graphs $Q_i, Q_j, Q_l \in \mathcal{Q}$ with morphisms h_i^j, h_i^l, h_j^l s.t. $h_i^l = h_j^l \circ h_i^j$, we use only h_i^j and h_j^l to form the rules in $P_Q(c)$;
2. either $Q_j \neq Q'$ or $V_G(Q_j) = V_G(Q')$, i.e. the structural part of the right hand-side is equal to the structural part of the completion Q' only in rules which add some structural edge.

Three final steps are needed at this point.

1. Check the generated rules against the other constraints, and filter out those which would certainly violate them. Namely, given a rule $p : L \to R \in P_Q(c)$ and a constraint $c : X \to Y$, we remove p from $P_Q(c)$ if there exists a morphism $R \to X$ but no morphism $R \to Y$. Moreover, the premise X is added as a NAC for all the other rules, i.e. with $R \neq Q'$, and the resulting rules included in $P(c)$.
2. Build the set IQY of all the intermediate graphs between Q' and Y, which do not contain an instance of X. In particular, since X', and as a consequence Q', already contained all nodes in $V_G(Y)$, these intermediate graphs can only differ from Q' by some additional edges, either structural or for those attributes which were in X (and are not in X' and therefore are not in Q' either). If some attribute edge is added in some of the graphs in IQY, we propagate these assignments backwards through all the graphs in $\mathcal{Q}$, in such a way that Q' is still the colimit of the diagram. We then correspondingly update the L and R for all attributes derived from these graphs.
3. If IQY is empty, i.e. Q' and Y differ by only one edge, we replace each rule in $P(c)$ of the form $p : L = K \to Q'$, by the set of rules $F(c) = \{p_F : L' \leftarrow K \to Y\}$, where each L' is obtained from L by adding one of the possible values for the attributes assigned in rule p (i.e. appearing in R but not in K), compatible with the other constraints. We call rules with $R = Y$ *final rules* and add NACs to prevent the formation of other instances of X.

Theorem 1 states the consistency with a constraint c of the resulting set $P(c)$.

Theorem 1. *Given a constraint $c : X \to Y$, a graph G satisfying c, and a rule $p : L \leftarrow K \to R \in P(c)$, the graph H obtained by applying p to G satisfies c.*

Proof. **[Sketch.]** G can satisfy c in two ways: either it contains X and Y, or it does not contain X. If it contains X, any rule from $P(c)$ which is not a final rule in $F(c)$ will not disrupt Y, as rules are only increasing. If G does not contain X, the graph H will only contain X if a final rule has been applied, as these are the only ones which make complete X. But in this case, the rule will also have provided a context Y for this occurrence of X. Note that different matches for X cannot be generated by other rules because of the added NACs.

Example Figure 9 shows the construction of the completion graph for the constraint $c_1 : X_1 \rightarrow Y_1$ from Figure 2, while Figure 10 presents the construction of the sets $\mathcal{Q}$ and $\mathcal{H}$ for this case. We have indicated only direct morphisms, and omitted those which can be derived by transitivity.

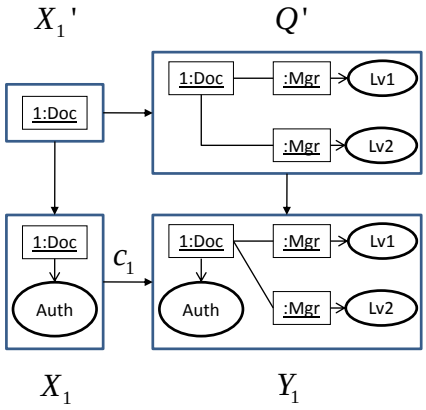


Fig. 9. The construction of the completion for $c_1 : X_1 \rightarrow Y_1$ in Figure 2.

The top of Figure 11 shows how the last rule derived from morphism $h_6^{Q'}$ would be modified. As the whole difference between X and Y is present, we can complete the right-hand side to be the full Y, i.e. presenting also an instance of X. Underneath this rule, we show one of its possible final versions, assuming that *preAuth* is a value in the domain of document states. Such a rule would be generated only if such a state is compatible with the presence of the association of the document with a manager of Level 1, i.e. if the *preAuth* state is not associated with some constraint forbidding the presence of the association with such a manager. Also note that, in the policy in our original scenario, the constraint $c_2 : X_2 \rightarrow Y_2$ from Figure 2 will filter out the rules derived from h_2^4 and $h_5^{Q'}$.

6 Incremental update

We analyse now the problem of incrementally updating the rules in $P(c)$ following a policy revision where some positive constraint $c : X \rightarrow Y$ is replaced by a positive constraint $c^u : X^u \rightarrow Y^u$. In particular, we consider the following possible modifications, where the inequalities indicate the existence of an injection from the smallest to the biggest graph. Moreover, the modifications are such that they preserve the image of the original premise in the original conclusion, for the common parts of X and X^u, and of Y and Y^u.

1. **(conclusion expansion)** $X = X^u$, $Y < Y^u$

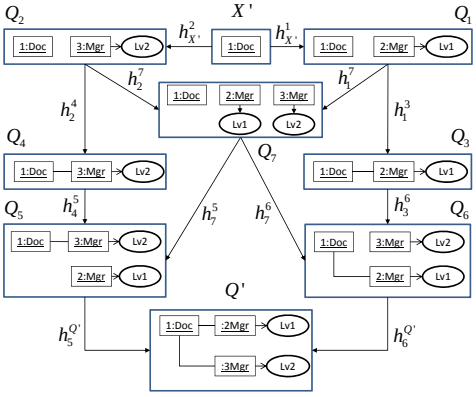


Fig. 10. The sets $\mathcal{Q}$ and $\mathcal{H}$ for the incremental procedure for $c_1 : X_1 \rightarrow Y_1$.

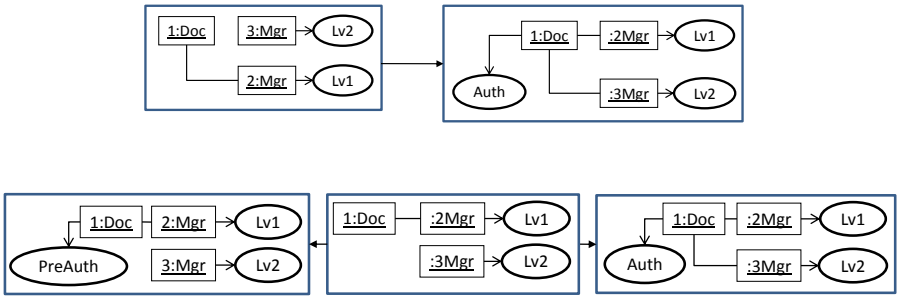


Fig. 11. Ensuring constraint c_1 (top) and one version updating the state (bottom).

2. **(conclusion reduction)** $X = X^u$, $Y > Y^u$
3. **(premise expansion)** $X < X^u$, $Y = Y^u$
4. **(premise reduction)** $X > X^u$, $Y = Y^u$
5. **(semantic reduction)** $X > X^u$, $Y < Y^u$
6. **(semantic expansion)** $X < X^u$, $Y > Y^u$
7. **(common expansion)** $X < X^u$, $Y < Y^u$
8. **(common reduction)** $X > X^u$, $Y > Y^u$

Note that we do not admit modifications which simultaneously add some elements and delete others from the premise or the conclusion of a constraint. Hence, if a premise is expanded, the conclusion can be reduced only by removing parts neither in the original nor the expanded premise.

We first consider the simple cases of premise expansion and conclusion reduction. From Figure 12 (left), modeling premise expansion, we observe that if a graph G satisfies c non trivially, i.e. G has a subgraph which is a match for

Y, it will also satisfy c^u under the same match. However, the rules in $P(c)$ can produce, in their right-hand sides, instances of X^u prior to creating the whole Y. For such rules, we need to expand their left-hand sides with the difference between X^u and Y. Moreover, we relax the NACs preventing the formation of X prior to the final rules of $P(c)$. More precisely, we extend each existing NAC by the difference between X^u and X, to make it prevent the formation of X^u.

For the case of conclusion reduction, centre of Figure 12, we observe that we can only remove elements which are not in X. In this case, the execution of the algorithm in Section 5 would produce a completion $Q^{u\prime}$ included in the original completion Q'. Therefore, we update the set $P(c)$ by removing all the rules presenting, either in the left-hand side or in the right-hand side, graphs greater than the difference between Y^u and Q'. Moreover, each rule presenting exactly the difference between Y^u and X in the right-hand side (such rules have to exist in $P(c)$), is now modified so as to also produce the missing part of X. As a consequence, we have Proposition 1.

Proposition 1. *The set $U(P(c))$ obtained by updating rules in $P(c)$ on premise expansion or conclusion reduction of c is equal to $P(c^u)$, the incremental procedure for c^u.*

The combination of premise expansion and conclusion reduction defines what is called semantic expansion, as $M(c) \subset M(c^u)$. Indeed, considering the diagram on the right of Figure 12, we observe that each graph G satisfying c^u also satisfies c. On the other hand, all graphs violating c because they present a match $m : X \to G$ for X which is not extendable to a match for Y are included in $M(c^u)$, if m cannot be extended to a match $m' : X^u \to G$ either.

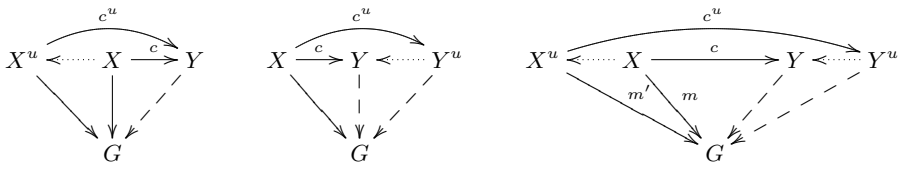


Fig. 12. Premise expansion (left), conclusion reduction (center) and semantic expansion (right).

The cases for premise reduction and conclusion expansion are discussed with reference to Figure 13. In the first case (on the left), some G in $M(c)$ can be not in $M(c^u)$, if G provides a match for X^u but not for X and Y. The difference between X^u and X can consist of the absence of some edges or of some nodes (possibly together with the associated edges). If only edges are removed from X, we consider that the evaluation of the completion $Q^{u\prime}$ would use the same graph X' as for the original constraint, but that these edges would now appear in Q'. On the other hand, there would be graphs Q_i in $\mathcal{Q}$ s.t. $X^u \leq Q_i$. Rules of the form $p : L \to R = Q_i$ in $P(c)$ must then be replaced with versions in which some

edge in $X \setminus X^u$ does not appear, while all the rules for which $R \cap X^u \neq \emptyset$ and $X^u \not< R$ must be completed with a NAC of the form $n : L \to X^u$. In the case that nodes are removed in X^u, they would however appear in $Q^{u\prime}$. Rules in $P(c)$ would therefore be modified creating, for each rule in which a node in $X \setminus X^u$ appears, a version without that node. As before, NACs of the form $n : L \to X^u$ must be added to the other rules.

For the case of conclusion expansion, $P(c)$ does not contain rules presenting graphs intermediate between Y and Y^u, which therefore have to be added. Moreover, we need to modify the final rules which produced Y, as they were the only ones having X as a subgraph for R. Let $\mathcal{L}_Y$ be the set of left-hand sides for final rules. We replace each rule of the form $p : L \to R$, $L \in \mathcal{L}_Y$, with the set of rules producing graphs intermediate between L and Y^u, but not including X.

Again, semantic reduction (right of Figure 13) derives from the combination of premise reduction and conclusion expansion and the relative modifications can be derived from the composition of the two processes.

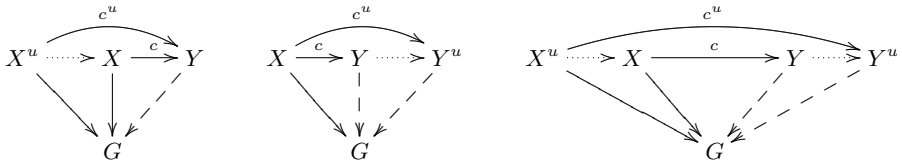


Fig. 13. Premise reduction (left), conclusion expansion (center) and semantic reduction (right).

In a similar way, common expansion and common reduction can be obtained by combining the above processes, in the following order:

- For common expansion, expand Y to Y^u first, and then expand X to X^u, in both cases expanding the morphism so as to preserve the image of X in Y.
- For common reduction, reduce X to X^u first, and then reduce Y to Y^u, in both cases reducing the morphism so as to preserve the image of X in Y.

One wonders whether a similar construction can be defined for the case of simultaneously removing and including elements in both premise and conclusion. Figure 14 illustrates the problem with this. In order to maintain incrementality, one would need to first identify the intersections X' and Y' between the original and the updated versions, update under common reduction for $X' \to Y'$, and then use common expansion to generate $P(c^u)$ for the new constraint $c^u : X^u \to Y^u$ (left). But in this case there is no guarantee that Y' is connected, as required in our model. A similar problem occurs if we first use common expansions and then common reduction, where we need the intersections X'' and Y'' (right).

Example Considering the scenario in Section 4, Figure 6 presents a case of conclusion expansion, making stricter requests on the authorisation process. Following the algorithm above, the construction of the new rules for this policy start

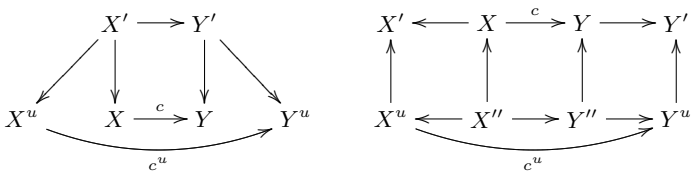


Fig. 14. Incorrect constructions for simultaneous removal and increment.

from the identification of the rules with $L \in \mathcal{L}_{Y_1}$; in this case rule $r_2 : L_2 \to R_2$ in Figure 5 (for the sake of the discussion, we omit the consideration of modifications of NACs). We then need to replace r_2 with the set of rules constructing the intermediate graphs between L_2 and the modified conclusion Y_6, with the proviso that X can appear only in the right-hand side of the rule building the whole Y_6. Figure 15 shows the result of replacing rule r_2 with rules r_2' and r_2^2.

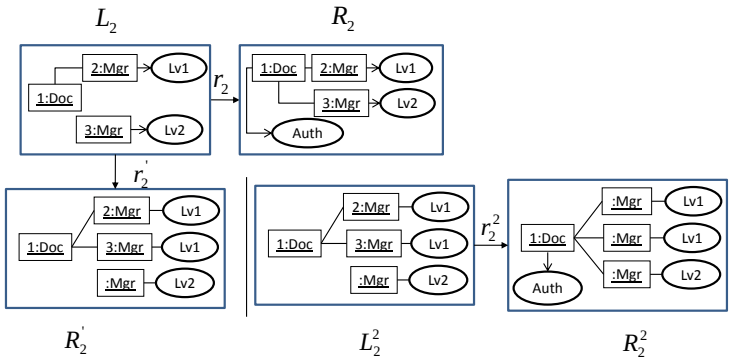


Fig. 15. Revising rule r_2 from the scenario.

7 Conclusions and future work

We have presented an approach to the construction of incremental procedures ensuring the application of rules consistently with atomic constraints and to the modification of such procedures in an incremental way after the modification of the original constraints. The approach works under a number of assumptions, which are reasonably met in practical cases. First, conclusions in constraints are defined by connected graphs, and with a specific relation with X. Second, constraint updates preserve the images for the preserved elements in the premise (i.e. we do not deal with revocation [6]). Third, constraints and rules are typed according to a metamodel maintaining distinct finite domains for each attribute

The procedures are defined as sets of rules, which can be exploited in different contexts, or organised in transformation units.

Several lines of research can be pursued from here. For example, modifications have been considered only for rules in an incremental procedure. However, a rule $p : L \to R$ can be consistent with a constraint $c : X \to Y$ even if $R > Y$, or $R < X'$, in which case $p \notin P(c)$. The general case of modifying such rules must then be studied. Also, we prevent rules from creating matches for X if they are not final rules. However, one could study situations in which to allow the creation of further matches than the one for which the extension to Y is created and define repair transformation units for these situations. We are also interested in more general cases relaxing conditions on Y. If we allow Y to be disconnected, then the construction in Figure 14 could be exploited for simultaneous reduction and expansion in premise and conclusion. Finally, one needs to consider what happens when arbitrary morphisms are considered instead of only injective ones. This problem can be investigated in versions where arbitrary morphisms can be used for matches of constraints, of rules, or both.

References

1. P. Bottoni, A. Fish, and F. Parisi Presicce. Preserving constraints in horizontal model transformations. In *Proc. GT-VMT 2010*, volume 29 of *ECEASST*, 2010.
2. H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundam. Inform.*, 74(1):135–166, 2006.
3. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.
4. K. Ehrig, J. M. Küster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In *Proc. FMOODS 2006*, volume 4037 of *LNCS*, pages 156–170. Springer, 2006.
5. A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.
6. Å. Hagström, S. Jajodia, F. Parisi-Presicce, and D. Wijesekera. Revocations-a classification. In *CSFW*, pages 44–58, 2001.
7. M. Janota, V. Kuzina, and A. Wasowski. Model construction with external constraints: An interactive journey from semantics to syntax. In *Proc. MoDELS 2008*, pages 431–445, 2008.
8. M. Koch, L. V. Mancini, and F. Parisi-Presicce. Graph-based specification of access control policies. *J. Comput. Syst. Sci.*, 71(1):1–33, 2005.
9. M. Koch and F. Parisi-Presicce. Uml specification of access control policies and their formal verification. *Software and System Modeling*, 5(4):429–447, 2006.
10. F. Orejas. Symbolic attributed graphs. In *Proc. GraMoT 2010*, to appear.
11. A. Rensink. Representing first-order logic using graphs. In *Proc. ICGT*, volume 3256 of *LNCS*, pages 319–335. Springer, 2004.
12. Y. Wei, C. Wang, and W. Peng. Graph transformations for the specification of access control in workflow. In *Proc. WiCOM '08*, pages 1–5, 2008.

Minimizing Finite Automata with Graph Programs*

Detlef Plump[1], Robin Suri[2], and Ambuj Singh[3]

[1] The University of York, UK
[2] Indian Institute of Technology Roorkee, India
[3] Indian Institute of Technology Kanpur, India

Abstract. GP (for Graph Programs) is a rule-based, nondeterministic programming language for solving graph problems at a high level of abstraction, freeing programmers from dealing with low-level data structures. In this case study, we present a graph program which minimizes finite automata. The program represents an automaton by its transition diagram, computes the state equivalence relation, and merges equivalent states such that the resulting automaton is minimal and equivalent to the input automaton. We illustrate how the program works by a running example and argue that it correctly implements the minimization algorithm of Hopcroft, Motwani and Ullman.

Key words: Graph programs, automata minimization, rule-based programming, correctness proofs

1 Introduction

GP is an experimental nondeterministic programming language for high-level problem solving in the domain of graphs. The language is based on conditional rule schemata for graph transformation, freeing programmers from implementing and handling low-level data structures for graphs. The prototype implementation of GP compiles graph programs into bytecode for an abstract machine, and comes with a graphical editor for programs and graphs. We refer to [7] for an overview of the language and to [6] for a description of the current implementation.

In this paper, we present a case study about solving a problem with GP that as first sight may not appear to be a graph problem: the minimization of finite automata. It is natural though to represent finite automata by their transition diagrams and to view the minimization process as a sequence of transformation steps on these diagrams. Programmers can visually construct corresponding rule schemata and control the application of these schemata by GP's commands.

We implement the minimization algorithm of Hopcroft, Motwani and Ullman [4] (see also [9]). This algorithm first computes the indistinguishability relation among states, called *state equivalence*, and then merges equivalent states to

* Work of the second and third author was done while visiting the University of York, funded by the Department of Computer Science.

obtain a minimal automaton that is equivalent to the input automaton. Two states are equivalent if processing strings from either state will have the same result with respect to acceptance. While state equivalence is usually computed by a table-filling algorithm, in our case we directly connect equivalent states with special edges. Once the equivalent states have been determined, we merge them by redirecting edges and removing isolated nodes.

In Section 5, we argue that our implementation is correct in that the graph program will transform every input automaton into an equivalent and minimal output automaton. This involves showing that the program terminates, that it correctly computes the state equivalence relation, and that the merging phase produces an automaton in which each equivalence class of states is represented by a unique state.

2 Graph Programs

We briefly review GP's conditional rule schemata and control constructs. Technical details (including the abstract syntax and operational semantics of GP) can be found in [7], as well as a number of example programs.

Conditional rule schemata are the "building blocks" of graph programs: a program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling the application of the schemata. Rule schemata generalise graph transformation rules in the double-pushout approach with relabelling [2], in that labels can contain expressions over parameters of type integer or string. Figure 1 shows a conditional rule schema consisting of the identifier `bridge` followed by the declaration of formal parameters, the left and right graphs of the schema, the node identifiers `1`, `2`, `3` specifying which nodes are preserved, and the keyword `where` followed by the condition `not edge(1,3)`.

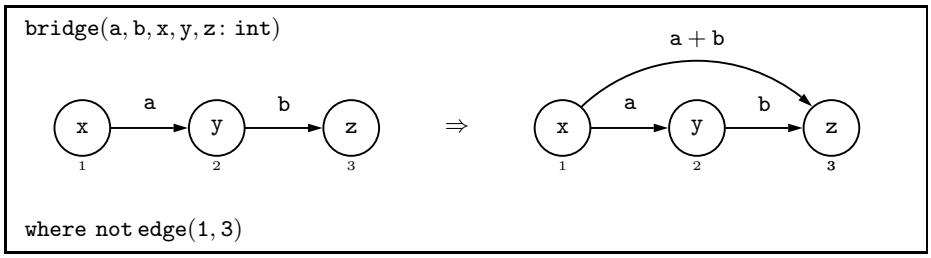


Fig. 1: A conditional rule schema

In the GP programming system [6], rule schemata are constructed with a graphical editor. Labels in the left graph comprise only variables and constants because their values at execution time are determined by graph matching. The condition of a rule schema is a Boolean expression built from arithmetic expressions and the special predicate `edge`, where all variables occurring in the condition must also occur in the left graph. The predicate `edge` demands the

(non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1,3)` in the condition of Figure 1 forbids an edge from node 1 to node 3 when the left graph is matched.

Conditional rule schemata represent possibly infinite sets of conditional graph transformation rules, and are applied according to the double-pushout approach with relabelling. A rule schema $L \Rightarrow R$ with condition Γ represents conditional rules $\langle\langle L^\alpha \leftarrow K \rightarrow R^\alpha\rangle, \Gamma^{\alpha,g}\rangle$, where K consists of the preserved nodes (which are unlabelled) and $\Gamma^{\alpha,g}$ is a predicate on graph morphisms $g\colon L^\alpha \rightarrow G$ (see [7]).

GP's commands for controlling rule-schema applications include the non-deterministic one-step application of a rule schema, the non-deterministic one-step application of a set $\{r_1, \ldots, r_n\}$ of rule schemata, the sequential composition $P; Q$ of programs P and Q, the as-long-as-possible iteration $P!$ of a program P, and the branching statement `if` C `then` P `else` Q for programs C, P and Q. The first four of these commands have the expected effects. The branching command first checks if executing C on the current graph G can produce a graph; if this is the case, then P is executed on G, otherwise Q is executed on G.

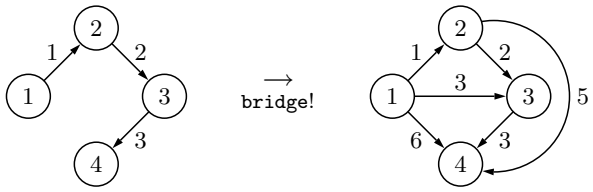


Fig. 2: An execution of the program `bridge!`

For example, Figure 2 shows an execution of the program `bridge!`. This program makes an input graph transitive in that for every directed path of the input, the output graph contains an edge from the first node to the last node of the path. Note that the edge with label 6 can be produced by applying `bridge` in two different ways, performing either the addition $3 + 3$ or $1 + 5$. In general, a program may produce many different output graphs for the same input. The semantics of GP assigns to every input graph the set of all possible output graphs (see [7, 8]).

3 Automata Minimization

Our starting point is the abstract minimization algorithm of Hopcroft, Motwani and Ullman [4] (see also [9]). To fix notation, we consider a deterministic finite automaton (DFA) as a system $A = (Q, \Sigma, \delta, q_0, F)$ where Q is the finite set of states, Σ is the input alphabet, $\delta\colon Q \times \Sigma \rightarrow Q$ is the transition function, q_0 is the initial state, and F is the set of final (or accepting) states. The extension of δ to strings is denoted by $\delta^*\colon Q \times \Sigma^* \rightarrow Q$.

Definition 1 States p and q of an automaton are *indistinguishable* if for all strings $w \in \Sigma^*$, $\delta^*(p, w) \in F$ if and only if $\delta^*(q, w) \in F$.

We say that p and q are *distinguishable* if they are not indistinguishable, that is, there must be some string $w \in \Sigma^*$ such that either $\delta(p, w) \in F$ and $\delta(q, w) \notin F$, or vice-versa. Clearly, indistinguishability of states is an equivalence relation. Henceforth we refer to this relation simply as *state equivalence*.

The following minimization algorithm first marks all unordered pairs of distinguishable states of an automaton A—thus representing state equivalence implicitly by all unmarked pairs of states. In a second phase, equivalent states are merged to form the states of the minimal automaton $\hat{A}$.

Algorithm 1

Marking phase

Stage 1:
for each $p \in F$ and $q \in Q - F$ **do** mark the pair $\{p, q\}$

Stage 2:
repeat
  for each non-marked pair $\{p, q\}$ **do**
    for each $a \in \Sigma$ **do**
      if $\{\delta(p, a), \delta(q, a)\}$ is marked **then** mark $\{p, q\}$
until no new pair is marked

{For each state p, the equivalence class of p consists of all states q for which the pair $\{p, q\}$ is not marked.}

Merging phase

Construct $\hat{A} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q_0}, \hat{F})$ as follows:
- $\hat{Q}$ consists of the state equivalence classes.
- $\hat{q_0}$ is the equivalence class containing q_0.
- For each $X \in \hat{Q}$ and $a \in \Sigma$, pick any $p \in X$ and set $\hat{\delta}(X, a) = Y$, where Y is the equivalence class containing $\delta(p, a)$.
- $\hat{F}$ consists of the equivalence classes containing states from F.

□

By the following lemma, the marking phase of Algorithm 1 correctly computes the state equivalence.

Lemma 1 ([4, 9]) *A pair of states is not marked by the marking phase of Algorithm 1 if and only if the states are equivalent.*

Using Lemma 1, the correctness of Algorithm 1 can be established.

Theorem 1 ([4]). *The automaton $\hat{A}$ produced by Algorithm 1 accepts the same language as A and is minimal.*

In the next section, we present an implementation of Algorithm 1 in GP. The correctness of the implementation is proved in Section 5.

4 Implementation in GP

We represent automata by their transition diagrams, that is, graphs in which nodes represent states and edges represent transitions. In the following, the terms 'node' and 'state', respectively 'edge' and 'transition' will often be used synonymously. We make the following assumptions about an input automaton:

1. The states have labels of the form x_i, where x is some integer and $i \in \{0, 1\}$. The component i is called a *tag*[4], we require that final states have tag 1 and that non-final states have tag 0. The integer x is arbitrary, except that the initial state, and only this state, has a label of the form 1_i.
2. The transitions are labelled with strings which represent the symbols in Σ.
3. To keep the presentation simple, we assume that all states are reachable from the initial state. (It is straightforward to write a graph program that removes all unreachable states.)

The graph program implementing Algorithm 1 is shown in Figure 3, where `mark`, `merge` and `clean_up` are *macros*. The rule schemata contained in the macros are discussed below.

```
main = mark; merge; clean_up

mark = distinguish!; propagate!; equate!
merge = init; add_tag!; (choose; add_tag!)!; disconnect!; redirect!
clean_up = remove_edge!; remove_node!; untag!
```

Fig. 3: GP program for automata minimization

We will explain each stage of the program in Figure 3, using as running example the minimization of the automaton in Figure 4. This automaton accepts all strings over $\{\mathtt{a}, \mathtt{b}\}$ that end in two `b`'s.

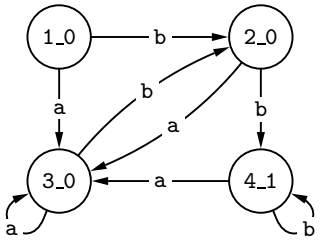


Fig. 4: Sample automaton with alphabet $\{\mathtt{a}, \mathtt{b}\}$

[4] In general, a label in GP has the form $x_1\_x_2\_\ldots\_x_n$ where each x_i is either an integer or a character string.

4.1 Marking Phase

We first need to determine which states are equivalent. For this, we implement the marking phase of Algorithm 1 in the macro `mark`. The macro's rule schemata are shown in Figure 5.

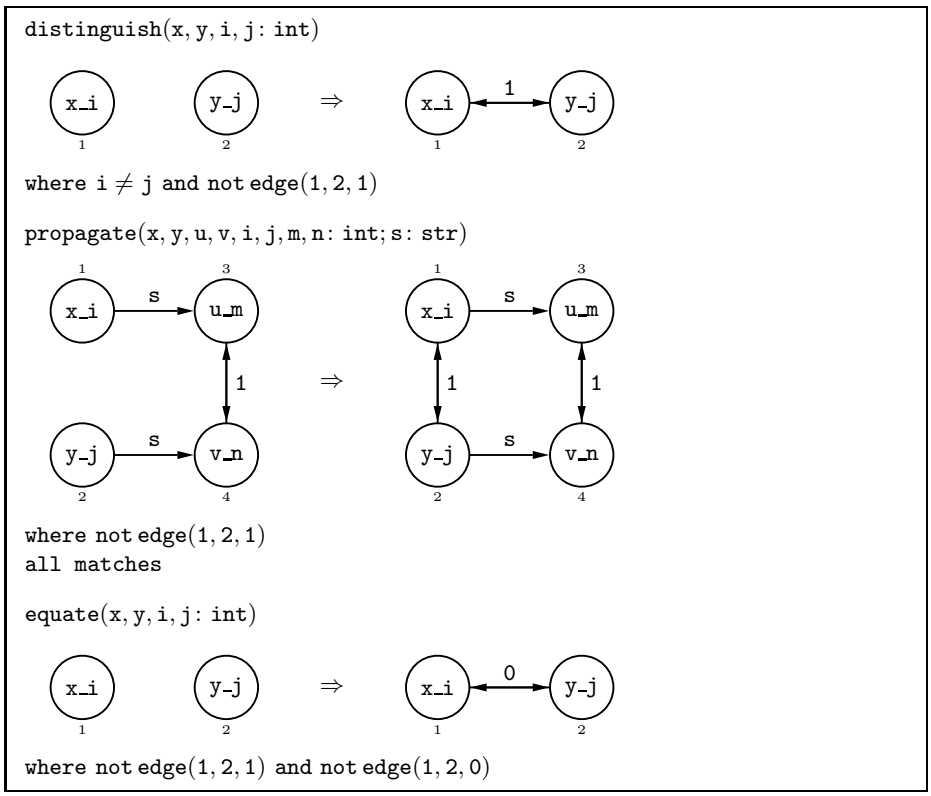


Fig. 5: Rule schemata of the macro `mark`

The subprogram `distinguish!` implements Stage 1 of Algorithm 1. Given two states such that one is a final state and the other is not, by assumption, the states carry tags `1` and `0` respectively. In this case we mark the states as distinguishable by connecting them with two `1`-labelled edges of opposite direction (drawn as a single edge with two arrowheads). The condition `not edge(1, 2, 1)` in `distinguish` forbids a `1`-labelled edge between nodes `1` and `2` to make sure that `distinguish!` terminates. The ternary `edge` predicate refines the binary predicate discussed in Section 2 in that it allows to specify the label of the forbidden edge.[5] See Figure 6 for the effect of `distinguish!` on the sample automaton, where we typeset new labels in italics.

[5] This predicate is not yet implemented in GP but will be included in the next release.

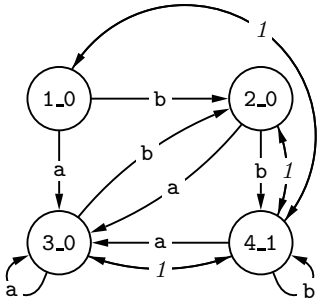


Fig. 6: Sample automaton after `distinguish!`

Next, the rule schema `propagate` looks for pairs of states that have not yet been discovered as distinguishable (and so are not linked by a `1`-edge). The states must have outgoing transitions with the same symbol, leading to states that have already been discovered as distinguishable. Again, a newly discovered pair of distinguishable states is marked by `1`-labelled edges with opposite directions. The subprogram `propagate!` thus implements the repeat-loop of Algorithm 1.

Rule schema `propagate` has the '`all matches`' attribute, meaning that nodes of the schema can be merged before the schema is applied. An alternative view is that `propagate` can be applied using non-injective graph morphisms. (See [1] for details and the equivalence of both views.) For the benefit of the reader, Figure 7 lists the standard rule schemata represented by `propagate` that are possibly applicable to an automaton. Other schemata obtained by node merging can be ruled out because our automata do not contain `1`-labelled loops and do not have states with multiple outgoing transitions labelled with the same symbol.

Lemma 1 guarantees that after termination of `propagate!`, all pairs of distinguishable states have been discovered. Thus we can mark the remaining pairs as equivalent, linking their states with `0`-labelled edges in the subprogram `equate!`. The effect of `propagate!` and `equate!` on the sample automaton is shown in Figure 8a and Figure 8b.

4.2 Merging Phase

After termination of the macro `mark`, the states of the input automaton are partitoned into equivalence classes: these are the subsets of states that are pairwise linked by `0`-labelled edges. Next we have to merge all the states in each partition into one state representing the partition. We need to ensure that all transitions to states that are not representing partitions are redirected to the unique states representing the partitions. Transitions outgoing from non-representative states can be removed, as can these states themselves. The merging process is implemented by the macro `merge`, whose rule schemata are shown in Figure 9.

We first consider the partition containing the initial state. The rule schema `init` marks this state as the unique representative of its partition by adding an

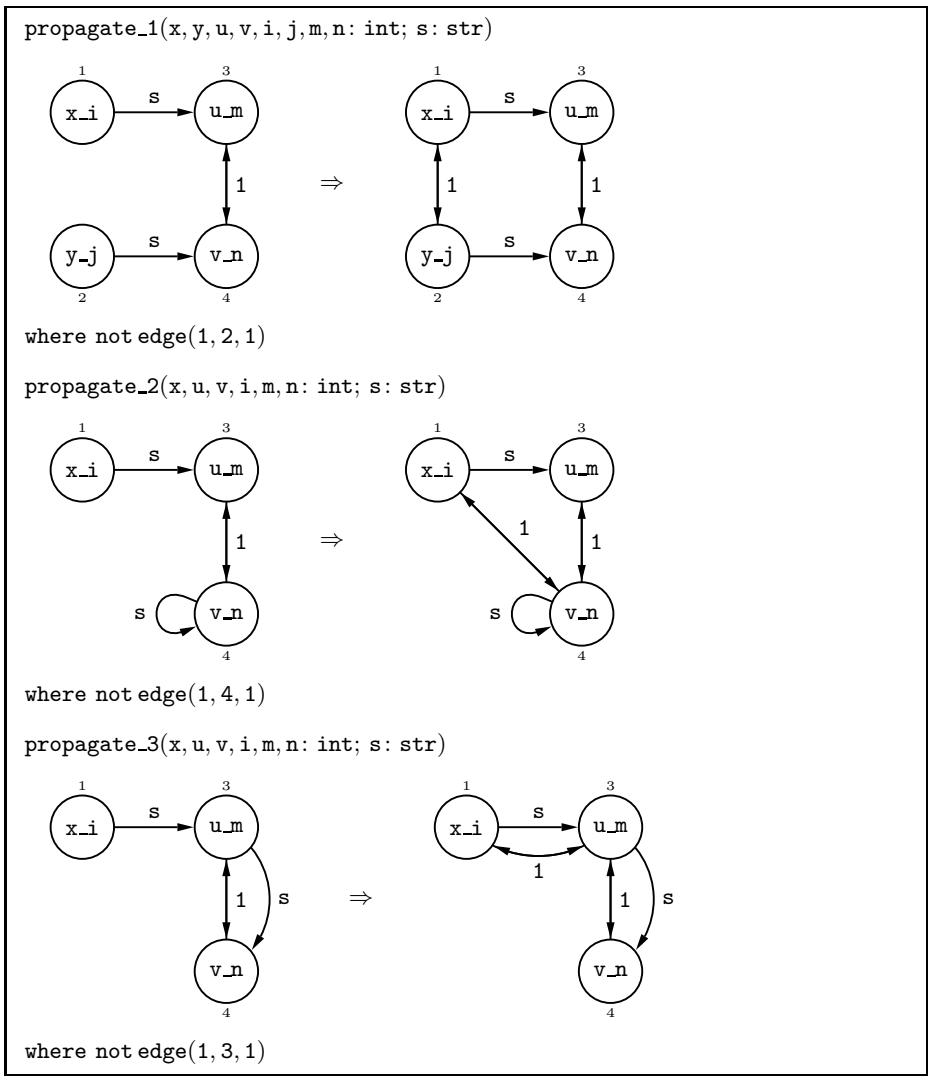
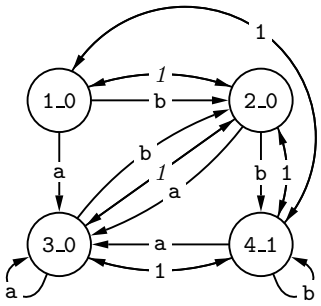


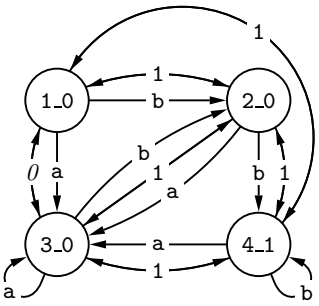
Fig. 7: Rule schemata represented by `propagate` using '`all matches`'

extra 0-tag to the state's label. Then the loop `add_tag!` marks all other states in the initial partition with an extra 1-tag. This marking procedure is repeated for all other partitions, by the nested loop (`choose`; `add_tag`!)!. In each iteration of the outer loop, some unmarked state is chosen as the unique representative of its partition and subsequently all other states in the partition are marked as non-representative states.

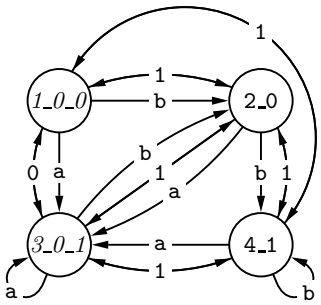
After all states have been marked as representatives or non-representatives, the rule schemata `disconnect` and `redirect` take care of the transitions leav-



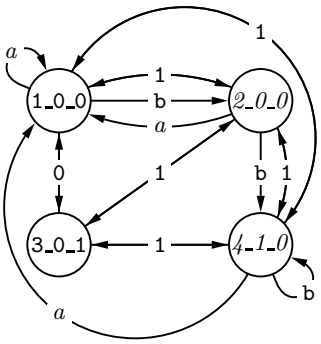
(a) After `propagate!`



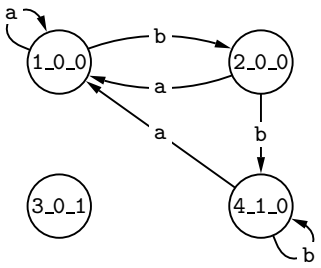
(b) After `equate!`



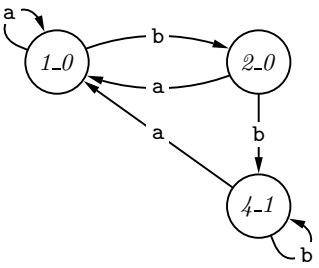
(c) After `init`; `add_tag!`



(d) After `redirect!`



(e) After `remove_edge!`



(f) After `untag!`

Fig. 8: Snapshots of the sample automaton

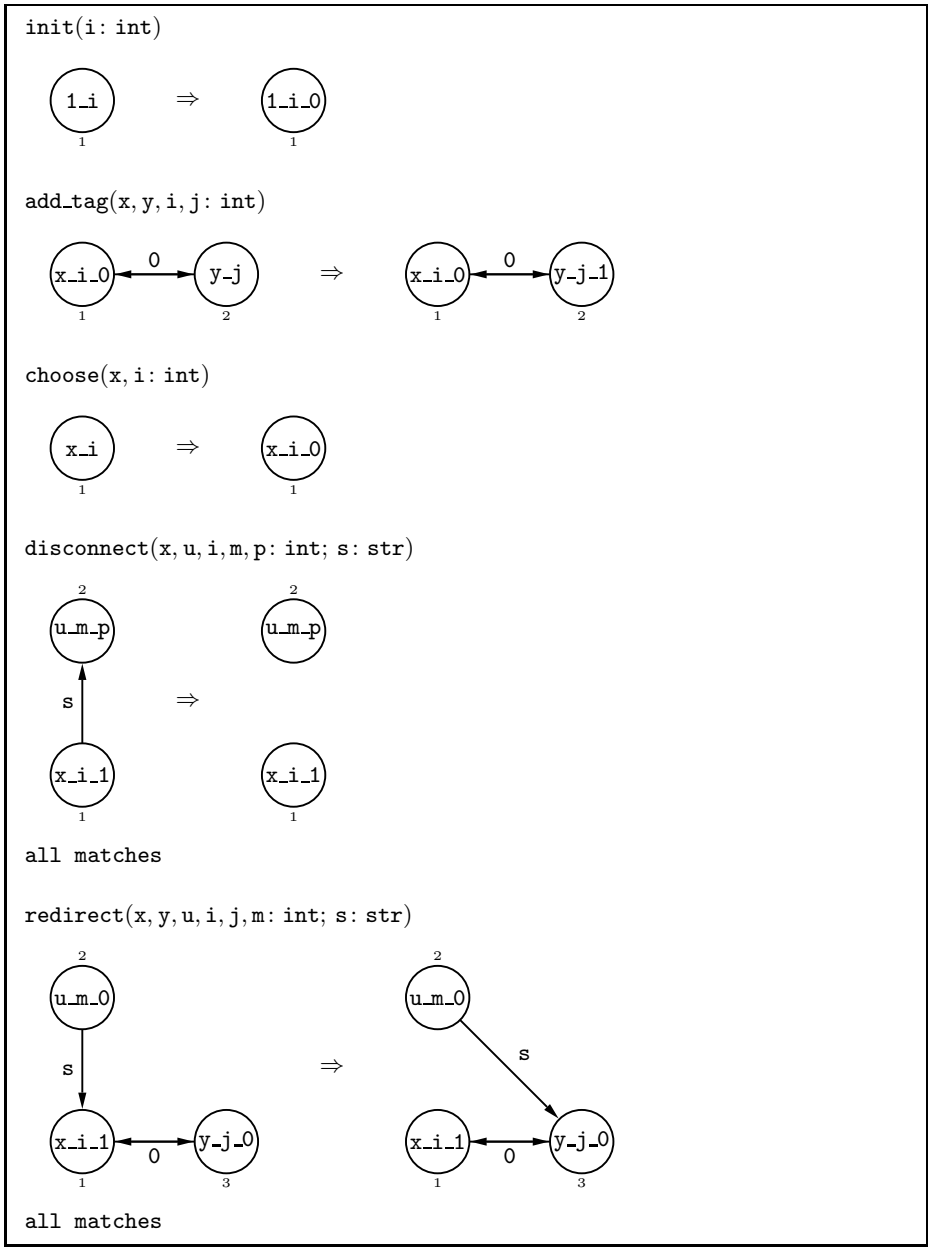


Fig. 9: Rule schemata of the macro `merge`

ing and reaching non-representative states. The loop `disconnect!` removes all outgoing transitions (including loops), as these are no longer needed, while `redirect!` redirects each transition reaching a non-representative state to the unique representative of that state's partition. Note that by the '`all matches`' attribute of `redirect`, transitions between equivalent states become loops at the representatives. The effect of `init`; `add_tag!` and the whole macro `merge` on the sample automaton is shown in Figure 8c and Figure 8d.

Finally, the rule schema `clean_up` exhaustively applies the rule schemata shown in Figure 10. The loop `remove_edge!` deletes all integer-labelled edges, as these auxiliary structures are no longer needed. Then `remove_node!` deletes all non-representative states—these states have become isolated. The remaining states are the unique representatives of their equivalence classes. Last but not least, `untag!` removes the auxiliary second tag of each state so that the remaining tag indicates, as before, whether a state is final or not. The resulting automaton is the unique minimal automaton equivalent to the input automaton (see next section). The automata resulting from `remove_edge!` and the overall program in our running example are shown in Figure 8e and Figure 8f.

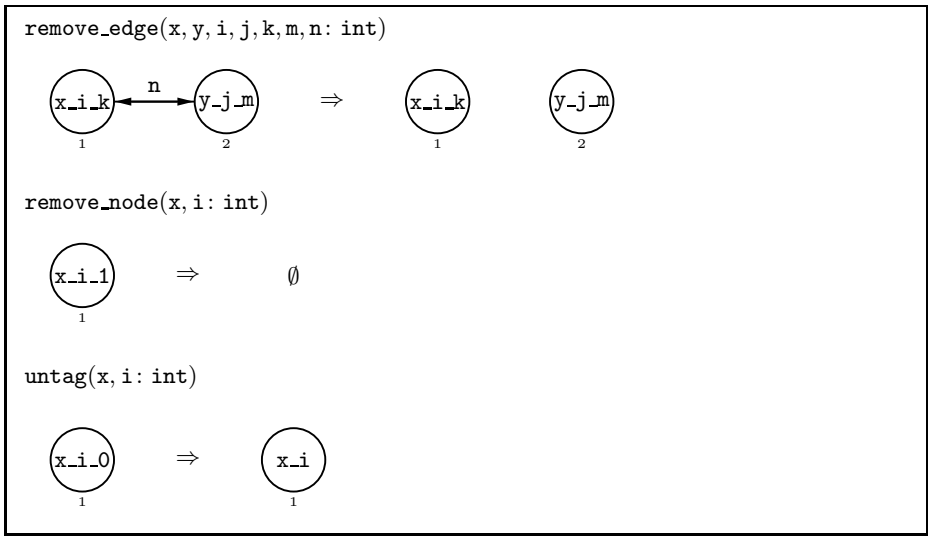


Fig. 10: Rule schemata of the macro `clean_up`

5 Correctness of the Implementation

In this section we prove some propositions which substantiate that the graph program of Figure 3 correctly implements Algorithm 1.

Proposition 1 *The program of Figure 3 terminates for every input automaton.*

Proof. By the conditions of the rule schemata `distinguish` and `propagate`, each application of these schemata reduces the number of state pairs that are not linked by 1-labelled edges of opposite direction. Similarly, each application of `equate` reduces the number of state pairs that are not linked by 0-labelled edges of opposite direction. Thus the macro `mark` terminates.

Each application of the rule schema `add_tag` reduces the number of states that do not have a label of the form x_i_1, where x and i are integers. Hence both the first loop `add_tag!` and the nested loop `(choose; add_tag!)!` terminate (note that `choose` does not affect labels of the form x_i_1). The loop `disconnect!` is trivially terminating as each application of `disconnect` reduces the number of edges in a graph. The loop `redirect!` terminates because each application of `redirect` reduces the sum of the degrees of nodes with a label of the form x_i_1. Thus the macro `merge` terminates, too.

The termination of the three loops in the macro `clean_up` is similarly easy to see. The rule schemata of the first two loops reduce the number of edges respectively the number of nodes, and each iteration of the loop `untag!` reduces the number of nodes with three tags. □

Proposition 2 *The macro* `mark` *links two distinct states by a* `0`*-labelled edge if and only if the states are equivalent.*

Proof. The loop `distinguish!` implements stage 1 of the marking phase of Algorithm 1 in that it links final states with non-final states by a `1`-labelled edge, marking such pairs as non-equivalent. Also, `propagate!` implements stage 2 of the marking phase: the three standard rule schemata represented by `propagate` (see Figure 7) cover the possible relations between the state pairs $\{p, q\}$ and $\{\delta(p, a), \delta(q, a)\}$ in the repeat-loop of Algorithm 1. In particular, they cover the special cases $p = \delta(p, a)$, $q = \delta(q, a)$, $p = \delta(q, a)$ and $q = \delta(p, a)$. Hence Lemma 1 implies that after termination of `propagate!`, two states are linked by a `1`-labelled edge if and only if they are not equivalent. The loop `equate!` then links two distinct states by a `0`-labelled edge if and only if they are not linked by a `1`-labelled edge, implying the proposition. □

Proposition 3 *After termination of the macro* `clean_up`*, two states are equivalent if and only if they are equal.*

Proof. Consider an equivalence class of states of the input automaton. Exactly one state in this class is selected either by the rule schema `init` (in the case of the initial state's class) or by the rule schema `choose` (in all other cases), and a `0`-tag is appended to the state's label. Then the loop `add_tag!` marks all other states in the equivalence class with an extra `1`-tag. Subsequently, `disconnect!` removes all transitions outgoing from `1`-tagged states and `redirect!` redirects away all transitions leading to `1`-tagged states. Hence, after termination of the macro `merge`, `1`-tagged states can be incident only to edges labelled with `0` or `1`. All these edges are deleted by the loop `remove_edge!`, so the `1`-tagged states become

isolated and are eventually removed by `remove_node!`. Thus, upon termination of the macro `clean_up`, from each equivalence class exactly one state remains in the resulting automaton. □

Proposition 4 *For every input automaton A, the automaton $\hat{A}$ produced by the program of Figure 3 is equivalent to A and minimal.*

Proof. By Theorem 1, Proposition 1 and Proposition 2, it suffices to show that the subprogram `merge`; `clean_up` correctly implements the merging phase of Algorithm 1. This can be seen as follows:

- By Proposition 3, each equivalence class of A is represented by its unique representative element in $\hat{A}$.
- The rule schema `init` selects the initial state of A as the representative of its class and `untag` makes this state the initial state of $\hat{A}$.
- Consider any equivalence class of states X, its representative $p \in X$ and any $a \in \Sigma$. If $\delta(p, a)$ is the representative of its equivalence class, then both states are marked with a `0`-tag in `merge` and the transition from p to $\delta(p, a)$ is preserved by the subprogram `disconnect!`; `redirect!`. Otherwise, if $\delta(p, a)$ does not represent its class, then it is marked with a `1`-tag in `merge`. In this case `redirect!` redirects the transition $p \to \delta(p, a)$ to the unique representative of the class of $\delta(p, a)$. Hence $\hat{\delta}(X, a)$, the equivalence class of $\delta(p, a)$, does not depend on the choice of p and thus is well-defined.
- In an equivalence class containing a final state, all states are final as otherwise the loop `distinguish!` would have linked the non-final states with the final state by `1`-labelled edges. Hence the representative of such a class is a final state.

□

6 Conclusion

We have shown how to minimize finite automata with rule-based, visual programming. Programmers need not be concerned with low-level data structures such as state tables but can directly manipulate the transition diagrams of automata. Moreover, GP's rule schemata and control constructs provide a convenient language for reasoning about the correctness of the implementation. Last but not least, the `all matches` option for rule schemata has proved to be useful for keeping the number of rule schemata small, and an extended `edge` predicate has been crucial for forbidding particular edges in the conditions of rule schemata.

The macro `merge` merges equivalent states by choosing representatives of equivalence classes, removing and redirecting transitions, and removing isolated states. A simpler implementation would use non-injective rule schemata to merge states directly—but such rule schemata are not available in GP. Non-injective rule schemata are also useful in other applications and may be realised in a future version of GP.

Finally, this case study could be extended by implementing more efficient automata minimization algorithms. We chose the algorithm of Hopcroft, Motwani and Ullman because of its simplicity, but its cubic running time is not optimal. More efficient algorithms include the quadratic algorithm of Hopcroft and Ullman [5] and Hopcroft's $n\log n$ algorithm [3].

Acknowledgement. We are grateful for the comments of an anonymous referee which helped to improve the presentation of this paper.

References

1. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
2. Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.
3. John E. Hopcroft. An $n\log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
4. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2007.
5. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
6. Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.
7. Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009.
8. Detlef Plump and Sandra Steinert. The semantics of graph programs. In *Proc. Rule-Based Programming (RULE 2009)*, volume 21 of *Electronic Proceedings in Theoretical Computer Science*, pages 27–38, 2010.
9. Jeffrey Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2009.

A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation

Ulrike Golas, Enrico Biermann, Hartmut Ehrig, and Claudia Ermel

Technische Universität Berlin, Germany
ugolas|enrico|ehrig|lieske@cs.tu-berlin.de

Abstract. Several different approaches to define the formal operational semantics of statecharts have been proposed in the literature, including visual techniques based on graph transformation. These visual approaches either define a compiler semantics (translating a concrete statechart into a semantical domain) or they define an interpreter using complex control structures. Based on the existing visual semantics definitions, it is difficult to apply the classical theory of graph transformations to analyze behavioral statechart properties due to the complex control structures. In this paper, we define an interpreter semantics for statecharts based on amalgamated graph transformation where rule schemes are used to handle an arbitrary number of transitions in orthogonal states in parallel. We build on an extension of the existing theory of amalgamation from binary to multi-amalgamation including nested application conditions to control rule applications for automatic simulation. This is essential for the interpreter semantics of statecharts. The theory of amalgamation allows us to show termination of the interpreter semantics of well-behaved statecharts, and especially for our running example, a producer-consumer system.

1 Introduction and Related Work

In [1], Harel introduced statecharts by enhancing finite automata by hierarchies, concurrency, and some communication issues. Over time, many versions with slightly differing features and semantics have evolved. In the UML specification [2], the semantics of UML state machines is given as a textual description accompanying the syntax, but it is ambiguous and explained essentially by examples. In [3], a structured operational semantics (SOS) for UML statecharts is given based on the preceding definition of a textual syntax for statecharts. The semantics uses Kripke structures and an auxiliary semantics using deduction, a semantical step is a transition step in the Kripke structure. This semantics is difficult to understand due to its non-visual nature. The same problem arises in [4], where labeled transition systems and algebraic specification techniques are used.

There are also different approaches to define a visual rule-based semantics of statecharts. One of the first was [5], where for each transition t a transition production p_t is derived describing the effects of the corresponding transition step.

A similar approach is followed in [6], where first a state hierarchy is constructed explicitly, and then a semantical step is given by a complex transformation unit constructed from the transition rules of a maximum set of independently enabled transitions. In [7], in addition, class and object diagrams are integrated. The approach highly depends on concrete statechart models and is not a general interpreter semantics for statecharts. Moreover, problems arise for nesting hierarchies, because the resulting situation is not fixed but also depends on other current or inactive states. In [8], the hierarchies of statecharts are flattened to a low-level graph representing an automaton defining the intended semantics of the statechart model. This is an indirect definition of the semantics, and again dependent on the concrete model, since the transformation rules have to be specified according to this model.

In [9], Varró defines a general interpreter semantics for statecharts. His intention is to separate syntactical and static semantic concepts (like conflicts, priorities etc.) of statecharts from their dynamic operational semantics, which is specified by graph transformation rules. To this end, he uses so-called model transition systems to control the application of the operational rules, which highly depend on additional structures encoding activation or conflicts of transitions and states.

The main advantage of our solution is that we do not need external control structures to cover the complex statecharts semantics: we define a state transition mainly by one interaction scheme followed by some clean-up rules. Therefore, our model-independent definition based on rule amalgamation is not only visual and intuitive but allows us to show termination and forms a solid basis for applying further graph transformation-based analysis techniques.

The rest of the paper is structured as follows. Section 2 gives a brief introduction to our model of statecharts as typed attributed graphs. In Section 3, we review the basic ideas of algebraic graph transformation [10] and give a short introduction to amalgamated transformation based on [11], which is used for the operational semantics of statecharts in Section 4. Based on the given semantics, we discuss the formal analysis of termination of semantical steps in statecharts. The operational semantics is demonstrated along a sample statechart modeling a producer-consumer system in Section 5. Finally, Section 6 concludes our paper and considers future work directions.

2 Modeling of Statecharts

In this section, we model statecharts by typed attributed graphs. We restrict ourselves to the most interesting parts of the statechart diagrams: we allow orthogonal regions as well as state nesting. But we do not handle entry and exit actions on states, nor extended state variables, and we allow guards only to be conditions over active states.

In Fig. 1, the sample statechart `ProdCons` is depicted modeling a producer-consumer system. When initialized, the system is in the state `prod`, which has three regions. There, in parallel a producer, a buffer, and a consumer may act.

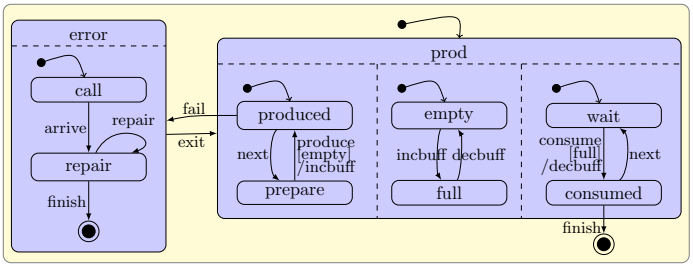


Fig. 1. Sample statechart `ProdCons`

The producer alternates between the states `produced` and `prepare`, where the transition `produce` models the actual production activity. It is guarded by a condition that the parallel state `empty` is also current, meaning that the buffer is empty and may receive a product, which is then modeled by the action `incbuff` denoted after the `/`-dash. Similarly to the producer, the buffer alternates between the states `empty` and `full`, and the consumer between `wait` and `consumed`. The transition `consume` is again guarded by the state `full` and followed by a `decbuff`-action emptying the buffer.

Two possible events may happen causing a state transition to leave the state `prod`: the consumer may decide to finish the complete run; or there may be a failure detected after the production leading to the `error`-state. Then, the machine has to be repaired before the `error`-state can be exited via the corresponding `exit`-transition and the standard behavior in the `prod`-state is executed again.

For our statechart language, we use typed attributed graphs, which are an extension of typed graphs by attributes [10]. We do not give details here, but use an intuitive approach to attribution, where the attributes of a node are given in a class diagram-like style. For the values of attributes in the rules we can also use variables.

The type graph TG_{SC} is given in Fig. 2. We use multiplicities to denote some constraints directly in the type graph. To obtain valid statechart models, some more constraints are needed which are described in the following.

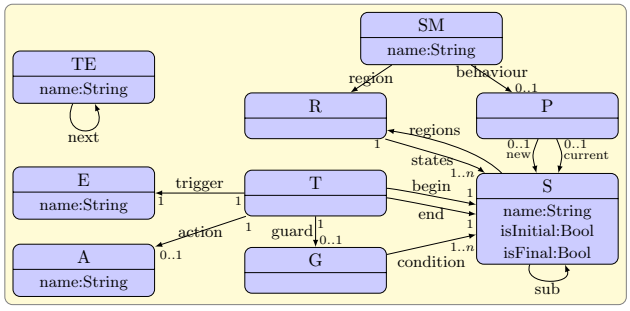


Fig. 2. Type graph TG_{SC} for statecharts

Each diagram consists of exactly one statemachine `SM` containing one or more regions `R`. A region contains states `S`, where state names are unique within each region. A state may again contain one or more regions. Each region is contained in either exactly one state or the statemachine. States may be initial (attribute value `isInitial = true`) or final (attribute value `isFinal=true`), each region has to contain exactly one initial and at most one final state, and final states cannot contain regions. Edge type `sub` is only necessary to compute all substates of a

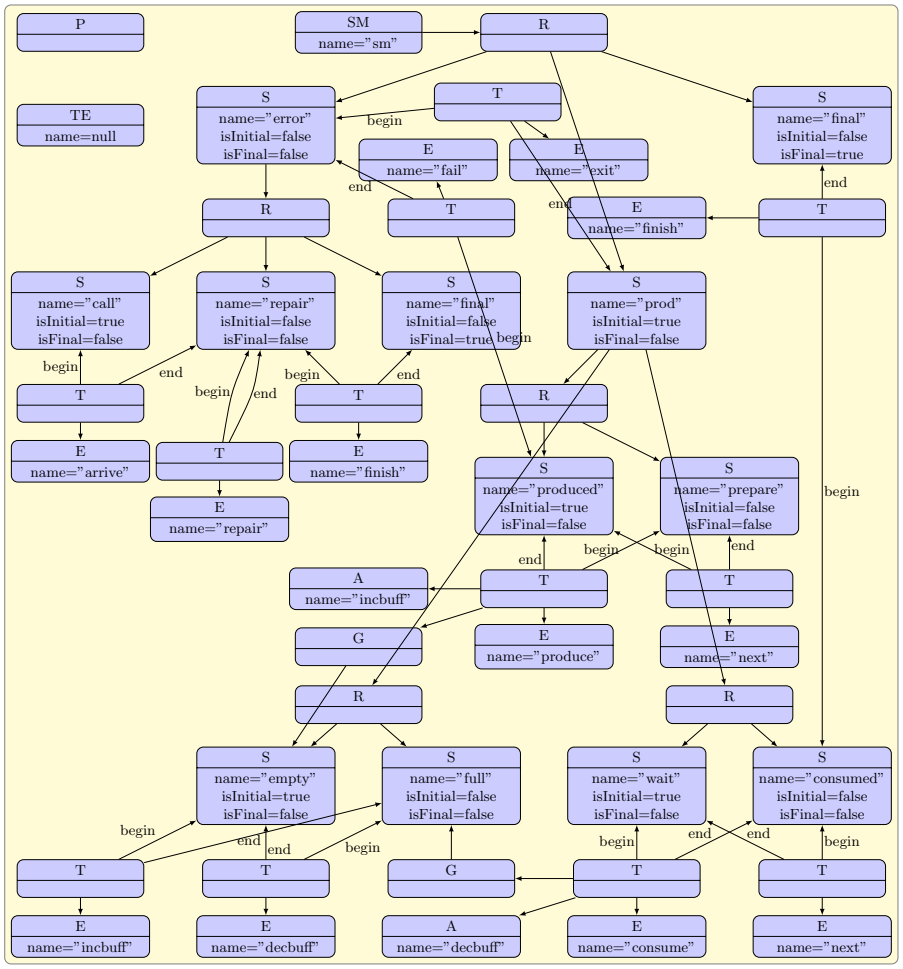


Fig. 3. Statechart `ProdCons` in abstract syntax

state, which we need for the definition of the semantics. This relation is computed in the beginning using the `states`- and `regions`-edges.

A transition `T` begins and ends at a state, is triggered by an event `E`, and may be restricted by a guard `G` and followed by an action `A`. A guard has one ore more states as condition. There is a special event with attribute value `name="exit"` reserved for exiting a state after the completion of all its orthogonal regions, which cannot have a guard condition. Final states cannot be the beginning of a transition and their name has to be `"name=final"`. Transitions cannot link states in different orthogonal regions of the same superstate.

A pointer `P` describes the active states of the statemachine. Note that newly inserted current states are marked by the `new`-edge, while for established current

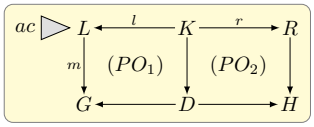
states the `current`-edge is used (which is assumed to be the standard type and thus not marked in our diagrams). This is due to our semantics definition, where we need to distinguish between states that were current before and states that just became current in the last state transition. Trigger elements `TE` describe the events which have to be handled by the statemachine. Note that this is not necessarily a queue because of orthogonal states, but for simplicity we call it event queue. There are at least the empty trigger element with attribute value `name=null` and exactly one pointer in each diagram.

In Fig. 3, the sample statechart `ProdCons` from Fig. 1 is depicted in abstract syntax. Nodes `P` and `TE` are added, which have to exist for a valid statechart model but are not visible in the concrete syntax. For simulating statechart runs, the event queue of the statechart (consisting of only one default element named `null` in Fig. 3) can be filled by events to be processed (see Fig. 9 in Section 5 for a possible event queue for our sample statechart).

3 Introduction to Amalgamated Graph Transformation

In this section, we review the basic ideas of algebraic graph transformation [10] and give a short introduction into amalgamated transformation based on [11], to be used for the interpreter semantics of statecharts in Section 4.

A graph grammar $GG = (RS, SG)$ consists of a set of rules RS and a start graph SG. A rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$ consists of a left-hand side L, an interface K, a right-hand side R, two injective graph morphisms $L \xleftarrow{l} K$ and $K \xrightarrow{r} R$, and an application condition ac on L. Applying a rule p to a graph G means to find a match m of L in G, given by a graph morphism $L \xrightarrow{m} G$ which satisfies the application condition ac, and to replace this matched part $m(L)$ by the corresponding right-hand side R of the rule. By $G \overset{p,m}{\Longrightarrow} H$, we denote the direct graph transformation where rule p is applied to G with match m leading to the result H. The formal construction of a direct transformation is a double-pushout (DPO) as shown in the diagram below with pushouts (PO_1) and (PO_2) in the category of graphs. The graph D is the intermediate graph after removing $m(L)$, and H is constructed as gluing of D and R along K.



A graph transformation is a sequence of direct transformations, denoted by $G \overset{*}{\Longrightarrow} H$, and the graph language $L(GG)$ of graph grammar GG is the set $L(GG) = \{G \mid \exists\, SG \overset{*}{\Longrightarrow} G\}$ of all graphs G derivable from SG.

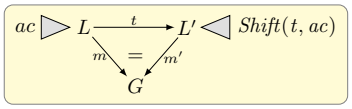
An important concept of algebraic graph transformation is parallel and sequential independence of graph transformation steps leading to the Local Church–Rosser and Parallelism Theorem [12], where parallel independent steps $G \overset{p_1,m_1}{\Longrightarrow} G_1$ and $G \overset{p_2,m_2}{\Longrightarrow} G_2$ lead to a parallel transformation $G \overset{p_1+p_2,m}{\Longrightarrow} H$ based on a parallel rule $p_1 + p_2$. If p_1 and p_2 share a common subrule p_0, the amalgamation theorem in [13] shows that a pair of "amalgamable" transformations $G \overset{(p_i,m_i)}{\Longrightarrow} G_i$ $(i = 1, 2)$ leads to an amalgamated transformation $G \overset{\tilde{p},\tilde{m}}{\Longrightarrow} H$ via

the amalgamated rule $\tilde{p} = p_1 +_{p_0} p_2$ constructed as gluing of p_1 and p_2 along p_0. The concept of amalgamable transformations is a weak version of parallel independence, and amalgamation can be considered as a kind of "synchronized parallelism".

For the interpreter semantics of statecharts we need an extension of amalgamation in [13] w.r.t. three aspects: first, we need a family of rules $p_1, \ldots, p_n$ with a common subrule p_0 for $n \geq 2$; second, we need typed attributed graphs [10] instead of "plain graphs", and third, we need rules with application conditions.

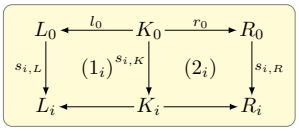
In the following, we formulate the extended amalgamation concept for a general notion of graphs and application conditions, where *general graphs* are objects in a weak adhesive HLR category [10] and *general application conditions* are nested application conditions [14], including positive and negative ones and their combinations by logic operators. For readers not familiar with weak adhesive HLR categories and nested application conditions, it is sufficient to think of rules based on graphs and (typed) attributed graphs with positive and/or negative application conditions (see [10] for more details). A match $L \xrightarrow{m} G$ satisfies a positive (negative) condition of the form $\exists a$ ($\neg\exists a$) for $L \xrightarrow{a} N$ if there is a (no) injective $q : N \to G$ with $q \circ a = n$. More general, $L \xrightarrow{m} G$ satisfies a nested condition of the form $\exists(a, ac_N)$ on L with condition ac_N on N if there is an injective $N \xrightarrow{q} G$ with $q \circ a = m$ and q satisfies ac_N. Note that $\forall(a, ac_N)$ is denoted as $\neg\exists(a, \neg ac_N)$ (see application conditions in Figs. 6 - 7).

An important concept is the shift of ac on L along a morphism $t : L \to L'$ s.t. for all $m' \circ t : L \to G$, m' satisfies $\mathit{Shift}(t, ac)$ if and only if $m = m' \circ t : L \to G$ satisfies ac [15].

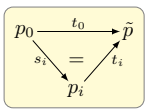


Based on [11], we are now able to introduce amalgamated rules and transformations with a common subrule p_0 of $p_1, \ldots, p_n$. A kernel morphism describes how the subrule is embedded into the larger rules.

Definition 1 (Kernel morphism). *Given rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$ for $i = 0, \ldots, n$, a* kernel morphism $s_i : p_0 \to p_i$ *consists of morphisms $s_{i,L} : L_0 \to L_i$, $s_{i,K} : K_0 \to K_i$, and $s_{i,R} : R_0 \to R_i$ such that in the diagram on the right (1_i) and (2_i) are pullbacks and (1_i) has a pushout complement for $s_{i,L} \circ l_0$, i.e. $s_{i,L}$ satisfies the gluing condition w.r.t. l_0. The pullbacks (1_i) and (2_i) mean that K_0 is the intersection of K_i with L_0 and also of K_i with R_0.*



Definition 2 (Amalgamated rule and transformation). *Given rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$ for $i = 0, .., n$ with kernel morphisms $s_i : p_0 \to p_i$ $(i = 1, \ldots, n)$, then the amalgamated rule $\tilde{p} = (\tilde{L} \longleftarrow \tilde{K} \longrightarrow \tilde{R}, \tilde{ac})$ of $p_1, \ldots, p_n$ via p_0 is constructed as the componentwise gluing of $p_1, \ldots, p_n$ along p_0, where $\tilde{ac}$ is the conjunction of $\mathit{Shift}(t_{i,L}, ac_i)$. $\tilde{L}$ is the gluing of $L_1, \ldots, L_n$ with shared L_0 leading to $t_{i,L} : L_i \to \tilde{L}$. Similar gluing constructions lead to $\tilde{K}$ and $\tilde{R}$ and we obtain*



kernel morphisms $t_i : p_i \to \tilde{p}$ *and* $t_i \circ s_i = t_0$ *for* $i = 1, \ldots, n$. *We call* p_0 *kernel rule, and* $p_1, \ldots, p_n$ *multi rules. An amalgamated transformation* $G \overset{\tilde{p}}{\Longrightarrow} H$ *is a transformation via the amalgamated rule* $\tilde{p}$.

Example 1 (Amalgamated rule construction). We construct an amalgamated rule for the initialization of a statemachine with two orthogonal regions. A pointer has to be linked to the statemachine and to the initial states of both the statemachine's regions. Rules are depicted in a compact notation where we do not show the interface K. It can be inferred by the intersection $L \cap R$. The mappings are given as numberings for nodes and can be inferred for edges. The kernel rule p_0 in Fig. 4 models the linking of the pointer to the statemachine. We have two multi-rules p_1 and p_2 modelling the linking of the pointer to the initial states of two different regions. In the amalgamated rule $\tilde{p}$, the common subaction (linking the pointer to the statemachine) is represented only once since the multi-rules p_1 and p_2 have been glued at the kernel rule p_0. The kernel morphisms are $t_i : p_i \to \tilde{p}$ for $i = 1, 2$.

Given a bundle of direct transformations $G \overset{p_i, m_i}{\Longrightarrow} G_i$ $(i = 1, .., n)$, where p_0 is a subrule of p_i, we want to analyze whether the amalgamated rule $\tilde{p}$ is applicable to G combining all direct transformations. This is possible if they are *multi-amalgamable*, i.e. the matches agree on p_0 and are parallel independent outside. This concept of multi-amalgamability is a direct generalization of amalgamability in [13] and leads to the following theorem [11].

Theorem 1 (Multi-amalgamation). *Given rules* $p_0, \ldots, p_n$, *where* p_0 *is a subrule of* p_i, *and multi-amalgamable direct transformations* $G \overset{p_i, m_i}{\Longrightarrow} G_i$ $(i = 1, \ldots, n)$, *then there is an amalgamated transformation* $G \overset{\tilde{p}, \tilde{m}}{\Longrightarrow} H$.

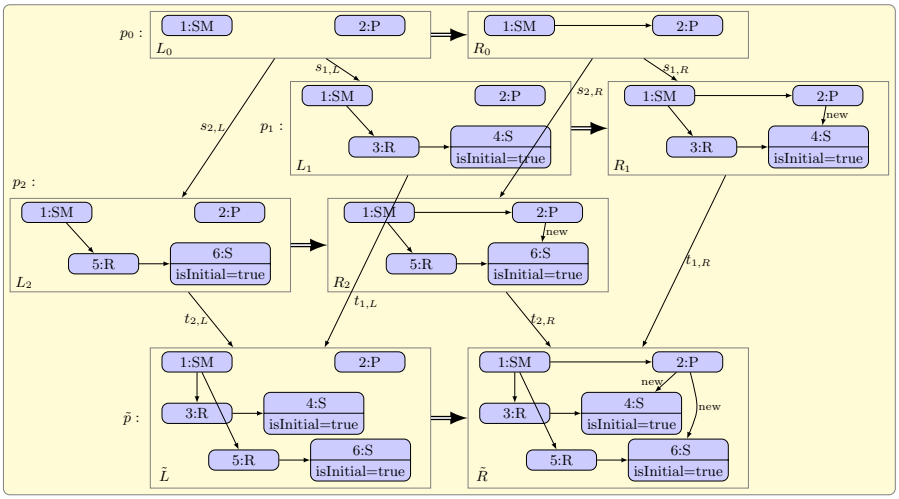


Fig. 4. Construction of amalgamated rule

Proof Idea: Using the properties of the multi-amalgamable bundle, we can show that $\tilde{m}$ with $\tilde{m} \circ t_{i,L} = m_i$ induced by the colimit is a valid match leading to the amalgamated transformation because the componentwise gluing is a colimit construction. For the complete proof see [16].

For many application areas, including the interpreter semantics of statecharts, we do not want to explicitly define the kernel morphisms between the kernel rule and the multi rules, but we want to obtain them dependent on the object to be transformed. In this case, only an interaction scheme $is = \{s_1, \ldots, s_k\}$ with kernel morphisms $s_i : p_0 \to p_j$ $(j = 1, \ldots, k)$ is given, which defines different bundles of kernel morphisms $s'_i : p_0 \to p'_i$ $(i = 1, \ldots, n)$ where each p'_i corresponds to some p_j for $j \leq k$.

Definition 3 (Interaction scheme). *A kernel rule p_0 and a set of multi rules $\{p_1, \ldots, p_k\}$ with kernel morphisms $s_i : p_0 \to p_i$ form an interaction scheme $is = \{s_1, \ldots, s_k\}$.*

Given an interaction scheme, we want to apply as many rules p_j as often as possible over a certain match of the kernel rule p_0. In the following, we consider *maximal weakly disjoint matchings*, where we require the matchings of the multi rules not only to be multi-amalgamable, but also disjoint up to the match of the kernel rule, and maximal in the sense that no more valid matches for any multi rule in the interaction scheme can be found.

Definition 4 (Maximal weakly disjoint matching). *Given an interaction scheme $is = \{s_1, \ldots, s_k\}$ and a tuple of matchings $m = (m_i : L'_i \to G)$, where each p'_i corresponds to some p_j for $j \leq k$, with transformations $G \overset{p'_i, m'_i}{\Longrightarrow} G_i$, then m forms a* maximal weakly disjoint matching *if the bundle $G \overset{p'_i, m'_i}{\Longrightarrow} G_i$ is multi-amalgamable, $m_i(L_i) \cap m'_i(L'_i) = m_0(L_0)$ for all $i \neq i'$, and for any rule p_j no other match $m' : L_j \to G$ can be found such that $((m_i), m')$ fulfills this property.*

Note, that we may find different maximal weakly disjoint matchings for a given interaction scheme, which may even lead to the same bundle of kernel morphisms. For a fixed maximal weakly disjoint match we can apply Thm. 1 leading to an amalgamated transformation $G \overset{\tilde{p}', \tilde{m}}{\Longrightarrow} H$, where $\tilde{p}'$ is the amalgamated rule of $p'_1, \ldots, p'_n$ via p_0.

Given a set IS of interaction schemes is and a start graph SG, we obtain an amalgamated graph grammar with amalgamated transformations via maximal matchings, defined by maximal weakly disjoint matchings of the corresponding multi rules.

Definition 5 (Amalgamated graph grammar). *An* amalgamated graph grammar *$AGG = (IS, SG)$ consists of a set IS of interaction schemes and a start graph SG. The* language *$L(AGG)$ of AGG is defined by $L(AGG) = \{G \mid \exists$ amalgamated transformation $SG \overset{*}{\Longrightarrow} G$ via maximal matchings$\}$.*

4 An Interpreter Semantics for Statecharts

The semantics of statecharts is modeled by amalgamated transformations, where one step in the semantics is modeled by several applications of interaction schemes. For the application of an interaction scheme we use maximal weakly disjoint matchings.

The termination of the interpreter semantics of a statechart in general depends on the structural properties of the simulated statechart. A simulation will terminate for the trivial cases that the event queue is empty, that no transition triggers an action, or that there is no transition from any active state triggered by the current head elements of the event queue. Since transitions may trigger actions which are added as new events to the queue it is possible that the simulation of a statechart may not terminate. Hence, it is useful to define structural constraints that provide a sufficient condition guaranteeing termination of the simulation in general for well-behaved statecharts, where we forbid cycles in the dependencies of actions and events.

Definition 6 (Well-behaved statecharts). *For a given statechart model, the* action–event graph *has as nodes all event names and an edge* (n_1, n_2) *if an event with name* n_1 *triggers an action named* n_2*.*

A statechart is called well-behaved *if it is finite, has an acyclic state hierarchy, and its action–event graph is acyclic.*

For example, the action–event graph of a statechart is cyclic if an event a triggers an action b, and the execution of the corresponding event b triggers action a. In this case, with only one external trigger element (either a or b) the statechart will run forever and does not terminate. An example of a well-behaved statechart is our statechart model in Fig. 1. It is finite, has an acyclic state hierarchy, and its action–event graph is acyclic, since the only action–event dependencies in our statechart occur between `produce` triggering `incbuff` and `consume` triggering `decbuff`.

For the *initialization step*, we provide a finite event queue and compute all substates of all states, which is not shown here. Then, the interaction scheme `init` is applied followed by the interaction scheme `enterRegions` applied as long as possible, which are depicted in Fig. 5. With `init`, the pointer is associated to the statemachine and all initial states of the statemachine's regions. The interaction scheme `enterRegions` handles the nesting and sets the current pointer also to the initial states contained in an active state. When applied as long as possible, all substates are handled. Note that only those initial substates become active that are contained in a hierarchy of nested initial states. The interaction scheme `enterRegions` also contains the identical kernel morphism $id_{p_{40}} : p_{40} \to p_{40}$ to ensure that the kernel rule is also applied in the lowest hierarchy level. For later use, also double edges are deleted and if the direct superstate is not marked by the pointer a `new` edge is added to it.

The initialization step (applying `init` once and `enterRegions` as long as possible) terminates because the application of the interaction scheme `enterRegions`

terminates: each application of `enterRegions` replaces one `new` edge with a `current` edge. The multi rules p_{41} and p_{42} create new `new`-edges on the next lower and upper levels of a hierarchical state, but if the state hierarchy is acyclic this interaction scheme is only applicable a finite number of times. The same holds for the multi rule p_{43} which deletes double edges, since the number of `current`- and `new`-edges is decreased. Thus, the transformation terminates.

Fact 1 (Termination of initialization step). *For well-behaved statecharts, the initialization step terminates.*

A *semantical step*, i.e. switching from one state to another, is done by applying the interaction scheme `transitionStep` shown in Fig. 6 followed by the interaction schemes `enterRegions!`, `leaveState1!`, `leaveState2!`, and `leaveRegions!` given in Fig. 5, Fig. 7, and Fig. 8 in this order, where `!` means that the corresponding interaction scheme is applied as long as possible.

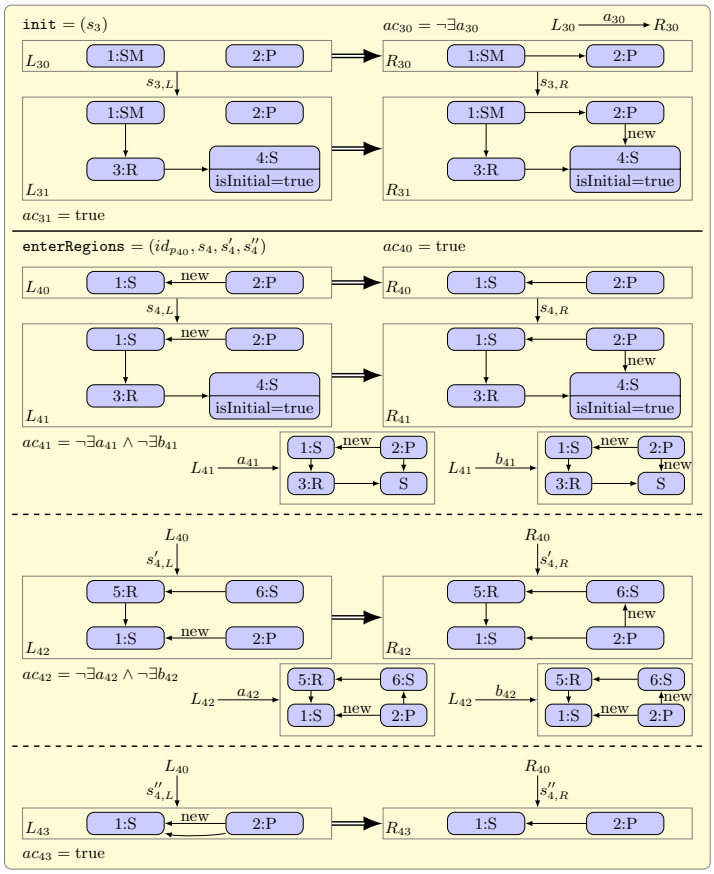


Fig. 5. The interaction schemes `init` and `enterRegions`

For a semantical step, the first trigger element (or one of the first if more than one action of different orthogonal substates may occur next) is chosen and deleted, while the corresponding state transitions are executed. `exit` trigger elements are handled with priority ensured by the application condition ac_{50}. A transition triggered by its trigger element is active if the state it begins at is active, its guard condition state is active, and it has no active substate where a transition triggered by the same event is active. These restrictions are handled be the application conditions ac_{51} and ac_{52}. Moreover, if an action is provoked, it has to be added as one of the first next trigger elements. The two multi rules

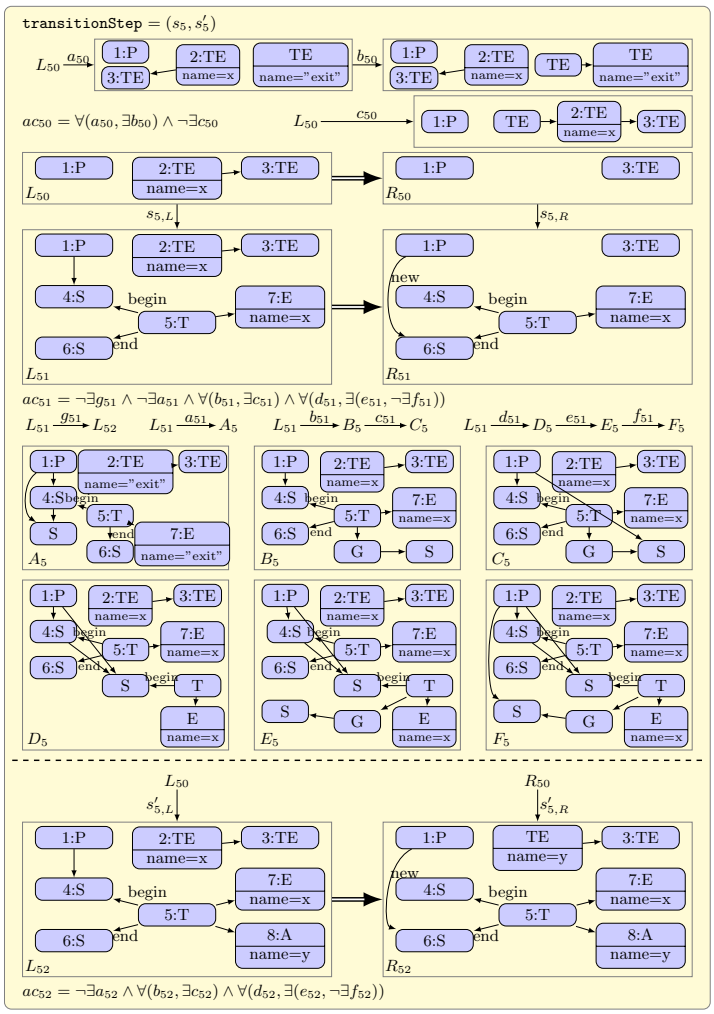


Fig. 6. The interaction scheme `transitionStep`

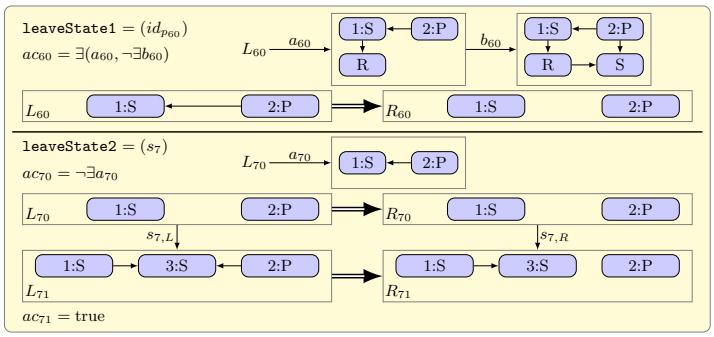


Fig. 7. The interaction schemes `leaveState1` and `leaveState2`

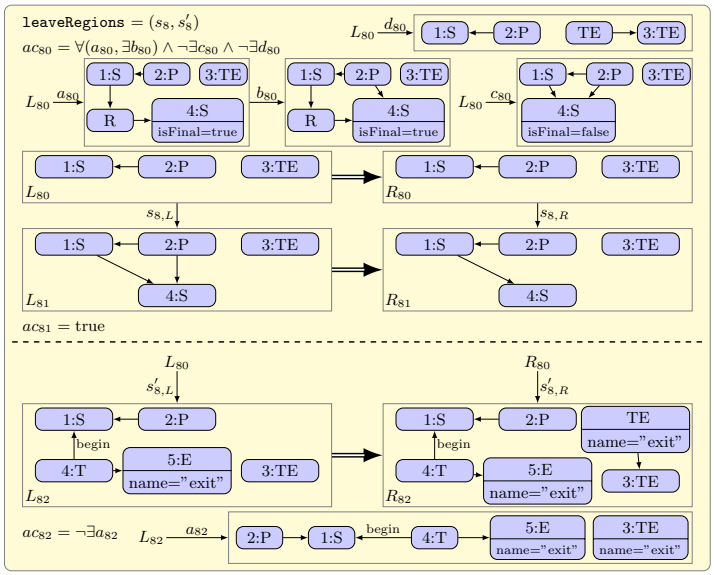


Fig. 8. The interaction scheme `leaveRegions`

of `transitionStep` handle the state transition with and without action, respectively. The application condition ac_{52} is not shown explicitly, but the morphisms $a_{52}, \ldots, f_{52}$ are similar to $a_{51}, \ldots, f_{51}$ containing an additional node `8:A`.

The interaction schemes `leaveState1`, `leaveState2`, and `leaveRegions` handle the correct selection of the active states. When for a yet active state with regions, by state transitions all states in one of its regions are no longer active, also this superstate is no longer active, which is described by `leaveState1`. The interaction scheme `leaveState2` handles the case that, when a state become inactive by a state transition, also all its substates become inactive. If for a state with orthogonal regions the final state in each region is reached then these final

states become inactive, and if the superstate has an `exit`-transition it is added as the next trigger element. This is handled by `leaveRegions`.

For the termination of a semantical step it is sufficient to show that the four interaction schemes `enterRegions`, `leaveState1`, `leaveState2`, and `leaveRegions` are only applicable a finite number of times. The interaction scheme `enterRegions` terminates as shown in Fact 1. The interaction schemes `leaveState1`, `leaveState2` as well as the multi rule p_{81} of `leaveRegions` reduce the number of active states in the statechart by deleting at least one `current` edge. The application of the second multi rule p_{82} of the interaction scheme `leaveRegions` prevents another match for itself because it creates the situation forbidden by its application condition ac_{82}. It follows that the application of each of these four interaction schemes as long as possible terminates.

Fact 2 (Termination of semantical steps). *Given a well-behaved statechart, each semantical step terminates.*

Combining our termination results we can conclude the termination of the statecharts semantics for well-behaved statecharts.

Theorem 2 (Termination of interpreter semantics). *For well-behaved statecharts with finite event queue, the interpreter semantics terminates.*

Proof. According to Facts 1 and 2, each initialization step and each semantical step terminates. Moreover, each semantical step consumes an event from the event queue. If it triggers an action, the acyclic action–event graph ensures that there are only chains of events triggering actions, but no cycles, such that after the execution of this chain the number of elements in the event queue actually decreases. Thus, after finitely many semantical steps the event queue is empty and the operational semantics terminates. □

5 Application to the Running Example

We now consider an initialization and a semantical step in our statechart example from Fig. 1. In the top of Fig. 9, we show an incoming event queue as needed for our system run to be processed. Note that the actions triggered by transitions do not occur here because they are started internally, while the other events have to be supplied from the environment. Below, the current states and their corresponding state transitions are depicted.

For simulation, we apply the rules for the semantics starting with the graph in abstract syntax in Fig. 3, extended by the event queue from Fig. 9 and all `sub`-edges marking that a state is a substate of its superstate.

For the initialization step, we apply the interaction scheme `init` from Fig. 5 followed by `enterRegions` as long as possible. With `init`, we connect the state machine and the pointer node, and in addition set the pointer to the `prod` state using a `new` edge. Now the only available kernel match for `enterRegions` is the

match mapping node `1` to the `prod` state, and with maximal matchings we obtain the bundle of kernel morphisms $(id_{p_{40}}, s_4, s_4, s_4)$, where node `4` in L_{41} is mapped to the states `produced`, `empty`, and `wait`, respectively. After applying the corresponding amalgamated rule, the current pointer is now connected to the state machine and state `prod`, and via `new` edges to the states `produced`, `empty`, and `wait`. Further applications of `enterRegions` using these three states for kernel matches, respectively, lead to the bundle $(id_{p_{40}})$, thus changing the `new` to `current` edges by its application. As result, the states `prod`, `produced`, `empty`, and `wait` are current, which is the initial situation for the statemachine as shown in Fig. 9. We do not find additional matches for `enterRegions` as we have only one level of nesting in our diagram, which means that the initialization is completed.

For a state transition, the interaction scheme `transitionStep` in Fig. 6 is applied, followed by the interaction schemes `enterRegions!`, `leaveState1!`, `leaveState2!`, and `leaveRegions!` given in Fig. 5, Fig. 7, and Fig. 8.

For the initial situation, the kernel rule p_{50} in Fig. 6 has to be matched such that node `2` is mapped to the first trigger element `next` and node `3` to `produce`, otherwise the application condition of the rule would be violated. For the multi rules, there are two events of name `next`, but since the state `consumed` is not current, only one match for L_{51} is found mapping node `4` to the current state `produced` and `6` to the state `prepare`. All application conditions are fulfilled, since this transition does not have a guard or action, and the state `produced` does not have any substates. Thus, the application of the bundle (s_5) deletes the first trigger element `next`, which is done by the kernel rule, and redirects the current pointer from `produced` to `prepare` via a `new` edge. An application of the interaction scheme `enterRegions` using the bundle $(id_{p_{40}})$ changes this `new` to a `current` edge. Since we do not find further matches for L_{40}, L_{60}, L_{71}, L_{81}, and L_{82}, the other interaction schemes cannot be applied. This means that the states `prod`, `prepare`, `empty`, and `wait` are now the current states, which is the situation after the state transition triggered by `next` as shown in Fig. 9. The procession of the remaining trigger elements works analogously.

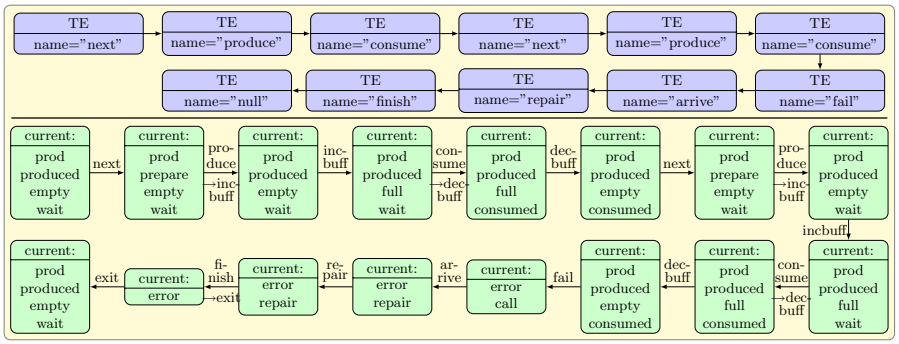


Fig. 9. Event queue and state transitions

According to Thm. 2, the simulation of our example terminates because our statechart is *well-behaved* and the event queue is finite.

6 Conclusion and Future Work

In this paper, we have defined a formal interpreter semantics for statecharts leading to a visual interpreter semantics. It is based on the theory of algebraic graph transformation and hence a solid basis for applying graph transformation-based analysis techniques. Unfortunately, the classical theory of graph transformations [12] is not adequate to model the interpreter semantics of statecharts because we need rule schemes to handle an arbitrary number of transitions in orthogonal states in parallel. In this paper, we have solved this problem using amalgamated graph transformation [11] in order to handle the interpreter semantics. As a first step towards the analysis of this semantics we have shown the termination of initialization and semantical steps and, more general, the termination of the interpreter semantics for *well-behaved* statecharts.

Our formal approach is also a promising basis to analyze other properties like confluence and functional behavior in the future. Since termination and local confluence implies confluence, it is sufficient to analyze local confluence. This has been done successfully for algebraic graph transformation based on standard rules and critical pairs [10]. It remains to extend this analysis from standard rules to amalgamated rules constructed by interaction schemes and to take into account maximal matchings as well as all essential amalgamated rules constructed from one interaction scheme.

Another interesting research area to be considered in future is the nesting of kernel morphisms, which may lead to a hierarchical interaction scheme such that a semantical step of the statechart is actually a direct amalgamated transformation over one interaction scheme, and we no longer need rules for redirecting the `current` pointer afterwards.

References

[1] Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming **8** (1987) 231–274

[2] OMG: Unified Modeling Language (OMG UML), Superstructure, Version 2.2. (2009)

[3] Beeck, M.: A Structured Operational Semantics for UML-statecharts. Software and Systems Modeling **1** (2002) 130–141

[4] Reggio, G., Astesiano, E., Choppy, C., Hussmann, H.: Analysing UML Active Classes and Associated State Machines - A Lightweight Formal Approach. In Maibaum, T., ed.: Fundamental Approaches to Software Engineering. Proceedings of FASE 2000. Volume 1783 of LNCS., Springer (2000) 127–146

[5] Maggiolo-Schettini, A., Peron, A.: A Graph Rewriting Framework for Statecharts Semantics. In Cuny, J., Ehrig, H., Engels, G., Rozenberg, G., eds.: Graph Grammars and Their Application to Computer Science. Volume 1073 of LNCS., Springer (1996) 107–121

[6] Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In Gogolla, M., Kobryn, C., eds.: Proceedings of UML 2001. Volume 2185 of LNCS., Springer (2001) 241–256
[7] Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J.: An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In Butler, M., Petre, L., Sere, K., eds.: Proceedings of IFM 2002. Volume 2335 of LNCS., Springer (2002) 11–28
[8] Gogolla, M., Parisi-Presicce, F.: State Diagrams in UML: A Formal Semantics Using Graph Transformations. In: Proceedings of ICSE 1998, IEEE (1998) 55–72
[9] Varró, D.: A Formal Semantics of UML Statecharts by Model Transition Systems. In Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G., eds.: Proceedings of ICGT 2002. Volume 2505 of LNCS., Springer (2002) 378–392
[10] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs. Springer (2006)
[11] Golas, U., Ehrig, H., Habel, A.: Multi-Amalgamation in Adhesive Categories. In: Proceedings of ICGT 2010. Volume 6372 of LNCS., Springer (2010) 346–361
[12] Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific (1997)
[13] Böhm, P., Fonio, H.R., Habel, A.: Amalgamation of Graph Transformations: A Synchronization Mechanism. JCSC **34**(2-3) (1987) 377–408
[14] Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. MSCS **19**(2) (2009) 245–296
[15] Ehrig, H., Habel, A., Lambers, L.: Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. ECEASST **26** (2010) 1–23
[16] Golas, U.: Multi-Amalgamation in M-Adhesive Categories: Long Version. Technical Report 2010/05, Technische Universität Berlin (2010)

The Pull-Tab Transformation

Abdulla Alqaddoumi[1] Sergio Antoy[2] Sebastian Fischer[3] Fabian Reck[3]

[1] Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, U.S.A.

[2] Computer Science Department
Portland State University
Portland, OR 97207, U.S.A.

[3] Institut für Informatik
Christian-Albrechts-Universität Kiel
D-24098 Kiel, Germany

Abstract. We present a new approach to the execution of functional logic programs. Our approach relies on definitional trees for the deterministic portions of a computation and on a graph transformation, called *pull-tab*, for the non-deterministic portions. This transformation moves, one level at a time, non-deterministic choices towards the root of the graph representing the state of a computation. With respect to need-based strategies for functional logic computations, our approach executes only localized graph replacements, a property that characterizes it as "pay as you go" and makes it suitable for parallel execution.

1 Introduction & Motivation

Non-deterministic programs are simpler to design and easier to reason about than their deterministic counterparts [4]. These advantages do not come for free. The burden unloaded from the programmer is placed on the execution mechanism. Loosely speaking, all the alternatives of a non-deterministic choice must be explored to some degree to ensure that no result of a computation is lost. Doing this efficiently is a long-standing problem.

There are three main approaches to the execution of non-deterministic steps in functional logic programs. This paper proposes a fourth approach with some interesting characteristics missing from the other approaches. We begin by proposing a simple example to present the existing approaches, to understand their limitations, and to compare their differences. Below, is a short program that we use as a running example. The syntax is Curry [10].

```
flip 0 = 1
flip 1 = 0                                    (1)
coin = 0 ? 1
```

We want to evaluate the expression

```
(flip x, flip x) where x = coin                    (2)
```

We recall that '`?`' is a library function, called *choice*, that returns either of its arguments, i.e., it is defined by the rules:

```
x ? _ = x
_ ? y = y                                          (3)
```

and that the `where` clause introduces a shared expression. Every occurrence of `x` in (2) has the same value throughout the entire computation according to the *call-time choice* semantics [13]. By contrast in `(flip coin, flip coin)` each occurrence of `coin` is evaluated independently of the other. Fig. 1 highlights the difference between these two expressions when they are represented as graphs.

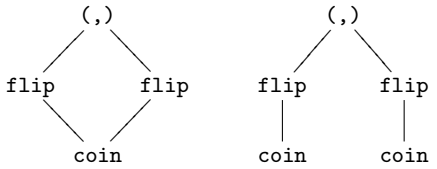


Fig. 1. Graph representations of (2) and `(flip coin, flip coin)`.

We recall that a *context* is an expression with a distinguished symbol called *hole* denoted '[]'. If C is a context, $C[x]$ is the expression obtained by replacing the hole in C with x. E.g., the expression in (2) can be written as $t[\texttt{coin}]$, where t is the context of `coin`. An expression rooted by a node labeled by the choice symbol is referred to as *a choice*.

1.1 Previous approaches

Backtracking is the most traditional approach to non-deterministic computations in functional logic programming. Evaluating a choice in some context, say $C[u?v]$, consists in selecting either argument of the choice, e.g., u (the criterion for selecting the argument is not relevant to our discussion), replacing the choice with the selected argument, which gives $C[u]$, and continuing the computation. In typical interpreters, if and when the computation of $C[u]$ completes, the result is consumed, e.g., printed, and the user is given the option to either terminate the execution or compute $C[v]$. Referring to our running example, $t[\texttt{0?1}]$ results in the evaluation of $t[\texttt{0}]$ followed by the evaluation of $t[\texttt{1}]$. Backtracking is well-understood and relatively simple to implement. It is employed in successful languages such as Prolog [14] and in language implementations such as PAKCS [11] and $\mathcal{TOY}$ [8]. The major objection to backtracking is its incompleteness. If the computation of $C[u]$ does not terminate, no result of $C[v]$ is ever obtained.

Copying (or *cloning*) fixes the inherent incompleteness of backtracking. Evaluating a choice in some context, say $C[u?v]$, consists in evaluating simultaneously

(e.g., by interleaving steps) and independently $C[u]$ and $C[v]$. In typical interpreters, if and when the computation of either completes, the result is consumed, e.g., printed, and the user is given the option to either terminate the execution or continue with the computation of the other. Referring to our running example, t[`0?1`] results in the simultaneous and independent evaluations of t[`0`] and t[`1`]. Copying is simpler than backtracking and it is used in some experimental implementations of functional logic languages [5, 18]. A significant optimization of copying consists in sharing (and thus computing only once) subexpressions of the context that are not on the spine of the choice (the path from the root to the choice). The major objection to copying is the significant investment of time and memory made when a non-deterministic step is executed. In well-designed programs, most alternatives of a choice fail to produce any result, hence portions of the copied context may never be used. For a contrived example, notice that in `1+(2+(...+(`n `` `div` coin)...))` an arbitrarily large context is copied when the choice is evaluated, but this context is almost immediately discarded.

Bubbling is an approach proposed to avoid the drawbacks of backtracking and copying [2, 15]. Bubbling is similar to copying, in that it copies a portion of the context of a choice to concurrently compute all its alternatives, but this portion of copied context is typically smaller than the entire context. We recall that in a rooted graph g, a node d is a *dominator* of a node n, proper when $d \neq n$, iff every path from the root of g to n contains d. An expression $C[u?v]$ can always be seen as $C_1[C_2[u?v]]$ in which the root of $C_2[\,]$ is a dominator of the choice. A trivial case arises when $C_1[\,] = [\,]$ and $C_2[\,] = C[\,]$. Evaluating a choice in some context, say $C[u?v]$, distinguishes whether or not C is empty. If C is the empty context, u and v are evaluated simultaneously and independently, as in copying, but there is no context to copy. Otherwise, the evaluation consists in finding C_1 and C_2 such that $C[u?v] = C_1[C_2[u?v]]$ and the root of C_2 is a proper dominator of the choice, and evaluating $C_1[C_2[u]?C_2[v]]$. If C_1 is the empty context, then bubbling is exactly as copying. Otherwise a smaller context, i.e., C_2 instead of C, is copied. Bubbling intends to reduce copying in the hope that some alternative of a choice will quickly fail. Referring to our running example, t[`0?1`] bubbles to the expression represented in the left-hand side of Fig. 2. Observe that the node labeled `(,)` is the immediate proper dominator of the choice.

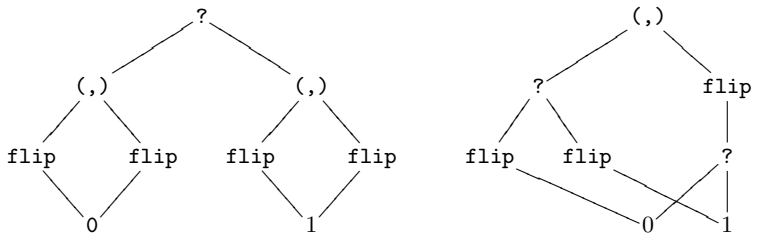


Fig. 2. Graph representation of the state of the computation of (2) after a bubbling (left side) and a pull-tab (right side) step.

Bubbling is more recent than the other approaches, it is not yet as well-understood, and it still is the subject of active investigation [7]. An objection to bubbling is the cost of finding a choice's immediate dominator and the risk of paying this cost repeatedly if no alternative of the choice fails. This cost entails traversing a possibly-large portion of the choice's context. Traversing the context is more efficient than copying it, since copying requires node construction in addition to the traversal, but it is still unappealing, since the cost of a non-deterministic step is not predictable and it may grow with the size of an expression.

2 Pulling the Tab

A *program* is a graph rewriting system [9, 16]. An *expression* is a rooted graph over the signature of the program. A *computation* is the repeated transformation of an expression by either a *rewrite* or a *pull-tab* step defined below. Rewrite steps are computed with standard techniques [1]. Informally, a pull-tab step moves a choice toward the root of an expression one level at a time. As in a rewrite, a (sub)expression of an expression is replaced. Textually, a (sub)expression of the form $f(t_1, \ldots, a_1 ? a_2, \ldots, t_k)$, where f is not a choice, is replaced by $f(t_1, \ldots, a_1, \ldots, t_k) ? f(t_1, \ldots, a_2, \ldots, t_k)$. For example, $((0+2) ? (1+2)) * 3$ is the pull-tab of $(0 ? 1) + 2 * 3$. If and when a choice reaches the root of an expression, its alternatives have no context and are evaluated independently of each other. The metaphor behind the name is to look at a path from the root of an expression down to a choice as a zipper in which the choice is a pull tab. As a choice is pulled up, the path opens into two strands, like a zipper, below the pull tab. Pulling a choice above a predecessor copies the smallest amount of context, i.e., the predecessor node only.

Unfortunately, the pull-tab transformation as sketched above may be unsound. Fig. 3 shows a state of the computation of (2) after some rewrite and pull-tab steps. The superscript of some symbols may be ignored for the time being. Without a corrective action, four results would be produced. In particular, the right argument of the left choice, i.e., `(1,0)`, is not intended by the semantics of current functional logic languages such as Curry [10] and $\mathcal{TOY}$ [8].

Unsoundness occurs when some choice has two predecessors, as in our running example. The choice will be pulled up along two paths creating four strands that eventually must be combined together. Some combinations will contain mutually exclusive alternatives, or in other words subexpressions impossible to obtain with the call-time choice semantics. In our running example, one such combination mixes a 1 originating from the left alternative of the initial choice with a 0 from the right alternative of *the same choice*. Avoiding expressions with mutually exclusive alternatives suffices to recover the soundness of the pull-tab strategy.

To avoid impossible combinations of subexpressions, we track the history of the non-deterministic steps of each expression. This history has been used in other aspects of functional logic computations [3, 6] under the name of "fingerprint." A node in a graph is decorated with information such as labeling and

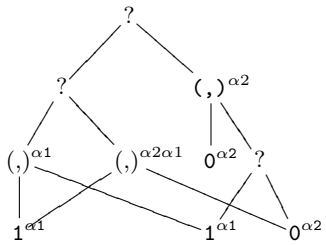


Fig. 3. States of the computation of (2) after both rewrite and pull-tab steps. Superscripts are fingerprints. α is the choice identifier of every node labeled by `?`. A node labeled by (,), the pair constructor, has fingerprint $\alpha 1 \alpha 2$. The subgraph at this node mixes the left and right arguments of a choice and consequently does not produce a result.

successor functions. For defining the pull-tab strategy, we extend the decorations of nodes. Let Ω be a denumerable set whose elements we call *choice identifiers* and denote by Greek letters. A *fingerprint* is a finite subset of $\Omega \times \{1, 2\}$ whose pairs we denote by juxtaposition. A node of each graph of a computation is decorated by a fingerprint and, if the node is labeled by the choice symbol, it is also decorated by a choice identifier.

A rewrite step preserves these decorations and assigns an empty fingerprint to any node introduced by the replacement and a fresh choice identifier if the node is a choice. A pull-tab step involves two nodes, a choice c and one of its predecessors p not labeled by a choice. Let α be the choice identifier of c and f the fingerprint of p. Informally, the step "moves up" c creating a new node c' and "splits" the predecessor p creating two new nodes, say p_1 and p_2. In the resulting expression, the choice identifier of c' is again α and the fingerprints of p_1 and p_2 are $f \cup \{\alpha 1\}$ and $f \cup \{\alpha 2\}$, respectively.

If the fingerprint of a node n contains $\alpha 1$ and $\alpha 2$, for some choice identifier α, the graph rooted by n is semantically impossible and should be eliminated. Fig. 3 shows an example of such a node, where superscripts denote fingerprints.

3 Current Work

We are developing a virtual machine based on the pull-tab strategy. The machine, about 1000 lines of commented Ruby [17] code, includes a rudimentary parser for the command line interpreter and a sophisticated printer for development purposes and the presentation of results. The machine executes multisteps [12] that, depending on the functional logic program being executed, may contain dozens or hundreds of elementary steps. Since both rewrite and pull-tab steps are localized graph replacements, we expect to be able to execute the elementary steps of a multistep in parallel with only a modest synchronization overhead.

References

1. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
2. S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, pages 61–70, Vienna, Austria, April 2006.
3. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, pages 73–82, Lisbon, Portugal, September 2009.
4. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85, April 2010.
5. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125, Lubeck, Germany, September 2005. Springer LNCS 3474.
6. Bernd Brassel and Frank Huch. On a tighter integration of functional and logic programming. In *APLAS'07: Proceedings of the 5th Asian conference on Programming languages and systems*, pages 122–138, Berlin, Heidelberg, 2007. Springer-Verlag.
7. D. Brown. Ph.D. dissertation, 2010. In progress.
8. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.
9. R. Echahed. Inductively sequential term-graph rewrite systems. In *Graph Transformations, 4th International Conference (ICGT 2008)*, pages 84–98, Leicester, UK, 2008. Springer, LNCS 5214.
10. M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*, 2006. Available at `http://www.informatik.uni-kiel.de/~curry`.
11. M. Hanus, editor. *PAKCS 1.9.1: The Portland Aachen Kiel Curry System*, 2008. Available at `http://www.informatik.uni-kiel.de/~pakcs`.
12. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991.
13. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
14. ISO. Information technology - Programming languages - Prolog - Part 1, 1995. General Core. ISO/IEC 13211-1, 1995.
15. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: The HO case. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, pages 147–162. Springer LNCS 4989, 2008.
16. D. Plump. Term graph rewriting. In H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg, editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.
17. D. Thomas and A. Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., 2001.
18. A. Tolmach, S. Antoy, and M. Nita. Implementing functional logic languages using multiple threads and stores. In *Proc. of the 2004 International Conference on Functional Programming (ICFP)*, pages 90–102, Snowbird, Utah, USA, September 2004. ACM.

Coinductive graph representation: the problem of embedded lists

Celia Picard and Ralph Matthes

Institut de Recherche en Informatique de Toulouse (IRIT),
C.N.R.S. and University of Toulouse III, France

Abstract. When trying to obtain formally certified model transformations, one may want to represent models as graphs and graphs as greatest fixed points. To do so, one is rather naturally led to define co-inductive types that use lists (to represent a finite but unbounded number of children of internal nodes). These concepts are rather well supported in the proof assistant Coq. However, their use in our intended applications may cause problems since the co-recursive call in the type definition occurs in the list parameter. When defining co-recursive functions on such structures, one will face guardedness issues, and in fact, the Coq system refuses those definitions by applying a syntactic criterion that is too rigid here.
We offer a solution using dependent types to overcome the guardedness issues that arise in our graph transformations. We also give examples of further properties and results, among which finiteness of represented graphs. All of this has been fully formalized in Coq.

1 The problem: explanation on an example

It is recognized that the on-going engineering effort for modeling and metamodeling has to be backed by rigorous formal methods. In this context, we aim at performing certified model transformations. In a first time, certification should be done by interactive theorem proving. This presupposes the representation of models and metamodels in the language of the theorem prover. We chose to represent models and metamodels as graphs and to use the Coq system [1] as a specification and verification tool. The Coq system offers a language with a rich notion of inductive and co-inductive types, i. e., data types that arise as least and greatest solutions of fixed-point equations, respectively.
This led us to represent node-labeled graphs with co-inductive types (in order to represent the infinite navigability in loops). The idea we had was that each node would have a label (a natural number for example) and a finite list of sons (graphs themselves). This type can be created through the following constructor:

Definition 1 (Graph, viewed coinductively).

$$mk\_Graph : nat \to (list\ Graph) \to Graph$$

[1] See `http://coq.inria.fr/`

$mk\_Graph\ n\ l$ constructs a graph from the natural number n and the list of graphs l. Since this is the greatest fixed point, no assumption about finite generation through mk_Graph is made. The empty list hides the base case.

Note 1 (Lists). For lists we use the *Caml* notation: [] for the empty list and $[a_1; a_2; \ldots]$ for an explicit enumeration.

Note 2. Here we deal with single-rooted connected graphs because they correspond best to co-inductive types. We are currently working on a more general definition, still within the expressive power of Coq.

Example 1 (A simple example that does not use co-recursion: just a leaf).

$$Leaf_n := mk\_Graph\ n\ []$$

Example 2 (Example of a finite graph). The graph of Figure 1 can be represented as a term of type *Graph* with the following co-recursive definition:

$$Finite\_Graph := mk\_Graph\ 0\ [mk\_Graph\ 1\ [Finite\_Graph]]$$

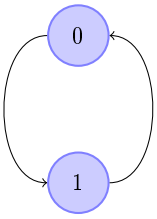


Fig. 1. Example of a finite graph

Note 3. This graph is finite but unfolds into an infinite (regular) tree, and thus allows infinite navigation.

Example 3 (Example of an infinite graph). To represent the graph of Figure 2 as a term of type *Graph*, we first define a family of infinite graphs, parameterized by the label of the first node:

$$Infinite\_Graph_n := mk\_Graph\ n\ [Infinite\_Graph_{n+1}]$$

The graph of Figure 2 corresponds to $Infinite\_Graph_0$.

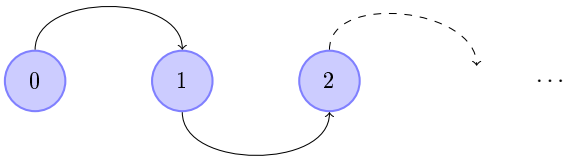


Fig. 2. Example of an infinite graph

Note 4. This graph is infinite and unfolds into an infinite irregular tree.

Until here, there is no problem with Coq. But if we try to apply a transformation on a graph, Coq complains. For example, it is forbidden to define the following co-recursive function that applies a function f to each label of a graph:

Definition 2. $applyF2G\ (f : nat \to nat)(mk\_Graph\ n\ l) :=$
$mk\_Graph\ (f\ n)\ (map\ (applyF2G\ f)\ l)$

Note 5. Of course, *map* is the usual mapping function that maps a function over all the elements of a list, i. e., $map\ f\ [a_1; a_2; \ldots] = [f\ a_1; f\ a_2; \ldots]$.

The reason why $applyF2G$ is not accepted by Coq is that the guardedness condition on co-inductive types is rather restrictive in Coq, and in this case, too restrictive. Indeed, in Coq the guardedness condition is based on productivity [6]. Technically speaking it says that a co-recursive call must always be the argument of some constructor of inductive or co-inductive type. Here, the co-recursive call is an argument of the *map* function, which is itself under the constructor. This is too indirect to satisfy the guardedness condition. For more details about the guardedness conditions in Coq see [3] and [11].
Basically, the idea of the guardedness condition is to ensure that potentially infinite objects are computable. This means that we can always obtain more information on the structure of the object in a finite amount of time. Consider the example of streams that are always infinite. The application of a filter on streams is actually a problem since we cannot ensure that the next "good" element will be found in a finite amount of time. But here the problem is quite different in nature: it is not about finding the next constructor but about the indirection of the co-recursive call through *map*. However, in the case of *map*, this indirection is harmless (we would only have to inspect in parallel the elements of that list). So, Coq's guardedness condition forbids us to write semantically well-formed definitions: guardedness restrictions go beyond syntactic well-formedness and normal typing constraints but are still of a syntactic nature and thus only approximate the semantic notion of productivity that guarantees well-definedness.

In this article, we offer and study a solution to overcome the problem with the guardedness condition for definitions involving graphs. In Section 2 we explain the solution, and we will see how it solves our problem in Section 3. Finally, in Section 4 we offer an extension of *ilist* to represent multiplicities in metamodels. All the work presented here has been formally proved in Coq (in the version 8.2). The whole development is available in [14].

2 The solution: *ilist*

We develop here a solution that allows us to bypass the guardedness condition.

2.1 The idea

The idea to solve the problem is to use a function that mimics the behaviour of lists (this idea has also been mentioned by Chlipala in [5]) Lists can easily be

seen as functions. If T is the type parameter, then a list can be considered as a function that associates to each element of a set of n elements (n being the length of the list) an element of type T. An element of the definition domain represents the position of the associated element in the list.

Example 4. The list [10; 2; 5] can be transformed into the function of Figure 3.

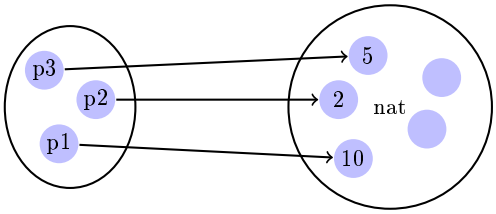


Fig. 3. Representation of the function corresponding to the list [10 ; 2 ; 5]

But to be able to represent such a function, we need to have a set of n elements.

2.2 *Fin* – a family of types for finite index sets

It is trivial to get an inductive type with n elements, for $n = 0, 1, 2, \ldots$ but it is not for an indeterminate n. Here we need n to be a parameter of the type. To represent a set of n elements, we have chosen to use the representation that has also been used by Altenkirch in [1] and by McBride and McKinna in [12]. We actually represent a family of sets parameterized by the number of elements they contain (in our case, the length of the list). This family is called *Fin*. *Fin* has type $nat \rightarrow Set$. It is defined through the two following constructors:

Definition 3 (Fin, viewed inductively).

$$first\ (n : nat) : Fin\ (n + 1)$$
$$succ\ (n : nat) : Fin\ n \rightarrow Fin\ (n + 1)$$

Note 6. *Fin* is a Generalized Algebraic Data Type (GADT). Those data types are also available in current implementations of the Haskell programming language.

First of all, we want to prove that *Fin* n indeed is a set of n elements.

Note 7. We use *card* to represent the cardinality of the set in an informal way.

Lemma 1. $\forall n,\ card\ \{i \mid i : Fin\ n\} = n$

Proof (by induction)

[Case 0] No constructor allows to create an element of type *Fin* 0. Therefore $card\ \{\ i \mid i\ :\ Fin\ 0\ \}\ =\ 0$

[Case n+1] With the constructor *succ*, we can construct as many elements of *Fin* $(n+1)$ as there are in *Fin* n. The constructor *first* allows us to construct one more element of *Fin* $(n + 1)$.
Therefore, $card\ \{i \mid i : Fin\ (n + 1)\} = card\ \{i \mid i : Fin\ n\} + 1$.
With the induction hypothesis, we have $card\ \{i \mid i : Fin\ (n+1)\} = n+1$. □

Note 8. This informal proof cannot be formalized in Coq because there is no such *card* operation. With the *card* operation the following result would have been a triviality.

Lemma 2. $\forall n\ m,\ n = m \Leftrightarrow Fin\ n = Fin\ m$

Proof

[Direction $\Rightarrow$] The proof here is straightforward, it is only a matter of rewriting and we directly have the property. In informal mathematics this would not even be stated.

[Direction $\Leftarrow$] This direction is much trickier than the first one. Indeed, the first idea we had was to show that all the elements of $Fin\ n$ are in $Fin\ m$ too, doing a type rewrite on the type of the element. However, Coq handles it poorly and does not allow us to do something like that (at least, we did not find a way to do it). In order to prove this property, we defined the type of segments of natural numbers (let's call it $NatSeg$): $NatSeg\ n := \{\ m \mid m\ <\ n\ \}$. We proved on it that if there exists a bijection between $NatSeg\ n$ and $NatSeg\ n'$ then $n = n'$. The proof is not straightforward here, but at least we could do it. [2] Then we could prove that there is a bijection between $Fin\ n$ and $NatSeg\ n$ and that therefore, $\forall n\ m,\ Fin\ n = Fin\ m \Rightarrow n = m$. □

Note 9. One may wonder why we did not use directly $NatSeg$ instead of *Fin* to represent a set of n elements. The reason is that it is much more comfortable to have an inductive type (with concrete finite elements). The elements of $NatSeg\ n$ contain a proof of $m < n$, and we consider *Fin* more elementary.

Note 10. Using the fact that there is a bijection between $Fin\ n$ and $NatSeg\ n$ and that the latter is a representation of $0...n-1$, we get an alternative (necessarily informal) proof of Lemma 1.

2.3 *Ilist* implementation

Now that we have the domain of our functions, we can define the type of functions itself (let's call it *ilistn*).

The function *ilistn* It has two parameters: the type of the elements of the list and its length. It is defined as follows:

Definition 4. $ilistn\ (T : Set)\ (n : nat) := Fin\ n \to T$

Now we have our function that "mimics" lists. To each element of a set of n elements, it associates an element of type T. However, one problem remains. Indeed, as we said, *ilistn* needs two parameters. But for a list, the length is not one of its parameters, it is inherent to it.

[2] We had the confirmation by other members of the Coq user community that no simple proof was known yet.

The list counterpart, *ilist* To solve this problem, we create a new type that combines the length of the list and the corresponding *ilistn*. We call it *ilist* :

Definition 5. $ilist\ (T : Set) := \{n : nat\ \&\ ilistn\ T\ n\}$

The two projection functions on *ilist* are called *lgti* (for the natural numbers part) and *fcti* for the *ilistn* part. If we call *CE* the constructor for elements of kind ... & ... (the dependent pair), then an element l of type $ilist\ T$ can be "reconstructed" as $CE\ (lgti\ l)\ (fcti\ l)$.
We can show that there is a bijection between *ilist* and lists. To do so, we define two functions (one for each direction, let's call them *ilist2list* and *list2ilist*). We show that the compositions $ilist2list \circ list2ilist$ and $list2ilist \circ ilist2list$ are extensionally equal, i. e., pointwise Leibniz equal, to the identity. So we finally have what we were looking for: a type equivalent to lists but not inductive.

An equivalence on *ilist* It is very useful to be able to compare two elements of the same type. Here, of course, we would like to be able to compare two elements of *ilist*. For *Fin* there was no problem, it is inductive and does not have any type parameter so Leibniz equality is fine.
But here, the problem is different. We intuitively see that in order to compare elements of *ilist*, we will have to compare two different things: the two parts of its definition. The first one, its length, is the easy one: it is a natural number, no problem here. But the second one is trickier. Indeed, we have to make sure that the elements of the two elements of *ilist* we are comparing are equivalent element-wise. And we have no insurance that they are comparable with Leibniz equality (actually, in our concrete problem here, they are not, they are only comparable through bisimulation). We thus define an inductive proposition (let's call it *ilist_rel* because it is the lifting of *ilist* to relations) that relates two elements of *ilist*. Apart from the elements of *ilist* we are comparing (let's call them l_1 and l_2) and the type parameter (let's call it T), the proposition needs a given base relation R on type T, of type $relation\ T$, which is a shorthand for $T \to T \to Prop$. Then, $ilist\_rel\ R$ has type $relation\ (ilist\ T)$.
Intuitively, we would like to define *ilist_rel* such that:

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel\ R\ l_1\ l_2 \Leftrightarrow$$
$$lgti\ l_1 = lgti\ l_2 \wedge (\forall i : Fin\ (lgti\ l_1), R\ (fcti\ l_1\ i)\ (fcti\ l_2\ i))$$

Unfortunately, this does not work. Indeed, $fcti\ l_2$ has type $Fin\ (lgti\ l_2) \to T$ and i has type $Fin\ (lgti\ l_1)$. We know that $lgti\ l_1 = lgti\ l_2$ but the types $Fin\ (lgti\ l_1)$ and $Fin\ (lgti\ l_2)$ are still syntactically different. Therefore, we must convert i to type $Fin\ (lgti\ l_2)$ (the hypothesis $lgti\ l_1 = lgti\ l_2$ ensures that we have the right to do it). In Coq, there is a special pattern matching feature that allows us to make this type rewrite. We do not detail it here, for more information see [16, Chapter 1.2.13 and 4.5.4].
In a context where $h : lgti\ l_1 = lgti\ l_2$ and $i : Fin\ (lgti\ l_1)$, we call i'_h the result of converting i to type $Fin\ (lgti\ l_2)$.

With this we can properly write our definition for *ilist_rel*:

Definition 6 (*ilist_rel*)**.**

$$\forall l_1\ l_2 : ilist\ T, ilist\_rel\ R\ l_1\ l_2 \Leftrightarrow$$
$$\exists h : lgti\ l_1 = lgti\ l_2,\ \forall i : Fin\ (lgti\ l_1), R\ (fcti\ l_1\ i)\ (fcti\ l_2\ i'_h)$$

Using advanced dependently typed pattern matching techniques, one can show that *ilist_rel* R is an equivalence relation if R is one.

Functions on *ilist* As we have a bijection between lists and *ilist*, we can redefine any function f that has lists as parameters and/or lists as result type. In particular that means that all the usual functions (and higher order functions) on lists have their counterpart on *ilist*. For example, the well-known *filter* function on lists gives this analogue on *ilist* (for P a predicate on the type of elements):

Definition 7. $ifilter\ P\ l := list2ilist\ (filter\ P\ (ilist2list\ l))$

And in general, any function f that would have the type $list\ T \to list\ T$ could be translated to *ilist* as a function f'. The function f' is defined as follows :

$$f' := list2ilist \circ f \circ ilist2list$$

However, this is only anecdotal as we embed f into another function and therefore we do not solve the guardedness issue. For example, if we defined an analogue of the *map* function (let's call it *imap*) with this method, we would have:

Definition 8 (imap, first try). $imap\ f\ l := list2ilist\ (map\ f\ (ilist2list\ l))$

But this does not solve our problem since the function f (which in our example is the co-recursive call) would still be embedded into the *map* function, which as we saw does not work.

imap We have to redefine the *map* function directly. This is actually quite easy since the part of the *ilist* that is affected by the *map* is the functional part (*ilistn*). So in fact, the *imap* function is little more than a composition of functions. What we have to do is to compose the functional part of the *ilist* with the function we have to apply and then recreate the *ilist*. The result has the same natural numbers part ($lgti\ l$) and a new functional one : $fun\ i \Rightarrow f\ (fcti\ l\ i)$:

Definition 9 (imap, suitable for guarded definitions).

$$imap\ (f : A \to B)\ (l : ilist\ A) := CE\ (lgti\ l)\ (fun\ i : Fin\ (lgti\ l) \Rightarrow f\ (fcti\ l\ i))$$

Here, the function f (and therefore in our example the co-recursive call) is directly under the constructor *CE*. This satisfies the guardedness restriction and solves our problem. So we see that the use of function spaces is considered less critical than the use of inductive types because they are more primitive. They are even part of the logical framework. This could not have been done on lists

since they are defined inductively and so should be the functions that manipulate them. There is no other way than recursion to define *map* on lists. All such higher-order functions add a layer between the constructor and the function given as a parameter. In the case this function is a co-recursive call, it can create, as we saw, a conflict with the guardedness conditions. As the *imap* function is not defined recursively, there is no layer added and as we said, in case of a co-recursive call the guardedness condition is satisfied.

Universal quantification For further definitions (see Section 3) we need to define a property on *ilist* that expresses that all the elements of an *ilist* satisfy a predicate P. We call it *iall* (it is the counterpart for the *for_all* function in Caml). It is defined as follows:

Definition 10 (iall).

$$iall\ (T : Set)\ (P : T \to Prop)\ (l : ilist\ T) : Prop := \forall i, P\ (fcti\ l\ i)$$

The *ilist* with only one element Only for comfort, we define the *ilist* that contains only one element and call it *singleton*. It consists of an *ilist* that has length 1 ($lgti\ singleton = 1$) and a constant function that associates any element of $Fin\ 1$ (but we know that there is only one element in $Fin\ 1$) to the element contained in the *ilist*. It is defined as follows and will be useful to deal with our examples (see Section 3.5):

Definition 11 (singleton).

$$singleton\ (T : Set)\ (t : T) : ilist\ T := CE\ 1\ (fun\ (\_ : Fin\ 1) \Rightarrow t)$$

3 Back to the original problem

Now, we can redefine the type *Graph* using *ilist* and define various functions and properties on it.

3.1 Definitions of *Graph* and *applyF2G*

The definition of *Graph* is identical to the previous one, except that lists are replaced by *ilist*. We define it through the following constructor:

Definition 12 (Graph, viewed co-inductively).

$$mk\_Graph : nat \to (ilist\ Graph) \to Graph$$

Now we can define the function *applyF2G* and Coq does not complain anymore:

Definition 13. $applyF2G\ (f : nat \to nat)\ (mk\_Graph\ n\ l) :=$
$mk\_Graph\ (f\ n)\ (imap\ (applyF2G\ f)\ l)$

3.2 An equivalence on *Graph*

We can also define all the other tools we need. In particular, we can define an equivalence relation on *Graph*. Indeed, as elements of *Graph* are coinductive, Leibniz equality cannot be used here (it is just too fine-grained), we need bisimulation. To relate two elements of *Graph*, we need (as we did for *ilist*) to relate their two parts. The label part is simple because, for natural numbers, we can use Leibniz equality (but in the general case where labels would be of a type T, we would need an equivalence relation on T, that could be in certain cases Leibniz equality). For the sons part, that is represented by an *ilist*, we will use the equivalence relation defined on *ilist*: *ilist_rel* (see Section 2.3). As the type parameter for the *ilist* is itself *Graph*, *ilist_rel* needs the equivalence relation on *Graph* as argument. So this relation must be defined coinductively. We call *label* and *sons* the two functions on *Graph* that return respectively the natural number and the *ilist* part of a *Graph*. They are such that the following lemma is correct:

Lemma 3. $\forall g : Graph, g = mk\_Graph\ (label\ g)\ (sons\ g)$

Note 11. Here we have the right to use Leibniz equality to compare two elements of *Graph* as they are definitionally equal for any g of the form $mk\_Graph\ n\ l$ (and not only bisimulated).

Finally, we can define the equivalence relation on *Graph* (let's call it *Geq*) as follows (defined co-inductively):

Definition 14 (Geq). $\forall g_1\ g_2 : Graph,\ Geq\ g_1\ g_2 \Leftrightarrow$
$label\ g_1\ =\ label\ g_2\ \wedge\ ilist\_rel\ Geq\ (sons\ g_1)\ (sons\ g_2)$

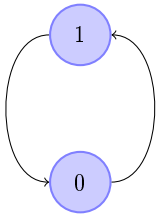
Note 12. If we had used lists instead of *ilist* we would have had another problem here. Indeed, usually lists are compared with Leibniz equality but here we cannot do that since the elements of the lists would only be comparable through bisimulation. Therefore, we would have had to define a new relation on lists that takes this into account. An example of a Coq implementation of such a relation can be seen in [14] (in the file *Listeq.v*).

It is possible to show that *Geq* is an equivalence relation using the same style of reasoning as for *ilist_rel*.

Note 13. To be equivalent, two elements of *Graph* would have to be constructed in the same way. But the two graphs of Figure 1 (see the expression of Example 5 below in Section 3.5) and Figure 4 would not be equivalent even though we might wish them to be. A coarser relation should be designed for this purpose.

3.3 Universal quantification on *Graph*

As we did with *ilist* (see Section 2.3), we define a property on *Graph*. It will be useful, in particular in Section 3.4. This property expresses that a predicate $P : Graph \to Prop$ on *Graph* is satisfied by a *Graph* g and all its descendants (sons, sons of its sons, and so on). As *Graph* is co-inductive, this property must be defined co-inductively too. We call it *G_all* and it is defined as follows:



This graph is represented by the following expression using Definition 12:
$Finite\_Graph' :=$
$mk\_Graph\ 1\ (singleton$
$\quad (mk\_Graph\ 0\ (singleton\ Finite\_Graph')))$

Fig. 4. Other representation of the graph of Figure 1

Definition 15 (G_all). $\forall P,\ \forall g,\ G\_all\ P\ g \Leftrightarrow P\ g \wedge iall\ (G\_all\ P)\ (sons\ g)$

3.4 Finiteness of *Graph*

Another interesting property on *Graph* is finiteness. It would be interesting for example to prove that the examples 2 and 3 indeed are respectively finite and infinite. Say that a *Graph* g is finite means that it contains a finite number of elements of *Graph*. This can be expressed by the fact that all the elements of *Graph* contained in g can fit into a finite list. This is the way we choose to define the finiteness of a *Graph*. We call the finiteness property *G_finite*. To define it, we need a predicate (let's call it *P_finite*) to check whether a *Graph* g is included in a list of graphs. By included we mean that there exists an element of the list that is related through bisimulation (Geq) with g. We use $\in$ to say that an element is in a list.

Definition 16. $P\_finite\ (lg : list\ Graph)\ (g : Graph) := \exists y,\ y \in lg \wedge Geq\ g\ y$

Thanks to it we can define *G_finite*.

Definition 17 (G_finite).

$$\forall g,\ G\_finite\ g \Leftrightarrow \exists lg : list\ Graph,\ G\_all\ (P\_finite\ lg)\ g$$

3.5 Proofs of finiteness and infiniteness

We want here to prove that the two examples 2 and 3 are respectively finite and infinite. First, we must redefine them with our new definition of *Graph*.

Example 5 (Redefinition of Finite_Graph). $Finite\_Graph :=$
$mk\_Graph\ 0\ (singleton\ (mk\_Graph\ 1\ (singleton\ Finite\_Graph)))$

Example 6 (Redefinition of Infinite_Graph$_n$).

$$Infinite\_Graph_n := mk\_Graph\ n\ (singleton\ Infinite\_Graph_{n+1})$$

Now we want to prove that Example 5 is finite, i. e., that Lemma 4 is true:

Lemma 4 (*Finite_Graph* **is finite).** *G_finite Finite_Graph*

Proof (by co-induction)

The proof here is quite easy. We must give a list containing all the elements of *Graph* contained in *Finite_Graph* and show that it actually contains them all. There are only two elements of *Graph* contained in *Finite_Graph*: *Finite_Graph* itself and *mk_Graph* 1 (*singleton Finite_Graph*). Therefore, the provided list is: [*Finite_Graph* ; *mk_Graph* 1 (*singleton Finite_Graph*)]. Now, we only have to prove that *Finite_Graph* is contained in the list (but it was designed for it !); that its sons are (it only has one son: *mk_Graph* 1 (*singleton Finite_Graph*), so it is in the list) and that the sons of its son are in the list too (this is *Finite_Graph* itself, so we use the co-inductive hypothesis). □

Similarly, we want to prove that $Infinite\_Graph_n$ is not finite.

Lemma 5 ($Infinite\_Graph_n$ **is infinite).** $\forall n, \neg\ G\_finite\ Infinite\_Graph_n$

To prove this, we use an auxiliary lemma that says that if the labels of a *Graph* are unbounded, then the *Graph* is infinite.

Lemma 6. $\forall g, (G\_finite\ g) \Rightarrow (\exists m,\ G\_all\ (fun\ x \Rightarrow label\ x \leq m)\ g)$

We do not detail the proof here but it is a straightforward co-induction.

Now, to prove Lemma 5, we only have to prove that the labels of $Infinite\_Graph_n$ are unbounded and we will have the result simply using Lemma 6. To prove that the labels of $Infinite\_Graph_n$ are unbounded, we show that $\forall m, m \geq n \Rightarrow Infinite\_Graph_m \subseteq Infinite\_Graph_n$

Note 14. We informally use the notation $\subseteq$ to say that a *Graph* is included in another.

With this, it is easy to show that the labels are unbounded (since the first label of $Infinite\_Graph_n$ is n).

Note 15. In a similar way, we can show that if the *number of sons* in a *Graph* is unbounded, then the *Graph* is infinite. However, it is also possible to construct elements of *Graph* in which the out-degree of a node is bounded and so are the labels and that are still infinite (see Figure 5 for an example). Here, the proof of infiniteness is much more difficult (it is part of [14]).

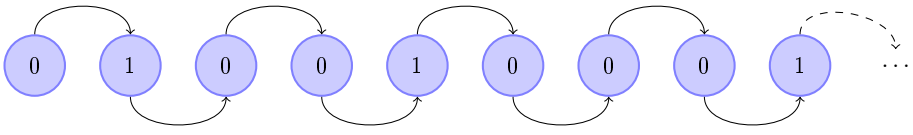


Fig. 5. Example of an infinite graph with bounded number of sons and bounded labels

3.6 *Graph* in *Graph*

We will need to represent the property asserting that an element *gin* of *Graph* is (strictly) included into another element *gout* of *Graph* (see Section 3.7). We

split the situation into two different cases: *gin* is part of *sons gout* or *gin* is included in one of *gout*'s sons. In Coq, the following definition is represented as an inductive property with two constructors.

Definition 18 (GinG).

$$\forall gin\ gout :\ Graph,\ GinG\ gin\ gout \Leftrightarrow \begin{cases} \exists i,\ Geq\ gin\ (fcti\ (sons\ gout)\ i) & or \\ \exists i,\ GinG\ gin\ (fcti\ (sons\ gout)\ i) \end{cases}$$

We can prove that this relation is transitive.

3.7 Cycles in *Graph*

It may also be useful to define a property about the existence of a cycle in an element of *Graph*. To do so, we use the property *GinG* defined above.
First of all, we define a property saying that an element g of *Graph* is itself a cycle (i. e., there is a non-empty path from the root to the root). This means that g is itself included in g. Therefore the definition of *isCycle* is straightforward.

Definition 19 (isCycle). $isCycle\ g \Leftrightarrow GinG\ g\ g$

Using this definition, it is easy to define the property of existence of a cycle in an element g of *Graph*. Just as we did for *GinG*, we divide the property into two cases. Either g is a cycle or one element of *sons g* has a cycle. As before, in Coq this is defined through two constructors.

Definition 20 (hasCycle).

$$\forall g :\ Graph,\ hasCycle\ g \Leftrightarrow \begin{cases} isCycle\ g & or \\ \exists i,\ hasCycle\ (fcti\ (sons\ g)\ i) \end{cases}$$

For a finite element of *Graph*, it is quite easy to prove the existence or non-existence of a cycle (for example, it is straightforward to prove that Example 5 has a cycle). However if there are many nodes, the proof might be long. Indeed, the proofs are constructive, that is, one will have to exhibit the cycle to prove that it exists or to look into each different path to show that there is none. This last operation may be tedious.

4 Multiplicity

In this section, we present an extension of *ilist* to represent multiplicities in metamodels representation. As said in Section 1, our final goal is to be able to represent big metamodels in Coq and then perform transformations on these models. But the very first thing to do is to represent them. There are various problems that arise then. For example, the representation of inheritance which we have not solved yet. Another one is the representation of multiplicity. For this, we have extended the concept of *ilist* to take into account multiplicity, i. e., an interval constraint on the out-degree.
First we need a property (let's call it *PropMult*) to say whether a number is between the two specified bounds of the multiplicity condition. The inferior bound (let's call it *inf*) always exists (it can be 0 but always has a value).

Therefore it has type *nat*. On the opposite, the superior bound (let's call it *sup*) may not exist (multiplicity "$*$"). Therefore it has type *option nat* (constructor *Some* if it exists, constructor *none* if not). The property is expressed as follows:

Definition 21 (PropMult). $\forall inf\ sup\ k$,

[Case $sup = Some\ s$**]** $k \geq inf \wedge k \leq s$
[Case $sup = None$**]** $k \geq inf$

Thanks to this property we can refine our *ilistn* (that was the set of functions of type $Fin\ n \to T$) to keep only the ones whose n satisfies *PropMult*.

Definition 22. $ilistnMult\ T\ inf\ sup\ n := \{ln : ilistn\ T\ n \mid PropMult\ inf\ sup\ n\}$

Note 16. Elements of *ilistnMult* are pairs formed by an element of *ilistn* and a proof of $PropMult\ inf\ sup\ n$, hence the type is empty if $PropMult\ inf\ sup\ n$ does not hold.

Now that we have *ilistnMult*, the definition of *ilistMult* (the counterpart of *ilist* with multiplicity) is straightforward. It is the same as the definition of *ilist* (see Section 2.3) but using *ilistnMult*.

Definition 23. $ilistMult\ T\ inf\ sup := \{n : nat\ \&\ ilistnMult\ T\ inf\ sup\ n\}$

We can define a relation and functions on *ilistMult* very much the same way as we did on *ilist*. Therefore we do not present them again here.
We can also show that there is a bijection between $ilistMult\ T\ 0\ None$ and list (we do it the same manner we did for *ilist*, defining *ilistMult2list* and *list2ilistMult* and showing that their compositions are extensionally equal to the identity).

Note 17 (multiplicities). The multiplicities 0 and *None* are explained by the fact that a list may have no element (empty list, so inf $= 0$) or a finite but unbounded number of elements (i. e., multiplicity "$*$", so $sup = None$).

Combining the lemmas about bijection between lists, *ilist* and *ilistMult*, we obtain that there exists a bijection between $ilist\ T$ and $ilistMult\ T\ 0\ None$.

The important result is that then all definitions written with $ilist\ T$ can be written equivalently with $ilistMult\ T\ 0\ None$. In particular the following definition of *GraphMult* is equivalent to *Graph*:

Definition 24 (GraphMult).

$$mk\_\ GraphMult : nat \to (ilistMult\ Graph\ 0\ None) \to Graph$$

5 Related Work

The work presented here shares concerns with other work. Amongst them, we can cite the work by Bertot and Komendantskaya in [3]. In their paper they treat the problem of representing streams as functions, to overcome Coq's guardedness issues. The main difference is that we need a finite definition set ($Fin\ n$) whereas they can just use *nat*. Recall that our problem was with the embedded *inductive* type of lists and not the co-inductive streams. In [7], Dams proposes an alternative solution to our problem in Coq. He defines everything co-inductively (so instead of lists, he has streams of sons) and then restricts what needs to be

finite by a property of finiteness. In that approach, programming is done with a bigger datatype and the proofs have to be carried out for the "good" elements. In [13], Niqui describes a general solution for the representation of bisimulation in Coq using category theory. However, as we tried to apply his theory, it seemed that only co-inductive embedded types could be treated (streams but not lists) with the given solution. Moreover, it did not seem possible to parameterize the bisimulation by an equivalence relation over the types of the elements.
Coq is not the only proof assistant to have guardedness issues. For example, they are present in Agda, another proof assistant based on predicative type theory. We studied the way guardedness issues are addressed in Agda. Danielsson describes it in [8] (see also extended case study with Altenkirch in [9]). The solution used is to redefine the types (for example the types of lists) adding a constructor for each problematic function (for example, *map*). However, this is based on a mixture of inductive and co-inductive constructors for a single datatype definition, which is not admissible in Coq and of experimental status even in Agda.
About graph representation in functional languages, we can mention the work by Erwig. In [10] he proposes a way to represent directed graphs using inductive types, where, in the inductive step, a new node is added, together with all its edges to and from previously introduced nodes. Being "new" or "previously introduced" is not part of the inductive specification but only of a more refined implementation. Moreover, there is no certification of these invariants for graph algorithms, although this might be interesting future work in expressive systems such as Coq. However, the main conceptual difference to our work is that in his representation, all nodes are represented at the same level (they are more or less elements of a list) while we actually wanted, for our own needs, to build into the construction navigability through the graph, including its loops.

6 Conclusion

In this paper, we have developed a complete solution to overcome Coq's guardedness condition when mixing the inductive type of lists with co-inductive types. The Coq development corresponding to this work is available in [14]. This framework can be extended with new features as needed. For the results we wanted to obtain, it worked well. Clearly, it would have been easier if a more refined guardedness criterion had been available in Coq but the last ten years have shown that getting the criterion right is a quite subtle issue.

However, we would now be interested in a more general solution to overcome the guardedness condition with any embedded inductive type (not only lists). But we realized that to do so, we needed to be more abstract. We are working on that now. In particular, we are studying the possibility to draw more inspiration from category theory. The work by Niqui in [13] might be a good start.

Moreover, the work we present has to be seen as part of a larger project where we are interested in a co-inductive representation of metamodels (see Section 4). We have solved the problem of multiplicity but the problem of inheritance/subtyping remains. Of course, we look for an extensible way to represent metamodels (they

may vary over time). Certainly, Poernomo's work on type theory for metamodels in [15] is relevant here. The work by Boulmé on FOCAL [4] that has been realized with Coq, will probably help in treating the inheritance problem.

Acknowledgement: This development was initiated by the original idea of Jean-Paul Bodeveix to use *ilist* to overcome the guardedness condition. We are grateful for several interesting suggestions by Silvano Dal Zilio and for the careful reading of a preliminary version by Martin Strecker.

References

1. Altenkirch, T.: A formalization of the strong normalization proof for system F in LEGO. In: Bezem, M., Groote, J.F. (eds.) Typed Lambda Calculi and Applications, International Conference, TLCA 1993. Lecture Notes in Computer Science, vol. 664, pp. 13–28. Springer (1993)
2. Berardi, S., Damiani, F., de'Liguoro, U. (eds.): Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers, Lecture Notes in Computer Science, vol. 5497. Springer (2009)
3. Bertot, Y., Komendantskaya, E.: Using structural recursion for corecursion. In: Berardi et al. [2], pp. 220–236
4. Boulmé, S.: Specifying in Coq inheritance used in computer algebra. Research report, LIP6 (2000), available on `www.lip6.fr/reports/lip6.2000.013.html`
5. Chlipala, A.: Is Coq being too conservative? Posting to Coq club, `http://logical.saclay.inria.fr/coq-puma/messages/d71fd3954d860d42#msg-285229ea3f28adef`
6. Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) Types for Proofs and Programs, International Conference, TYPES 1993. Lecture Notes in Computer Science, vol. 806, pp. 62–78. Springer (1993)
7. Dams, C.: Is Coq being too conservative? Posting to Coq club, `http://logical.saclay.inria.fr/coq-puma/messages/d71fd3954d860d42#msg-7946fd74eb4de604`
8. Danielsson, N.A.: Beating the productivity checker using embedded languages. In: Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR (2010)
9. Danielsson, N.A., Altenkirch, T.: Subtyping, declaratively. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction (MPC'10). Lecture Notes in Computer Science, vol. 6120, pp. 100–118. Springer (2010)
10. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. 11(5), 467–492 (2001)
11. Giménez, E., Castéran, P.: A tutorial on [co-]inductive types in Coq (2007), `www.labri.fr/perso/casteran/RecTutorial.pdf`
12. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. 14(1), 69–111 (2004)
13. Niqui, M.: Coalgebraic reasoning in Coq: Bisimulation and the lambda-coiteration scheme. In: Berardi et al. [2], pp. 272–288
14. Picard, C., Matthes, R.: Formalization in Coq for this article, `www.irit.fr/~Celia.Picard/Coq/Coind_Graph/`
15. Poernomo, I.: Proofs-as-model-transformations. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) International Conference on Model Transformation, ICMT 2008. Lecture Notes in Computer Science, vol. 5063, pp. 214–228. Springer (2008)
16. The Coq Development Team: The Coq proof assistant reference manual, `http://coq.inria.fr`

Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions

Ulrike Golas, Hartmut Ehrig, and Frank Hermann

Technische Universität Berlin, Germany
{ugolas|ehrig|frank}@cs.tu-berlin.de

Abstract. Triple graph grammars are a successful approach to describe exogenous model transformations. Source and target models are related by some connection part, triple rules describe the simultaneous construction of these parts, and forward and backward rules can be derived modeling the forward and backward model transformations. As shown already for the specification of visual models by typed attributed graph transformation, the expressiveness of the approach can be enhanced significantly by using application conditions, which are known to be equivalent to first order logic on graphs.
We extend triple rules with a specific form of application conditions, which enhance the expressiveness of formal specifications for model transformations. In the main technical results, we show how to extend results concerning information preservation, termination, correctness, and completeness of model transformations to the case with application conditions. We illustrate our approach and results with a model transformation from statecharts to Petri nets.

1 Introduction

Specification of models and model transformations play a central role in model-driven software development. For the specification of visual models and languages, it is common practice to use UML modeling techniques for the concrete syntax with underlying typed attributed graph transformation for the abstract syntax. The visual language can be defined in a declarative way by a meta-model with OCL-constraints or – on the abstract level – by a type graph and suitable graph constraints. Alternatively, the visual language can be generated on the abstract level by typed attributed graph grammars [1]. It is well-known that the expressiveness of such generative approaches can be enhanced by using graph grammar rules with negative application conditions (NACs), or even more by using nested application conditions in the sense of [2], which are known to be equivalent to first order logic on graphs and more expressive than NACs.

For the specification of model transformations, triple graph grammars (TGGs) are a well-established formalism [3], where several extensions of the original TGG definitions have been published in [4, 5, 6], and various kinds of applications have

been presented [7, 8, 9]. Formal properties concerning information preservation, termination, correctness, and completeness of model transformations have been studied already in [10, 11] based on triple rules without NACs, where the decomposition and composition theorem for triple graph transformation sequences in [12] plays a fundamental role. In [13], this theorem has been extended to triple rules with NACs, but not yet to nested application conditions [2].

It is the main aim of this paper to extend the theory of model transformations based on TGGs to rules with general nested application conditions, short application conditions, in order to enhance the expressiveness of model transformations including the generation of the source and target languages by corresponding source and target rules. As a case study, we consider a model transformation from statecharts to Petri nets, where we use a combination of positive and negative application conditions as available in the framework of general application conditions, but not in the more restrictive framework of NACs.

As first main result, we show that the decomposition and composition theorem can be extended to rules with application conditions. This allows to enhance the expressiveness of model transformations and to extend in our second and third main result termination, correctness, completeness, and backward information preservation to this more general framework.

This paper is organized as follows. In Sec. 2, we review triple rules and application conditions. Our case study is presented in Sec. 3 and used as illustrating example in Sec. 4, where we define model transformations based on TGGs with application conditions leading to the three main results. A conclusion including related and future work is presented in Sec. 5.

2 Review of Triple Graph Transformation and Application Conditions

Triple graph grammars [3] are a well known approach for bidirectional model transformations. In [5], the basic concepts of triple graph grammars are formalized in a set-theoretical way, which is generalized and extended in [12] to typed, attributed graphs.

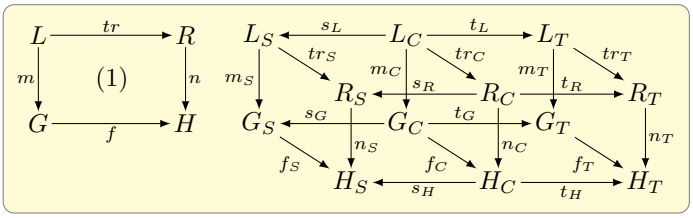
A *triple graph* $G = (G_S \xleftarrow{s_G} G_C \xrightarrow{t_G} G_T)$ consists of graphs G_S, G_C, and G_T, called source, connection, and target component, and two graph morphisms s_G and t_G mapping the connection to the source and target components. A triple graph morphism $f : G_1 \to G_2$ matches the single components and preserves the connection part.

The typing of a triple graph is done in the same way as for standard graphs via a type graph TG - in this case a triple type graph - and a typing morphism $type_G$ from the graph G into this type graph leading to the *typed triple graph* $(G, type_G)$. A typed triple graph morphism $f : (G_1, type_{G_1}) \to (G_2, type_{G_2})$ is a triple graph morphism f such that $type_{G_2} \circ f = type_{G_1}$.

Triple graphs and typed triple graphs, together with the component-wise compositions and identities, form the categories **TripleGraphs** and **TripleGraphs$_{\mathbf{TG}}$**. Moreover, these categories can be extended to weak adhesive HLR

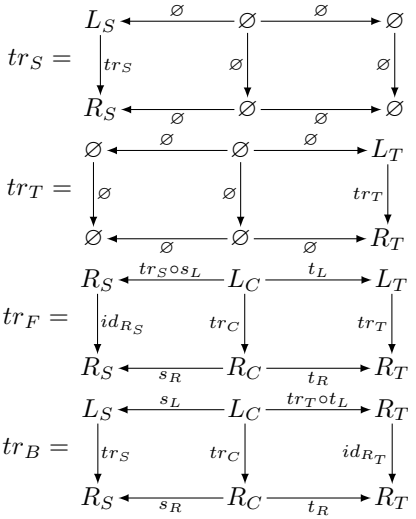
categories [1] with the class $\mathcal{M}$ of injective morphisms which allows us to instantiate the theory to transformations of triple graphs. We consider both triple graphs and typed triple graphs, but do not explicitly mention the typing.

A *triple rule* $tr = (L \overset{tr}{\to} R)$ consists of triple graphs L and R, and an $\mathcal{M}$-morphism $tr : L \to R$. Since triple rules are non-deleting, we do not need a span of morphisms for a rule. A *direct triple transformation* $G \overset{tr,m}{\Longrightarrow} H$ of a triple graph G via a triple rule tr and a match $m : L \to G$ is given by the pushout (1), which is constructed as the component-wise pushouts in the S-, C-, and T-components, where the morphisms s_H and t_H are induced by the pushout of the connection component. Note, that due to the structure of the triple rules, double and single pushout approach are equivalent in this case.

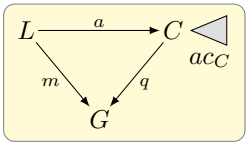


A *triple graph transformation system* $TGS = (TR)$ is based on triple graphs and a set of rules TR over them. A *triple graph grammar* $TGG = (TR, S)$ contains in addition a triple start graph S. For triple graph grammars, the generated language is defined by $VL = \{G \mid \exists \text{ triple transformation } S \overset{*}{\Rightarrow} G \text{ via rules in } TR\}$. Moreover, the source language $VL_S = \{G_S \mid (G_S \overset{s_G}{\leftarrow} G_C \overset{t_G}{\to} G_T) \in VL\}$ contains all standard graphs that are the source component of a derived triple graph. Similarly, the target language $VL_T = \{G_T \mid (G_S \overset{s_G}{\leftarrow} G_C \overset{t_G}{\to} G_T) \in VL\}$ contains all derivable target components.

From a triple rule, we can derive a source rule tr_S and a target rule tr_T, which specify the changes done by this rule in the source and target components, respectively. Moreover, the forward rule tr_F and the backward rule tr_B describe the changes done by the rule to the connection and target resp. source parts, assuming that the source resp. target rules have been applied already. Intuitively, the source rule creates a source model, which can then be transformed by the forward rules into the corresponding target model. This means that the forward rules define the actual model transformation from source to target models. Vice versa, the target rules create the target model, which can then be transformed into a source model applying the backward rules. Thus, the backward rules define the backward model transformation from target to source models.



An important extension is the use of rules with suitable *application conditions* as done in the next sections. These include positive application conditions of the form $\exists a$ for a morphism $a : L \to C$, demanding a certain structure in addition to L, and also negative application conditions $\neg\exists a$, forbidding such a structure. A match $m : L \to G$ satisfies $\exists a$ ($\neg\exists a$) if there is a (no) $\mathcal{M}$-morphism $q : C \to G$ satisfying $q \circ a = m$. In more detail, we use nested application conditions [2], short application conditions, where true is an application conditions, which is always satisfied. For a basic application condition $\exists(a, ac_C)$ on L with an application condition ac_C on C, in addition to the existence of q it is required that q satisfies ac_C.We use $\exists a$ as a short notion for $\exists(a, \text{true})$. In general, we write $m \models \exists(a, ac_C)$ if m satisfies $\exists(a, ac_C)$, and application conditions are closed under boolean operations. Moreover, $ac_C \cong ac'_C$ denotes the semantical equivalence of ac_C and ac'_C on C.



3 Model Transformation from Statecharts to Petri Nets

In this section, we define a model transformation from a variant of UML statecharts [14] to Petri nets using triple rules and application conditions. Statecharts may have orthogonal regions as well as state nesting. As a small restriction, we do not handle entry and exit actions, do not allow extended state variables, allow guards only to be conditions over active states, and allow only a depth of two for hierarchies of states. For the target language of Petri nets, we use nets with inhibitor arcs, contextual arcs, and open places. A transition with an inhibitor arc from a place (denoted by a filled dot instead of an arrow head) is only enabled if there is no token on this place. A contextual arc between a place and a transition (denoted by an edge without arrow heads), also known as read arc in the literature, means that this token is required for firing, but remains on the place. Open places allow the interaction with the environment, i.e. token may appear or disappear without firing a transition within the net. We assume all places to be open. With these restrictions for statecharts and extensions for Petri nets we are able to define a model transformation from statecharts to Petri nets which preserves the semantical behavior, at least on an informal level.

In Fig. 1, the statechart `Prod Cons` is depicted modeling a producer-consumer system. When initialized, the system is in the state `prod`, which has three regions. There, in parallel a producer, a buffer, and a consumer may act. The producer alternates between the states `produced` and `prepare`, where the tran-

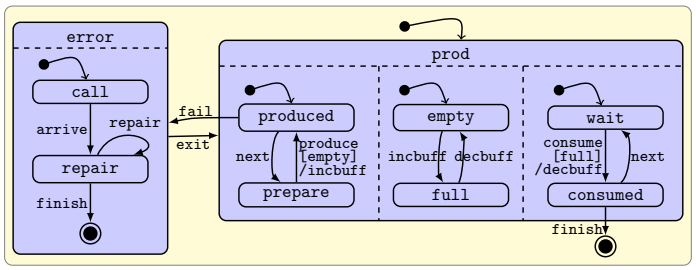


Fig. 1. The example statechart in concrete syntax

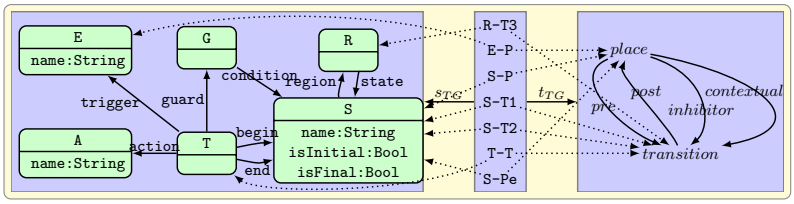


Fig. 2. The triple type graph

sition `produce` models the actual production activity. It is guarded by a condition that the parallel state `empty` is also current, meaning that the buffer is empty and may actually receive a produce, which is then modeled by the action `incbuff` denoted after the `/`-dash. Similarly to the producer, the buffer alternates between the states `empty` and `full`, and the consumer between `wait` and `consumed`. The transition `consume` is again guarded by the state `full` and followed by a `decbuff`-action emptying the buffer.

Two possible events may happen causing a state transition leaving the state `prod`: the consumer may decide to finish the complete run or there may be a failure detected after the production leading to the `error`-state. Then, the machine has to be repaired before the `error`-state can be exited via the corresponding `exit`-transition and the standard behavior in the `prod`-state is executed again.

For the modeling, we use typed attributed graphs, which are an extension of typed graphs by attributes [1]. We do not give details here, but use an intuitive approach to attribution, where the attributes of a node are given in a class diagram-like style. For the values of attributes in the rules we can also use variables. Note, that for the typing of the edges we omit the edge types if they are clear from the node types they are connecting.

In Fig. 2, the triple type graph is depicted, containing in the left the source component of statecharts in abstract syntax, in the right the target component of Petri nets, and the connection component inbetween. To obtain valid statechart models, some constraints are needed which are described in the following but are not shown explicitly.

Each diagram consists of at least one state `S` containing one or many regions `R`, which again contain states. States may be initial (attribute value `isInitial = true`) or final (attribute value `isFinal=true`), each region has to contain exactly one initial and at most one final state, and final states cannot contain regions. A transition `T` begins and ends at a state, is triggered by an event `E`, and may be restricted by a guard `G` and followed by an action `A`. A guard has one or more states as conditions. There is a special event with attribute value `name="exit"` reserved for exiting a state after the completion of all its orthogonal regions, which cannot have a guard condition. Final states cannot be the beginning of a transition.

In the following, we present the triple rules that create simultaneously the statechart model, the connection part, and the corresponding Petri net. In general, each state of the statechart model is connected to a place in the Petri net.

Transitions between states are mapped to Petri net transitions, and fire when the corresponding state transition occurs. Also, events are connected to places, where all events with the same name share the same Petri net place. They are connected via a contextual arc to their corresponding transition thus enabling the simultaneous firing of all enabled Petri net transitions when a token is placed there. By using contextual arcs it is possible that all transitions connected to an event with this name are enabled. Otherwise, we would not be able to fire all these transitions concurrently. They would not be independent but compete for the token. For independence, we had to know in advance how many of these transitions will fire to allocate that number of tokens on the event's place. For a guard, the Petri net transition of its transition in the statechart diagram is the target of a pre and post arc from the place connected to the condition. Thus, we check also in the Petri net that this condition is fulfilled before firing the transition. Each state that may contain regions is connected via `S-T1` to a transition that is the target of pre arcs from all places of final states and inhibitor arcs from all other places in its regions, while the superstate's place is a contextual place. This makes sure that, when all substates are final, these substates are no longer current and, if it exists, the `exit`-action of the superstate can be initiated. Similarly, each substate is connected via `S-T2` to a transition which is the target of a pre arc from its superstate. This makes sure that, when a state transition leaves this superstate, also all substates are no longer current. Each region is connected via `R-T3` to a transition which makes sure that, when no state inside this region is current, also the superstate is deactivated. For the handling of the special `"exit"`-events, each state which may be a superstate is connected via `S-Pe` to a place which handles the proper execution of this event regarding `T1`- and `T3`-transitions.

For the initialization and the semantical steps, all places corresponding to currently active states will be marked. Note, that for the handling of the hierarchical (de)activation the proper open places may fire triggered by the corresponding semantical rules for the statecharts. Thus, the Petri net for itself shows different semantical behavior than the statechart, but every semantical statechart step can be simulated by the Petri net.

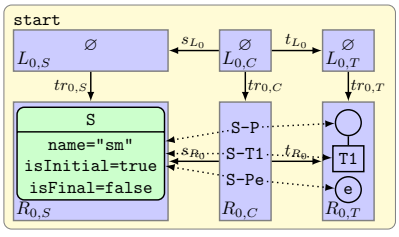


Fig. 3. The rule `start`

The start graph is the empty graph, and the first rule to be applied exactly once is the triple rule `start` shown in Fig. 3, creating the outermost state and its corresponding places and transition. In Fig. 4, the triple rule `newRegion` is depicted which allows to create a new region of a state. Since each region has to have an initial state, this initial state is created and connected to a place. Also the additional `T1`-, `T2`-, and `T3`-transitions are created and connected accordingly. The application condition forbids that the superstate is final or already a substate of another state. Note, that we allow parameters for the rules. Thus, the user has to declare the name of the newly created state when

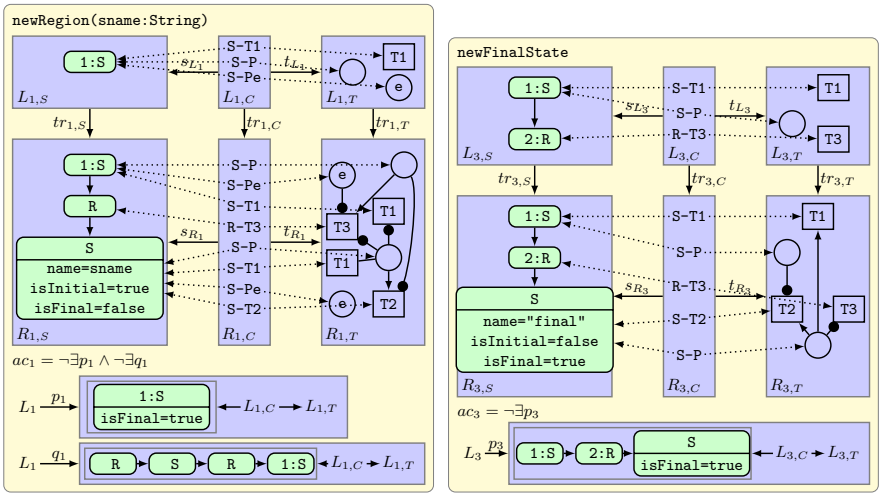


Fig. 4. The rules `newRegion` and `newFinalState`

applying this triple rule. For the creation of other than initial and final states, the triple rule `newState` is used which is very similar to the rule `newRegion` and thus not depicted here. The only difference between both rules is that `newState` already contains the region in the left-hand side which is otherwise created by `newRegion`. Moreover, the application condition is extended by ensuring that there is not already a state with the new name in this region.

In the right of Fig. 4, the triple rule for creating final states is shown. A corresponding place is created in the target component, which is connected to the `T1`-transition of the superstate, inhibits the `T3`-transition of the region, and there is a new transition with the superstate as inhibitor connected by `S-T2`. The application condition of this rule makes sure that only one final state per region is allowed.

For the creation of a new transition, the triple rules `newTransition` in Fig. 5 and `newTransitionOldEvent` (not depicted) are used. A new transition in the source part connected with a new Petri net transition in the target part is created, and in case of

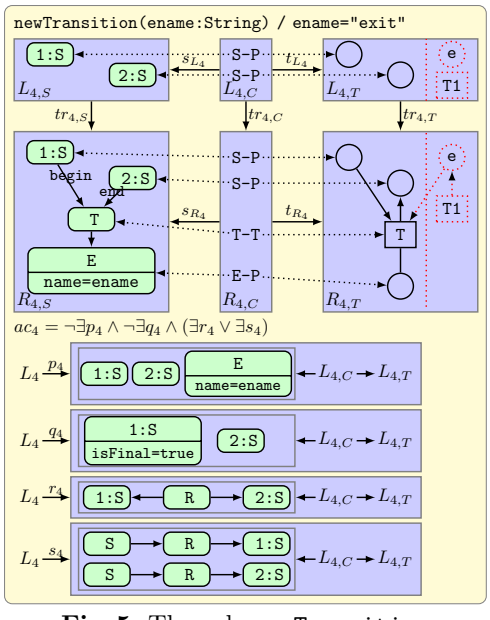


Fig. 5. The rule `newTransition`

a new event, this event is connected with a new place. Moreover, for `exit`-transitions the rules have to be extended handling the `e`-place as depicted by the dotted elements in the target component of Fig. 5. For `newTransitionOldEvent`, the left-hand side already contains an event and the corresponding place such that the transition is connected with this place. The application conditions forbid that the `begin`-state is a final state (q_4), and make sure that either states within the same region (r_4) or in different states (s_4) are connected. Note, that in general we use boolean formulas over application conditions because pure NACs are not powerful enough.

In Fig. 6, the triple rule `newGuard` is shown which creates the guard conditions of a transition. The guard condition is a state, whose corresponding place is connected as pre and post place of the corresponding net transition. The application conditions ensure that only one guard per transition is allowed and that a transition with `exit`-event is not guarded at all. In addition, the rule `nextGuard`, which is not depicted, only adds the edge and the corresponding pre and post arcs for an already existing guard. Moreover, the rule `newAction` (not depicted) adds an action at a transition in the statechart model if none is present, while the Petri is not changed at all.

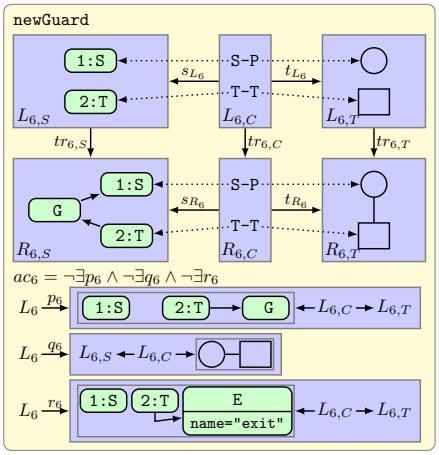


Fig. 6. The rule `newGuard`

The statechart example given in Fig. 1 can be constructed by the application of the following triple rules: $1 \times$ `start`, $5 \times$ `newRegion`, $5 \times$ `newState`, $2 \times$ `newFinalState`, $9 \times$ `newTransition`, $1 \times$ `newTransitionExit`, $2 \times$ `newTransitionOldEvent`, $2 \times$ `newGuard`, $2 \times$ `newAction`. Choosing a proper transformation sequence and the right matches, the result in the source component is our statechart example.

In the target component we find the Petri net depicted in Fig. 7, where we have labeled the places and transitions with the names of the corresponding statechart elements and correspondence node names to ease the recognition. Moreover, we do not show unconnected `T1`-transitions and `e`-places.

The source rules including suitable derived application conditions represent a generating grammar for our statechart models. All models are typed over the type graph and respect the specified constraints. For the target rules, only a subset of Petri nets can be generated, but all models obtained from transformations using the target rules are well-formed, because they are typed over the Petri net type graph and we cannot generate double arcs. This is due to the fact that the rules either create only arcs from or to a new element or the multiple application is forbidden as in the rule `newGuard` as part of the application condition.

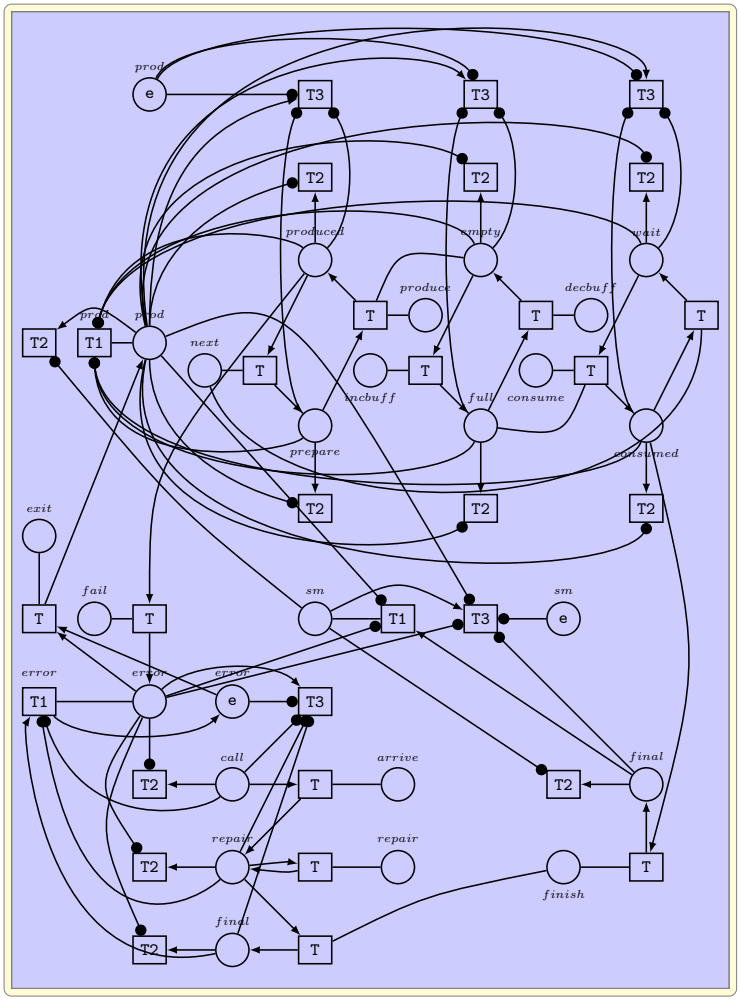


Fig. 7. The corresponding Petri net

4 Model Transformation Based on Triple Graph Transformation with Application Conditions

As shown in the example in Section 3, rules with application conditions are more expressive and allow to restrict the application of the rules. Thus, we enhance triple rules and combine a triple rule tr without application conditions with an application condition ac over L. Then a triple transformation is applicable if the match m satisfies the application condition ac. From now on, a triple rule denotes a rule with application conditions, while the absence of application conditions is explicitly mentioned.

Definition 1 (Triple rule and transformation). *A* triple rule $tr = (tr : L \to R, ac)$ *consists of triple graphs* L *and* R*, an* $\mathcal{M}$*-morphism* $tr : L \to R$*, and an application condition* ac *over* L*.*

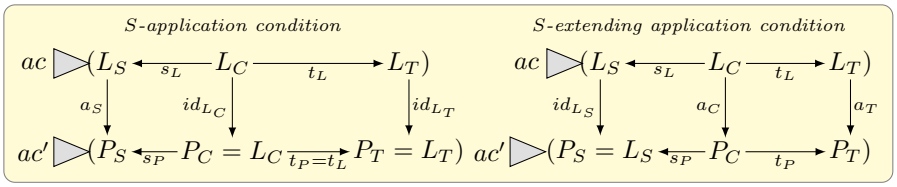
A direct triple transformation $G \overset{tr,m}{\Longrightarrow} H$ *of a triple graph* G *via a triple rule* tr *and a match* $m : L \to G$ *with* $m \models ac$ *is given by the direct triple transformation* $G \overset{\overline{tr},m}{\Longrightarrow} H$ *via the corresponding triple rule without application conditions.*

Example 1. Examples for triple rules using application conditions have been shown in Section 3.

For the extension of the derived rules with application conditions, we need more specialized application conditions that can be assigned to the source and forward rules.

Definition 2 (Special application conditions). *Given a triple rule* $tr : L \to R$*, an application condition* $ac = \exists(a, ac')$ *over* L *with* $a : L \to P$ *is an*

- S-application condition *if* a_C*,* a_T *are identities, i.e.* $P_C = L_C$*,* $P_T = L_T$*, and* ac' *is an* S*-application condition over* P*, and*
- S-extending application condition *if* a_S *is an identity, i.e.* $P_S = L_S$*, and* ac' *is an* S*-extending application condition over* P*.*



Moreover, true *is an* S*- and* S*-extending application condition, and if* ac*,* ac_i *are* S*- or* S*-extending application conditions so are* $\neg ac$*,* $\wedge_{i\in\mathcal{I}} ac_i$*, and* $\vee_{i\in\mathcal{I}} ac_i$*.*

For the assignment of the application condition ac to the derived rules, the application condition has to be consistent to the source and forward rules, which means that we must be able to decompose ac into S- and S-extending application conditions.

Definition 3 (S-consistent application condition). *Given a triple rule* $tr = (tr : L \to R, ac)$*, then* ac *is* S-consistent *if it can be decomposed into* $ac \cong ac'_S \wedge ac'_F$ *such that* ac'_S *is an* S*-application condition and* ac'_F *is an* S*-extending application condition.*

Example 2. All triple rules in Section 3 have S-consistent application conditions. For example, the application condition ac_4 of the rule `newTransition` in Fig. 5 is an S-application condition, thus no decomposition is necessary. Moreover, the application condition ac_6 of the rule `newGuard` in Fig. 6 can be decomposed into the S-application condition $\neg\exists p_6 \wedge \neg\exists r_6$ and the S-extending application condition $\neg\exists q_6$.

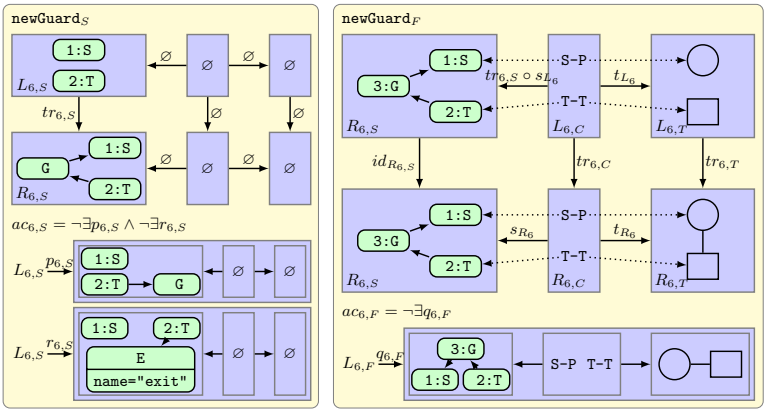


Fig. 8. The source and forward rules of `newGuard`

For an S-consistent application condition, we obtain the application conditions of the source and forward rules from the S- and S-extending parts of the application condition, respectively.

Definition 4 (Derived rules with application conditions). *Given a triple rule $tr = (tr : L \to R, ac)$ with S-consistent $ac \cong ac'_S \wedge ac'_F$ we translate ac'_S to an application condition ac_S on $(L_S \leftarrow \varnothing \to \varnothing)$ using only the source morphisms of ac'_S and similarly ac'_F to an application condition ac_F on $(R_S \leftarrow L_C \to L_T)$ using only the connection and target morphisms of ac'_F. This leads to the* source rule (tr_S, ac_S) *and the* forward rule (tr_F, ac_F).

Example 3. In Fig. 8, the source and forward rules `newGuard`$_S$ and `newGuard`$_F$ of the rule `newGuard` in Fig. 6 are shown. The S-application condition $\neg\exists p_6 \wedge \neg\exists r_6$ is translated to the source rule, where the source graphs of the original application conditions are kept, but the connection and target graphs are empty now. The S-extending application condition $\neg\exists q_6$ is translated to the forward rule, where the source graph is adapted to the new left-hand side.

Now we want to analyze how a triple transformation can be decomposed into a transformation applying first the source rules followed by the forward rules. Match consistency of the decomposed transformation means that the comatches of the source rules define the source part of the matches of the forward rules. This helps us to define suitable forward model transformations, which have to be source consistent to ensure a valid model. Note, that triple transformation sequences always satisfy the application conditions of the corresponding rules.

Definition 5 (Source and match consistency). *Given a sequence $(tr_i)_{i=1,\dots,n}$ of triple rules with S-consistent application conditions leading to corresponding sequences $(tr_{iS})_{i=1,\dots,n}$ and $(tr_{iF})_{i=1,\dots,n}$ of source and forward rules. A triple transformation sequence $G_{00} \overset{tr_S^*}{\Longrightarrow} G_{n0} \overset{tr_F^*}{\Longrightarrow} G_{nn}$ via first $tr_{1S}, \dots, tr_{nS}$*

and then $tr_{1F}, \ldots, tr_{nF}$ *with matches* m_{iS} *and* m_{iF} *and comatches* n_{iS} *and* n_{iF}*, respectively, is* match consistent *if the source component of the match* m_{iF} *is uniquely defined by the comatch* n_{iS}*.*

A triple transformation $G_{n0} \overset{tr_F^*}{\Longrightarrow} G_{nn}$ *is called* source consistent *if there is a match consistent sequence* $G_{00} \overset{tr_S^*}{\Longrightarrow} G_{n0} \overset{tr_F^*}{\Longrightarrow} G_{nn}$*.*

We can split a transformation $G_0 \overset{tr_1}{\Longrightarrow} G_1 \Rightarrow \ldots \overset{tr_n}{\Longrightarrow} G_n$ into transformations $G_0 \overset{tr_{1S}}{\Longrightarrow} G'_0 \overset{tr_{1F}}{\Longrightarrow} G_1 \Rightarrow \ldots \overset{tr_{nS}}{\Longrightarrow} G'_{n-1} \overset{tr_{nF}}{\Longrightarrow} G_n$. But to apply first the source and then the forward rules, these have to be independent in a certain sense. In the following theorem, we show that such a decomposition into a match consistent transformation can be found and, vice versa, each match consistent transformation can be composed to a transformation via the corresponding triple rules if the application conditions are S-consistent. This result is an extension of the corresponding result for triple transformations without application conditions [12] and with negative application conditions [13]. It is essential for concepts and results of model transformations with application conditions below.

Theorem 1 (Decomposition and composition). *For triple transformation sequences with* S*-consistent application conditions the following holds:*

1. ***Decomposition:*** *For each triple transformation sequence* $G_0 \overset{tr_1}{\Longrightarrow} G_1 \Rightarrow \ldots \overset{tr_n}{\Longrightarrow} G_n$ *there is a corresponding match consistent triple transformation sequence* $G_0 = G_{00} \overset{tr_{1S}}{\Longrightarrow} G_{10} \Rightarrow \ldots \overset{tr_{nS}}{\Longrightarrow} G_{n0} \overset{tr_{1F}}{\Longrightarrow} G_{n1} \Rightarrow \ldots \overset{tr_{nF}}{\Longrightarrow} G_{nn} = G_n$*.*
2. ***Composition:*** *For each match consistent triple transformation sequence* $G_{00} \overset{tr_{1S}}{\Longrightarrow} G_{10} \Rightarrow \ldots \overset{tr_{nS}}{\Longrightarrow} G_{n0} \overset{tr_{1F}}{\Longrightarrow} G_{n1} \Rightarrow \ldots \overset{tr_{nF}}{\Longrightarrow} G_{nn}$ *there is a triple transformation sequence* $G_{00} = G_0 \overset{tr_1}{\Longrightarrow} G_1 \Rightarrow \ldots \overset{tr_n}{\Longrightarrow} G_n = G_{nn}$*.*
3. ***Bijective Correspondence:*** *Composition and Decomposition are inverse to each other.*

Proof idea. Similar to [12], the proof is based on the Concurrency Theorem and the Local Church–Rosser Theorem, but now with application conditions as shown in [15]. We use the fact that $tr_i = tr_{iS} *_{E_i} tr_{iF}$ and that the transformations via tr_{iS} and tr_{jF} are sequentially independent for $i > j$, which can be extended to triple rules with application conditions by showing the compatibility of the application conditions due to S-consistency. Thus, the proof from [12] can be done analogously for rules with application conditions. □

Based on source consistent forward transformations we define model transformations, where we assume that the start graph is the empty graph.

Definition 6 (Model transformation). *A* (forward) model transformation sequence $(G_S, G_0 \overset{tr_F^*}{\Longrightarrow} G_n, G_T)$ *is given by a source graph* G_S*, a target graph* G_T*, and a source consistent forward transformation* $G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ *with* $G_0 = (G_S \xleftarrow{\varnothing} \varnothing \xrightarrow{\varnothing} \varnothing)$ *and* $G_{n,T} = G_T$*.*

A (forward) model transformation $MT_F : VL_S \Rrightarrow VL_T$ *is defined by all (forward) model transformation sequences.*

Example 4. As explained for our example transformation in Section 3, applying the corresponding source rule sequence to the empty start graph we obtain our statechart example. This statechart model can be transformed into the Petri net via the forward rules. This triple transformation is source consistent, since the matches of the source parts for the forward rules are uniquely defined by the comatches of the source rules. Thus, we actually obtain a model transformation sequence from the statechart model in Fig. 1 to the Petri net in Fig. 7.

For all notions and results concerning source and forward rules, we obtain the dual notions and results for target and backward rules. Thus, an application condition ac is T-consistent if it can be decomposed into $ac \cong ac'_T \wedge ac'_B$, where ac'_T is a T-application condition with identities a_S, a_C and ac'_B is a T-extending application condition with identity a_T. This leads to the corresponding target and backward rules with application conditions and the dual composition and decomposition properties hold for triple transformation sequences with T-consistent application conditions. Moreover, a backward model transformation sequence $(G_T, G'_0 \overset{tr^*_B}{\Longrightarrow} G'_n, G_S)$ is based on a target consistent backward transformation $G'_0 \overset{tr^*_B}{\Longrightarrow} G'_n$ with $G'_0 = (\varnothing \xleftarrow{\varnothing} \varnothing \xrightarrow{\varnothing} G_T)$ and $G'_{n,S} = G_S$.

4.1 Results for Model Transformations with Application Conditions

Based on Thm. 1 we can show correctness, completeness, backward information preservation, and termination of model transformations. The first result shows that transformations are correct and complete regarding the source and target languages.

Theorem 2 (Correctness and completeness w.r.t. VL_S, VL_T). *Each model transformation sequence $(G_S, G_0 \overset{tr^*_F}{\Longrightarrow} G_n, G_T)$ and $(G_T, G'_0 \overset{tr^*_B}{\Longrightarrow} G'_n, G_S)$ is correct with respect to the source and target languages, i.e. $G_S \in VL_S$ and $G_T \in VL_T$.*

For each $G_S \in VL_S$ there is a corresponding $G_T \in VL_T$ such that there is a model transformation sequence $(G_S, G_0 \overset{tr^*_F}{\Longrightarrow} G_n, G_T)$. Similarly, for each $G_T \in VL_T$ there is a corresponding $G_S \in VL_S$ such that there is a model transformation sequence $(G_T, G'_0 \overset{tr^*_B}{\Longrightarrow} G'_n, G_S)$.

Proof. If $G_0 \overset{tr^*_F}{\Longrightarrow} G_n$ is source consistent we have a match consistent sequence $\varnothing \overset{tr^*_S}{\Longrightarrow} G_0 \overset{tr^*_F}{\Longrightarrow} G_n$ by Def. 5 . By composition in Thm. 1 there is a triple transformation $\varnothing \overset{tr^*}{\Longrightarrow} G_n$ with $G_S = G_{n,S} \in VL_S$ and $G_T \in VL_T$.

For $G_S \in VL_S$ there exists a triple transformation $\varnothing \overset{tr^*}{\Longrightarrow} G$, which can be decomposed by Thm. 1 into a match consistent sequence $\varnothing \overset{tr^*_S}{\Longrightarrow} G_0 = (G_S \xleftarrow{\varnothing} \varnothing \xrightarrow{\varnothing} \varnothing) \overset{tr^*_F}{\Longrightarrow} G$, and by definition $(G_S, G_0 \overset{tr^*_F}{\Longrightarrow} G, G_T)$ is the required model transformation sequence with $G_T \in VL_T$.

Dually, this holds for backward model transformation sequences. □

Example 5. Since our example in Section 3 represents a well-defined model transformation sequence, our statechart and Petri net are correct. Moreover, for each valid statechart model we obtain a correct Petri net model, and vice versa. Note, that for the backward translation this only holds for Petri nets which are correct w.r.t. our target language, and not the language of all well-formed Petri nets.

A forward model transformation from G_S to G_T is backward information preserving concerning the source component if there is a backward transformation sequence from G_T leading to the same source graph G_S.

Definition 7 (Backward information preserving). *A forward transformation sequence $G \overset{tr_F^*}{\Longrightarrow} H$ is* backward information preserving *if for the triple graph $H' = (\varnothing \xleftarrow{\varnothing} \varnothing \xrightarrow{\varnothing} H_T)$ there is a backward transformation sequence $H' \overset{tr_B^*}{\Longrightarrow} G'$ with $G'_S \cong G_S$.*

This theorem is an extension of the corresponding result in [12] to triple transformations with application conditions.

Theorem 3 (Backward information preservation). *If all triple rules are S- and T-consistent, a forward transformation $G \overset{tr_F^*}{\Longrightarrow} H$ is backward information preserving if it is source consistent.*

Proof. If $G \overset{tr_F^*}{\Longrightarrow} H$ is a source consistent sequence then by Def. 5 there exists a match consistent sequence $\varnothing \overset{tr_S^*}{\Longrightarrow} G \overset{tr_F^*}{\Longrightarrow} H$ leading to the triple transformation sequence $\varnothing \overset{tr^*}{\Longrightarrow} H$ using Thm. 1. From the decomposition, we also obtain a match consistent sequence $\varnothing \overset{tr_T^*}{\Longrightarrow} H' \overset{tr_B^*}{\Longrightarrow} H$ using the target and backward rules, with $H'_T = H_T$ and $H'_C = H'_S = \varnothing$. Thus, $G \overset{tr_F^*}{\Longrightarrow} H$ is backward information preserving. □

Example 6. The Petri net in Fig. 7 can be transformed into the statechart in Fig. 1 using the backward rules of our model transformation in the same order as the forward rules were used for the forward transformation. Indeed, this holds for each Petri net obtained of a model transformation sequence from a valid statechart model.

If the source and target rules are creating, forward and backward transformation sequences are terminating, which means that we do not find infinite model transformation sequences. Together with local confluence, this would lead to confluence and functional behavior of model transformations.

Theorem 4 (Termination). *Consider a source model $G_S \in VL_S$ (target model $G_T \in VL_T$) and a set of triple rules such that G_S (G_T) and all rule components are finite on the graph part and the triple rules are creating on the source (target) component. Then each model transformation sequence $(G_S, G_0 \overset{tr_F^*}{\Longrightarrow} G_n, G_T)$ $((G_T, G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n, G_S))$ is terminating, i.e. any extended sequence $G_0 \overset{tr_F^*}{\Longrightarrow} G_n \overset{tr'^+_F}{\Longrightarrow} G_m$ $(G'_0 \overset{tr_B^*}{\Longrightarrow} G'_n \overset{tr'^+_B}{\Longrightarrow} G'_m)$ is not source (target) consistent.*

Proof. Let $G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ be a source consistent forward sequence such that $\varnothing \overset{tr_S^*}{\Longrightarrow} G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ is match consistent, i.e. each comatch $n_{i,S}$ determines the source component of the match $m_{i,F}$. Thus, also each forward match $m_{i,F}$ determines the corresponding comatch $n_{i,S}$. By uniqueness of pushout complements along $\mathcal{M}$-morphisms the comatch $n_{i,S}$ determines the match $m_{i,S}$ of the source step, thus $m_{i,F}$ determines $m_{i,S}$ $(*)$.

If $G_0 \overset{tr_F^*}{\Longrightarrow} G_n \overset{tr_{(n+1,F)},m_{(n+1,F)}}{\Longrightarrow} G_{n+1} \overset{tr''^*_F}{\Longrightarrow} G_m$ is a source consistent forward sequence then there is a corresponding source sequence $\varnothing \overset{tr_S^*}{\Longrightarrow} G' \overset{tr_{n+1,S}}{\Longrightarrow} G'' \overset{tr''^*_S}{\Longrightarrow} G_0$ leading to match consistency of the complete sequence $\varnothing \Rightarrow^* G_m$. Using $(*)$ it follows that $G' \cong G_0$, which implies that we have a transformation step $G_0 \overset{tr_{n+1,S}}{\Longrightarrow} G'' \subseteq G_0$, because triple rules are non-deleting. This is a contradiction to the precondition that each rule is creating on the source component implying that $G' \ncong G_0$. Therefore, the forward transformation sequence $G_0 \overset{tr_F^*}{\Longrightarrow} G_n$ cannot be extended and is terminating.

Dually, this can be shown for backward model transformation sequences. □

Example 7. All triple rules in our example in Section 3 are finite on the graph part and source creating. Thus, all model transformation sequences based on finite statechart models are terminating. Note, that this does not hold for the backward direction, since the rule `newAction` is not target creating. Thus, the corresponding backward rule can be applied infinitary often.

5 Conclusion

In this paper, we have extended the theory of model transformations based on TGGs to rules with nested application conditions [2], which are known to be equivalent to first order logic on graphs. This enhances the expressiveness of model transformations including that of the generation of source and/or target languages. We have discussed in detail a model transformation from statecharts to Petri nets, where the use of application conditions allows to specify and translate more general statecharts then those considered in [1] using an inplace model transformation. We have presented main results for termination, correctness, completeness, and information preservation extending those for the case with NACs in [13] and without NACs in [12].

Our new results are based on the Local Church–Rosser, Parallelism, and Concurrency Theorems with nested application conditions in [15]. As future work it remains to extend also the results concerning functional behaviour in [16] and [17] to the case of rules with nested application conditions based on the "on-the-fly construction" in [11]. This would allow to meet the "Grand Research Challenge of the TGG Community" in [4] for our enhanced framework. Moreover, it is open to show that our model transformation from statecharts to Petri nets is semantically correct, where the semantics of the source and target language could be based on a suitable operational semantics. For statecharts, an operational semantics based on amalgamated graph transformation is presented in [18].

References

[1] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs. Springer (2006)

[2] Habel, A., Pennemann, K.H.: Correctness of High-Level Transformation Systems Relative to Nested Conditions. MSCS **19**(2) (2009) 245–296

[3] Schürr, A.: Specification of Graph Translators With Triple Graph Grammars. In Tinhofer, G., ed.: Proceedings of WG 1994. Volume 903 of LNCS., Springer (1994) 151–163

[4] Schürr, A., Klar, F.: 15 Years of Triple Graph Grammars. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: Proceedings of ICGT 2008. LNCS, Springer (2008) 411–425

[5] König, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. ENTCS **148**(1) (2006) 113–150

[6] Guerra, E., Lara, J.: Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach. Technical Report UC3M-TR-CS-2006-00, Universidad Carlos III, Madrid, Spain (2006)

[7] Taentzer, G., Ehrig, K., Guerra, E., Lara, J., Lengyel, L., Levendovsky, T., Prange, U., Varró, D., Varró-Gyapay, S.: Model Transformation by Graph Transformation: A Comparative Study. In: Proceedings of MTP 2005. (2005)

[8] Guerra, E., Lare, J.: Model View Management with Triple Graph Grammars. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Proceedings of ICGT 2006. Volume 4178 of LNCS., Springer (2006) 351–366

[9] Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application scenarios. Technical Report TR-ri-07-284, University of Paderborn, Germany (2007)

[10] Ehrig, H., Ermel, C., Hermann, F.: On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars. In Karsai, G., Taentzer, G., eds.: Proceedings of GraMoT 2008, ACM (2008)

[11] Ehrig, H., Ermel, C., Hermann, F., Prange, U.: On-the-Fly Construction, Correctness and Completeness of Model Transformations Based on Triple Graph Grammars. In Schürr, A., Selic, B., eds.: Proceedings of MODELS 2009. Volume 5795 of LNCS., Springer (2009) 241–255

[12] Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In Dwyer, M., Lopes, A., eds.: Proceedings of FASE 2007. Volume 4422 of LNCS., Springer (2007) 72–86

[13] Ehrig, H., Hermann, F., Sartorius, C.: Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. ECEASST **18** (2009) 1–18

[14] OMG: Unified Modeling Language, Superstructure, Version 2.2. (2009)

[15] Ehrig, H., Habel, A., Lambers, L.: Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. ECEASST **26** (2010) 1–23

[16] Hermann, F., Ehrig, H., Orejas, F., Golas, U.: Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In: Proceedings of ICGT 2010. Volume 6372 of LNCS., Springer (2010) 155–170

[17] Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. (2010) Submitted to MDI Workshop at MODELS 2010.

[18] Golas, U., Biermann, E., Ehrig, H., Ermel, C.: A Visual Interpreter Semantics for Statecharts Based on Amalgamated Graph Transformation. In: Proceedings of GCM 2010. (2010)

Weakest Liberal Preconditions relative to HR* Graph Conditions

Hendrik Radke
hendrik.radke@informatik.uni-oldenburg.de

Carl v. Ossietzky Universität Oldenburg, Germany*

Abstract. Graph conditions are very important for graph transformation systems and graph programs in a large variety of application areas. With HR* graph conditions, non-local graph properties like "there exists a path of arbitrary length" or "the graph is cycle-free" can be expressed. Together with graph programs, these conditions form a framework for writing programs over graphs, and specifying invariants and properties for these graphs. This paper takes a step towards automating the verification of graph programs with pre- and postconditions. Using Dijkstra's approach, the postcondition is transformed "over the program" to a weakest precondition. The correctness problem is thus reduced to the problem whether or not the precondition implies the weakest precondition, which can be tackled with a theorem prover.

1 Introduction

Formal methods, like the verification of programs with respect to formal system properties, play an important role in the development of trustworthy systems. In our approach, we use graphs to model real-world states and double-pushout graph transformation rules [EEPT06] to describe state changes. Structural properties of the system are described by graph conditions. In [HP09,Pen09], nested graph conditions have been discussed as a formalism to describe structural properties. Nested conditions are expressively equivalent to first-order graph formulas and can express local properties in the sense of Gaifman [Gai82]. In [HR10], HR^+ graph conditions are introduced which are more expressive than monadic second-order graph formulas [Cou97]. These conditions make it possible to express non-local properties like the existence of a path of arbitrary length, the connectedness or circle-freeness of a graph. In this paper, we propose HR^* conditions generalizing HR and HR^+ conditions. Our goal is to check the correctness of graph programs relative to a HR^* pre- and postcondition. Following Dijkstra's approach [Dij76], the correctness of a program relative to pre- and postconditions can be shown by constructing a weakest precondition from the program and the postconditon. A weakest precondition is constructed by first transforming the postcondition into a right HR^* application condition for the program,

* This work is supported by the German Research Foundation (DFG), grants GRK 1076/1 (Graduate School on Trustworthy Software Systems).

then a transformation from the right to a left application condition and finally, from the left application condition to the weakest precondition. This is a generalization of the transformations from [HPR06], where the variables and the corresponding replacement systems have to be regarded. This way, the correctness problem can be reduced to the problem whether the precondition implies the weakest precondition.

Example 1. A small example may illustrate how HR conditions look and how they can be used to form non-local conditions.

$$c_x = \exists(\bullet_1 \xrightarrow{+} \bullet_2), \text{ with } + ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \longrightarrow \bullet \xrightarrow{+} \bullet_2$$

The condition c_x has the meaning "There is a path of arbitrary length from the image of node 1 to the image of node 2". The paths of arbitrary length are represented by the hyperedge labeled +. Replacing the hyperedge according to the replacement rules, we gain a series of graphs without hyperedges $\bullet_1 \longrightarrow \bullet_2$, $\bullet_1 \longrightarrow \bullet \longrightarrow \bullet_2$, $\bullet_1 \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet_2$, ..., i.e. paths of arbitrary length from node 1 to node 2.

The paper is organized as follows. Section 2 introduces HR* conditions and graph transformation rules. In Section 3, basic transformations for HR* conditions are defined. These transformations are used in Section 4 to define the transformation of programs and postconditions into weakest preconditions. The paper is closed with Section 5, where the conclusion is drawn and further work is suggested.

2 HR* conditions

Graphs with variables consist of nodes, edges, and hyperedges. Edges have one source and one target and are labeled by a symbol of an alphabet; hyperedges have an arbitrary sequence of attachment nodes (indicated by *tentacles* between the hyperedge and the attachment node) and are labeled by variables.

Definition 1 (Graphs with variables). *Let* $\mathrm{C} = \langle \mathrm{C_V}, \mathrm{C_E}, \mathrm{X} \rangle$ *be a fixed, finite label alphabet where* X *is a set of variables with a mapping* $\mathrm{rank}\colon \mathrm{X} \to \mathbb{N}_0$ *defining the rank of each variable. A* graph (with variables) *over* C *is a system* $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{Y}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{att}_G, \mathrm{lv}_G, \mathrm{le}_G, \mathrm{ly}_G)$ *consisting of finite sets* V_G, E_G, *and* Y_G *of* nodes *(or* vertices*)*, edges, *and* hyperedges, source *and* target functions $\mathrm{s}_G, \mathrm{t}_G\colon \mathrm{E}_G \to \mathrm{V}_G$, *an* attachment function $\mathrm{att}_G\colon \mathrm{Y}_G \to \mathrm{V}_G^*$[1], *and* labeling functions $\mathrm{lv}_G\colon \mathrm{V}_G \to \mathrm{C_V}$, $\mathrm{le}_G\colon \mathrm{E}_G \to \mathrm{C_E}$, $\mathrm{ly}_G\colon \mathrm{Y}_G \to \mathrm{X}$ *such that, for all* $y \in \mathrm{Y}_G$, $|\mathrm{att}_G(y)| = \mathrm{rank}(\mathrm{ly}_G(y))$. *The set of all graphs with variables we call* $\mathcal{G}_X$, *while* $\mathcal{G}$ *denotes the set of all graphs without variables, i.e. with* $\mathrm{Y}_G = \emptyset$ *for any* $G \in \mathcal{G}$.

[1] This also includes hyperedges with zero tentacles.

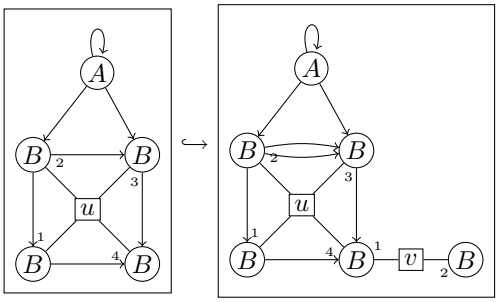
Remark 1. The definition extends the well-known definition of graphs [Ehr79] by the concept of hyperedges in the sense of [Hab92]. Graphs with variables also may be seen as special hypergraphs where the set of hyperedges is divided into a set of edges labelled with terminal symbols and a set of hyperedges labelled by nonterminal symbols.

We extend the definition of graph morphisms to the case of graphs with variables.

Definition 2 (Graph morphisms with variables). *A* (graph) morphism (with variables) $g\colon G \to H$ *consists of functions* $g_{\mathrm{V}}\colon \mathrm{V}_G \to \mathrm{V}_H$, $g_{\mathrm{E}}\colon \mathrm{E}_G \to \mathrm{E}_H$, *and an injective*[2] *function* $g_{\mathrm{Y}}\colon \mathrm{Y}_G \to \mathrm{Y}_H$ *that preserve sources, targets, attachment nodes, and labels, that is,* $\mathrm{s}_H \circ g_{\mathrm{E}} = g_{\mathrm{V}} \circ \mathrm{s}_G$, $\mathrm{t}_H \circ g_{\mathrm{E}} = g_{\mathrm{V}} \circ \mathrm{t}_G$, $\mathrm{att}_H = g_{\mathrm{V}}^* \circ \mathrm{att}_G$, $\mathrm{lv}_H \circ g_{\mathrm{V}} = \mathrm{lv}_G$, $\mathrm{le}_H \circ g_{\mathrm{E}} = \mathrm{le}_G$, *and* $\mathrm{ly}_H \circ g_{\mathrm{Y}} = \mathrm{ly}_G$. *(For a mapping* $g\colon A \to B$, *the free symbolwise extension* $g^*\colon A^* \to B^*$ *is defined by* $g^*(a_1 \dots a_k) = g(a_1) \dots g(a_k)$ *for all* $k \in \mathbb{N}$ *and* $a_i \in A$ $(i = 1, \dots, k)$.*) We call* $\mathrm{Dom}(g) = G$ *the* domain *of* g *and* $\mathrm{Ran}(g) = H$ *the* codomain *of* g. *The term* $g(G)$ *denotes the result of the application of the functions in* g *on graph* G.

A morphism g *is* injective *(*surjective*) if* g_{V}, g_{E}, *and* g_{Y} *are injective (surjective), and an* isomorphism *if it is both injective and surjective. In the latter case* G *and* H *are* isomorphic, *which is denoted by* $G \cong H$. *The* composition $h \circ g$ *of* g *with a graph morphism* $h\colon H \to M$ *consists of the composed functions* $h_{\mathrm{V}} \circ g_{\mathrm{V}}$, $h_{\mathrm{E}} \circ g_{\mathrm{E}}$, *and* $h_{\mathrm{Y}} \circ g_{\mathrm{Y}}$. *For a graph* G, *the* identity $\mathrm{id}_G\colon G \to G$ *consists of the identities* $\mathrm{id}_{G_{\mathrm{V}}}$, $\mathrm{id}_{G_{\mathrm{E}}}$, *and* $\mathrm{id}_{G_{\mathrm{Y}}}$ *on* G_{V}, G_{E}, *and* G_{Y}, *respectively.*

Example 2. Consider graphs G, H over the label alphabet $\mathrm{C} = \langle \{A, B\}, \{\square\}, \mathrm{X} \rangle$ where the symbol $\square$ stands for the invisible edge label and is not drawn and $\mathrm{X} = \{u, v\}$ is a set of variables that have rank 4 and 2, respectively. The graph G contains five nodes with the labels A and B, respectively, seven edges with label $\square$ which is not drawn, and one hyperedge of rank 4 with label u. Additionally, the graph H contains a node, an edge, and a hyperedge of rank 2 with label v.



The drawing of graphs with variables combines the drawing of graphs in [Ehr79] and the drawing of hyperedges in [Hab92,DHK97]: Nodes are drawn by

[2] Injectivity of g_Y ensures that replacement morphisms (defined later) can be composed properly, see also [HR10].

circles carrying the node label inside, edges are drawn by arrows pointing from the source to the target node and the edge label is placed next to the arrow, and hyperedges are drawn as boxes with attachment nodes where the i-th tentacle has its number i written next to it and is attached to the i^{th} attachment node and the label of the hyperedge is inscribed in the box. For visibility reasons, we sometimes write $\bullet \xrightarrow{x} \bullet$ instead of $\bullet {}^{1}\!\!-\!\!\boxed{x}\!\!-\!\!{}_{2}\bullet$. Arbitrary graph morphisms are drawn by the usual arrows "$\rightarrow$"; the use of "$\hookrightarrow$" indicates an injective graph morphism. The actual mapping of elements is conveyed by indices, if necessary.

Hyperedges do not only play a static part as building blocks of graphs with variables, but also a dynamic part as place holders for graphs. While a hyperedge is attached to a sequence of attachment nodes, a graph that should replace it must be equipped with an equally long sequence of nodes. This node sequence controls which attachment point of the hyperedge is fused with which node from the replacing graph.

Definition 3 (Pointed graphs with variables). *A* pointed graph with variables $\langle G, \mathrm{pin}_G \rangle$ *is a graph with variables G together with a sequence* $\mathrm{pin}_G = v_1 \dots v_n$ *of pairwise distinct nodes from G. We write* $\mathrm{rank}(G)$ *for the number n of nodes. For $x \in \mathrm{X}$ with* $\mathrm{rank}(x) = n$, $x^\bullet$ *denotes the pointed graph with the nodes $v_1, \dots, v_n$, one hyperedge attached to $v_1 \dots v_n$, and sequence $v_1 \dots v_n$.*

In [PH96,Pra04], variables are substituted by arbitrary graphs. In this paper, variables are replaced by graphs generated by a hyperedge replacement system.

Definition 4 (HR system). *A* hyperedge replacement (HR) system $\mathcal{R}$ *is a finite set of replacement pairs of the form x/R where x is a variable and R a pointed graph with* $\mathrm{rank}(x) = \mathrm{rank}(R)$.

Given a graph G, the application of the replacement pair x/R to a hyperedge y with label x proceeds in two steps:

1. *Remove the hyperedge y from G, yielding the graph $G - \{y\}$.*
2. *Construct the disjoint union $(G - \{y\}) + R$ and fuse the i^{th} node in* $\mathrm{att}_G(y)$ *with the i^{th} attachment point of R, for $i = 1, \dots, \mathrm{rank}(y)$, yielding the graph H.*

Then G directly derives *H by x/R applied to y, denoted by $G \Rightarrow_{x/R,y} H$ or $G \Rightarrow_{\mathcal{R}} H$ provided $x/R \in \mathcal{R}$. A sequence of direct derivations $G \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} H$ is called a* derivation *from G to H, denoted by $G \Rightarrow^*_{\mathcal{R}} H$. For every variable x, $\mathcal{R}(x) = \{G \in \mathcal{G}_X \mid x^\bullet \Rightarrow^*_{\mathcal{R}} G\}$ denotes the set of all graphs derivable from $x^\bullet$ by $\mathcal{R}$.*

Example 3. The hyperedge replacement system $\mathcal{R}$ with the rules given in Backus-Naur form $+ ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \longrightarrow \bullet \xrightarrow{+} \bullet_2$ generates the set of all directed paths from node 1 to node 2.

In the following, let $\mathcal{R}$ be a fixed HR-system.[3] Hyperedge replacement systems define replacements.

Definition 5 (Replacement). *Let $\mathcal{G}_X^\bullet$ denote the set of all pointed graphs. Let G be a graph and $\mathrm{Y} \subseteq \mathrm{Y}_G$ be a set of hyperedges to be replaced. A mapping* $\mathrm{repl}\colon \mathrm{Y} \to \mathcal{G}_X^\bullet$ *is a* base for replacement *in G if, for all $y \in \mathrm{Y}$,* $\mathrm{repl}(y) \in \mathcal{R}(\mathrm{ly}_G(y))$. $\mathrm{Dom}(\mathrm{repl}) = \mathrm{Y}$ *is the* domain *of* repl. *The replacement of* Y *in G by* repl, *denoted by* $\mathrm{repl}(G)$, *is obtained from G by simultaneously replacing all hyperedges y in Y by* $\mathrm{repl}(y)$. *We write $G \Longrightarrow^{\mathrm{repl}} H$ if $H \cong \mathrm{repl}(G)$.*

Replacement morphisms consist of a base for replacement and a injective graph morphism. For the composition of graph and replacement morphisms, see [HR10].

Definition 6 (Replacement morphisms). *A replacement morphism $\langle \mathrm{repl}, g \rangle$ consists of a base for replacement* repl *in G and an injective graph morphism $G \hookrightarrow^g H'$. It is* injective *if g is injective.*

Notation 1 *In order to discern different types of morphisms in graphs, different types of arrows are used. Replacement morphisms are symbolized by double tips ($\longrightarrow\!\!\!\!\!\rightarrow$) and replacements by double shafts ($\Longrightarrow$).*

We now define HR* conditions, a slight extension of HR$^+$ conditions [HR10]. Instead of having conditions $X \sqsubseteq \boxed{\mathrm{Y}}$, HR* conditions have "splitting" conditions of the form $\exists(\boxed{\mathrm{Y}} \stackrel{\circ}{=} X \oplus \boxed{\mathrm{Y'}}, c)$ that decompose the hyperedge Y into X and a new hyperedge Y'. This allows us to scope in on a part X of the replacement of Y, while at the same time keeping an injective representation of Y.

Definition 7 (union of graphs).

For two graphs with variables G_1, G_2, the union $G_1 \oplus G_2$ is defined as the pushout object G' constructed from $g_1\colon G_0 \to G_1$ and $g_2\colon G_0 \to G_2$, where G_0 constitutes the common parts of G_1 and G_2 (as indicated by indices in G_1 and G_2).

$$\begin{array}{ccc} G_0 & \hookrightarrow & G_1 \\ \downarrow & (PO) & \downarrow \\ G_2 & \hookrightarrow & G' \end{array}$$

Definition 8 (HR* (graph) condition). HR* conditions *are inductively defined as follows. For a graph with variables P,* true *is a condition over P. For every morphism $a\colon P \to C$ and every condition c over C, $\exists(a, c)$ is a condition over P. For every graph with variables C and every condition c over C, $\exists(P \stackrel{\circ}{=} C, c)$ is a condition over P, where $P = P' \oplus Q$, $C = C' \oplus Q$, P' is a graph*

[3] This is to prevent the tedious inclusion of $\mathcal{R}$ in every condition and improve readability.

induced by a set of hyperedges in P, and C' is an arbitrary graph in $\mathcal{G}$. Boolean formulas over conditions over P are conditions over P: For a condition c over P, $\neg c$ is a condition over P, and for an index set J and HR^* *conditions $(c_j)_{j\in J}$ over P, $\wedge_{j\in J} c_j$ is a* HR^* *condition over P. A* HR^* *condition is* finite *if every index set J in this* HR^* *condition is finite.*

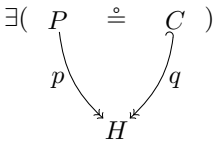
Furthermore, the following abbreviations are used: false abbreviates $\neg$true, $\exists a$ abbreviates $\exists(a, \text{true})$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, $\vee_{j\in J} c_j$ abbreviates $\neg \wedge_{j\in J} \neg c_j$, $\forall(P' \stackrel{\circ}{=} C', c)$ abbreviates $\neg\exists(P' \oplus Q \stackrel{\circ}{=} C' \oplus Q, \neg c)$ (i.e. identical parts can be omitted), and $\forall(P \stackrel{\circ}{=} C, c)$ abbreviates $\neg\exists(P \stackrel{\circ}{=} C, \neg c)$.

Remark 2. In the following, HR^* conditions are shortly called *conditions*. Conditions in the context of graphs are called *constraints*, and conditions in the context of rules are called *application conditions*.

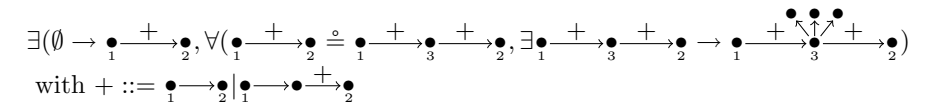
We define the satisfaction of HR^* conditions, as an extension of [Pen09].

Definition 9 (satisfaction of HR^* conditions). *A replacement morphism $p\colon P \twoheadrightarrow H$* satisfies *$\exists(a, c)$ iff there is an injective replacement morphism q such that $q \circ a = p$ and q satisfies c. A replacement morphism p* satisfies *$\exists(P \stackrel{\circ}{=} C, c)$ if there is an injective replacement morphism $q\colon C \hookrightarrow\!\!\!\!\twoheadrightarrow H$ such that $p(P) = q(C)$ and q satisfies c.*
The satisfaction is extended to Boolean formulas over conditions in the usual way, i.e., a replacement morphism p satisfies *true, p* satisfies *$\neg c$ iff p does not satisfy c, and p* satisfies *$\wedge_{i\in I} c_i$ iff p satisfies all c_i $(i \in I)$. A graph G* satisfies *the condition c, if c is a condition over $\emptyset$ and the morphism $\emptyset \to G$ satisfies c. We write $p \models c$ [$G \models c$] to denote that a replacement morphism p [a graph G] satisfies c.*

Remark 3. For non-injective replacement morphisms, conditions of the form $\exists(P \stackrel{\circ}{=} C, c)$ are not satisfied.



Example 4. The following example extends upon the one from the introduction. It shows a HR^* condition expressing "There is a path from the image of node 1 to the image of node 2, and all images of nodes on this path (i.e. that can be reached by splitting the path into two subpaths) have at least three outgoing edges". Note that while the nodes 1 and 2 are common to the left- and right-hand side of the splitting condition, the hyperedges on both sides are not identified.



HR* conditions include HR+ conditions, i.e. for every HR+ condition c, there is an equivalent HR* condition that satisfies the same morphisms. A condition $\overset{\bullet}{\scriptstyle x} \sqsubseteq \boxed{\text{Y}}$ is transformed into a HR* condition $\exists(\boxed{\text{Y}} \overset{\circ}{=} \overset{\bullet}{\scriptstyle x}\!\!-\!\!\boxed{\text{Y'}}, \text{true})$, where the replacement system for Y' is the same as for Y, with the addition that node x can exactly once be identified with a derived node. For HR+ conditions with edges, $\overset{\bullet}{\scriptstyle 1}\xrightarrow{\text{x}}\overset{\bullet}{\scriptstyle 2} \sqsubseteq \boxed{\text{Y}}$ is transformed into $\exists(\boxed{\text{Y}} \overset{\circ}{=} \overset{\bullet}{\scriptstyle 1}\overset{\text{x}}{\rightarrow}\overset{\bullet}{\scriptstyle 2}\!\!-\!\!\boxed{\text{Y'}}, \text{true})$, where the two nodes 1, 2 together with edge x can exactly once be identified with two nodes and a derived edge.

The advantage of HR+ conditions over HR*conditions is that they allow expressions over parts of replacement while retaining a fully injective semantics. In contrast, the semantics for HR* conditions is based on replacement morphisms that are injective up to replacement, i.e. in the replacements for the hyperedges, nodes and edges may be joined. The author deems the fully injective semantics more comfortable and intuitive.

3 Basic transformations of HR* conditions

In this section, we define rules and generalize the basic transformations for nested graph conditions in [HPR06] to HR* graph conditions.

Definition 10 (rules). *A* plain rule $p = \langle L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R \rangle$ *is a pair of injective graph morphisms l, r with common domain K called* interface. *L is called the* left-hand side *and R the* right-hand side. *A* left (right) application condition *is a condition over L (R). A* rule $\rho = \langle p, \text{ac}_L, \text{ac}_R \rangle$ *consists of a plain rule p together with a left and a right application condition* ac_L *and* ac_R*, respectively.*

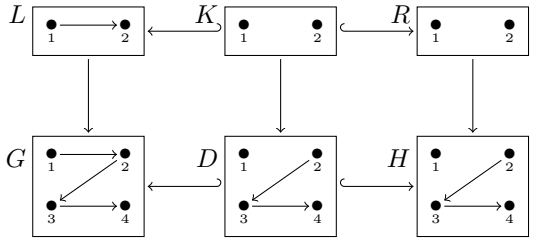
$$\begin{array}{ccccc} L & \hookleftarrow & K & \hookrightarrow & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle m^*} \\ G & \hookleftarrow & D & \hookrightarrow & H \end{array}$$

Given a plain rule p and a morphism $m\colon L \to G$, a direct derivation *consists of two pushouts (1) and (2). We write $G \Rightarrow_{p,m,m^*} H$, $G \Rightarrow_p H$, or short $G \Rightarrow H$. The morphism m is called* match *and m^* is called* comatch. *Given a rule $\rho = \langle p, \text{ac} \rangle$ and a morphism $K \to D$, there is a direct derivation $G \Rightarrow_{p,m,m^*} H$ if $G \Rightarrow_{p,m,m^*} H$, $m \models \text{ac}_L$, and $m^* \models \text{ac}_R$.*

Pushout (1) dictates that the match must satisfy the *dangling condition*. This means that any node to be deleted (i.e. in $G - D$) must not have an edge to a node which is not deleted (i.e. a node in D). Otherwise, the pushout construction (1) would leave D with "dangling" edges which have no source or target node and D would not be a proper graph in $\mathcal{G}$.

Example 5. The (plain) rule $\rho_x = \langle \overset{\bullet}{\scriptstyle 1}\!\!\longrightarrow\!\!\overset{\bullet}{\scriptstyle 2} \hookleftarrow \overset{\bullet}{\scriptstyle 1} \qquad \overset{\bullet}{\scriptstyle 2} \hookrightarrow \overset{\bullet}{\scriptstyle 1} \qquad \overset{\bullet}{\scriptstyle 2} \rangle$ deletes an edge between two nodes 1 and 2. The nodes may be identified, so the rule could also be applied on a single node with a loop and delete the loop.

The application of this rule on a graph is shown below.



Now, we define a transformation that converts a constraint into a right application condition for a given rule.

Lemma 1 (transformation of constraints into application conditions). *For every* HR^* *constraint* c *over* P *and every graph morphism* $m\colon P \to P'$*, there is an application condition* $\mathrm{A}(m,c)$ *such that, for every replacement morphism* $p\colon P' \twoheadrightarrow P''$*,* $p \models \mathrm{A}(m,c) \iff p \circ m \models c$*.*

This transformation is used to transform a postcondition (over $\emptyset$) into a right application condition (over the right-hand side R of the rule).

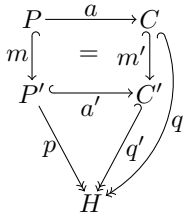
Construction 1 *For a morphism* $m\colon P \to P'$*, transformation* A *is inductively defined as follows:*

$$\begin{array}{ccc} P & \xrightarrow{a} & C \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle m'} \\ P' & \xrightarrow{a'} & C' \end{array} \qquad \mathrm{A}(m, \exists(P \xrightarrow{a} C, c)) = \bigvee_{(a',m') \in \mathcal{F}} \exists\,(a', \mathrm{A}(m', c)),\ \textit{where}\ \mathcal{F} = \{(a',m') \mid \textit{(1) commutes}, m'\ \textit{injective}, (a',m')\ \textit{jointly epimorphic}\}$$

For $\mathrm{A}(m, \exists(P \stackrel{\circ}{=} C, c))$*, where* $P = Q \oplus P_1$*,* $C = Q \oplus C_1$ $\quad \begin{array}{ccccc} P & \hookleftarrow & Q & \hookrightarrow & C \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle m'} \\ P' & \hookleftarrow & Q' & \hookrightarrow & C' \end{array}$
let $\mathrm{A}(m, \exists(P \stackrel{\circ}{=} C, c)) = \exists(\mathrm{Ran}(m) \stackrel{\circ}{=} m'(C), \mathrm{A}(m', c))$ *if the pushout complement* (1) *can be constructed and* (2) *is a pushout, and* false *otherwise.*
For Boolean formulas over conditions and true*, the construction of* A *is straightforward:* $\mathrm{A}(m, \neg c) = \neg \mathrm{A}(m,c)$*,* $\mathrm{A}(m, \wedge_{j \in J} c_j) = \wedge_{j \in J} \mathrm{A}(m, c_j)$*, and* $\mathrm{A}(m, \mathrm{true}) = \mathrm{true}$*.*

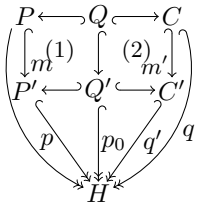
Proof. By structural induction over conditions.
Basis. For $c = \mathrm{true}$, we have $\mathrm{A}(m,c) = \mathrm{true} = c$.
Hypothesis. Assume that $p \models \mathrm{A}(m,c') \Leftrightarrow p \circ m \models c'$ holds for condition c'.
Step. We proceed by case distinction over the structure of c. For the Boolean conditions $c = \neg c', c = c' \wedge c''$, the proof is straightforward. For details, see [Pen09].

Case 1: $c = \exists(a, c')$. Assume that p satisfies $\mathrm{A}(m, c) = \bigvee_{(a',m')\in\mathcal{F}} \exists\,(a', \mathrm{A}(m', c'))$. By definition of $\models$, there is an injective morphism q' such that $p = q' \circ a'$ and $q' \models \mathrm{A}(m', c')$. Let $q = q' \circ m'$. Then we have $p \circ m = q \circ a$ and $q' \circ m' \models c'$. By the induction hypothesis, $q' \circ m' \models c' \Leftrightarrow q \models \mathrm{A}(m', c')$, completing this part of the proof.



Assume that $p \circ m \models \exists(a, c')$. By definition of $\models$, there is an injective morphism q such that $p \circ m = q \circ a$ and $q \models c'$. We construct jointly epimorphic graph morphisms $\langle m', a' \rangle$ such that $m' \circ a = a' \circ m$, and an injective replacement morphism q' with $q' \circ m' = q$. By the induction hypothesis, $q \models c' \Leftrightarrow q' \models \mathrm{A}(m', c')$.

Case 2: $c = \exists(P \mathrel{\mathring{=}} C, c)$. Assume that p satisfies $\mathrm{A}(m, c) = \exists(P' \mathrel{\mathring{=}} C', \mathrm{A}(m', c'))$ where (1) and (2) are pushouts. We let $q = q' \circ m'$, yielding $(p \circ m)(P) = q(C)$ and, by the induction hypothesis, $q' \models \mathrm{A}(m', c') \Leftrightarrow q' \circ m' \models c'$, implying $p \circ m \models \exists(P \mathrel{\mathring{=}} C, c')$.



Assume that $p \circ m \models c$. Since m' is injective, we can construct a $q' \colon C' \to H$ such that $q = q' \circ m'$ and the diagram above commutes, yielding $p(P) = q'(C)$. By the induction hypothesis, $q' \circ m' \models c' \Leftrightarrow q' \models \mathrm{A}(m', c')$, thus $p \models \mathrm{A}(m, c')$. □

Example 6. We now show the application of A on rule ρ_x from example 5 and condition c_x from introductory example 1. The right morphism of ρ_x is $r_x =$  .

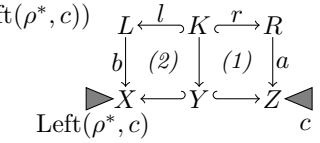
$$\mathrm{A}(r_x, \exists(\bullet_1 \xrightarrow{+} \bullet_2)) \;=\; \exists(\bullet_1 \xrightarrow{+} \bullet_2 \quad \bullet_a \quad \bullet_b) \vee \exists(\bullet_{1=a} \xrightarrow{+} \bullet_{2=b}) \vee \exists(\bullet_{1=b} \xrightarrow{+} \bullet_{2=a}) \vee$$
$$\exists(\bullet_b \quad \bullet_{1=a} \xrightarrow{+} \bullet_2) \vee \exists(\bullet_a \quad \bullet_{1=b} \xrightarrow{+} \bullet_2) \vee \exists(\bullet_1 \xrightarrow{+} \bullet_{2=a} \quad \bullet_b) \vee \exists(\bullet_1 \xrightarrow{+} \bullet_{2=b} \quad \bullet_a)$$

In a next step, we transform the right application condition of a rule ρ into a left one. Basically, this encompasses the reverse application of ρ to the condition itself.

Lemma 2 (from right to left application conditions). *There is a transformation* Left *such that, for every* HR^* *application condition* ac *of a rule* $\rho = \langle L \hookleftarrow K \hookrightarrow R \rangle$ *and for all direct derivations* $G \underset{\rho,m,m^*}{\Longrightarrow} H$, $m \models \mathrm{Left}(\rho, \mathrm{ac}) \Leftrightarrow m^* \models \mathrm{ac}$.

Construction 2 *The transformation* Left *applies the reverse of the rule to each morphism and object in the condition, yielding* false *whenever the dangling condition is not met.*

For a condition $\exists(a,c)$, $\mathrm{Left}(\rho, \exists(a,c)) = \exists(b, \mathrm{Left}(\rho^*, c))$ *if* $\langle r, a\rangle$ *has a pushout complement (1) and* $\rho^* = \langle X \hookleftarrow Y \hookrightarrow Z\rangle$ *is the rule derived by constructing pushout (2). Otherwise,* $\mathrm{Left}(\rho, \exists(a, \mathrm{ac})) = \mathrm{false}$.



For a condition $\exists(R \triangleq C_R, c)$, $\mathrm{Left}(\rho, \exists(R \triangleq C_R, c)) = \exists(L \triangleq C_L, \mathrm{Left}(\rho^*, c))$, *where* $R = R' \oplus P'$, $K = K' \oplus P'$, $L = L' \oplus P'$, *and* $\rho^* = \langle L' \oplus C' \hookleftarrow K' \oplus C' \hookrightarrow R' \oplus C'\rangle$ *is constructed from* ρ *by replacing* P' *with* C'. *Since* P' *contains only hyperedges,* P' *is unchanged by the rule.*

For Boolean formulas over conditions, the construction is straightforward: $\mathrm{Left}(\rho, \mathrm{true}) = \mathrm{true}$, $\mathrm{Left}(\rho, \neg c) = \neg\,\mathrm{Left}(\rho, c)$ *and* $\mathrm{Left}(\rho, \bigwedge_{j\in J} c_j) = \bigwedge_{j\in J} \mathrm{Left}(\rho, c_j)$.

Proof. By induction over the structure of ac.
Basis. For $\mathrm{ac} = \mathrm{true}$, we have $m \models \mathrm{Left}(\rho, \mathrm{true}) = \mathrm{true} \Leftrightarrow \mathrm{true} \Leftrightarrow m^* \models \mathrm{true}$.
Hypothesis. Assume that $m \models \mathrm{Left}(\rho, c) \Leftrightarrow m^* \models c$ holds for application condition c.
Step. We proceed by case distinction over the structure of ac. For Boolean formulas over conditions and $\exists(a,c)$, the proof is only sketched; for details, refer to [HPR06].

Case 1: $\mathrm{ac} = \neg c$. Then $m \models \mathrm{Left}(\rho, \neg c) \Leftrightarrow m \models \neg\,\mathrm{Left}(\rho, c) \Leftrightarrow \neg m \models \mathrm{Left}(\rho, c) \Leftrightarrow \neg m^* \models c \Leftrightarrow m^* \models \neg c$.

Case 2: $\mathrm{ac} = c \wedge c'$. Then $m \models \mathrm{Left}(\rho, c) \wedge c' \Leftrightarrow m \models \mathrm{Left}(\rho, c) \wedge m \models \mathrm{Left}(\rho, c') \Leftrightarrow m^* \models c \wedge m^* \models c' \Leftrightarrow m^* \models c \wedge c'$.

Case 3: $\mathrm{ac} = \exists(a,c)$. Assume that (m,a) has a pushout complement. Then $m \models \mathrm{Left}(\rho, \mathrm{ac}) \Leftrightarrow m \models \exists(b, \mathrm{Left}(\rho^*, c))$. For the derivation, we can decompose the pushouts of the derivation $G \underset{\rho,m,m^*}{\Longrightarrow} H$ such that $m^* = q \circ a$ and $m = q' \circ b$. By the hypothesis, $q' \models \mathrm{Left}(\rho^*, c) \Leftrightarrow q \models c$, thus $m^* \models \exists(a,c)$. If (m,a) has no pushout complement, $m \models \mathrm{Left}(m, \mathrm{ac}) \Leftrightarrow m \models \mathrm{false}$ and there is no pushout such that $m^* \models c$, i.e. $m^* \models \mathrm{false}$. Case 4: $\mathrm{ac} = \exists(R \oplus P' \triangleq R \oplus C', c)$. Assume that $m \models \mathrm{Left}(\rho, \exists(R' \oplus P' \triangleq R' \oplus C', c)$. Then $m \models \exists(L' \oplus P' \triangleq L' \oplus C', \mathrm{Left}(\rho^*, c)) \Leftrightarrow \exists q.m(L' \oplus P') = q(L' \oplus C')$ and $q \models \mathrm{Left}(\rho^*, c)$. By the induction hypothesis, $q \models \mathrm{Left}(\rho^*, c) \Leftrightarrow q^* \models c$, thus $m \models \exists(L \oplus P' \triangleq L \oplus C', \mathrm{Left}(\rho^*, c))$. □

Example 7. We use example 6 to demonstrate the workings of transformation Left.

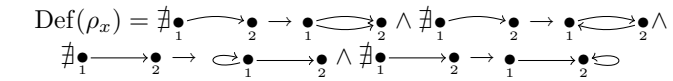
$$\mathrm{Left}(A(r_x, c_x)) = \exists(\bullet_1 \xrightarrow{+} \bullet_2 \quad \bullet_a \to \bullet_b) \vee \exists(\bullet_{1=a} \underset{2=b}{\overset{+}{\rightrightarrows}} \bullet_b) \vee \exists(\bullet_{1=b} \underset{2=a}{\overset{+}{\rightrightarrows}} \bullet_a) \vee$$
$$\exists(\bullet_b \leftarrow \bullet_{1=a} \xrightarrow{+} \bullet_2) \vee \exists(\bullet_a \to \bullet_{1=b} \xrightarrow{+} \bullet_2) \vee \exists(\bullet_1 \xrightarrow{+} \bullet_{2=a} \to \bullet_b) \vee \exists(\bullet_1 \xrightarrow{+} \bullet_{2=b} \leftarrow \bullet_a)$$

Furthermore, we need a transformation that expresses the applicability of a rule.

Lemma 3 (applicability of a rule [HP09]). *There is a transformation* Def *from rules into application conditions such that, for every rule* ρ *and every morphism* $m\colon L \to G$, $m \models \mathrm{Def}(\rho) \Leftrightarrow \exists H.G \underset{\rho,m,m^*}{\Longrightarrow^*} H$.

Construction 3 *For a plain rule* $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$*, let* $\mathrm{Def}(p) = \wedge_{a \in A} \nexists a$*, where* A *is the set of all graph morphisms* $a\colon L \to L'$*, where* L' *is obtained from* L *by adding an edge with both end points in* L*, and for which* $\langle l, a \rangle$ *has no pushout complement. For a rule* $\rho = \langle p, \mathrm{ac} \rangle$ *with application condition, let* $\mathrm{Def}(\rho) = \mathrm{Def}(p) \wedge \mathrm{ac}_L \wedge \mathrm{Left}(\rho, \mathrm{ac}_R)$.

Example 8. For our example rule ρ_x, we have

$\mathrm{Def}(\rho_x) =$ 

Now, a transformation C is defined that transforms left application conditions into constraints over $\emptyset$. As in [HPR06], this is done by ensuring that the left application condition is valid for every morphism $\emptyset \to L$.

Lemma 4 (From application conditions to constraints). *For every application condition* ac *over* L*, there is a condition* $\mathrm{C}(\mathrm{ac})$ *such that for every graph* G*,* $G \models \mathrm{C}(\mathrm{ac}) \Leftrightarrow \forall m\colon L \to G.m \models \mathrm{ac}$.

Construction 4 *Let* $\mathcal{E}(P)$ *denote the set of all epimorphisms with domain* P*. Define* $\mathrm{C}(\mathrm{ac}) := \wedge_{e \in \mathcal{E}(L)} \forall(e \circ i, \mathrm{C}_e(\mathrm{ac}))$*, where* $i : \emptyset \to L$ *is the unique morphism from* $\emptyset$ *to* L*.* $\mathrm{C}_e(\mathrm{ac})$ *is defined over the structure of conditions as* $\mathrm{C}_e(\exists(a, \mathrm{ac})) = \exists(a', \mathrm{ac})$ *if there is a factorization* $a = a' \circ e$*, where* a' *is an injective morphism and* e *an epimorphism, and* false *otherwise, and* $\mathrm{C}_e(\exists(P \stackrel{\circ}{=} C, c)) = \exists(P \stackrel{\circ}{=} C, c)$*. For Boolean conditions,* C_e *is defined the usual way.*

Proof. By induction over the structure of ac. Let $\mathrm{C}_i(\mathrm{ac}) = \wedge_{e \in \mathcal{E}(L)} \forall(e \circ i, \mathrm{ac})$.
Basis. For $\mathrm{ac} = \mathrm{true}$, we have $\mathrm{C}(a, \mathrm{true}) = \mathrm{C}_i(\mathrm{true}) = \mathrm{true} = \mathrm{ac}$.
Hypothesis. Assume that $G \models \mathrm{C}(\mathrm{ac}) \Leftrightarrow \forall m\colon L \to G.m \models \mathrm{ac}$ holds for application condition c.
Step. We proceed by case distinction over the structure of ac.

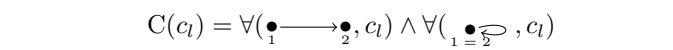
Case 1: $\mathrm{ac} = \neg c$. $G \models \mathrm{C}(\rho, \mathrm{ac}) \Leftrightarrow G \models \mathrm{C}_i(\neg \mathrm{C}_e(c)) = \neg \mathrm{C}_i(\mathrm{C}_e(c)) \Leftrightarrow m \models \mathrm{ac}$.

Case 2: $\mathrm{ac} = c \wedge c'$. Then $G \models \mathrm{C}(c \wedge c') \Leftrightarrow G \models \mathrm{C}_i(\mathrm{C}_e(c) \wedge \mathrm{C}_e(c'))$
$\Leftrightarrow m \models c \wedge c'$.

Case 3: $\mathrm{ac} = \exists(a, c)$. Then $G \models \mathrm{C}(\mathrm{ac}) \Leftrightarrow G \models \mathrm{C}_i(\mathrm{C}_e(\mathrm{ac}))$
$\Leftrightarrow \mathrm{C}_i(\exists a'.a = a' \circ e \wedge G \models \exists(a', c)) \vee \mathrm{false})$
$\Leftrightarrow \mathrm{C}_i(\exists a'.a = a' \circ e \wedge m \models c \vee \mathrm{false}) \Leftrightarrow m \models \mathrm{ac}$.

Case 4: $\mathrm{ac} = \exists(P \stackrel{\circ}{=} C, c)$. $G \models \mathrm{C}(\mathrm{ac}) \Leftrightarrow G \models \mathrm{C}_i(\mathrm{C}_e(\mathrm{ac}))$
$\Leftrightarrow \mathrm{C}_i(G \models \mathrm{ac}) \Leftrightarrow \mathrm{C}_i(m \models c) \Leftrightarrow m \models \mathrm{ac}$. □

Example 9. We continue with the term from example 7 and apply transformation C to it. Let $c_l = \mathrm{Left}(A(\rho_x, c_x))$.

$\mathrm{C}(c_l) =$ 

4 Weakest liberal preconditions of HR* conditions

We now use the transformations defined in the last section to transform conditions over rules and programs. Furthermore, we show that the transformed conditions are weakest liberal preconditions.

Graph programs are defined as in [HP01,HP09].

Definition 11 (Graph program). *Graph programs are inductively defined as follows:*

1. *Every rule is a program.*
2. *Every finite set $\mathcal{S}$ of programs is a program.*
3. *Given programs P and Q, sequential composition $(P;Q)$ and iteration $P\downarrow$ are programs.*

We can now define the semantics of graph programs.

Definition 12 (Semantics of graph programs). *The semantics of a program is a binary relation $[\![P]\!] \subseteq \mathcal{G} \times \mathcal{G}$. For every rule ρ, every set $\mathcal{S}$ of programs, and every pair of programs P and Q, $[\![p]\!] = \{\langle G, H\rangle \,|G \Rightarrow_p H\}, [\![\mathcal{S}]\!] = \bigcup_{P\in\mathcal{S}} [\![P]\!]$ for finite sets $\mathcal{S}$ of programs, $[\![(P;Q)]\!] = [\![Q]\!] \circ [\![P]\!], [\![P\downarrow]\!] = \{\langle G, H\rangle \mid \langle G, H\rangle \in [\![P]\!]^* \wedge \nexists M. \langle H, M\rangle \in [\![P]\!]\}$, where $[\![P]\!]^*$ is the reflexive-transitive closure of $[\![P]\!]$.*

Definition 13 (liberal precondition). *For a program P and a condition d, a condition c is a* liberal precondition relative to *d if for all graphs G satisfying c, $\langle G, H\rangle \in [\![P]\!]$ implies $H \models d$ for all H. A liberal precondition c is a* weakest liberal precondition *of P relative to d, denoted by* $\mathrm{wlp}(P, d)$*, if any precondition of P relative to d implies c. A (weakest) liberal precondition is a* (weakest) precondition *if $G \Rightarrow_\rho H$ for some H and P terminates for G.*

Combining the basic transformations from Section 3, we can now define a transformation of a program and a postcondition into a weakest (liberal) precondition similar to [HPR06].

Theorem 1 (weakest liberal precondition). *There is a transformation* Wlp *such that for every program P and for every condition* post*,* $\mathrm{Wlp}(P, \mathrm{post})$ *is a weakest liberal precondition of P and* post*.*

Construction 5 *For any postcondition d, any rule ρ, any set $\mathcal{S}$ of programs, and any programs P, Q, let*

$$\begin{aligned}
\mathrm{Wlp}(\rho, d) &= C(\mathrm{Def}(p) \Rightarrow \mathrm{Left}(p, A(r, d))) \\
\mathrm{Wlp}(\mathcal{S}, d) &= \bigwedge_{P\in\mathcal{S}} \mathrm{Wlp}(P, d) \\
\mathrm{Wlp}((P;Q), d) &= \mathrm{Wlp}(P, \mathrm{Wlp}(Q, d)) \\
\mathrm{Wlp}(P\downarrow, d) &= \mathrm{Wlp}(d \vee \bigwedge_{i=1}^{\infty} \mathrm{Wlp}(P^i, d), \mathrm{Wlp}(P, \mathrm{false}) \Rightarrow d)
\end{aligned}$$

where P^i is defined inductively as $P^1 = P$ and $P^{i+1} = (P^i; P)$.

Proof. For rules ρ, we show that $\mathrm{Wlp}(P, d)$ is a weakest liberal precondition. For all objects G, we have

$$
\begin{aligned}
& G \models Wlp(p,d) \Leftrightarrow G \models^m \mathrm{C}(\mathrm{Def}(\rho) \Rightarrow \mathrm{Left}(\rho, \mathrm{A}(\rho, d))) && \text{(Def. Wlp)} \\
& \Leftrightarrow \forall L \to G.m \models^m \mathrm{Def}(\rho) \Rightarrow \mathrm{Left}(\rho, \mathrm{A}(\rho, d)) && \text{(Def. C)} \\
& \Leftrightarrow \forall L \to G.m \models^m \mathrm{Def}(\rho) \Rightarrow m \models^{m^*} \mathrm{Left}(\rho, \mathrm{A}(\rho, d)) && \text{(Def. } \models\text{)} \\
& \Leftrightarrow \forall L \to G, R \to H.m \models^m \mathrm{Def}(\rho) \Rightarrow m^* \models^{m^*} \mathrm{A}(\rho, d) && \text{(Def. Left)} \\
& \Leftrightarrow \forall L \to G, R \to H.(G \Rightarrow_{\rho,m,m^*} H) \Rightarrow H \models d && \text{(Def. A, Def)} \\
& \Leftrightarrow \forall H.G, H \in \rho \Rightarrow H \models d && \text{(Application of } \rho\text{)} \\
& \Leftrightarrow G \text{ is a weakest liberal precondition.}
\end{aligned}
$$

For composed graph programs, see the proof in [HPR06].

Similar to [HPR06], a weakest liberal precondition can be transformed into a weakest precondition. It remains to be shown that the additional requirements for weakest preconditions can be met.

5 Conclusion

We have presented a transformation of HR^* postconditions over graph programs to weakest liberal HR^* preconditions. These conditions are an extension of HR and HR^+ conditions of [HR10] which allows the partitioning of graphs. We have shown that a HR^* constraint can be transformed into a right application condition, that a right application condition can be transformed into a left, and that left application conditions, including the implicit dangling condition, can be transformed into a precondition in the form of a HR^* constraint. These transformations are used to construct a weakest liberal precondition for a program respective to a postcondition. Thus the problem whether a program is weakly correct relative to its pre- and postcondition is reduced to the problem whether the weakest liberal precondition implies the precondition.

As further work, the result for weakest liberal preconditions shall be generalized for weakest preconditions. Furthermore, a theorem prover similar to ProCon [Pen09] shall be developed that can check whether a HR^* condition implies another HR^* condition.

References

[Cou97] Bruno Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In *Descriptive Complexity and Finite Models: Proceedings of a DIMACS Workshop, Chapter 2*, 1997.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, pages 95–162. World Scientific, 1997.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer, Berlin, 2006.

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer, 1979.

[Gai82] H. Gaifman. On local and non-local properties. In J. Stern, editor, *Proceedings of the Herbrand symposium: Logic Colloquium'81*, pages 105–135. North Holland Pub. Co., 1982.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *LNCS*. Springer, Berlin, 1992.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *LNCS*, pages 230–245. Springer, 2001.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, pages 1–52, 2009.

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Graph Transformations (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460. Springer, 2006.

[HR10] Annegret Habel and Hendrik Radke. Expressiveness of graph conditions with variables. In *Int. Colloquium on Graph and Model Transformation on the occasion of the 65th birthday of Hartmut Ehrig*, volume 30 of *Electronic Communications of the EASST*, 2010. to appear.

[Pen09] Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.

[PH96] Detlef Plump and Annegret Habel. Graph unification and matching. In *Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 75–89. Springer, 1996.

[Pra04] Ulrike Prange. Graphs with variables as an adhesive HLR category. Private communication, 2004.