GCM 2014

The Fifth International Workshop on Graph Computation Models

Proceedings

York, United Kingdom, 21st July 2014

Editors:
Rachid Echahed, Annegret Habel and Mohamed Mosbah

Contents

Preface

This volume contains the proceedings of the Fifth International Workshop on *Graph Computation Models (GCM 2014*[1]*)*. The workshop took place in York, UK, on 21st July, 2014, as part of ICGT 2014 (the seventh edition of the International Conference on Graph Transformation) and of STAF 2014 (Software Technologies: Applications and Foundations).

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 (GCM 2006), in Leicester, UK (GCM 2008), in Enschede, The Netherlands (GCM 2010) and in Bremen, Germany (GCM 2012).

These proceedings contain the abstracts of two invited talks and eight accepted papers. All submissions were subject to careful refereeing. The topics of accepted papers include theoretical aspects of graph transformation, proof techniques 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 2014, especially the Programme 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 ICGT 2014 and STAF 2014 Organizing Committees for their help in organizing GCM 2014 in York, UK.

July, 2014 Rachid Echahed, Annegret Habel and Mohamed Mosbah
Programme co-chairs of GCM 2014

[1] GCM 2014 web site: http://gcm2014.imag.fr
[2] GCM web site : http://gcm-events.org

Programme committee of GCM 2014

Rachid Echahed	University of Grenoble, France (co-chair)
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

Sandra Alves
Dominique Duval
Paolo Baldan
Nils Erik Flick
Sabine Kuske
Christoph Peuser
Frederic Prost
Hendrik Radke
Caroline von Totth

On the Notion of Graphs in Graph Theory and Graph Transformation

Gabriele Taentzer

Philipps-Universität Marburg, Germany

Graphs are regularly used to model structures as they occur e.g. in chemistry, biology, physics, social sciences, linguistics and especially, in computer science. We consider a variety of graph examples including typical tasks to be performed on these graphs. These tasks are basically concerned with analyzing graph properties as considered in graph theory or changing graphs stepwise as done by graph transformations. We will discuss how graph representations depend on the operations to be performed and hence, how they differ within the areas of graph theory and graph transformation.

We apply domain analysis to analyze variabilities and commonalities of graph concepts and develop a feature model to give an overview on core features of graphs and to understand how they are related to each other. (This work is inspired by the domain analysis of model transformation approaches presented by Czarnecki and Helsen.) For selected feature combinations, i.e. graph variants, suitable graph representations as well as their advantages and disadvantages wrt. selected graph operations are discussed.

Well-Structured Graph Transformation Systems

Barbara König

Universität Duisburg-Essen, Germany

Graph transformation systems (GTSs) can be seen as well-structured transition systems (WSTSs), thus obtaining decidability results for certain classes of GTSs. It was shown that well-structuredness can be obtained using the minor ordering as a well-quasi-order. We extend this idea to obtain a general framework in which several types of GTSs can be seen as (restricted) WSTSs. We instantiate this framework with the subgraph ordering and the induced subgraph ordering. Furthermore we present the tool UNCOVER and discuss runtime results.

Graph Transformation with Symbolic Attributes via Monadic Coalgebra Homomorphisms

Wolfram Kahl

McMaster University, Hamilton, Ontario, Canada, kahl@cas.mcmaster.ca

Abstract. We show how a coalgebraic approach leads to more natural representations of many kinds of graph structures that in the algebraic approach are frequently dealt with using ad-hoc constructions. For the case of symbolically attributed graphs, we demonstrate how using substituting coalgebra homomorphisms in double-pushout rewriting steps yields a powerful and easily understandable transformation mechanism.

Keywords: Transformation of symbolically attributed graphs, attributed graphs as coalgebras, categoric approach to graph transformation

1 Introduction

An attributed graph is a graph where (some of) the items (nodes and edges) carry "attributes", which are taken from some attribute datatypes. Formal treatments of datatypes [EM85, BKL$^+$91, BM04] typically characterise datatypes as *algebras*, or "sets with operations"; they are most frequently implemented as software libraries where the sets are only abstract entities, the operations are executable code, and only the elements of the sets are represented as static data. Graphs, too, can be characterised as algebras, most prominently in the "algebraic approach to graph transformation" [CMR$^+$97, EHK$^+$97, EEPT06]. However, the sets in question are the sets of nodes and edges, and the "operations" are the incidence relations; the whole algebra, understood as a graph, is typically represented as static data.

In attributed graphs, these two conflicting views of algebras come together, and formalisations that consider an attributed graph as a single algebra that includes both graph item sorts and attribute value sorts do not correspond to the way attributed graphs are understood in terms of data organisation. For graph transformation, the theory of the algebraic approach also contributes to the necessity of keeping the graph algebra separate from the attribute value algebra, since pushouts of graph structures, customarily considered as unary algebras [Löw90, CMR$^+$97], can be calculated component-wise, while for typical attribute value algebras, this is not the case. Indeed, most applications have no need to transform the attribute value algebras, since most transformation concepts for attributed graphs expect the transformation results to be attributed over the same attribute datatypes. An exception to this consideration are symbolic

attributes, which can easily be drawn from term algebras over different variable sets during different stages of transformation.

Unary algebras are in fact also co-unary coalgebras, and many kinds of graphs that do not fit the mould of unary algebras can actually naturally be considered as more general coalgebras. This argument was first made in [Kah14]; in current paper we continue that development and show how to use substituting coalgebra homomorphisms for DPO rewriting of symbolically attributed graphs.

After discussing related work in the next section and providing necessary notation in Sect. 3, we explain the basic technicalities for modelling graph structures using coalgebras in Sect. 4. Using the example of edge-labelled and node-attributed graphs, we move to substituting coalgebra homomorphisms in Sect. 5. The resulting category is an instance of the monadic product coalgebra categories introduced in [Kah14]; we summarise definition and basic results in Sect. 6. The resulting pushouts are used in Sect. 7 to obtain direct derivations of attributed graphs. We contrast our approach with the adhesive approach of [EEPT06] in more detail in Sect. 8. Appendix A contains a characterisation of monomorphisms in categories of substitutions.

2 Related Work

Löwe *et al.* [LKW93] appear to have been the first to consider attributed graphs in the context of the algebraic approach to graph transformation; they propose to extend the customary unary graph structure signature with an arbitrary attribute signature, and a set of unary attribution operators connecting the two. These attribution operators typically may have as their source special sorts of attribute carriers, which can be deleted and re-created for relabelling. König and Kozioura [KK08] essentially follow the approach of [LKW93], but choose a rigid organisation of unlabelled nodes, and labelled hyperedges with a single attribute the sort of which is determined by the edge label. Homomorphisms include algebra homomorphisms. In a rule (L, R, α, g), the two rule sides L and R are attributed graphs over the term algebra over a globally fixed set of variables, with L attributed only with variables, and only variables occurring in L may occur in R. The rule morphism is defined by an injective node mapping α; rule morphisms are not defined for edges and attributes, and therefore are a special case of partial morphisms. (The Boolean guard term g controls applicability of the rule.) Matches need to be injective on edges; rewrite steps preserve the data algebra. In the double-pushout approach, [HKT02, EPT04] use attribution edges connecting graph items with attribute values, and are essentially predecessors of [EEPT06], the approach of which is discussed in more detail in Sect. 8. All the above consider arbitrary attribute algebras for the application graphs, with term algebras a special case.

For the "symbolic graphs" of [Ore11, OL10b], the data algebra is not considered an explicit part of the graph structure; instead, a "symbolic graph" is an E-graph over a sorted variable set together with a set of formulae (most typically equations) that may refer to constants drawn from the data algebra.

Since the conventional $\mathcal{M}$-adhesive approach does not cover rule applications that change attributes, [Gol12] presents a variant of adhesive categories that softens the adhesive restrictions to only affect the pushouts that are actually needed during transformation, avoiding spurious non-unifiability problems for attributes. Similarly, Habel and Plump [HP12] restrict the class of morphisms to be used in "vertical" roles in the rewriting steps, to be able to capture the relabelling DPO graph transformations of [HP02, Plu09] which use partially labelled interface graphs. A different approach to relabelling is that of Rebout [RFS08], which combines de-facto-partial attribution relations with a special mechanism for relabelling via "computations" in the left-hand side of the rule.

Rutten's overview article [Rut00] is useful for general theory of coalgebras. Related with our current work is the part of the coalgebra literature that deals with combining algebras and coalgebras; one approach considers separate algebraic and coalgebraic structures in the same carriers, for example Kurz and Hennicker's "Institutions for Modular Coalgebraic Specifications" [KH02]. A further generalisation are "dialgebras" [Hag87, PZ01], which have a single carrier X, and operations $f_i : F_i\, X \to G_i\, X$, where both F_i and G_i are polynomial functors.

3 Notation and Background: Categories and Monads

We assume familiarity with the basics of category theory; for notation, we write "$f : A \to B$" to declare that morphism f goes from object $\mathcal{A}$ to object $\mathcal{B}$, and use ";" as the associative binary *forward composition* operator that maps two morphisms $f : A \to B$ and $g : B \to C$ to $(f ; g) : A \to C$. The identity morphism for object A is written $\mathbb{I}_A$. We assign ";" higher priority than other binary operators, and assign unary operators higher priority than all binary operators.

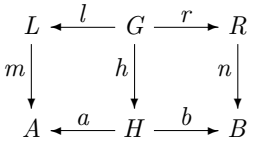
The category of sets and functions is denoted by *Set*.

A *functor* $\mathcal{F}$ from one category to another maps objects to objects and morphisms to morphisms respecting the structure generated by $\to$, $\mathbb{I}$, and composition; we denote functor application by juxtaposition both for objects, $\mathcal{F}\ A$, and for morphisms, $\mathcal{F}\ f$. Although we use forward composition of morphisms, we use backward composition "$\circ$" for functors, with $(\mathcal{G} \circ \mathcal{F})\ A = \mathcal{G}\ (\mathcal{F}\ A)$.

A *monad* on a category $\mathcal{C}$ consists is a functor $\mathcal{M} : \mathcal{C} \to \mathcal{C}$ for which there are two natural transformations ("polymorphic morphisms") $\eta_A : A \to \mathcal{M}\ A$ and $\mu_A : \mathcal{M}\ (\mathcal{M}\ A) \to \mathcal{M}\ A$ satisfying $\eta_{\mathcal{M}\ A} ; \mu_A = \mathbb{I}$ and $\mathcal{M}\ \eta_A ; \mu_A = \mathbb{I}$ and $\mathcal{M}\ \mu_A ; \mu_A = \mu_{\mathcal{M}\ A} ; \mu_A$. Important monads are the List monad, and the term monad $\mathcal{T}_\Sigma$ for any (algebraic) signature Σ. For the former, $\mu_{\mathsf{List},A} : \mathsf{List}\ (\mathsf{List}\ A) \to \mathsf{List}\ A$ is the function that flattens (or concatenates) lists of lists. For the latter, $\mathcal{T}_\Sigma\ V$ is the set of terms with elements of set V used as variables; the function $\mu_{\mathcal{T}_\Sigma, V} : \mathcal{T}_\Sigma\ (\mathcal{T}_\Sigma\ V) \to \mathcal{T}_\Sigma\ V$ maps nested terms (or terms using V-terms as variables) into "flattened" V-terms. Each monad $\mathcal{M}$ on $\mathcal{C}$ induces the so-called *Kleisli category* $\mathbb{K}_\mathcal{M}$ that has the same objects as $\mathcal{C}$, but $\mathcal{C}$-morphisms $A \to \mathcal{M}\ B$ as morphisms from A to B. Kleisli-composition of $f : A \to \mathcal{M}\ B$ with $g : B \to \mathcal{M}\ C$ will be written $f \fatsemi g$; this is defined by $f \fatsemi g = f ; (\mathcal{M}\ g) ; \mu_C$.

In the term monad $\mathcal{T}_\Sigma$, Kleisli morphisms are substitutions $\sigma : V_1 \to \mathcal{T}_\Sigma\ V_2$, and Kleisli composition is just composition of substitutions.

The double-pushout (DPO) approach to high-level rewriting [CMR+97] uses transformation rules that are spans $L \xleftarrow{l} G \xrightarrow{r} R$ in an appropriate category between the **l**eft-hand side L, **g**luing object G, and **r**ight-hand side R. A direct transformation step from object A to object B via such a rule is given by a double pushout diagram, with **h**ost object H, where m is called the **m**atch.



4 The Coalgebra View of Graph Structures

In the context of the algebraic approach to graph transformation, graph structures have traditionally been presented as unary algebras [Löw90, CMR+97]. However, as such they are the intersection between algebras and coalgebras, and in [Kah14], we showed how more general coalgebras are useful in modelling graph features. Recall: Given a (unary) functor F,

- an F-algebra $A = (C_A, f_A)$ is an object C_A together with a morphism $f_A : F\ C_A \to C_A$
- an F-coalgebra $A = (C_A, f_A)$ is an object C_A together with a morphism $f_A : C_A \to F\ C_A$.

Whereas non-unary algebras allow structured types for the arguments of their operations, non-unary coalgebras allow structured types for their results. Also, while in practical algebras, the shape of the arguments can typically be described by a polynomial functor, more general functors are routinely considered for the shape of the results in coalgebras.

In the signatures for such coalgebras, we therefore allow additional syntax for such functors, like List, with fixed interpretation, just like the product functor $\times$ that is used for the argument shapes of non-unary algebras. In general, we assume a language of functor symbols (with arity), and a *signature* introduces first, after "**sorts:**", a list of *sort symbols*, and then, after "**ops:**", a list of *function symbols* (or *operation symbols*), and for each operation symbol, an argument type expression and a result type expression (separated by "$\to$") each built from the functor symbols and the sort symbols.

- An *algebraic signature* has only single sort symbols as *result* types.
- An *coalgebraic signature* has only single sort symbols as *argument* types.

For example, the following is a coalgebraic signature for directed hypergraphs where each hyperedge has a sequence of source nodes and a sequence of target nodes, and each node is labelled with an element of the constant set L:

$$\begin{aligned} \mathsf{sigDHG} := \langle\ & \textbf{sorts: } \mathsf{N}, \mathsf{E} \\ & \textbf{ops: } \mathsf{src} : \mathsf{E} \to \mathsf{List}\ \mathsf{N} \\ & \qquad \mathsf{trg} : \mathsf{E} \to \mathsf{List}\ \mathsf{N} \\ & \qquad \mathsf{nlab} : \mathsf{N} \to L \quad \rangle \end{aligned}$$

The coalgebra functor corresponding to sigDHG is a functor between product categories because of the two sorts:

$$F_{\mathsf{sigDHG}}\ (N\ ,\ E)\ =\ (L\ ,\ ((\mathsf{List}\ N) \times (\mathsf{List}\ N)))$$

Since in algebras, all operations must have a sort as result, modelling labelled graphs as algebras always has to employ the trick of declaring the label sets as additional sorts, and then consider the subcategory that has algebras with a fixed choice for these label sets, and morphisms that map them only with the identity. Similarly, list-valued source and target functions are frequently considered for algebraic graph transformation, but with ad-hoc definitions for morphisms and custom proofs of their properties. In contrast, declaring these features via a coalgebra signature such as sigDHG directly captures the mathematical intent.

5 Attributed Graphs as Coalgebras

The expressive power of coalgebraic signatures extends to attributed graphs without any effort. For example, the following is a coalgebraic signature for edge-labelled (with label set $\mathcal{L}$) and node-attributed graphs, with symbolic attributes taken from the term algebra over some term signature Σ and with variables from the variable carrier set for sort V:

$$\begin{aligned}\mathsf{sigSNAG}_\Sigma := \langle\ &\textbf{sorts: } \mathsf{N}, \mathsf{E}, \mathsf{V}\\ &\textbf{ops: } \mathsf{src} : \mathsf{E} \to \mathsf{N}\\ &\qquad\ \mathsf{trg} : \mathsf{E} \to \mathsf{N}\\ &\qquad\ \mathsf{lab} : \mathsf{E} \to \mathcal{L}\\ &\qquad\ \mathsf{attr} : \mathsf{N} \to \mathcal{T}_\Sigma\ \mathsf{V}\quad \rangle\end{aligned}$$

The resulting homomorphism concept only allows renaming of variables:

Fact 5.1 A $\mathsf{sigSNAG}_\Sigma$-coalgebra homomorphism $F : G_1 \to G_2$ consists of three mappings $F_\mathsf{N} : \mathsf{N}_1 \to \mathsf{N}_2$ and $F_\mathsf{E} : \mathsf{E}_1 \to \mathsf{E}_2$ and $F_\mathsf{V} : \mathsf{V}_1 \to \mathsf{V}_2$ satisfying the following conditions:

$$\begin{aligned} F_\mathsf{E} \fatsemi \mathsf{src}_2 &= \mathsf{src}_1 \fatsemi F_\mathsf{N} \qquad & F_\mathsf{E} \fatsemi \mathsf{lab}_2 &= \mathsf{lab}_1\\ F_\mathsf{E} \fatsemi \mathsf{trg}_2 &= \mathsf{trg}_1 \fatsemi F_\mathsf{N} \qquad & F_\mathsf{N} \fatsemi \mathsf{attr}_2 &= \mathsf{attr}_1 \fatsemi \mathcal{T}_\Sigma\ F_\mathsf{V} \end{aligned}$$ □

DPO rewriting in this category would have to rely on deletion and re-creation of attribute carrying nodes to implement relabelling, similar to [LKW93, KK08]. In addition we also lack the ability to instantiate rules via variable substitution as part of the morphism concept, and might therefore be tempted to add such instantiation outside the DPO rewriting framework, as in [PS04].

The homomorphism concept for $\mathsf{sigSNAG}_\Sigma$-coalgebras can be "fixed" to allow substitution, by also adapting the morphism conditions to take the substituted variables inside the image terms of the attribution function into account:

Definition 5.2 We define the category SNAG_Σ to have $\mathsf{sigSNAG}_\Sigma$-coalgebras as objects, and a morphism $F : G_1 \to G_2$ consists of three mappings typed as shown to the left, satisfying the conditions shown to the right:

$$\begin{array}{l} F_\mathsf{N} : \mathsf{N}_1 \to \mathsf{N}_2 \\ F_\mathsf{E} : \mathsf{E}_1 \to \mathsf{E}_2 \\ F_\mathsf{V} : \mathsf{V}_1 \to \mathcal{T}_\Sigma\ \mathsf{V}_2 \end{array} \qquad \begin{array}{l} F_\mathsf{E} \fatsemi \mathsf{src}_2 = \mathsf{src}_1 \fatsemi F_\mathsf{N} \\ F_\mathsf{E} \fatsemi \mathsf{trg}_2 = \mathsf{trg}_1 \fatsemi F_\mathsf{N} \end{array} \qquad \begin{array}{l} F_\mathsf{E} \fatsemi \mathsf{lab}_2 \ = \mathsf{lab}_1 \\ F_\mathsf{N} \fatsemi \mathsf{attr}_2 = \mathsf{attr}_1 \mathbin{⨟} F_\mathsf{V} \end{array} \qquad \square$$

Note that F_V is now a morphism in the Kleisli category of the term monad $\mathcal{T}_\Sigma$, and accordingly the homomorphism condition for F_V employs Kleisli composition $\mathbin{⨟}$. It is not hard to verify that this category is well-defined — the key to the proof is to recognise that the F_V components are substitutions and compose via Kleisli composition of the term monad.

6 Monadic Product Coalgebras

In [Kah14], we introduced the concept of "monadic product coalgebras" as abstract setting for graph structures with substituting homomorphisms, which distinguishes "graph item sorts" from "variable sorts" by setting the formalisation in the product category $\mathcal{C}_1 \times \mathcal{C}_2$, assuming:

- two categories $\mathcal{C}_1$ and $\mathcal{C}_2$,
- a monad $\mathcal{M}$ on $\mathcal{C}_2$,
- a functor $\mathcal{F}$ from $\mathcal{C}_1 \times \mathcal{C}_2$ to $\mathcal{C}_1$.

In terms of coalgebraic signatures, this implements the restriction that sorts mentioned as monad arguments do not occur as source sorts of operators, and that the monad must not depend on sorts that do occur as source sorts of operators. This restriction is satisfied by all simple kinds of symbolically attributed graphs where the monad is typically a term monad, is applied only to sets of free variables, and these variables do not otherwise participate in the graph structure.

Definition 6.1 ([Kah14]) An *$\mathcal{M}$-$\mathcal{F}$-product-coalgebra* A is a triple $(A_1, A_2, \mathsf{op}_A)$ consisting of
- an object A_1 of $\mathcal{C}_1$, and
- an object A_2 of $\mathcal{C}_2$, and
- a morphism $\mathsf{op}_A : A_1 \to \mathcal{F}\ (A_1,\ \mathcal{M}\ A_2)$

A *$\mathcal{M}$-$\mathcal{F}$-product-coalgebra homomorphism* f from $(A_1, A_2, \mathsf{op}_A)$ to $(B_1, B_2, \mathsf{op}_B)$ is a pair (f_1, f_2) consisting of a $\mathcal{C}_1$-morphism f_1 from A_1 to B_1 and a morphism f_2 from A_2 to B_2 in the Kleisli category of $\mathcal{M}$ such that

$$f_1 \fatsemi \mathsf{op}_B = \mathsf{op}_A \fatsemi \mathcal{F}\ (f_1,\ \mathcal{M}\, f_2 \fatsemi \mu)\ .$$

Morphism composition is composition of the corresponding product category. □

This morphism composition is well-defined, and induces a category. If we let $\mathcal{M}_0$ be the product monad of the identity monad on $\mathcal{C}_1$ and $\mathcal{M}$, then we see that $\mathcal{M}$-$\mathcal{F}$-product-coalgebra homomorphism are in fact morphisms of the Kleisli category $\mathbb{K}_{\mathcal{M}_0}$, and also use its composition. If we further define $\mathcal{F}_0$ as endofunctor on $\mathcal{C}_1 \times \mathcal{C}_2$ by $\mathcal{F}_0(X_1, X_2) = (\mathcal{F}(X_1, X_2), \mathbb{1})$, then an $\mathcal{M}$-$\mathcal{F}$-product-coalgebra is indeed a $(\mathcal{F}_0 \circ \mathcal{M}_0)$-coalgebra. (This factorisation is further explored in [Kah14], and is too general for pushout creation).

Example 6.2 The category SNAG_Σ of Def. 5.2 is equivalent to the category of $\mathcal{T}_\Sigma$-$\mathcal{F}_{\mathsf{sigSNAG}}$-product-coalgebras, where $\mathcal{C}_1 = \mathit{Set} \times \mathit{Set}$ for nodes and edges, $\mathcal{C}_2 = \mathit{Set}$ for variables (or terms), and

$$\mathcal{F}_{\mathsf{sigSNAG}}\ ((N, E), T) = (T, (N \times N \times \mathcal{L}))\ .$$

The four constituents of the result type of $\mathcal{F}_{\mathsf{sigSNAG}}$ correspond to the four operators attr, src, trg, and lab of $\mathsf{sigSNAG}$, with attr being the first constituent, since it is the only operator starting from sort N, while the remaining three all start from sort E.

Since $\mathcal{M}_0$ is a product monad, pushouts in $\mathbb{K}_{\mathcal{M}_0}$ are calculated componentwise, that is, they consist of a pushout in $\mathcal{C}_1$ and a pushout in the Kleisli category of $\mathcal{M}$, and we have:

Theorem 6.3 ([Kah14]) The forgetful functor from the category of $\mathcal{M}$-$\mathcal{F}$-product-coalgebra homomorphisms to the Kleisli category of $\mathcal{M}_0$ creates pushouts. □

More explicitly, if a span $B \xleftarrow{f} A \xrightarrow{g} C$ of $\mathcal{M}$-$\mathcal{F}$-product-coalgebra homomorphisms is given, and also a cospan $(B_1, B_2) \xrightarrow{h} (D_1, D_2) \xleftarrow{k} (C_1, C_2)$ in $\mathbb{K}_{\mathcal{M}_0}$ that is a pushout for the Kleisli morphisms underlying F and G, then (D_1, D_2) can be extended to a $\mathcal{M}$-$\mathcal{F}$-product-coalgebra $D = (D_1, D_2, \mathsf{op}_D)$ such that $B \xrightarrow{h} D \xleftarrow{k} C$ is a pushout for $B \xleftarrow{f} A \xrightarrow{g} C$ in the $\mathcal{M}$-$\mathcal{F}$-product-coalgebra category.

Together with the equivalence of categories of Example 6.2, pushouts for node-attributed graphs essentially reduce to unification for their variable components (due to the fact that *Set* as underlying category has pushouts):

Corollary 6.4 A span $B \xleftarrow{F} A \xrightarrow{G} C$ in the category SNAG_Σ of node-attributed graphs (as $\mathsf{sigSNAG}_\Sigma$ structures) has a pushout if F_V and G_V, as substitutions, have a pushout. □

7 DPO Transformation with Substituting Homomorphisms

Most DPO approaches to attributed graph transformation insist that the "data algebra" supplying the attribute values remains unchanged by transformation. In

contrast, our approach has the data algebra generated by a monad from selected carrier sets — most typically, the data algebra is the term algebra of the variable carrier set. And since variables are just elements of one of the carrier sets, adding and deleting variables is as easy as adding and deleting nodes and edges.

For rewriting of symbolically attributed graphs, we organise the variable set V_G of the gluing graph as a coproduct $V_G = T_G + R_G$ of

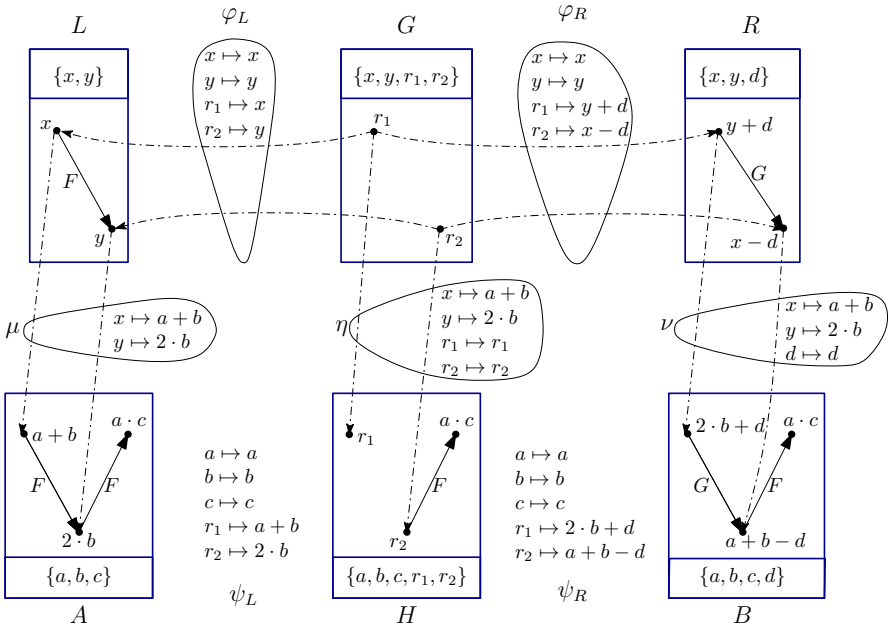
- the set T_G of *transfer variables*, and
- the set R_G of *replacement variables*.

We demand that

- the graph parts (node and edge components) of the rule morphisms are injective,
- the rule morphisms map transfer variables injectively to variables,
- all replacement variables occur in attributes of graph items.

A (rule) morphism satisfying these conditions is called *rigid*.

For human-oriented presentation, and for simplifying the technical arguments below, the transfer-variable parts of the rule morphisms, namely $\varphi_{L,T} : T_G \to \mathcal{T}_\Sigma\ V_L$ and $\varphi_{R,T} : T_G \to \mathcal{T}_\Sigma\ V_R$, will be subset inclusions, with $T_G = V_L \cap V_R$. In the following example DPO drawing, we explicitly list the variable set for each graph, and the variable component (substitution) for each homomorphism (but we do not indicate the obvious node and edge mappings for the application span $A \longleftarrow H \longrightarrow B$).



Here, the transfer variables are x, and y, and the replacement variables are r_1 and r_2; the latter are mapped to different terms on the two rule sides, thus

implementing "re-attribution". Furthermore, variable d is "added by the RHS"; if the host graph H had already contained a variable d, then the d of the RHS would have been mapped to some fresh variable in the result graph B.

Note that φ_L is *not* a monomorphism in the category SNAG_Σ of Def. 5.2: Consider a graph Z with empty node and edge sets and with variable set $\{z\}$, and homomorphisms

- $\lambda_1 : Z \to G$ with $\lambda_{1,V}(z) = x$ and
- $\lambda_2 : Z \to G$ with $\lambda_{2,V}(z) = r_1$,

then $\lambda_1 \fatsemi \varphi_L = \lambda_2 \fatsemi \varphi_L$, but $\lambda_1 \neq \lambda_2$. (The homomorphism φ_R "accidentally" happens to be a monomorphism, but it would not be one if, e.g., $x - d$ were replaced with $x-1$. See Appendix A for more information about monomorphisms in categories of substitutions.)

Although the replacement variables in the example above correspond to undefined labelling in, e.g., [HP12], this is not their only possible use; replacement variables can also occur deeper in the term structure of graph item attributes. This could be used for example to emulate multiple attributes via record-valued single attributes, and then replacing selected attributes could employ gluing nodes with attributes like "$\mathit{pair}(x, r_1)$" or even "$\langle a_1 \mapsto x, a_2 \mapsto r_1, a_3 \mapsto \pi \cdot r_2 \rangle$".

Existence of a pushout complement in the category SNAG_Σ requires, besides the conventional gluing condition for the graph part, the following additional clause for the attribution part:

Definition 7.1 (Variable gluing condition) Each deleted variable (i.e., each variable in $V_L - V_G$) is mapped by the matching μ to a variable in A that does not occur in attributes outside the image of the deleted part of L (which is the "dangling" aspect), and also does not occur in the result of μ_V for any other variable (which is the "identification" aspect). □

The pushout complement is obtained via the following steps:

- The graph part (nodes and edges) is constructed as the pushout complement of the graph part of $G \xrightarrow{\varphi_L} L \xrightarrow{\mu} A$.
- We then calculate a least unifier $\gamma : R_G \to \mathcal{T}_\Sigma\ R_G$ that simultaneously unifies (without instantiating any variables of A) all pairs

$$\big(\ \mu_V\ (\mathsf{attr}_G(n_1)),\quad \mu_V\ (\mathsf{attr}_G(n_2))\ \big)$$

  for different preimages $n_1 \neq n_2 \in N_G$ of nodes identified by μ, that is, with and $\mu_N(\varphi_{L,N}(n_1)) = \mu_N(\varphi_{L,N}(n_2))$. Such a least unifier exists since the matching μ proves unifiability.
- The variable set H_V is the disjoint union of the preserved variables of A, that is, $V_P := V_A - \mu_V(V_L - V_G)$, with the replacement variables of R_G that occur in the range of γ. (Variables that have been unified away must be removed.)

– For the attribution function, we then define (note that γ and μ_V replace disjoint sets of variables):

$$\mathsf{attr}_H(n) = \begin{cases} \gamma\ (\mu_V\ (\mathsf{attr}_G(m))) & \text{if } n = \eta_N(m) \text{ with } m \in N_G \\ \mathsf{attr}_A(\psi_{L,N}(n)) & \text{if } \psi_{L,N}(n) \notin \mu_N(N_L) \end{cases}$$

– The substitution η_V is the identity on replacement variables, and coincides with μ_V on transfer variables.
– The substitution $\psi_{L,V}$ is the identity on preserved variables, and coincides with $\varphi_{L,V} \fatsemi \mu_V$ on replacement variables.

Commutativity $\eta_V \fatsemi \psi_{L,V} = \varphi_{L,V} \fatsemi \mu_V$ is then trivial; the attribute preservation properties

$$\eta_{\mathsf{N}} \mathbin{;} \mathsf{attr}_H = \mathsf{attr}_G \fatsemi \eta_{\mathsf{V}} \qquad \text{and} \qquad \psi_{L,\mathsf{N}} \mathbin{;} \mathsf{attr}_A = \mathsf{attr}_H \fatsemi \psi_{L,\mathsf{V}}$$

are trivial when γ is trivial (that is, when μ_N does not identify any nodes), and in general require careful analysis for the different variable sets.

The variable gluing condition (Def. 7.1) is essential to show the universality property of the cospan $L \xrightarrow{\mu} A \xleftarrow{\psi_L} H$; altogether we obtain:

Theorem 7.2 (Existence and uniqueness of the pushout-complement) For $G \xrightarrow{\varphi_L} L \xrightarrow{\mu} A$ in the category SNAG, if φ_L is a rigid morphism, then a pushout complement $G \xrightarrow{\eta} H \xrightarrow{\psi_L} A$ exists iff the extended gluing condition is satisfied. If a pushout complement exists, it is unique up to isomorphism. □

Since the category SNAG does not have all pushouts, the right-hand side of a rule might contribute additional application conditions. Now we define a SNAG-*transformation rule* to be a span $L \xleftarrow{\varphi_L} G \xrightarrow{\varphi_R} R$ of rigid SNAG-homomorphisms. With that restriction, it is easy to see that right-hand side pushouts always exist at least if the matching μ does not identify any nodes. In the case of node identification via μ, the extended gluing condition is only sufficient for construction and well-definedness of the pushout complement; for the construction of the result graph, the following additional condition is necessary:

– *Attribute identification condition*: There is a unifier $\delta : V_R \to \mathcal{T}_\Sigma\ V_R$ that simultaneously unifies all pairs

$$\big(\ \mu_V\ (\mathsf{attr}_R(\varphi_{R,N}(n_1))),\quad \mu_V\ (\mathsf{attr}_R(\varphi_{R,N}(n_2)))\ \big)$$

for different preimages $n_1 \neq n_2 \in N_G$ of nodes identified by μ, that is, with and $\mu_N(\varphi_{L,N}(n_1)) = \mu_N(\varphi_{L,N}(n_2))$.

Theorem 7.3 (Existence of direct derivation) For $A \xleftarrow{\mu} L \xleftarrow{\varphi_L} G \xrightarrow{\varphi_R} R$ in the category SNAG, if φ_L and φ_R are rigid morphisms and the attribute identification condition is satisfied in addition to the gluing condition for $G \xrightarrow{\varphi_L} L \xrightarrow{\mu} A$, then the usual double-pushout diagram for a direct derivation from A via the rule $L \xleftarrow{\varphi_L} G \xrightarrow{\varphi_R} R$ and the matching μ can be constructed. □

8 Comparison with Attributed Graph Transformation in the Adhesive Approach

In the adhesive HLR approach to attributed graph transformation, presented in detail in [EEPT06, Chapters 8–12], each attributed graph contains its own Σ-algebra for attribute values. Attributes are associated with graph items (nodes or edges) through special "attribute edges" that have a graph item as source and an attribute value as target. This has the advantage that the source of a matching needs to have only those attributes defined that are relevant for the matching, but also has the disadvantage that "attribute names" require separate mechanisms for distinguishing attributes belonging to different names (achieved via "typing", i.e., move to a slice category, in [EEPT06, Def. 8.7]), for enforcing existence (achieved via "constraints" in [EEPT06, Section 12.1]), and for enforcing uniqueness (apparently requiring tuning a global parameter of the "constraint" mechanism, see [EEPT06, Example 12.2]).

For the implementation AGG, [EEPT06, p. 308] mentions that "AGG allows neither graphs which are only partially attributed, nor several values for one type. This restriction is natural, [...]. In the theory, this restriction can be expressed by adding [...] constraints [...]". We agree that "this restriction is natural", and we consider coalgebras a far more natural way to incorporate this restriction into the theory of attributed graphs: Any number of attribution operators can be added to a coalgebraic signature, each of these is then necessarily interpreted (implemented) as a (conceptually separate) total function in all coalgebras for that signature.

The typed attributed graph transformation rules of [EEPT06, Chapter 9] are restricted to a "term algebra with variables" for attributes, and all three graphs of a rule $L \longleftarrow G \longrightarrow R$ share the same term algebra. Therefore, "[the] definition of the match $[L \to A]$ already requires an assignment for all variables" [EEPT06, p. 183], including those that one might otherwise consider as "introduced in the RHS". The fact all horizontal morphisms are restricted to isomorphisms on the value algebra implies that that algebra cannot be modified by transformations. In particular, if the application graph contains a term algebra, it is not possible to add or delete variables.

9 Conclusion and Outlook

The theory of coalgebras, where operations map carrier set elements to arbitrary types constructed via functors from all carrier sets, provides inherent flexibility for modelling of graph structures that is sorely missing from the theory of unary algebras traditionally employed for modelling of graph structures in the "algebraic" approach to graph transformation. In particular, coalgebras can model attributed graph structures effortlessly. In contrast, the non-unary value algebras needed for practical applications of attribution form an alien element in the traditional unary algebras modelling graph structures, and therefore require complex auxiliary constructions to properly capture even the simple fact that

attribution is a total function from (e.g.) nodes to attribute values, as explained in the previous section.

While the traditional approach handles substitution (typically as a special case of evaluation) via algebra homomorphisms, we use the approach of [Kah14] to handle substitution via Kleisli composition, by factoring the coalgebra functor over an appropriate monad.

In the current paper, we restricted our attention to the term monad, and therefore only considered symbolic attribution with terms in more detail; due to the fact that the set of variables is one of the carrier sets of our coalgebras, adding and deleting variables via DPO transformations is essentially as easy as adding and deleting nodes or edges. This facility of adding variables results in a symbolic rewriting system that is closer in spirit to that of [OL10a] than to the point of view of [EEPT06], where additional variables in the RHS need to be instantiated as part of the rule application (and indeed already as part of the matching, for technical reasons).

Even though the Kleisli category of the term monad does not have all pushouts (since not all terms are unifiable), we still managed to obtain a rule concept with an application condition that is an only slightly strengthened gluing condition, and that guarantees that a DPO rewriting step can be constructed. Interestingly, the rule sides, although injective on their graph parts, are *not* monomorphisms in our coalgebra category, so none of the current $\mathcal{M}$-adhesive, $\mathcal{W}$-adhesive [Gol12], or $\mathcal{M},\mathcal{N}$-adhesive [HP12] approaches is directly applicable. The general approach of e.g., [Gol12], should however still be applicable to DPO rewriting in categories of monadic product coalgebras — the concrete instance of a $\mathcal{W}$-adhesive category of attributed graphs presented in [Gol12] (implicitly) uses, for enabling attribute change, a partiality monad, which, like the term monad, is a "guarded monad" [GLDM05]; we conjecture that guarded monads might be used to unify the two approaches.

For future work, we therefore hope to identify an appropriate variant of the adhesive HLR approaches that does not require monomorphisms for the rule sides, and still supports typical HLR results. The results of Sect. 7 should then be generalised to arbitrary $\mathcal{M}$-$\mathcal{F}$-product-coalgebra categories, which will probably require monads with membership relation for the gluing condition.

Besides the DPO-based HLR approaches, we also plan to investigate applying to monadic product coalgebras for example also the sesqui-pushout (SqPO) approach of [CHHK06], which is aplied to attributed graph transformation in [DEPR14]. While the approach of Sect. 7 can only delete variables that are matched to variables, sesqui-pushout rewriting should give us more flexibility there. It may even be advantageous to explore applying the fibred apprach to rewriting [Kah97], as this provies a principled approach to distinguish the different ways substitution and/or partiality are employed in the horizontal versus the vertical arrows of "double-square rewriting".

We also plan to investigate moving from term algebras to more general algebras as target types for attributions.

References

[BKL+91] Bidoit, M., Kreowski, H.-J., Lescanne, P., Orejas, F., Sanella, D. (eds.) Algebraic System Specification and Devalopment — A Survey and Annotated Bibliography, LNCS, vol. 501. Springer (1991)

[BM04] Bidoit, M., Mosses, P. D.: CASL User Manual, LNCS (IFIP Series), vol. 2900. Springer (2004). With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki

[CMR+97] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. In [Roz97], Chapt. 3, pp. 163–245

[CEKR02] Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G. (eds.) ICGT 2002, LNCS, vol. 2505 (2002)

[CHHK06] Corradini, A., Heindel, T., Hermann, F., Knig, B.: Sesqui-Pushout Rewriting. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006, LNCS, vol. 4178, pp. 30–45. Springer Berlin Heidelberg (2006)

[DEPR14] Duval, D., Echahed, R., Prost, F., Ribeiro, L.: Transformation of Attributed Structures with Cloning. In Gnesi, S., Rensink, A. (eds.) FASE 2014, LNCS, vol. 8411, pp. 310–324. Springer Berlin Heidelberg (2014)

[EM85] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Springer (1985)

[EHK+97] Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic Approaches to Graph Transformation, Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In [Roz97], Chapt. 4, pp. 247–312

[EPT04] Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In [PPBE04], pp. 161–177

[EEPT06] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)

[EEKR12] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012, LNCS, vol. 7562. Springer (2012)

[GLDM05] Ghani, N., Lüth, C., De Marchi, F.: Monads of Coalgebras: Rational Terms and Term Graphs. Mathematical Structures in Computer Science 15, 433–451 (2005)

[Gol12] Golas, U.: A General Attribution Concept for Models in $\mathcal{M}$-Adhesive Transformation Systems. In [EEKR12], pp. 187–202

[HP02] Habel, A., Plump, D.: Relabelling in Graph Transformation. In [CEKR02], pp. 135–147

[HP12] Habel, A., Plump, D.: $\mathcal{M},\mathcal{N}$-Adhesive Transformation Systems. In [EEKR12], pp. 218–233

[Hag87] Hagino, T.: A Categorical Programming Language. PhD thesis, Edinburgh University (1987)

[HKT02] Heckel, R., Küster, J. M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In [CEKR02], pp. 161–176

[Kah97] Kahl, W.: A Fibred Approach to Rewriting — How the Duality between Adding and Deleting Cooperates with the Difference between Matching and Rewriting. Technical Report 9702, Fakultät für Informatik, Universität der Bundeswehr München (1997). `http://www.cas.mcmaster.ca/~kahl/Publications/TR/Kahl-1997b.html`

[Kah14] Kahl, W.: Categories of Coalgebras with Monadic Homomorphisms. In Bonsangue, M. (ed.) CMCS 2014, LNCS. Springer (2014). (To appear.) Agda theories available via `http://RelMiCS.McMaster.ca/RATH-Agda/`.

[KK08] König, B., Kozioura, V.: Towards the Verification of Attributed Graph Transformation Systems. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008, LNCS, vol. 5214, pp. 305–320 (2008)

[KH02] Kurz, A., Hennicker, R.: On Institutions for Modular Coalgebraic Specifications. Theoretical Computer Science 280, 69–103 (2002)

[Löw90] Löwe, M.: Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. Technical Report 90/05, TU Berlin (1990)

[LKW93] Löwe, M., Korff, M., Wagner, A.: An Algebraic Framework for the Transformation of Attributed Graphs. In Sleep, M., Plasmeijer, M., van Eekelen, M. (eds.) Term Graph Rewriting: Theory and Practice, pp. 185–199. Wiley (1993)

[OL10a] Orejas, F., Lambers, L.: Delaying Constraint Solving in Symbolic Graph Transformation. In Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010, LNCS, vol. 6372, pp. 43–58 (2010)

[OL10b] Orejas, F., Lambers, L.: Symbolic Attributed Graphs for Attributed Graph Transformation. 30, 2.1–2.25 (2010)

[Ore11] Orejas, F.: Symbolic Graphs for Attributed Graph Constraints. J. Symbolic Comput. 46, 294–315 (2011)

[PPBE04] Parisi-Presicce, F., Bottoni, P., Engels, G. (eds.) ICGT 2004, LNCS, vol. 3256 (2004)

[PS04] Plump, D., Steinert, S.: Towards Graph Programs for Graph Algorithms. In [PPBE04], pp. 128–143

[Plu09] Plump, D.: The Graph Programming Language GP. In: CAI 2009, LNCS, vol. 5725, pp. 99–122 (2009)

[PZ01] Poll, E., Zwanenburg, J.: From Algebras and Coalgebras to Dialgebras. ENTCS 44, 289–307 (2001)

[RFS08] Rebout, M., Féraud, L., Soloviev, S.: A Unified Categorical Approach for Attributed Graph Rewriting. In Hirsch, E., Razborov, A., Semenov, A., Slissenko, A. (eds.) CSR 2008, LNCS, vol. 5010, pp. 398–409. Springer (2008)

[Roz97] Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations. World Scientific, Singapore (1997)

[Rut00] Rutten, J. J.: Universal coalgebra: a theory of systems. Theoretical Computer Science 249, 3–80 (2000)

A Monomorphisms in Substitution Categories

Let $\mathcal{T}_\Sigma$ denote the term functor for signature Σ, that is, $\mathcal{T}_\Sigma\ X$ is the set of Σ-terms with elements of set X as variables. Let $\mathsf{FV}(t)$ denote the set of (free) variables occurring in term t.

$\mathcal{T}_\Sigma$ is an endofunctor on the category Set, and naturally extends to a monad. Substitutions, that is, functions $X \to \mathcal{T}_\Sigma\ Y$, are morphisms in the Kleisli category of the term monad $\mathcal{T}_\Sigma$, and therefore compose via Kleisli composition which is defined for arbitrary $F : X \to \mathcal{T}_\Sigma\ Y$ and $G : Y \to \mathcal{T}_\Sigma\ Z$ as follows, where

$\mu_{\mathcal{T}_\Sigma} : \forall A : \mathcal{T}_\Sigma\ (\mathcal{T}_\Sigma\ A) \to \mathcal{T}_\Sigma\ A$ is the "join" natural transformation of the term monad:

$$F \fatsemi G = F \mathbin{;} \mathcal{T}_\Sigma\ G \mathbin{;} \mu_{\mathcal{T}_\Sigma}$$

Monomorphisms

In any monad, if the "return" natural transformation produces monomorphisms (which it does for $\mathcal{T}_\Sigma$), then monomorphisms in the Kleisli category of this monad are also monomorphisms in the underlying category. Monomorphisms F of the underlying category that are preserved by the monad functor give rise to monomorphisms $F \mathbin{;} \eta$ in the Kleisli category.

The term functor preserves monomorphisms: An injective variable mapping $F : V_1 \to V_2$ gives rise to an injective term mapping $\mathcal{T}_\Sigma\ F : \mathcal{T}_\Sigma\ V_1 \to \mathcal{T}_\Sigma\ V_2$ that only renames variables. The resulting substitution $F \mathbin{;} \eta : V_1 \to \mathcal{T}_\Sigma\ V_2$ is an injective variable renaming, which is easily seen to be a mono also in the category of substitutions.

The category of substitutions has pushouts along such variable renamings; these pushouts implement name-clash-avoiding extension of the domain of substitutions.

Monomorphisms in the category of substitutions cannot map any variables to ground terms. For σ, being a monomorphism in the category of substitutions means application of σ does not unify any two different terms, which is not easy to check directly. However, we can show that monomorphisms in the Kleisli category of the term monad are those substitutions that do not identify variables with different terms (for finite substitutions, this condition directly translates into a decision procedure):

Theorem A.1 A substitution $\sigma : V_1 \to \mathcal{T}_\Sigma\ V_2$ is a monomorphism in the category of substitutions iff for every variable $v : V_1$ and every term $t : \mathcal{T}_\Sigma\ V_1$, we have:

$$\sigma\ v = \sigma\ t \qquad \Rightarrow \qquad v = t\ .$$

Proof. "$\Rightarrow$" follows directly by applying the monomorphism property to the two terms v and t.
"$\Leftarrow$": Assume that σ satisfies the given condition. To show that σ is a monomorphism in the category of substitutions it suffices to show that for any two terms $t, u : \mathcal{T}_\Sigma\ V_1$ with $\sigma\ t = \sigma\ u$, we have $t = u$. Since this is actually equivalent to restricting V_0 to a one-element set, it suffices to show that for all terms $t_1, t_2 : \mathcal{T}_\Sigma\ V_1$ with $\sigma t_1 = \sigma t_2$ we have $t_1 = t_2$.

- If $t = v$ is a variable, then $\sigma\ v = \sigma t = \sigma\ u$, from which the given property yields $v = u$. (The case where u is a variable is analogous.)
- If $t = f(t_1, \ldots t_n)$ and $u = g(u_1, \ldots u_n)$, then $\sigma t = \sigma u$ implies $f = g$ and $\sigma\ t_i = \sigma\ u_i$, from which the induction hypothesis yields $t_i = u_i$ for all i, implying $t = u$. □

An interaction net encoding of Gödel's System $\mathcal{T}$

Ian Mackie and Shinya Sato

Abstract. The graph rewriting system of interaction nets has been very successful for the implementation of the lambda calculus. In this paper we show how the ideas can be extended to encode Gödel's System $\mathcal{T}$—the simply typed λ-calculus extended with numbers. Surprisingly, using some results about System $\mathcal{T}$, we obtain a very simple system of interaction nets that is significantly more efficient for the evaluation of programs.

1 Introduction

Gödel's System $\mathcal{T}$ [7] is the simply typed λ-calculus, with function and product types, extended with natural numbers. It is a very simple system, yet has enormous expressive power—well beyond that of primitive recursive functions.

Interaction nets [9] are a model of computation, based on graph rewriting. They are user defined rewrite systems and because we can write systems which correspond to term rewriting systems we can see them as specification languages. But, because we must also explain all the low-level details (such as copying and erasing) then we can see them as a low-level operational semantics or more specifically, as an implementation language. Supporting this latter point, we remark that in general graph rewriting locating (by graph matching) a reduction step is considered an expensive operation, but in interaction nets there is a very simple mechanism to locate a redex (called an active pair in interaction net terminology), and there is no need to use expensive matching algorithms. There are interesting aspects of interaction nets for parallel evaluation—we will hint at some of these aspects later in the paper.

Over the last years there have been several implementations of the λ-calculus using interaction nets. These include optimal reduction [8], encodings of existing strategies [12, 15], and new strategies [13, 14]. One of the first algorithms to implement Lévy's [11] notion of optimal reduction for the λ-calculus was presented by Lamping [10]. Asperti et al. [3] devised BOHM (Bologna Optimal Higher-Order Machine) building on the ideas of Lamping.

The purpose of this paper is to add to this list of interaction net implementations and to bring together on one hand the successful study of encoding λ-calculus and related systems into interaction nets mentioned above, together with the result that Gödel's System $\mathcal{T}$ can be encoded with the linear λ-calculus and an iterator [2]. Specifically, there are redundancies in System $\mathcal{T}$—copying and erasing can be done either by the iterator or by the λ-calculus. We can remove the copy and erasing power of the λ-calculus, and still keep the expressive power. Taking this further, we can also get primitive recursive functions as a subset of this system. The key motivation for bringing these works together is

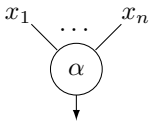
that the linear λ-calculus can be very easily encoded into interaction nets, and therefore there is a hope for a very efficient implementation of this language.

Overview. The rest of this paper is structured as follows. In the next section we recall the basic notations of interaction nets, to fix notation, and also give the definition of linear System $\mathcal{T}$. In Section 3 we give a compilation of the calculus into interaction nets and give the dynamics of the system together with some examples. In Section 4 we discuss the use of this work, and finally we conclude in Section 5.

2 Background

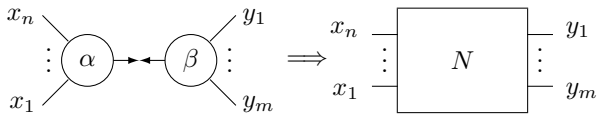
2.1 Interaction nets

In the graphical rewriting system of interaction nets [9], we have a set Σ of *symbols*, which are names of the nodes in our diagrams. Each symbol has an arity ar that determines the number of *auxiliary ports* that the node has. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n + 1$ *ports:* n auxiliary ports and a distinguished one called the *principal port*.



Nodes are drawn variably as circles, triangles or squares. A *net* built on Σ is an undirected graph with nodes at the vertices. The edges of the net connect nodes together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*.

Two nodes $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair*, which is the interaction nets analogue of a redex. A rule $((\alpha, \beta) \Longrightarrow N)$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



The most powerful property of this graph rewriting system is that it is one-step confluent: the order of rewriting is not important, and all sequences of rewrites are of the same length (in fact they are permutations). This has practical consequences: the diagrammatic transformations can be applied in any order, or even in parallel, to give the correct answer.

2.2 System $\mathcal{T}$

In this section we recall the main notions of Gödel's System $\mathcal{T}$. This is a functional calculus, or an applied λ-calculus, with function and product types and natural numbers. Intuitively, we can think of it as a minimal higher-order language that is an extension to the simply typed λ-calculus (allowing numbers to be represented rather than through encodings). From an alternative perspective, it is a language that has greater computational power than primitive recursive functions (we can define Ackermann's function for instance).

We refer the reader to [7] for a detailed description of System $\mathcal{T}$. In [2] it was shown that there are redundancies in this calculus: copying and erasing can be done either in the λ-calculus or using the iterator. This lead to a much simpler presentation using the linear λ-calculus. In this paper we simplify further by introducing pattern matching. There is nothing deep in this step, but it allows us to present the same computational power as System $\mathcal{T}$ in a very simple syntax. In the following we assume familiarity with the λ-calculus [4], and also some basic recursion theory.

Table 1 summarises the syntax of our linear version of System $\mathcal{T}$. The first four lines give the linear λ-calculus with pairs. The construct $\lambda p.t$ is the usual abstraction, extended to allow patterns of variables or pairs of patterns (as defined at the bottom of the table). The remaining three rules define the syntax for constructing numbers and the iteration. We work with terms modulo α-conversion as usual.

The notion of pattern requires a little explanation. When we write $\lambda p.t$, if the pattern p is a variable, say x, then we have the usual abstraction. However, we allow richer patterns built from pairs. It is through these patterns that we are able to access the components of the pairs constructed in the syntax (so we do not need explicit projection functions).

In Figure 1 we give the typing rules for this calculus. We write judgements as $p_1 : A_1, \ldots, p_n : A_n \vdash t : B$. The typing rules capture the linear variable constraints in an alternative way. We remark that we have used the linear notation for types as they are all linear functions.

Our version of linear System $\mathcal{T}$ has a number of useful properties: it is confluent, strongly normalising and reduction preserves types. Reduction also preserves the variable constraints, and reduction is adequate to give normal forms for programs of type nat. The following defines the reduction, and we explain some concepts below.

Definition 1 (Reduction). *The reduction rules for calculus are given below:*

Reduction		*Condition*
$(\lambda p.t)v$	$\longrightarrow [p \ll v].t$	$\mathsf{fv}(v) = \varnothing$
$\mathsf{iter}\ (\mathsf{S}\ t)\ u\ v$	$\longrightarrow \mathsf{iter}\ t\ (vu)\ v$	$\mathsf{fv}(v) = \varnothing$
$\mathsf{iter}\ 0\ u\ v$	$\longrightarrow u$	$\mathsf{fv}(v) = \varnothing$

The construct $[p \ll v].t$ is a matching operation, defined as:

$$[x \ll v].t \longrightarrow t[v/x]$$
$$[\langle p, q\rangle \ll \langle t, u\rangle].t \longrightarrow [p \ll t].[q \ll u].t$$

Terms	**Variable Constraint**	**Free Variables** (fv)
x	$-$	$\{x\}$
tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\lambda p.t$	$\mathsf{bv}(p) \subseteq \mathsf{fv}(t)$	$\mathsf{fv}(t) \setminus \mathsf{bv}(p)$
$\langle p, q\rangle$	$\mathsf{fv}(p) \cap \mathsf{fv}(q) = \varnothing$	$\mathsf{fv}(p) \cup \mathsf{fv}(q)$
0	$-$	$\varnothing$
$\mathsf{S}\ t$	$-$	$\mathsf{fv}(t)$
$\mathsf{iter}\ t\ u\ v$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(v) = \varnothing$ $\mathsf{fv}(t) \cap \mathsf{fv}(v) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(v)$
Pattern	**Variable Constraint**	**Bound Variables** (bv)
x	$-$	$\{x\}$
$\langle p, q\rangle$	$\mathsf{bv}(p) \cap \mathsf{bv}(q) = \varnothing$	$\mathsf{bv}(p) \cup \mathsf{bv}(q)$

Table 1. Terms

Context

$$\frac{}{x : A \vdash x : A}\ (\mathsf{Var}) \qquad \frac{\Gamma, p : A, q : B \vdash t : C}{\Gamma, \langle p, q\rangle : A \otimes B \vdash t : C}\ (\mathsf{Pattern\ Pair})$$

$$\frac{\Gamma, p : A, q : B, \Delta \vdash t : C}{\Gamma, q : B, p : A, \Delta \vdash t : C}\ (\mathsf{Exchange})$$

Logical Rules:

$$\frac{\Gamma, p : A \vdash t : B}{\Gamma \vdash \lambda p.t : A \multimap B}\ (\multimap\mathsf{Intro}) \qquad \frac{\Gamma \vdash t : A \multimap B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}\ (\multimap\mathsf{Elim})$$

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash \langle t, u\rangle : A \otimes B}\ (\mathsf{Pair})$$

Numbers:

$$\frac{}{\Gamma \vdash 0 : \mathsf{nat}}\ (\mathsf{Zero}) \qquad \frac{\Gamma \vdash t : \mathsf{nat}}{\Gamma \vdash \mathsf{S}\ t : \mathsf{nat}}\ (\mathsf{Succ})$$

$$\frac{\Gamma \vdash t : \mathsf{nat} \quad \Delta \vdash u : A \quad \Theta \vdash v : A \multimap A}{\Gamma, \Delta, \Theta \vdash \mathsf{iter}\ t\ u\ v : A}\ (\mathsf{Iter})$$

Fig. 1. Linear System $\mathcal{T}$

Substitution is a meta-operation defined as usual, and reductions can take place in any context. Matching forces evaluation of terms, and will always succeed. The conditions on the rules are used to preserve the linearity of them terms.

The matching operation is inspired by that of the ρ-calculus [5]. $\lambda p.t$ is a *generalized abstraction*–it can be seen as a λ-abstraction on a pattern p instead of a single variable. $[p \ll u].t$ is a *matching constraint* denoting a matching problem $p \ll u$ whose solutions will be applied to t.

2.3 Examples

Here we give a few examples to illustrate how to use the syntax and what programs look like.

- Addition, multiplication and exponentiation can be defined as:

$$\begin{aligned}\mathsf{add} &= \lambda mn.\mathsf{iter}\ m\ n\ (\lambda x.\mathsf{S}x)\\ \mathsf{mult} &= \lambda mn.\mathsf{iter}\ m\ 0\ (\mathsf{add}\ n)\\ \mathsf{exp} &= \lambda mn.\mathsf{iter}\ n\ (\mathsf{S}\ 0)\ (\mathsf{mult}\ m)\end{aligned}$$

  Note in particular that each function satisfies the linearity constraints.
- When we need to copy of erase, we can do that as shown in the following examples for numbers:

$$\begin{aligned}C &= \lambda x.\mathsf{iter}\ x\ \langle 0,0\rangle\ (\lambda\langle a,b\rangle.\langle \mathsf{S}a, \mathsf{S}b\rangle)\\ \mathsf{fst} &= \lambda\langle n,m\rangle.\mathsf{iter}\ m\ n\ (\lambda x.x)\end{aligned}$$

  Where $C : \mathsf{nat} \multimap \mathsf{nat} \otimes \mathsf{nat}$, and $\mathsf{fst} : \mathsf{nat} \otimes \mathsf{nat} \multimap \mathsf{nat}$.
- Ackermann's function is a standard example of a non primitive recursive function:

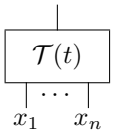
$$\begin{aligned}ack(0,n) &= \mathsf{S}\ n\\ ack(\mathsf{S}\ n, 0) &= ack(n, \mathsf{S}\ 0)\\ ack(\mathsf{S}\ n, \mathsf{S}\ m) &= ack(n, ack(\mathsf{S}\ n, m))\end{aligned}$$

  In a higher-order functional language, there is an alternative definition that we can write in our syntax:

$$\mathsf{ack} = \lambda m.\lambda n.(\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda xy.\mathsf{iter}\ (\mathsf{S}\ y)\ (\mathsf{S}\ 0)\ x))n$$

3 Interaction net encoding

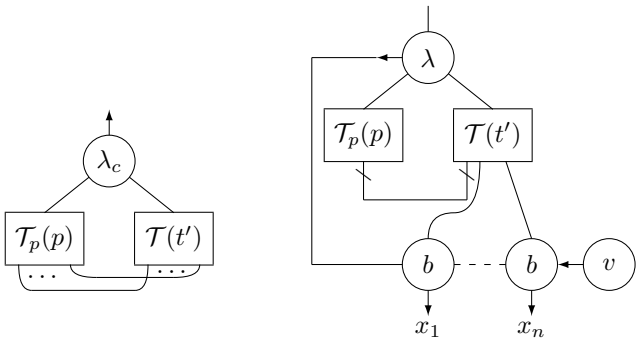
In this section we give a translation $\mathcal{T}(\cdot)$ of linear System $\mathcal{T}$ terms into interaction nets. A term t with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated as a net $\mathcal{T}(t)$ with the root edge at the top, and n free edges corresponding to the free variables:



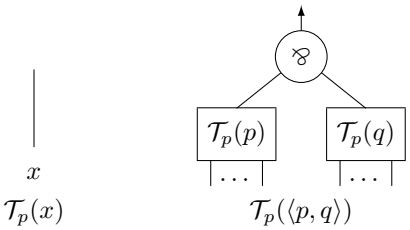
The labelling of free edges is just for the translation (and convenience), and is not part of the system. The agents that need for this compilation will be introduced on demand, and we give the interaction rules later in the section. We will occasionally make some assumptions about the order of the free edges to make the diagrams simpler below.

Variable. When t is a variable, say x, then $\mathcal{T}(t)$ is translated into an edge:

Abstraction. If t is an abstraction, say $\lambda p.t'$, then there are two alternative translations of the abstraction, which are given as follows:



In these diagrams, we use an auxiliary function for the translation of patterns $\mathcal{T}_p(p)$ which is given by the following two rules.



If p is a variable, then it is translated into an edge. Otherwise, if it is a pair pattern, then it is translated as shown in the right hand diagram above.

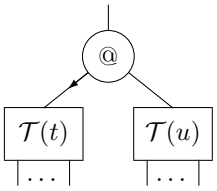
Returning to the compilation of abstraction, in the first case, shown on the left in the above diagram, is when $\mathsf{fv}(\lambda p.t') = \varnothing$. Here we use an agent λ_c to represent a *closed abstraction* and we explicitly connect the occurrence of the variable of the body of the abstraction to the λ_c agent.

The second case, shown on the right, is when $\mathsf{fv}(\lambda p.t') = \{x_1, \ldots, x_n\}$. Here we introduce three different kinds of agent: λ of arity 3, for abstraction, and two kinds of agent representing a list of free variables. An agent b is used for each free variable, and we end the list with an agent v. The idea is that there is a pointer to the free variables of an abstraction; the body of the abstraction is

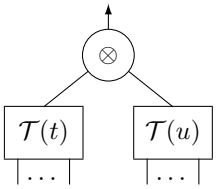
encapsulated in a box structure. We assume, without loss of generality, that the (unique) occurrence of the variable x is in the leftmost position of $\mathcal{T}(t')$.

It is worth noting that a closed term will never become open during reduction, but crucially for this system to work, terms may become closed during reduction. The distinction between open and closed terms is crucial in the dynamics of the interaction system that is given later.

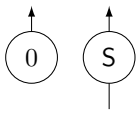
Application. If t is an application, say uv, then $\mathcal{T}(uv)$ is given by the following net, where we have introduced an agent @ of arity 2 corresponding to an application.



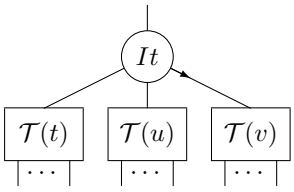
Pair. If t is a pair, say $\langle u, v\rangle$, then $\mathcal{T}(\langle u, v\rangle)$ is given by the following net, where we have introduced an agent $\otimes$ of arity 2 corresponding to a pair.



Numbers. A number will be represented by a chain of successor agents (S), terminating with a zero (0) agent. S has one auxiliary port, and 0 has none:



Iterator. To encode $\mathsf{iter}\ t\ u\ v$ we introduce one new agent as shown below. The principal port of this agent points to the function v, because we must wait for this to become a closed term before starting the interaction process.



3.1 Example

We complete this section by giving an example to illustrate how represent programs as interaction nets. In Figure 2 we give the net corresponding to the Ackermann function: $\mathcal{T}(\lambda m.(\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda xy.\mathsf{iter}\ (\mathsf{S}\ y)\ (\mathsf{S}\ 0)\ x)))$ (note that we have used η-conversion to slightly simplify this net)

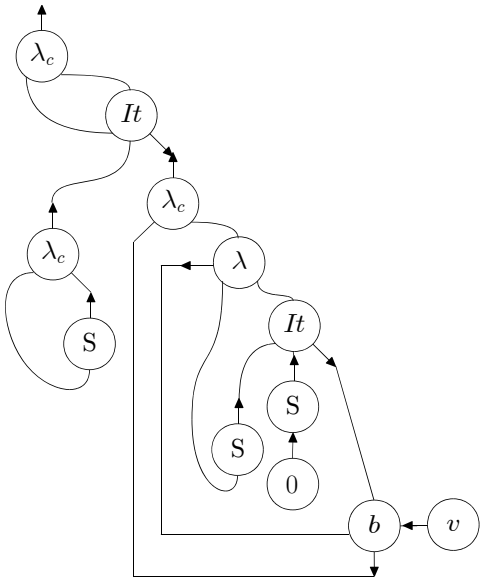


Fig. 2. Ackermann function

3.2 Reduction

In Figure 3 we summarise most of the interaction rules for this system. The first rule deals with β-reduction, the next with pair pattern matching, and the next four deal with substitution. The final three rules are for duplication and erasing, where we use α to range over all other agents in the system. There are three additional rules not in the figure that we explain in more details that implement iteration. When iterator agent interacts with a closed abstraction we have the following rule:

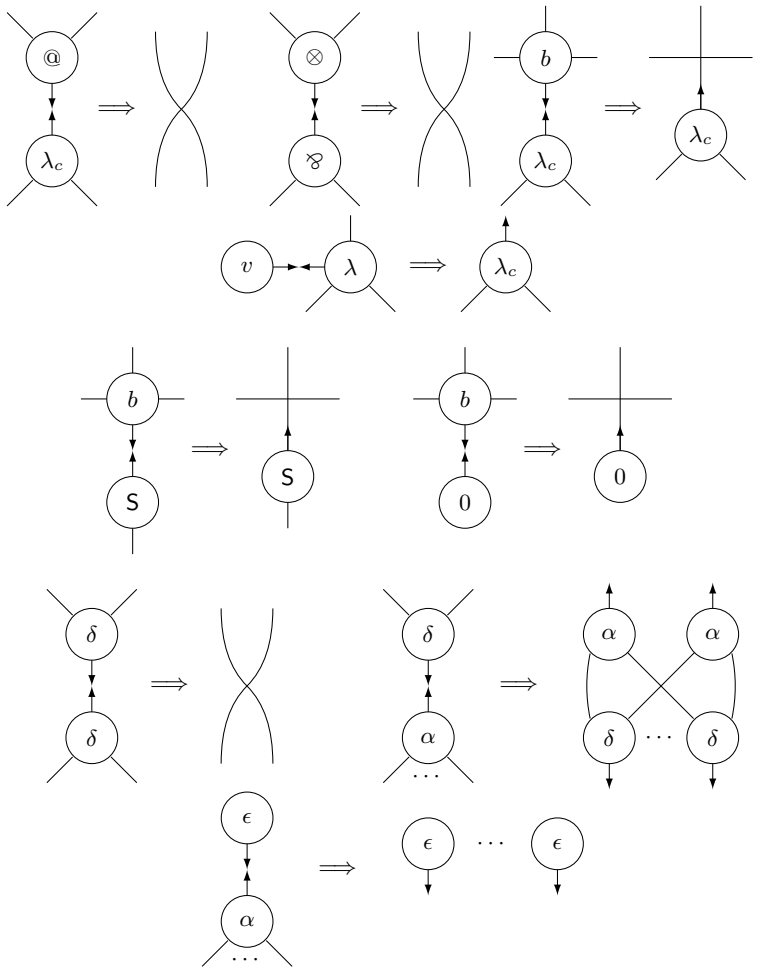
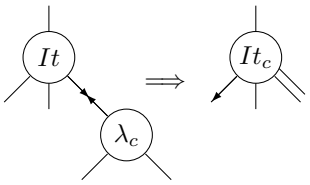
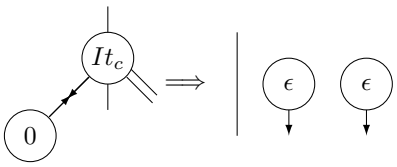


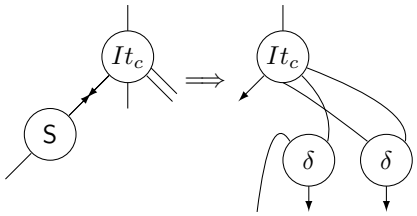
Fig. 3. Interaction Rules



This rule creates a new agent It_c that will interact with numbers. The agent also holds on to the body and the variable edge of the abstraction. The two rules for the It_c agent are as follows. The first rule is when we erase the function, and connect the result to the base value.



The final rule is when we unfold one level of iteration. Here the function is duplicated with δ agents, and one copy is applied to the base value as required. Because the function being duplicated is closed, the duplication process is easily proved to be correct.



These rules are all that we need to implement our linear System $\mathcal{T}$. By showing that we simulate the reduction rules for each case of the iterator we get the following result.

Theorem 1. *Let t be a closed linear System $\mathcal{T}$ term of base type (*nat*), then $\mathcal{T}(t) \Downarrow N$, where N is a representation of a number (i.e., built from zero and successors).*

With a little extra effort, we could have given a translation of Gödel's System $\mathcal{T}$ directly to this system of interaction nets. However, we have significantly simplified this encoding using a result from [1] stating that the linear version is as powerful as the non-linear version.

Using a result of Dal Lago [6]: if we take the linear λ-calculus where iterated functions must be *closed by construction* (*i.e.,* $\mathsf{fv}(v) = \varnothing$ in $\mathsf{iter}\ t\ u\ v$) then this system captures exactly the primitive recursive functions. If we are building functions to iterate that must be closed by construction, then we no longer need

the box structure to identify when a term becomes closed. The consequence of this result here is that we can eliminate b v and λ agents, so that λ_c and @ are sufficient to encode the linear λ-calculus.

Theorem 2. *An interaction system built from the agents* 0, S, *It,* It_c, δ, ϵ, λ_c, *and* @ *is complete for primitive recursive functions.*

The encoding of the linear λ-calculus as a system of interaction nets is particularly simple, since substitution is implemented for free: β-reduction is a constant time operation. This is a consequence of the fact that substitution is essentially implemented as an assignment.

What we have achieved therefore is a very simple, no overheads, implementation of Gödel's System $\mathcal{T}$ and primitive recursive functions.

4 Discussion

Very few people write programs with unary arithmetic (zero and successor). Nevertheless, the same techniques are used to represent lists and other data-structures. Our belief is that to understand complex languages and make them efficient, it is fruitful to start with simple subsets and build up. This is the approach we have taken in this paper.

Implementations of the λ-calculus are made complicated by the non-linear aspects of the calculus. Using the linear λ-calculus with iterators gives a simpler formulation of many algorithms, and even simpler when the iterated function is closed. It is possible to use compilation techniques to transform a non-linear algorithm in System $\mathcal{T}$ to our linear version, and also close some functions automatically. We will try to report on some of these aspects in the final version of this paper.

5 Conclusion

We have given a very simple and efficient implementation of Gödel's System $\mathcal{T}$ using the graph rewriting formalism of interaction nets. The aim of this paper was initially to apply some of the ideas used for the representation of the λ-calculus in interaction nets to the the linear version of System $\mathcal{T}$ to investigate if the resulting system provides a useful implementation technique. The experimental results have confirmed that this is a useful approach.

This work is a building block in a larger programme of research to investigate when interaction nets are useful for the evaluation of programs (either because they are more efficient than standard techniques, of if they offer some other advantage such as parallelism, small run-time system, etc.). A first step in this direction is the study an extension with simple data-types, in particular lists, and investigate if list processing algorithms can also give comparable results as in this paper.

References

1. Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. The power of linear functions. In Z. Ésik, editor, *Proceedings of the 15th EACSL Conference on Computer Science Logic (CSL'06)*, volume 4207 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2006.
2. Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Gödel's system T revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500, 2010.
3. Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. The bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, November 1996.
4. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, second, revised edition, 1984.
5. Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.
6. U. Dal Lago. The geometry of linear higher-order recursion. In P. Panangaden, editor, *Proceedings of the 20th Annual IEEE Symposium on logic in Computer Science, LICS 2005*, pages 366–375. IEEE Computer Society Press, June 2005.
7. Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
8. Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, January 1992.
9. Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.
10. John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, January 1990.
11. Jean-Jacques Lévy. Optimal reductions in the lambda calculus. In J. P. Hindley and J. R Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
12. Sylvain Lippi. λ-calculus left reduction with interaction nets. *Mathematical Structures in Computer Science*, 12(6), 2002.
13. Ian Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, September 1998.
14. Ian Mackie. An interaction net implementation of closed reduction. In Sven-Bodo Scholz and Olaf Chitil, editors, *IFL*, volume 5836 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2008.
15. François-Régis Sinot. Call-by-name and call-by-value as token-passing interaction nets. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2005.

Graph Isomorphism and Edge Graph Isomorphism

Edel Sherratt

Department of Computer Science, University of Wales, Penglais, Aberystwyth SY23 3DB, Wales UK

Abstract. A chemical structure generator enumerates the structural isomers that can be constructed from a collection of atoms or molecular fragments. The Abermol structure generator represents molecules and fragments as coloured graphs, whose vertices stand for atoms, coloured by the elements they represent, and whose edges represent bonds, coloured to represent different kinds of bond. At each enumeration step Abermol identifies and eliminates duplicate structures, which it then eliminates. This entails transforming edge- and vertex-coloured graphs to simple vertex-coloured graphs for isomorphism testing by McKay's nauty software. This paper presents the theorems underpinning isomorphism testing in Abermol. The theorems argue that if two possibly disconnected graphs have the same number of isolated vertices, and the same number of K_3 connected components, then isomorphism of the edge graphs corresponding to their remaining components is logically equivalent to isomorphism of the original graphs. This allows direct representation of molecular graphs as edge- and vertex-coloured graphs, and subsequent transformation to equivalent edge graphs for isomorphism testing.

1 Introduction

A chemical structure generator takes as input atoms or molecular fragments or both and generates an isomorph-free enumeration of the structural isomers that can be constructed from its inputs. Molecules and fragments are represented as graphs, where vertices stand for atoms, and edges for bonds.

The vertices are 'coloured' [1] by the atoms they represent, so that, for instance, all vertices representing carbon atoms are coloured 'C'. Edges can also be coloured to represent different bond types. Alternatively, loop-free multigraphs [1], allow, for example, double or triple bonds to be represented as two or three edges in the molecular graph.

The best known chemical structure generators include isomer generators and generators of molecular graphs containing given substructures. For example, Molgen [2,3] enumerates all the molecular graphs that correspond to a given formula. This is also the case with Houdini[4], and the generator defined by Faulon and Churchwell [5], which generates all structures matching a given signature.

These systems all generate complete molecules, which facilitates isomorphism testing, for example by taking account of the characteristic vertex connectivities of saturated molecular graphs [5] .

If we also wish to generate disconnected molecules or fragments, representing intermediate products in biochemical networks, then alternative strategies are needed. This paper presents the key theorem that underpins the approach to isomorphism testing used in Abermol, a chemical structure generator designed to enumerate structures that represent complete molecules or molecule fragments.

A brief overview of isomorphism testing is provided below. This is followed by a proof that subject to some readily checked conditions, isomorphism of two edge graphs is logically equivalent to isomorphism of the parent graphs from which they are derived. This allows transformation of simple edge- and vertex-coloured molecular graphs to be transformed to simple vertex-coloured graphs for isomorphism testing. The paper concludes with a short discussion of the application of this result within Abermol.

2 Isomorphism testing for molecular graphs

Every structure generator needs a fast effective way to avoid generating duplicate structures. If molecular graphs are not isomorphic, then they represent distinct structures[1]

Abermol uses nauty [6,7], to test for isomorphism. Nauty provides very fast reliable isomorphism testing for simple vertex coloured graphs. However, molecular graphs have both edge and vertex colouring.

A loop-free multigraph [1] can be transformed to a simple graph by the inclusion of dummy nodes to split duplicate edges into pairs of edges. Abermol takes a different approach, transforming edge- and vertex-coloured graphs to simple graphs called *edge graphs* before testing for isomorphism. Isomorphism of two edge graphs implies isomorphism of original graphs provided some readily checked conditions are met by graphs that contain three mutually adjacent edges.

This result is proved below for simple, uncoloured graphs, and its application to structure generation is briefly outlined.

3 Definitions and Useful Facts

Some basic definitions and useful facts about graphs are stated below. These will be used in proving the relationship between edge graph isomorphism and graph isomorphism.

Throughout this and subsequent sections, all sets are assumed to be finite and discrete, because this is true of molecular graphs.

Definition 1. *Simple Graph*

A simple graph G = (V,E) consists of a nonempty set V of vertices, and a set E of unordered pairs of distinct elements of V called edges. If $G = (V, E)$ *is a simple graph, then the undirected edge* $\{a, b\}$ *of G is also written ab or ba.*

[1] Graphs that are isomorphic may nonetheless represent. distinct structures. For example, N(H)(H) and O(H)(H) represent distinct structures although their molecular graphs are isomorphic. Of course, this is not an issue if only fully connected structures containing all given elements are generated.

Definition 2. *Subgraph*

Let $G = (V, E)$ be a simple graph. A graph $G' = (V', E')$ is a subgraph of G iff $V' \subset V$ and $E' \subset E$

Definition 3. *Partition*

Let $G = (V, E)$ be a simple graph. The list of graphs $(G_1 = (V_1, E_1), \ldots, G_n = (V_n, E_n)$ partitions G if $(V_1, \ldots, V_n)$ partitions the set V and $(E_1, \ldots E_n)$ partitions the set E.

Definition 4. *Image of a graph*

Let $G = (V, E)$ be a simple graph and let f be a function defined on V. Then V^f, E^f and G^f are the images of V, E and G under f, defined as follows:

$$V^f = \{f(a) \mid a \in V\}$$
$$E^f = \{f(a)f(b) \mid ab \in E\}$$
$$G^f = (V^f, E^f)$$

Definition 5. *Connectivity*

Let $G = (V, E)$ be a simple graph. Let $a \in V$.

- *The set of edges incident on a is $A = \{e \in E \mid a \in e\}$.*
- *If $e, e' \in E$ are two distinct edges of G, then e and e' are coincident edges iff they share a common vertex, that is, iff $\exists\ a \in V : e \cap e' = \{a\}$*
- *The degree of a is $\mid A \mid$, the cardinality of the set of edges incident on a.*
- *a is an isolated vertex iff $\mid A \mid = 0$.*
- *Let $b \in V$. There is a walk from a to b in G iff $a = b$ or there is a sequence $(a_1, \ldots, a_n)$ where $a = a_1$, $b = a_n$ and $\{a_1a_2, \ldots, a_{n-1}a_n\} \subset E$*
- *G is connected iff $\forall v, v' \in V$: there is a walk from v to v' in G.*
- *The null graph $G = (\emptyset, \emptyset)$ is connected.*
- *A connected component of G is a connected subgraph G' of G such that there is no other connected subgraph of G that contains G'.*
  If $G' = (V', E')$ is a connected subgraph of G, then G' is a connected component of G iff $\forall v \in V \backslash V', v' \in V'$: there is no walk from v to v'.

Definition 6. *Edge preservation and Isomorphism*

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple graphs.

1. *A function $f : V_1 \to V_2$ preserves the edges of G_1 provided $ab \in E_1 \iff f(a)f(b) \in E_2$.*
2. *G_1 and G_2 are isomorphic, written $G_1 \cong G_2$, iff there is a bijective function $f : V_1 \to V_2$ that preserves the edges of G_1.*

Definition 7. *Edge graph*

Let $G = (V, E)$ be a simple graph, with no isolated vertices.

Then $G^{\mathcal{E}} = (E, E^{\mathcal{E}})$, where $E^{\mathcal{E}} = \{ee' \mid \exists a \in V : e \cap e' = \{a\}\}$, is the edge graph corresponding to G. $G^{\mathcal{E}}$ is a simple graph whose vertices are the edges of

G, and whose edges are the pairs of coincident edges of G. The edge graph is not defined for graphs with isolated vertices [2].

Figure 1 illustrates a graph and some edges and vertices its corresponding edge graph. The complete edge graph corresponding to the original graph is illustrated in Figure 2.

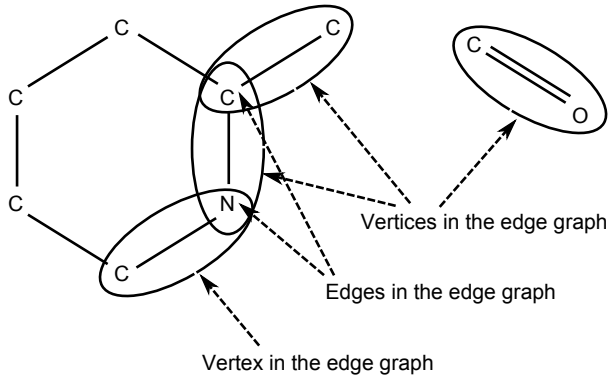


Fig. 1. Edges and vertices in an edge graph. The double line in the upper right does not indicate two edges, but an edge colouring indicating a chemical double bond. The single lines are edges coloured to represent single bonds.

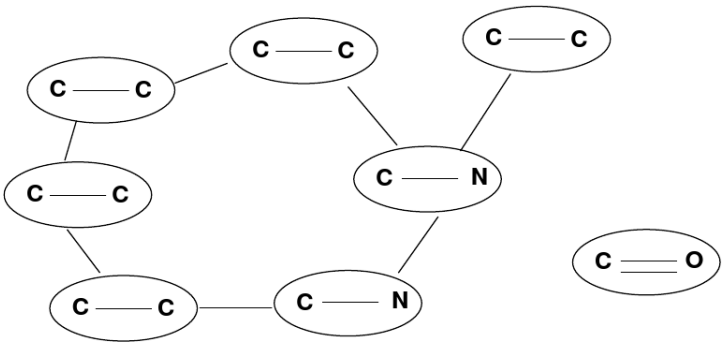


Fig. 2. An edge graph labelled with the original graph elements

[2] This not a problem for chemical structure generation (nor for isomorphism testing), as simple comparison is sufficient to determine whether or not two collections of isolated vertices are ismorphic.

Some useful facts are listed below. Their proofs are straightforward and are not included.

1. **The union of a collection of bijective functions is a bijective function**
   Let $f_1 : V_1 \to W_1, \ldots, f_n : V_n \to W_n$ be a list of bijective functions such that $\forall 1 \leq i, j \leq n : i \neq j \Rightarrow V_i \cap V_j = \emptyset$ and $W_i \cap W_j = \emptyset$. Then $f : V \to W = \bigcup_{1 \leq i \leq n} f_i$ is also a bijective function where $V = \bigcup_{1 \leq i \leq n} V_i$ and $W = \bigcup_{1 \leq i \leq n} W_i$.
2. **Any subset of a bijective function is a bijective function**
   Let $f : V \to W$ be a bijective function. Let $g \subset f$. Then g is a bijective function.
3. **Inverse image of a subset is the subset**
   Let $f : V \to W$ be a bijective function. Let $A \subset V$. Then $(A^f)^{f^{-1}} = A$
4. **Graph and Subgraph Isomorphism**
   Let $G = (V, E)$ and $G' = (V', E')$ be two simple graphs such that $G \cong G'$. Let $f : V \to V'$ be a bijective function such that $\{a, b\} \in E \iff \{f(a), f(b)\} \in E'$. If $H = (V_H, E_H)$ is a subgraph of G, then $H^f = (V_H^f, E_H^f)$ is a subgraph of G^f and $H \cong H^f$.
5. **Connected image of a connected subgraph**
   Let $G = (V, E)$ and $G' = (V', E')$ be two simple graphs such that $G \cong G'$. Let $f : V \to V'$ be a bijective function such that $\{a, b\} \in E \iff \{f(a), f(b)\} \in E'$. If $K = (V_K, E_K)$ is a connected subgraph of G, then $K^f = (V_K^f, E_K^f)$ is a connected subgraph of G^f
6. **Partitioning a graph into its connected components**
   Let $G = (V, E)$ be a simple graph. For all $a, b \in V$, let aRb denote the relationship that is satisfied iff there is a walk from a to b.
   Then R is an equivalence relation whose equivalence classes partition of G into its connected components.

4 Graph Isomorphism and Edge Graph Isomorphism

This section proves that two graphs are isomorphic iff

- they have equal numbers of isolated vertices
- they have equal numbers of K_3 components
- their connected components with two or more vertices have edge graphs that are pairwise isomorphic

The proof is structured as follows.

1. First it is observed that two graphs are isomorphic iff their connected components are pairwise isomorphic. This allows the rest of the proof to focus on connected graphs. In particular, it means that isolated vertices, which represent isolated atoms in a chemical structure, can simply be compared when checking for duplicate structures.

2. Different cases arise when we consider connected graphs with isomorphic edge graphs. These are handled as follows.
   Suppose $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two simple connected graphs, where $\mid V_1 \mid > 1$, $\mid V_2 \mid > 1$ and $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.
   Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$. That is, $f_{\mathcal{E}}$ is a bijection that defines edge graph isomorphism.

   2.1 Lemmas 1 and 2 deal with the case that the image under $f_{\mathcal{E}}$ of the set of edges incident on any given vertex in the G_1 is the set of edges incident on a given vertex of G_2.

      2.1.1 If the vertex of G_2 is not unique, then each of the graphs consists of two vertices connected by a single edge, and so the graphs are isomorphic. (Lemma 1).

      2.1.2 If the vertex of G_2 is unique, then the bijection between the edge graphs determines an edge-preserving bijection between the vertices of the original graphs, and so they are isomorphic. (Lemma 2).

   2.2 Lemmas 3 and 4 deal with the case where there is at least one vertex of G_1 whose incident edges are mapped by $f_{\mathcal{E}}$ to a set of edges that are not incident on a single vertex of G_2.
   In this case, $f_{\mathcal{E}}$ either maps a K_3 to an S_4 subgraph or vice versa (Lemma 3). Lemma 4 shows that the S_4 and K_3 subgraphs in fact constitute the entire graph.

3. Theorem 1 proves that if two simple graphs with no isolated vertices have isomorphic edge graphs, and if they have the same number of K_3 connected components, then they are isomorphic.
4. Finally, Theorem 2 shows that two simple graphs are isomorphic iff they have the same number of isolated vertices, the same number of K_3 components and if their edge graphs are isomorphic.

4.1 Two Graphs are Isomorphic iff their Connected Components are Pairwise Isomorphic

Let $G = (V, E)$ and $G' = (V', E')$ be two simple graphs.

Let $(G_1 = (V_1, E_1), \ldots, G_n = (V_n, E_n))$ partition G into its connected components, and let $(G'_1 = (V'_1, E'_1), \ldots, G'_m = (V'_m, E'_m))$ partition G' into its connected components.

Then $G \cong G'$ iff $m = n$ and there is a permutation p of $1 \ldots n$ such that $\forall 1 \leq i \leq n : G_i \cong G'_{p(i)}$.

This useful observation is easily proved and means that graph isomorphism can be considered in terms of isomorphism of connected subgraphs. In particular, isolated vertices can be compared, and edge graph isomorphism can be used when dealing with graphs that have at least one edge.

4.2 Incident edges on a vertex of one edge graph map to incident edges on a vertex of isomorphic edge graph.

Suppose G_1 and G_2 have isomorphic edge graphs, and suppose $f_{\mathcal{E}}$ is an edge-preserving, bijective mapping between those edge graphs such that the image under $f_{\mathcal{E}}$ of the set of edges incident on any given vertex in the G_1 is the set of edges incident on a given vertex of G_2.

Lemma 1 argues that if the vertex of G_2 is not unique, then each of the graphs consists of two vertices connected by a single edge, and so the graphs are isomorphic.

If on the other hand, the vertex of G_2 is unique, then the bijection between the edge graphs determines an edge-preserving bijection between the vertices of G_1 and G_2, and so they are isomorphic. (Lemma 2)

Lemma 1. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple connected graphs, where $\mid V_1 \mid > 1$, $\mid V_2 \mid > 1$ and $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.*

Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$.

If $\forall a \in V_1 : \exists x \in V_2 : A^{f_{\mathcal{E}}} = X$ and $\exists a \in V_1; x, y \in V_2 : x \neq Y$ and $X = A^{f_{\mathcal{E}}} = Y$
then $G_2 = (\{x, y\}, \{xy\})$ and $\exists b \in V_1 : G_1 = (\{a, b\}, \{ab\})$ and so $G_1 \cong G_2$.

Proof Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple connected graphs, where $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.

Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$.

Let $a \in V_1$, and let $x, y \in V_2 : x \neq y$ and $X = A^{f_{\mathcal{E}}} = Y$ and let $e \in A$

Then $f(e) \in X$ and $f(e) \in Y$; that is $f(e) = xy$, and so $X = Y = \{xy\} = A^{f_{\mathcal{E}}}$, but then $A = (A^{f_{\mathcal{E}}})^{f_{\mathcal{E}}^{-1}} = (\{xy\})^{f_{\mathcal{E}}^{-1}} = \{e\}$. So $\exists b \in V_1 : e = ab$ and $A = \{e\} = \{ab\}$

Also $E_1 = A$, for let $e' \in E_1$. Then, since G_1 is connected, eRe'.

Suppose $e \neq e'$

Then there is a sequence $(e_1e_2, \ldots, e_{n-1}e_n)$, where $e = e_1$, $e_n = e'$ and $\{e_1e_2, \ldots, e_{n-1}e_n\} \subset E_1^{\mathcal{E}}$

$ee_2 \in E_1^{\mathcal{E}} \Rightarrow f_{\mathcal{E}}(e)f_{\mathcal{E}}(e_2) \in E_2^{\mathcal{E}}$

So $x \in f_{\mathcal{E}}(e_2)$ or $y \in f_{\mathcal{E}}(e_2)$, but since $X = \{xy\} = Y$, this means that $e_2 \in \{xy\}$, but $e_2 \neq xy$, so it cannot be the case that $ee_2 \in E_1^{\mathcal{E}}$, and hence it must be the case that $e' = e$. So $E_1 \subset A$, and since by definition $A \subset E_1$, we can conclude that $E_1 = A$

So $E_1 = \{ab\}$, and since G_1 is connected, $V_1 = \{a, b\}$ and so $G_1 = (\{a, b\}, \{ab\})$

Similarly, $E_2 = \{xy\}$, $V_2 = \{x, y\}$ and $G_2 = (\{x, y\}, \{xy\})$

Hence $G_1 \cong G_2$.

Lemma 2. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple connected graphs, where $\mid V_1 \mid > 1$, $\mid V_2 \mid > 1$ and $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.*

Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$.

If $\forall a \in V_1 : \exists x \in V_2 : A^{f_{\mathcal{E}}} = X$ and $\forall a \in V_1; x, y \in V_2 : X = A^{f_{\mathcal{E}}} = Y \Rightarrow x = y$, then the function $f : V_1 \to V_2 : a \mapsto x \iff A_{\mathcal{E}}^f = X$ is bijective and preserves the edges of G_1, and so $G_1 \cong G_2$.

Proof Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple connected graphs, where $\mid V_1 \mid > 1$, $\mid V_2 \mid > 1$ and $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.

Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$, where $\forall a \in V_1 : \exists x \in V_2 : A^{f_{\mathcal{E}}} = X$ and $\forall a \in V_1; x, y \in V_2 : X = A^{f_{\mathcal{E}}} = Y \Rightarrow x = y$.

Let $f = \{(a, x) \in V_1 \times V_2 : A^{f_{\mathcal{E}}} = X\}$

1. f is a function
   Let $(a, x), (a, y) \in f$. Then $A^{f_{\mathcal{E}}} = X$ and $A^{f_{\mathcal{E}}} = Y$. So $X = A^{f_{\mathcal{E}}} = Y$
   But $\forall a \in V_1; x, y \in V_2 : X = A^{f_{\mathcal{E}}} = Y \Rightarrow x = y$.
   So $x = y$ and f is a function.
2. f is injective
   Let $(a, x), (b, x) \in f$.
   Suppose $a \neq b$. Then $A = B$, since $A^{f_{\mathcal{E}}} = X = B^{f_{\mathcal{E}}}$. But $\forall e \in A : e \in B$, so $A = B = \{ab\}$. So $X = \{f_{\mathcal{E}}(ab)\} = \{xy\}$ for some $y \in V_2$
   Also, if $A = B = \{ab\}$ and $A^{f_{\mathcal{E}}} = B^{f_{\mathcal{E}}} = X = \{xy\}$, then $Y = \{xy\}$, for let $e \in Y : e \neq xy$. Then $e(xy) \in E_2^{\mathcal{E}}$, and so $(f_{\mathcal{E}}^{-1}(e))(ab) \in E_a^{\mathcal{E}}$. and so $f_{\mathcal{E}}^{-1}(e) \in A$ or $f_{\mathcal{E}}^{-1}(e) \in B$, and $f_{\mathcal{E}}^{-1}(e) \neq ab$
   But since $A = B = \{ab\}$ this is not the case, and so $\forall e \in Y : f_{\mathcal{E}}^{-1}(e) = ab$.
   so $A = B = \{ab\}$, and $A^{f_{\mathcal{E}}} = B^{f_{\mathcal{E}}} = \{xy\}$
   But this means that $A^{f_{\mathcal{E}}} = X$ and $A^{f_{\mathcal{E}}} = Y$ with $x \neq y$ which contradicts $\forall y \in V_2 : A^{f_{\mathcal{E}}} = Y \Rightarrow x = y$
   So $a = b$ and f is injective.
3. f is surjective
   Let $z \in V_2$, let $e \in Z$ and let $ab = f_{\mathcal{E}}^{-1}(e)$. Let $x \in V_2 : A^{f_{\mathcal{E}}} = X$ and let $y \in V_2 : B^{f_{\mathcal{E}}} = Y$
   That is $(a, x), (b, y) \in f$
   Now $x \neq y$, since $a \neq b$ and f is injective.
   Also $e \in A^{f_{\mathcal{E}}} = X$, since $f_{\mathcal{E}}^{-1}(e) = ab \in A$. and $e \in B^{f_{\mathcal{E}}} = Y$, since $f_{\mathcal{E}}^{-1}(e) = ab \in B$
   So $e = xy$, and, since $z \in e$, $z = x$ or $z = y$.
   If $z = x$, then $f(a) = z$, and if $z = y$ then $f(b) = z$.
   In either case $\exists a \in V_1 : (a, x) \in f$ and so f is surjective.
4. f preserves the edges of G_1
   - Let $ab \in E_1$
     Let $x \in V_2 : A^{f_{\mathcal{E}}} = X$ and let $y \in V_2 : B^{f_{\mathcal{E}}} = Y$
     So $f(a) = x$ and $f(b) = y$. Since $a \neq b$ and since f is injective, $x \neq y$
     Let $e = f_{\mathcal{E}}(ab) \in E_2$. Then $e \in (A^{f_{\mathcal{E}}} \cap B^{f_{\mathcal{E}}}) = X \cap Y$. So $e = xy \in E_2$.
     So $ab \in E_1 \Rightarrow f(a)f(b) \in E_2$
   - Now let $a, b \in V_1$ with $f(a)f(b) \in E_2$
     $a \neq b$ since $f(a) \neq f(b)$ and since f is a function. Let $x = f(a)$ and let $y = f(b)$. Then $xy \in V_2$ where $X = A^{f_{\mathcal{E}}}$ and $Y = B^{f_{\mathcal{E}}}$. So $f_{\mathcal{E}}^{-1}(xy) \in E_1$.
     But $f_{\mathcal{E}}^{-1}(xy) \in X^{f_{\mathcal{E}}^{-1}}(xy) = A$. And $f_{\mathcal{E}}^{-1}(xy) \in Y^{f_{\mathcal{E}}^{-1}}(xy) = B$.
     And since $a \neq b$, $A \neq B$ and $f_{\mathcal{E}}^{-1}(xy) \in A \cap B = \{ab\}$.
     But $xy = f(a)f(b) \in E_2$ and so $f_{\mathcal{E}}^{-1}(xy) \in E_1$. That is $ab \in E_1$. So $f(a)f(b) \in E_2 \Rightarrow ab \in E_1$.

So $f : V_1 \to V_2$ is a bijective function that preserves the edges of G_1 and hence $G_1 \cong G_2$

4.3 What if $f_{\mathcal{E}}$ does not determine f?

This section concerns non-isomorphic graphs with isomorphic edge graphs. These always involve K_3 and S_4 subgraphs, whose edge graphs are all K_3. The problem is illustrated in Figure 3. This situation occurs if G_1 and G_2 have isomorphic

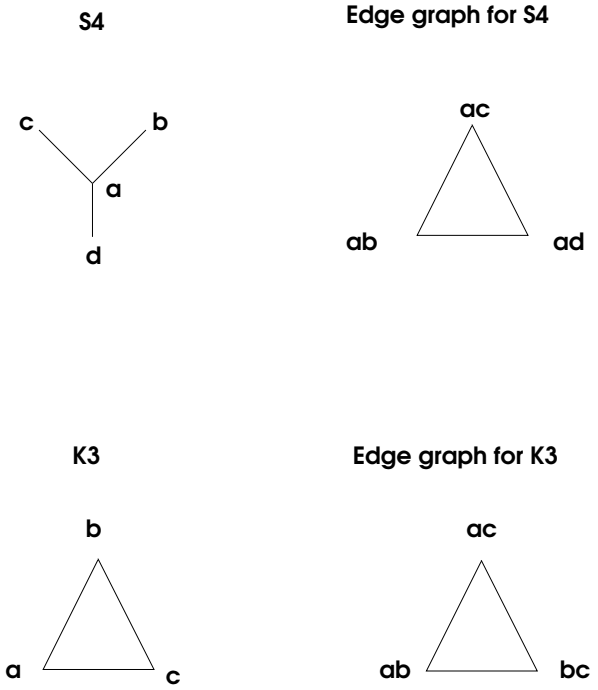


Fig. 3. The edge graph for a K_3 graph is a K_3 graph, but so too is the edge graph for an S_4 graph.

edge graphs, but there is at least one vertex of G_1 whose incident edges are mapped by an edge-preserving bijection to edges of G_2 that are not incident on any given vertex of G_2. Lemma 3 shows that this involves a mapping of a K_3 to an S_4 subgraph or vice versa. Lemma 4 shows that the S_4 and K_3 subgraphs in fact constitute all of G_1 and G_2.

Lemma 3. *Let* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *be two simple connected graphs, where* $\mid V_1 \mid > 1$, $\mid V_2 \mid > 1$ *and* $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.

Let $f_{\mathcal{E}} : E_1 \to E_2$ *be a bijective function that preserves the edges of* $G_1^{\mathcal{E}}$

If $\exists\ a \in V_1 : \forall x \in V_2 : A^{f_{\mathcal{E}}} \neq X$ *then either* $\exists a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, ad \mapsto yz$ *or* $\exists a, b, c \in V_1; x, y, z, w \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, bc \mapsto xw$

Proof Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple connected graphs, where $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.

Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$
Let $a \in V_1 : \forall x \in V_2 : A^{f\varepsilon} \neq X$
Let $ab \in A$ and let $xy = f_{\mathcal{E}}(ab)$
$xy \in X$ and $xy \in Y$
Now $X \neq A^{f\varepsilon}$, and $Y \neq A^{f\varepsilon}$ since $\nexists x \in V_2 : A^{f\varepsilon} = X$
So ($X \not\subset A^{f\varepsilon}$ or $A^{f\varepsilon} \not\subset X$) and ($Y \not\subset A^{f\varepsilon}$ or $A^{f\varepsilon} \not\subset Y$)
That is ($\exists e \in A^{f\varepsilon} : e \notin X$ or $\exists e \in X : e \notin A^{f\varepsilon}$) and ($\exists e' \in A^{f\varepsilon} : e \notin Y$ or $\exists e' \in Y : e \notin A^{f\varepsilon}$)

- Let $e \in A^{f\varepsilon} : e \notin X$
  Now $e \neq xy$, since $e \notin X$, so $f_{\mathcal{E}}^{-1}(e) \neq ab$. But $f_{\mathcal{E}}^{-1}(e) \in A$, since $e \in A^{f\varepsilon}$.
  So $(f_{\mathcal{E}}^{-1}(e))(ab) \in E_1^{\mathcal{E}}$, and so $e(xy) \in E_2^{\mathcal{E}}$
  Hence $e \in Y$.
- Let $e \in X : e \notin A^{f\varepsilon}$
  Now $e \neq xy$, since $e \notin A^{f\varepsilon}$ and $xy \in A^{f\varepsilon}$ So $e(xy) in E_2^{\mathcal{E}}$, and $(ab)(f_{\mathcal{E}}^{-1}(e)) \in E_1^{\mathcal{E}}$.
  Now $f_{\mathcal{E}}^{-1}(e) \notin A$, so $f_{\mathcal{E}}^{-1}(e) \in B$

  Similarly, if $e' \in A^{f\varepsilon}$ and $e' \notin Y$, then $e' \in X$,
  and if $e' \in Y : e' \notin A^{f\varepsilon}$, then $f_{\mathcal{E}}^{-1}(e') \in B$
  This gives the following possibilities:

1. $\exists e, e' \in A^{f\varepsilon} : e \in Y, e \notin X$, $e' \in X$ and $e' \notin Y$. Let $e, e' \in A^{f\varepsilon}$ with $e \in Y$, $e \notin X$, $e' \in X$ and $e' \notin Y$. Then e, e' and xy are all distinct members of E_2 since $e \in Y$ but $e' \notin Y$, $e \notin X$ but $xy \in X$ and $e' \notin Y$ but $xy \in Y$
   $\exists c, d \in V_1 : e = ac$ and $e' = ad$, since $f_{\mathcal{E}}^{-1}(e), f_{\mathcal{E}}^{-1}(e') \in A$
   Also $(f_{\mathcal{E}}^{-1}(e))(f_{\mathcal{E}}^{-1}(e')) \in E_1^{\mathcal{E}}$, and so $ee' \in E_2^{\mathcal{E}}$
   So $\exists z \in V_2 : e = yz, e' = xz$, since $e \in Y$ and $e' \in X$
   So $\exists c, d \in V_1, z \in V_2 : f : ac \mapsto xz, ad \mapsto yz$. Hence $\exists\ a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, ad \mapsto yz$.
2. $\exists e, e' \in B^{f\varepsilon} : e \in X, e' \in Y$ and $e, e' \notin A^{f\varepsilon}$
   Let $e, e' \in B^{f\varepsilon} : e \in X, e' \in Y$ and $e, e' \notin A^{f\varepsilon}$
   Then e, e' and xy are all distinct members of E_2 since $xy \in A^{f\varepsilon}$, but $e \notin A^{f\varepsilon}$, $xy \in A^{f\varepsilon}$, but $e' \notin A^{f\varepsilon}$ and $e \in X$, but $e' \notin X$
   Now $f_{\mathcal{E}}^{-1}(e), f_{\mathcal{E}}^{-1}(e') \in B$, so $\exists c, d \in V_1 : f_{\mathcal{E}}^{-1}(e) = bc$ and $f_{\mathcal{E}}^{-1}(e') = bd$.
   Also $f_{\mathcal{E}}^{-1}(e), f_{\mathcal{E}}^{-1}(e') \in E_1^{\mathcal{E}}$, so $ee' \in E_2^{\mathcal{E}}$ and, since $e \in X$ and $e' \in Y$, $\exists z \in V_2 : e = xz$ and $e' = yz$
   So $\exists c, d \in V_1, z \in V_2 : f_{\mathcal{E}}^{-1} : bc \mapsto xz, bd \mapsto yz$
   Hence $\exists\ a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, bc \mapsto xz, bd \mapsto yz$, which, renaming the bound variables a and b, is logically equivalent to $\exists\ a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, ad \mapsto yz$,
3. $\exists e, e' \in X : e \notin A^{f\varepsilon}, e \in B^{f\varepsilon}$, $e' \in A^{f\varepsilon}$, and $e' \notin Y$
   Let $e, e' \in X : e \notin A^{f\varepsilon}, e \in B^{f\varepsilon}$, $e' \in A^{f\varepsilon}$, and $e' \notin Y$
   Then e, e' and xy are all distinct members of E_2 since $e \notin A^{f\varepsilon}$ but $e' \in A^{f\varepsilon}$, $e \notin A^{f\varepsilon}$ but $xy \in A^{f\varepsilon}$ and $e' \notin Y$ but $xy \in Y$

$\exists z, w \in V_2 : e = xw$ and $e' = xz$, since $e, e' \in X$
Also $ee' \in E_2^{\mathcal{E}}$, and so $(f_{\mathcal{E}}^{-1}(e))(f_{\mathcal{E}}^{-1}(e')) \in E_1^{\mathcal{E}}$. So $\exists c \in V_1 : f_{\mathcal{E}}^{-1}(e) = bc$ and $f_{\mathcal{E}}^{-1}(e') = ac$, since $f_{\mathcal{E}}^{-1}(e) \in B$ and $f_{\mathcal{E}}^{-1}(e') \in A$
So $\exists c \in V_1 : z, w \in V_2 : f : ac \mapsto xz, bc \mapsto xw$
Hence $\exists a, b, c \in V_1; x, y, z, w \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, bc \mapsto xw$

4. $\exists e, e' \in Y : e \notin X, e \in A^{f\varepsilon}$, $e' \notin A^{f\varepsilon}$, and $e' \in B^{f\varepsilon}$
Let $e, e' \in Y : e \notin X, e \in Y, e \in A^{f\varepsilon}$, $e' \notin A^{f\varepsilon}$, and $e' \in B^{f\varepsilon}$
Then e, e' and xy are all distinct members of E_2, and since $e \notin X$ but $xy \in X$, $e' \notin A^{f\varepsilon}$ but $xy \in A^{f\varepsilon}$ and $e \in A^{f\varepsilon}$ but $e' \notin A^{f\varepsilon}$,
$\exists z, w \in V_2 : e = yw$ and $e' = yz$, since $e, e' \in Y$
Also $ee' \in E_2^{\mathcal{E}}$, and so $(f_{\mathcal{E}}^{-1}(e))(f_{\mathcal{E}}^{-1}(e')) \in E_1^{\mathcal{E}}$. So $\exists c \in V_1 : f_{\mathcal{E}}^{-1}(e) = ac$ and $f_{\mathcal{E}}^{-1}(e') = bc$, since $f_{\mathcal{E}}^{-1}(e) \in A$ and $f_{\mathcal{E}}^{-1}(e') \in B$
Hence $\exists a, b, c \in V_1; x, y, z, w \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, bc \mapsto xz, ac \mapsto yw$ which, renaming the bound variables a and b, is logically equivalent to Hence $\exists a, b, c \in V_1; x, y, z, w \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, bc \mapsto xw$

Hence $\exists a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, ad \mapsto yz$ or $\exists a, b, c \in V_1; x, y, z, w \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, bc \mapsto xw$

4.4 S_4 subgraph of G_1 maps to K_3 subgraph of G_2

Lemma 4. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple connected graphs, where $\mid V_1 \mid > 1$, $\mid V_2 \mid > 1$ and $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.*
Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$
If $\exists a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, ad \mapsto yz$
then either $\exists w : V_2 = \{x, y, z, w\}$ and $G_1 \cong G_2$, or $V_2 = \{x, y, z\}$, $G_2 = (\{x, y, z\}, \{xy, xz, yz\})$ and $G_1 = (\{a, b, c, d\}, \{ab, ac, ad\})$.

Proof Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two simple connected graphs, where $\mid V_1 \mid > 1$, $\mid V_2 \mid > 1$ and $G_1^{\mathcal{E}} \cong G_2^{\mathcal{E}}$.
Let $f_{\mathcal{E}} : E_1 \to E_2$ be a bijective function that preserves the edges of $G_1^{\mathcal{E}}$
Let $a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, ad \mapsto yz$
Consider A, B, C, D, X, Y, Z in turn.

1. $A = \{ab, ac, ad\}$
For if $e \in A \backslash \{ab, ac, ad\}$, then $e(ab), e(ac), e(ad) \in E_1^{\mathcal{E}}$ and so $(f_{\mathcal{E}}(e))(xy), (f_{\mathcal{E}}(e))(xz)), (f_{\mathcal{E}}(e))(yz) \in E_2\mathcal{E}$.
So ($x \in f_{\mathcal{E}}(e)$ or $y \in f_{\mathcal{E}}(e)$) and ($x \in f_{\mathcal{E}}(e)$ or $z \in f_{\mathcal{E}}(e)$) and ($y \in f_{\mathcal{E}}(e)$ or $z \in f_{\mathcal{E}}(e)$)
Now this is only satisfied if $f_{\mathcal{E}}(e) \in \{xy, xz, yz\}$. But $f_{\mathcal{E}}(e) \in \{xy, xz, yz\} \Rightarrow e \in \{ab, ac, ad\}$ which contradicts $e \in A \backslash \{ab, ac, ad\}$.
So $A \backslash \{ab, ac, ad\} = \emptyset$ and $A = \{ab, ac, ad\}$
2. $B \subset \{ab, bc, bd\}$, $C \subset \{ac, bc, cd\}$ and $D \subset \{ad, bd, cd\}$
Let $e \in B \backslash \{ab\}$. Then $e(ab) \in E_1^{\mathcal{E}}$, and so $(f_{\mathcal{E}})(xy) \in E_2^{\mathcal{E}}$
Also, $f_{\mathcal{E}} \notin \{xy, xz, yz\}$ since $e \notin \{ab, ac, ad\}$
So if $x \in f_{\mathcal{E}}(e)$ then $(f_{\mathcal{E}}(e))(xz) \in E_2^{\mathcal{E}}$. So $(ac)e \in E_1^{\mathcal{E}}$, and, since $e \in B \backslash \{ab\}$, $e = bc$.

Similarly, if $y \in f_{\mathcal{E}}(e)$, $e = bd$
Hence $B \subset \{ab, bc, bd\}$
By a similar argument, $C \subset \{ac, bc, cd\}$ and $D \subset \{ad, bd, cd\}$

3. $V_1 = \{a, b, c, d\}$ and $E_1 \subset \{ab, ac, ad, bc, bd, cd\}$
$a, b, c, d \in V_1$ since $ab, ac, ad \in E_1$
Suppose $v \in V_1 \backslash \{a, b, c, d\}$
Then, since G_1 is connected, $\exists v_1, \ldots v_n \in V_1$ such that $v_1 \in \{a, b, c, d\}$, $v_n = v$ and $\forall 1 \leq i < n : v_i v_{i+1} \in E_1$
Let j be the maximum value in $1 \ldots n-1$ such that $v_j \in \{a, b, c, d$. This maximum exists since $v_1 \in \{a, b, c, d\}$, and $v_n = v \notin \{a, b, c, d\}$
Now $v_j v_{j+1} \in E_1$ since $\forall 1 \leq i < n : v_i v_{i+1} \in E_1$, and $v_j v_{j+1} \in A \cup B \cup C \cup D$, since $v_j \in \{a, b, c, d\}$, but $v_j v_{j+1} \notin A \cup B \cup C \cup D$, since $A \cup B \cup C \cup D \subset \{ab, ac, ad, bc, bd, cd\}$ and $v_{j+1} \notin \{a, b, c, d\}$
But this contradication means that $V_1 \backslash \{a, b, c, d\} = \emptyset$, and, since $a, b, c, d \in V_1$, $V_1 = \{a, b, c, d\}$.
Moreover, $E_1 \subset \{vw \mid v, w \in V_1\} = \{ab, ac, ad, bc, bd, cd\}$.

4. If $bc \in E_1$ or $bd \in E_1$ or $cd \in E_1$, then $G_1 \cong G_2$
If $bc \in E_1$, then $(bc)(ab), (bc)(ac) \in E_1^{f_\varepsilon}$ and so $(f_{\mathcal{E}}(bc))(xy), (f_{\mathcal{E}}(bc))(xz) \in E_2^{f_\varepsilon}$
So $x \in f_{\mathcal{E}}(bc)$ or $(f_{\mathcal{E}}(bc)) = yz$
Now $f_{\mathcal{E}}(bc) \neq yz$, since $bc \neq ad$ So $x \in f_{\mathcal{E}}(bc)$, but $f_{\mathcal{E}}(bc) \notin \{xy, xz\}$, since $bc \notin \{ab, ac\}$ So $\exists w \in V_2 \{x, y, z\} : f_{\mathcal{E}}(bc) = xw$
Similarly, if $bd \in E_1$, then $\exists w' \in V_2 : f_{\mathcal{E}}(bd) = yw'$
Moreover, if $bc, bd \in E_2$, and if $f_{\mathcal{E}}(bc) = xw$ and $f_{\mathcal{E}}(bd) = yw'$, then $w = w'$
For suppose $bc, bd \in E_1$. Then $(bc)(bd) \in E_1^{\mathcal{E}}$ So $(xw)(yw') \in E_2^{\mathcal{E}}$ and so $w = w'$
So if $bc, bd \in E_1$, then $\exists w \in V_2 : f_{\mathcal{E}}(bc) = xw$ and $f_{\mathcal{E}}(bd) = yw$.
Similarly, if $bc, cd \in E_2$ then $\exists w \in V_2 : f_{\mathcal{E}}(bc) = xw$ and $f_{\mathcal{E}}(cd) = zw$ and if $bd, cd \in E_2$ then $\exists w \in V_2 : f_{\mathcal{E}}(bc) = yw$ and $f_{\mathcal{E}}(cd) = zw$ and if $bc, bd, cd \in E_2$ then $\exists w \in V_2 : f_{\mathcal{E}}(bc) = xw$, $f_{\mathcal{E}}(bc) = yw$ and $f_{\mathcal{E}}(cd) = zw$
Now $E_1 \subset \{ab, ac, ad, bc, bd, cd\}$, and $\{ab, ac, ad\} \subset E_1$
Suppose $bc \in E_1$ and $bd, cd \notin E_1$. That is, suppose $E_1 = \{ab, ac, ad, bc\}$.
Then $E_1^{f_\varepsilon} = \{xy, xz, yz, xw\} = E_2$
Also $V_2 = \{x, y, z, w\} = \{\alpha \in e \mid e \in E_2\}$, since G_2 is connected.
So $G_1 = (\{a, b, c, d\}, \{ab, ac, ad, bc\})$ and $G_2 = (\{x, y, z, w\}, \{xy, xz, xw, yz\})$, and the function $f : a \mapsto x, b \mapsto y, c \mapsto z, d \mapsto w$ is a bijective function that preserves the edges of G_1 and so $G_1 \cong G_2$
Similarly, if any one of bd, cd is in E_1 then $G_1 \cong G_2$.
Now suppose $bc, bd \in E_1$, but $cd \notin E_1$. That is, suppose $E_1 = \{ab, ac, ad, bc, bd\}$.
Then $E_1^{f_\varepsilon} = \{xy, xz, yz, xw, yw\} = E_2$, and $V_2 = \{x, y, z, w\}$
So $G_1 = (\{a, b, c, d\}, \{ab, ac, ad, bc, bd\})$ and $G_2 = (\{x, y, z, w\}, \{xy, xz, xw, yz, yw\})$ and again the function $f : a \mapsto x, b \mapsto y, c \mapsto z, d \mapsto w$ is a bijective function that preserves the edges of G_1 and so $G_1 \cong G_2$
Similarly, if bc, cd or if bd, cd are in E_1 then $G_1 \cong G_2$.
Finally, if $E_1 = \{ab, ac, ad, bc, bd, cd\}$, then $E_1^{f_\varepsilon} = \{xy, xz, xw, yz, yw, zw\} = E_2$ and once again the function $f : a \mapsto x, b \mapsto y, c \mapsto z, d \mapsto w$ is a bijective function that preserves the edges of G_1 and so $G_1 \cong G_2$

Hence if $\exists a, b, c, d \in V_1; x, y, z \in V_2 : f_{\mathcal{E}} : ab \mapsto xy, ac \mapsto xz, ad \mapsto yz$
then either $\exists w : V_2 = \{x, y, z, w\}$ and $G_1 \cong G_2$,
or $V_2 = \{x, y, z\}, G_2 = (\{x, y, z\}, \{xy, xz, yz\})$ and $G_1 = (\{a, b, c, d\}, \{ab, ac, ad\})$.

4.5 Graph isomorphism and edge graph isomorphism

Theorem 1. *Two graphs are isomorphic if their edge graphs are isomorphic and if they have equal numbers of K_3 components.*

Let $G = (V, E)$ and $G' = (V', E')$ be two simple graphs with no isolated vertices.

If $G^{\mathcal{E}} \cong (G')^{\mathcal{E}}$ and
if $\mid \{H : H$ is a K_3 subgraph of $G\} \mid \; = \; \mid \{H' : H'$ is a K_3 subgraph of $G'\} \mid$,
then $G \cong G'$.

Proof Let $(G_1^{\mathcal{E}} = (V_1^{\mathcal{E}}, E_1^{\mathcal{E}}), \ldots, G_n^{\mathcal{E}} = (V_n^{\mathcal{E}}, E_n^{\mathcal{E}}))$ partition $G^{\mathcal{E}}$ into its connected components, and let $((G_1')^{\mathcal{E}} = ((V_1')^{\mathcal{E}}, (E_1')^{\mathcal{E}}), \ldots, (G'_m)^{\mathcal{E}}_m = ((V'_m)^{\mathcal{E}}, (E'_m)^{\mathcal{E}}))$ partition $(G')^{\mathcal{E}}$ into its connected components.

Then, since $G^{\mathcal{E}} \cong (G')^{\mathcal{E}}$, $m = n$ and there is a permutation p of $(1, \ldots n)$ such that $\forall 1 \leq i \leq n : G_i^{\mathcal{E}} \cong (G'_{p(i)})^{\mathcal{E}}$

Let p be a permutation of $(1, \ldots n)$ such that $\forall 1 \leq i \leq n : G_i^{\mathcal{E}} \cong (G'_{p(i)})^{\mathcal{E}}$

Then $\forall 1 \leq i \leq n, G_i \cong G'_{p(i)}$, or $G_i = (\{a, b, c, d\}, \{ab, ac, ad\})$ and $(G_{p(i)})') = (\{x, y, z\}, \{xy, xz, yz\})$, or $G_i = (\{a, b, c\}, \{ab, ac, cd\})$ and $(G'_{p(i)}) = (\{x, y, z, w\}, \{xy, xz, xw\})$, since $G_i^{\mathcal{E}} \cong (G'_{p(i)})^{\mathcal{E}}$

That is, either $G_i \cong G'_{p(i)}$, or else G_i is a K_3 graph and $G'_{p(i)}$ is an S_4 graph, or vice versa.

Let $k = \mid \{G_i \mid G_i$ is a K_3 graph and $G_i \cong G_i'\} \mid$
and let $l = \mid \{G_i \mid G_i$ is a K_3 graph and $G_i \ncong G_i'\}$

Now $k = \mid \{G_i \mid G_i$ is a K_3 graph and $G_i \cong G_i'\} \mid$
$= \mid \{G_i' \mid G_i'$ is a K_3 graph and $G_i \cong G_i'\} \mid$

And so $l = \{G_i' \mid G_i'$ is a K_3 graph and $G_i \ncong G_i'\}$, since $k + l = \mid \{1 \leq i \leq n :$ G_i is a K_3 graph $\} \mid \; = \; \mid \{1 \leq i \leq n : G_i'$ is a K_3 graph $\} \mid$,

Hence $\mid \{G_i \mid G_i$ is K_3 and G_i' is $S_4\} \mid = \mid \{G_i' \mid G_i'$ is K_3 and G_i is $S_4\} \mid$

So we can define a permutation q of $1 \ldots n$ such that

- $q(i) = p(i)$ if $G_i \cong G_i'$
- $q(i) = p(j)$ and $q(j) = p(i)$ for some j such that G_i is K_3 and $G'_{p(i)}$ is S_4, and G_j is S_4 and $G'_{p(j)}$ is K_3.

But this means that $(G_1 = (V_1, E_1), \ldots, G_n = (V_n, E_n))$ is a partition of G into its connected components, and $(G_1' = (V_1', E_1'), \ldots, G_n' = (V_n', E_n'))$ is a partition of G' into its connected components, and q is a permutation of $1 \ldots n$ such that $G_i \cong G_i'$ for all $1 \leq i \leq n$, and so $G \cong G'$

Theorem 2. *Let $G = (V, E)$ and $G' = (V', E')$ be two simple graphs.*
Let $V_I = \{v \in V \mid v$ is isolated in $G\}$ and $V_I' = \{v \in V' \mid v$ is isolated in $G'\}$.
Then $G \cong G'$ iff

– $\mid V_I \mid = \mid V' \mid$.
– $\mid \{H : H \textit{ is a } K_3 \textit{ subgraph of } G\} \mid \; = \; \mid \{H' : H' \textit{ is a } K_3 \textit{ subgraph of } G'\} \mid$
– $(V \backslash V_I, E)^{\mathcal{E}} \cong (V' \backslash V'_I, E')$

Proof Partition G into into $(V_I, \emptyset)$ and $(V \backslash V_I, E)$ and partition G' into $(V'_I, \emptyset)$ and $(V' \backslash V'_I, E')$.

1. $\Leftarrow$
   $(V_I, \emptyset) \cong (V'_I, \emptyset)$ since $\mid V_I \mid = \mid V' \mid$.
   Also, $(V \backslash V_I, E) \cong (V' \backslash V'_I, E')$ since $(V \backslash V_I, E)^{\mathcal{E}} \cong (V' \backslash V'_I, E')^{\mathcal{E}}$ and since $(V \backslash V_I, E)$ and $(V' \backslash V'_I, E')$ each contain the same number of K_3 subgraphs.
   Hence $G \cong G'$.
2. $\Rightarrow$
   Suppose $G \cong G'$ Let $f : V \to V'$ be a bijective function such that $\{a, b\} \in E \iff \{f(a), f(b)\} \in E'$
   – a is isolated in G iff $f(a)$ is isolated in G' and so $\mid V_I \mid = \mid V' \mid$
   – ab, ac and bc form the edges of a K_3 subgraph of G iff $f(a)f(b)$, $f(a)f(c)$ and $f(b)f(c)$ form the edges of a K_3 subgraph of G'. Hence $(V \backslash V_I, E)$ and $(V' \backslash V'_I, E')$ each contain the same number of K_3 subgraphs.
   – For every $ab \in E$, define $f^{\mathcal{E}} : ab \mapsto f(a), f(b)$
   Then $f^{\mathcal{E}} : E \to E'$ is bijective, and $\forall \{e_1, e_2\} \in E^{\mathcal{E}}$, $f^{\mathcal{E}}(\{e_1, e_2\}) \in (E')^{\mathcal{E}}$
   Hence $(V \backslash V_I, E)^{\mathcal{E}} \cong (V' \backslash V'_I, E')$

5 Application to Structure Generation

If the vertex colouring of each vertex in an edge graph is defined as a triple that comprises the edge and vertex colours from the original graph, then this result can be extended to coloured graphs in a straightforward way. This provides the basis for transforming molecular graphs to simple graphs before passing them to Nauty for isomorphism testing.

The alternative approach – namely to use multiple edges and dummy vertices to represent different bond types – gives rise to smaller graphs (with fewer vertices and edges) when dealing with fully connected structures containing mainly single bonds.

The approach used in Abermol leads to smaller graphs when dealing with structures containing different bond types, or structures that are not fully connected. This is particularly useful during molecular structure generation, when structures are typically not fully connected.

This approach also provides greater flexibility for representing different kinds of bonds. For example, aromatic bonds are represented by an appropriate label, which does not affect the representation of other bond types.

6 Conclusion

The theorems underpinning a practical approach to isomorphism testing were presented above.

Following the principles of orderly enumeration described by McKay [8], Abemol starts with a set of bonds (graph edges) that represent fragments of molecules. It computes all the ways in which the set can be extended by adding one bond. The resulting collection is then reduced so that no two sets represent isomeric molecules. The process is repeated for each of the new sets until no further extension is possible.

At each stage in the process, all the sets of bonds have the same cardinality, so that sets from previous stages cannot represent isomers of previously generated molecules or fragments. At each stage, sets are also grouped to avoid testing structures such as NH_2 and H_2O for isomorphism.

Isomorphism testing was carried out by nauty [6,7]. Since nauty deals with simple vertex-coloured graphs, Abermol transforms the molecular graph so that bonds are represented as coloured vertices, and bonds that share a common atom are connected by simple edges. This requires special treatment of K_3 and S_4 graphs, as exposed by the theorem above.

In this way, the theorems developed above have been used to facilitate a flexible, extensible and intuitively appealing representation of molecular structures, that also allows for efficient identification of duplicate molecules and fragments.

References

1. Faulon, J.-L.: Isomorphism, automorphism partitioning and canonical labelling can be solved in polynomial time for molecular graphs. Journal of Chemical Information and Computer Science, **38** (1998) 432–444
2. Benecke, C., Grund, R., Hohberger, R., Kerber, A., Laue, R., Wieland, Th.: MOLGEN+, a generator of connectivity isomers and stereoisomers for molecular structure elucidation. Anal. Chim. Acta. **314** (1995) 141–147;
   see also the MOLGEN structure generation page: http://www.mathe2.uni-bayreuth.de/molgen4/, accessed June 30, 2014
3. Braun, J., Gugisch, R., Kerber, A., Laue, R., Meringer, M., Rücker. C.: MOLGEN-CID – A Canonizer for Molecules and Graphs Accessible through the Internet. Journal of Chemical Information and Computer Science (2003)
4. Korytko, A., Schulz, K.-P. , Madison, M.S., Munk, M.E.: HOUDINI: A new approach to Computer-based Structure Generation. Journal of Chemical Information and Computer Science. **43** (2003) 1434–1446
5. Faulon, J.-L., Churchwell, C.J.: The Signature Molecular Descriptor. 2. Enumerating Molecules from their Extended Valence Sequences. Journal of Chemical Information and Computer Science. **43** (2003) 721–734
6. McKay, B.D.: Practical graph isomorphism. Congressus Numerantum **30** (1981) 45–87
7. McKay, B.D.: The nauty User's Guide (Version 2.2). The nauty page: http://cs.anu.edu.au/people/bdm/nauty/, accessed June 30, 2014
8. McKay, B.D.: Isomorph-free exhaustive generation. Journal of Algorithms **26** (1998) 306–324

Towards distributed bigraphical reactive systems

Distributed computing of bigraphical embeddings*

Alessio Mansutti Marco Peressotti Marino Miculan

Laboratory of Models and Applications of Distributed Systems,
Department of Mathematics and Computer Science, University of Udine, Italy
alessio.mansutti@gmail.com, {marco.peressotti, marino.miculan}@uniud.it

Abstract. The bigraph embedding problem is crucial for many results and tools about bigraphs and bigraphical reactive systems (BRS). There are algorithms for computing bigraphical embedding but these are designed to be run locally and assume a complete view of the guest and host bigraphs, putting large bigraphs and BRS out of their reach. To overcome these limitations we present a *decentralized algorithm* for computing bigraph embeddings that allows us to distribute both state and computation over several concurrent processes. Among various applications, this algorithm offers the basis for distributed BRS simulations where non-interfering reactions are carried out concurrently.

1 Introduction

Bigraphical Reactive Systems (BRSs) [10, 15] are a flexible and expressive metamodel for ubiquitous computation. In the last decade, BRSs have been successfully applied to the formalization of a wide range of domain-specific calculi and models, from traditional programming languages to process calculi for concurrency and mobility, from business processes to systems biology; a non exhaustive list is [1, 3, 4, 6, 12, 13]. Recently, BRSs have found a promising applications in *structure-aware agent-based computing:* the knowledge about the (physical) world where the agents operate (e.g., drones, robots, etc.) can be conveniently represented by means of BRSs [16, 20]. BRSs are appealing also because they provide a range of general results and tools, which can be readily instantiated with the specific model under scrutiny: simulation tools, systematic construction of compositional bisimulations [10], graphical editors [7], general model checkers [18], modular composition [17], stochastic extensions [11], etc.

This expressive power stems from the rich structure of *bigraphs*, which form the states of a bigraphic reactive system. A bigraph is a compositional data structure describing at once both the locations and the logical connections of (possibly nested) components of a system. To this end, bigraphs combine two independent graphical structures over the same set of *nodes*: a hierarchy of *places*, and a hypergraph of *links*. Intuitively, places can be used for representing physical positions of agents, while links represent logical connections between agents. A simple example is shown in Figure 1.

* Work partially supported by MIUR PRIN project 2010LHT4KM, *CINA*.

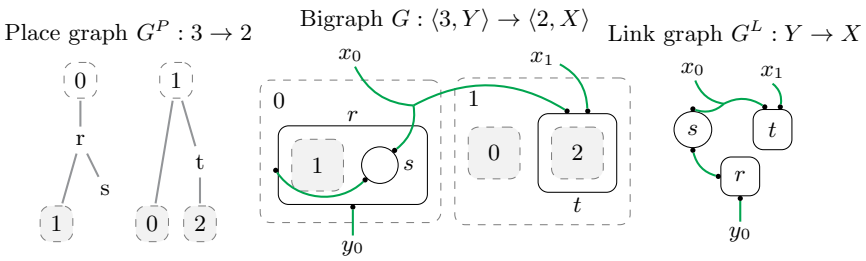


Fig. 1. Forming a bigraph from a place graph and a link graph.

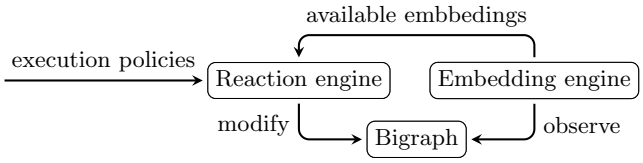


Fig. 2. An abstract bigraphical machine.

Like graph rewriting [19], the behaviour of a BRS is defined by a set of *(parametric) reaction rules*, which can modify a bigraph by replacing a *redex* with a *reactum*, possibly changing agents' positions and connections.

Bigraphical reactive systems can be run (or simulated) by the abstract machine depicted in Figure 2 (or variants of it). This machine is composed by two main modules: the *embedding engine* and the *reaction engine*. The former is responsible of keeping track of every occurrence of the redexes into the machine state. The latter is responsible of carrying out the reactions, in two steps: (a) choosing an occurrence of a redex among those provided by the embedding engine and (b) updating the machine state by performing the chosen rewrite operation. The selection of the reaction is driven by user-provided execution policies.

Therefore, computing bigraph embeddings is a central issue in any implementation of a BRS abstract machine. The problem is known to be NP-complete [2], and some algorithms (or reductions) can be found in the literature [8, 14, 21]. However, existing algorithms assume a complete view of both the guest and the host bigraphs. This hinders the scalability of BRS execution tools, especially on devices with low resources (like embedded ones). Moreover, in a truly distributed setting (like in multi-agent systems [12]) the bigraph is scattered among many machines; gathering it to a single "knowledge manager" in order to calculate embeddings and apply the rewriting rules, would be impractical.

In this paper, we aim to overcome these problems, by introducing an algorithm for computing bigraphical embeddings in distributed settings where bigraphs are spread across several cooperating processes. This decentralized algorithm does not impose a complete view of the host bigraph, but retains the fundamental property of (eventually) computing every possible embedding for

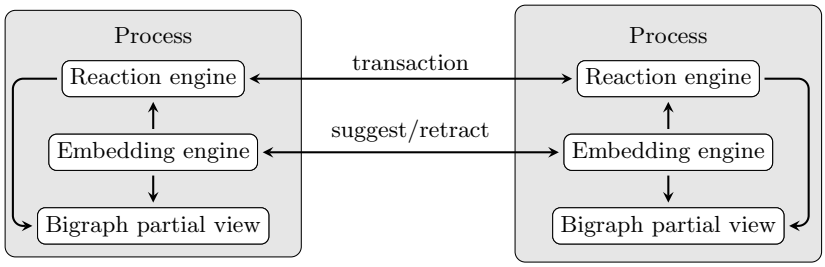


Fig. 3. Distributed bigraphical machine.

the given host. Thanks to the decentralized nature of the algorithm, this solution can scale to bigraphs that cannot fit into the memory of a single process, hence too large to be handled by existing implementations. Moreover, the algorithm is parallelized: several (non-interfering) reductions can be identified and applied at once. In this paper we consider distributed hosts only since guests are usually redexes of parametric reaction rules and hence small enough to be handled even in presence of scarce computational resources.

Thanks to this result we are able to define a decentralized variation of the abstract bigraphical machine illustrated above. The architecture of this new *distributed bigraphical machine* is sketched in Figure 3. Both computation and states are distributed over a family of processes. Each process has only a partial view of the global state and negotiates updates to its piece of the global bigraph with its "neighbouring processes". In order to simplify the exposition we assume reliable asynchronous point-to-point communication between reliable processes. These are mild assumptions for a distributed system and can be easily achieved e.g. over unreliable channels.

Synopsis In Section 2 we briefly recall the notion of bigraphs and bigraphical reactive systems. In Section 3 we recall the notion of bigraph embedding, introduce the notion of *partial embedding* and study their ordering (which are at the base of our algorithm). In Section 4 and Section 5 we describe the distributed bigrapical machine and its components; especially the distributed algorithm for solving the embedding problem at the core of this paper. Conclusions and final remarks are discussed in Section 6.

2 Bigraphical reactive systems

In this section we briefly recall the notion of Bigraphical Reactive Systems (BRS) referring the interested reader to [15]. The key point of BRSs is that "the model should consist in some sort of reconfigurable space". Agents may interact in this space, even if they are spatially separated. This means that two agents may be adjacent in two ways: they may be at the same *place*, or they may be connected by a *link*. This leads to the definition of *bigraphs* as a data structure

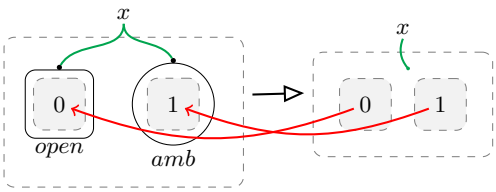


Fig. 4. Open reaction rule of the Ambient Calculus.

for representing the state of the system. A bigraph can be seen as an enriched hyper-graph combining two independent graphical structures over the same set of *nodes*: a hierarchy of *places*, and a hyper-graph of *links*.

Definition 1 (Bigraph [15, Def. 2.3]). *Let Σ be a bigraphical signature (i.e. a set of types, called* controls*, denoting a finite arity). A* bigraph *G over Σ is an object $(V_G, E_G, \mathrm{ctrl}_G, \mathrm{prnt}_G, \mathrm{link}_G) : \langle m_G, X_G\rangle \to \langle n_G, Y_G\rangle$ composed of two substructures (cf. Figure 1): a* place graph *$G^P = (V_G, \mathrm{ctrl}_G, \mathrm{prnt}_G) : m_G \to n_G$ and a* link graph *$G^L = (V_G, E_G, \mathrm{ctrl}_G, \mathrm{link}_G) : X_G \to Y_G$. The set V_G is a finite set of nodes and to each of them is assigned a control in Σ by the* control map *$\mathrm{ctrl}_G : V_G \to \Sigma$. The set E_G is a finite set of names called* edges*.*

These structures present an inner interface (composed by m_G and X_G) and an outer one (n_G, Y_G) along which can be composed with other of their kind as long as they do not share any node or edge. In particular, X_G and Y_G are finite sets of names and m_G and n_G are finite ordinals.

On the side of G^P, nodes, sites and roots are organized in a forest described by the parent map *$\mathrm{prnt}_G : V_G \uplus m_G \to V_G \uplus n_G$ s.t. sites are leaves and roots are n_G.*

On the side of G^L, nodes, edges and names of the inner and outer interface forms a hyper-graph described by the link map *$\mathrm{link}_G : P_G \uplus X_G \to E_G \uplus Y_G$ which is a function from X_G and ports P_G (i.e. elements of the finite ordinal associated to each node by its control) to edges E_G and names in Y_G.*

The dynamic behaviour of a system is described in terms of *reactions* of the form $a \twoheadrightarrow a'$ where a, a' are agents, i.e. bigraphs with inner interface $\langle 0, \emptyset\rangle$. Reactions are defined by means of graph rewrite rules, which are pairs of bigraphs (R_L, R_R) equipped with a function η from the sites of R_R to those of R_L called *instantiation rule*. A bigraphical encoding for the open reaction rule of the Ambient Calculus is shown in Figure 4 where redex and reactum are the bigraph on the left and the one on the right respectively and the instantiation rule is drawn in red. A rule fires when its redex can be embedded into the agent; then, the matched part is replaced by the reactum and the parameters (i.e. the substructures determined by the redex sites) are instantiated accordingly with η.

3 Partial bigraph embeddings

The following definitions are mainly taken from [9], with minor modification to simplify the presentation of the distributed embedding algorithm (cf. Section 5).

As usual, we will exploit the orthogonality of the link and place graphs, by defining *link and place graph embeddings* separately and then combine them to extend the notion to bigraphs. We then introduce *partial bigraph embeddings*, define an ordering on them and study the atomic $CPO_\perp$ structure presented by the set of partial embeddings for any given pair of guest and host bigraphs. This structure is fundamental for the algorithm we present in Section 5.

Link graph Intuitively an embedding of link graphs is a structure preserving map from one link graph (the *guest*) to another (the *host*). As one would expect from a graph embedding, this map contains a pair of injections: one for the nodes and one for the edges (i.e., a support translation). The remaining of the embedding map specifies how names of the inner and outer interfaces should be mapped into the host link graph. Outer names can be mapped to any link; here injectivity is not required since a context can alias outer names. Dually, inner names can mapped to hyper-edges linking sets of points in the host link graph and such that every point is contained in at most one of these sets.

Definition 2 (Link graph embedding [9, Def 7.5.1]). *Let $G : X_G \to Y_G$ and $H : X_H \to Y_H$ be two concrete link graphs. A* link graph embedding *$\phi : G \hookrightarrow H$ is a map $\phi \triangleq \phi^{\mathsf{v}} \uplus \phi^{\mathsf{e}} \uplus \phi^{\mathsf{i}} \uplus \phi^{\mathsf{o}}$ (assigning nodes, edges, inner and outer names respectively) subject to the following conditions:*

(LGE-1) *$\phi^{\mathsf{v}} : V_G \rightarrowtail V_H$ and $\phi^{\mathsf{e}} : E_G \rightarrowtail E_H$ are injective;*
(LGE-2) *$\phi^{\mathsf{i}} : X_G \rightarrowtail \wp(X_H \uplus P_H)$ is fully injective: $\forall x \neq x' : \phi^{\mathsf{i}}(x) \cap \phi^{\mathsf{i}}(x') = \emptyset$;*
(LGE-3) *$\phi^{\mathsf{o}} : Y_G \to E_H \uplus Y_H$ in an arbitrary partial map;*
(LGE-4) *$\mathrm{img}(\phi^{\mathsf{e}}) \cap \mathrm{img}(\phi^{\mathsf{o}}) = \emptyset$ and $\mathrm{img}(\phi^{\mathsf{i}}) \cap \mathrm{img}(\phi^{\mathsf{port}}) = \emptyset$;*
(LGE-5) *$\phi^{\mathsf{p}} \circ \mathrm{link}_G^{-1}\big|_{E_G} = \mathrm{link}_H^{-1} \circ \phi^{\mathsf{e}}$;*
(LGE-6) *$\mathrm{ctrl}_G = \mathrm{ctrl}_H \circ \phi^{\mathsf{v}}$;*
(LGE-7) *$\forall p \in X_G \uplus P_G : \forall p' \in (\phi^{\mathsf{p}})(p) : (\phi^{\mathsf{h}} \circ \mathrm{link}_G)(p) = \mathrm{link}_h(p')$*

where $\phi^{\mathsf{p}} \triangleq \phi^{\mathsf{i}} \uplus \phi^{\mathsf{port}}$, $\phi^{\mathsf{h}} \triangleq \phi^{\mathsf{e}} \uplus \phi^{\mathsf{o}}$ and $\phi^{\mathsf{port}} : P_G \rightarrowtail P_H$ is $\phi^{\mathsf{port}}(v, i) \triangleq (\phi^{\mathsf{v}}(v), i)$).

The first three conditions are on the single sub-maps of the embedding. Condition (LGE-4) ensures that no components (except for outer names) are identified; condition (LGE-5) imposes that points connected by the image of an edge are all covered. Finally, conditions (LGE-6) and (LGE-7) ensure that the guest structure is preserved i.e. node controls and point linkings are preserved.

Place graph Like link graph embeddings, place graph embeddings are just a structure preserving injective map from nodes along with suitable maps for the inner and outer interfaces. In particular, a site is mapped to the set of sites and nodes that are "put under it" and a root is mapped to the host root or node that is "put over it" splitting the host place graphs in three parts: the guest image, the context and the parameter (which are above and below the guest image).

Definition 3 (Place graph embedding [9, Def 7.5.4]). *Let $G : n_G \to m_G$ and $H : n_H \to m_H$ be two concrete place graphs. A* place graph embedding *$\phi : G \hookrightarrow H$ is a map $\phi \triangleq \phi^{\mathsf{v}} \uplus \phi^{\mathsf{s}} \uplus \phi^{\mathsf{r}}$ (assigning nodes, sites and regions respectively) subject to the following conditions:*

(PGE-1) $\phi^{\mathsf{v}} : V_G \rightarrowtail V_H$ *is injective;*
(PGE-2) $\phi^{\mathsf{s}} : n_G \rightarrowtail \wp(n_H \uplus V_H)$ *is fully injective;*
(PGE-3) $\phi^{\mathsf{r}} : m_G \to V_H \uplus m_H$ *in an arbitrary map;*
(PGE-4) $\mathrm{img}(\phi^{\mathsf{v}}) \cap \mathrm{img}(\phi^{\mathsf{r}}) = \emptyset$ *and* $\mathrm{img}(\phi^{\mathsf{s}}) \cap \mathrm{img}(\phi^{\mathsf{v}}) = \emptyset$*;*
(PGE-5) $\forall r \in m_G : \forall s \in n_G : \mathrm{prnt}^{*}_{H}\phi^{\mathsf{r}}(r) \cap \phi^{\mathsf{s}}(s) = \emptyset$*;*
(PGE-6) $\phi^{\mathsf{c}} \circ \mathrm{prnt}_{G}^{-1}\big|_{V_G} = \mathrm{prnt}_{H}^{-1} \circ \phi^{\mathsf{v}}$*;*
(PGE-7) $\mathrm{ctrl}_G = \mathrm{ctrl}_H \circ \phi^{\mathsf{v}}$*;*
(PGE-8) $\forall c \in n_G \uplus V_G : \forall c' \in \phi^{\mathsf{c}}(c) : (\phi^{\mathsf{f}} \circ \mathrm{prnt}_G)(c) = \mathrm{prnt}_H(c')$*;*

where $\phi^{\mathsf{f}} \triangleq \phi^{\mathsf{v}} \uplus \phi^{\mathsf{r}}$ *and* $\phi^{\mathsf{c}} \triangleq \phi^{\mathsf{v}} \uplus \phi^{\mathsf{s}}$.

Conditions in the above definition follows the structure of Definition 2, the main notable difference is (PGE-5) which states that the image of a root can not be the descendant of the image of another. Conditions (PGE-1), (PGE-2) and (PGE-3) are on the three sub-maps composing the embedding; conditions (PGE-4) and (PGE-5) ensure that no components are identified; (PGE-6) imposes surjectivity on children and the last two conditions require the guest structure to be preserved by the embedding map.

Bigraph Finally, bigraph embeddings can now be defined as maps being composed by an embedding for the link graph with one for the place graph consistently with the interplay of these two substructures. In particular, the interplay is captured by a single additional condition ensuring that points in the image of an inner names reside in the parameter defined by the place graph embedding (i.e. are inner names or ports of some node under a site image).

Definition 4 (Bigraph embedding [9, Def 7.5.14]). *Let* $G : \langle n_G, X_G\rangle \to \langle m_G, Y_G\rangle$ *and* $H : \langle n_H, X_H\rangle \to \langle m_H, Y_H\rangle$ *be two concrete bigraphs. A* bigraph embedding $\phi : G \hookrightarrow H$ *is a map given by a place graph embedding* $\phi^{\mathsf{P}} : G^P \hookrightarrow H^P$ *and a link graph embedding* $\phi^{\mathsf{L}} : G^L \hookrightarrow H^L$ *subject to the consistency condition:*

(BGE-1) $\mathrm{img}(\phi^{\mathsf{i}}) \subseteq X_H \uplus \{(v, i) \in P_H \mid \exists s \in n_G : k \in \mathbb{N} : \mathrm{prnt}^{k}_{H}(v) \in \phi^{\mathsf{s}}(s)\}$.

Partial bigraph embeddings In the following we relax the above definition to allow partiality and formally represent intermediate steps of the algorithm we present in Section 5. Basically, a partial bigraph embedding is a partial map subject to the same conditions of a total embedding up-to partiality.

Definition 5 (Partial bigraph embedding). *Let* $G : \langle n_G, X_G\rangle \to \langle m_G, Y_G\rangle$ *and* $H : \langle n_H, X_H\rangle \to \langle m_H, Y_H\rangle$ *be two concrete bigraphs. A* partial bigraph embedding $\phi : G \rightharpoonup H$ *is a partial map subject, where defined, to the same conditions of Definition 4.*

Partial embeddings represent partial or intermediate steps towards a total embedding. This is reflected by the obvious ordering given by the point-wise lifting of the anti-chain order to partial maps. In particular, given two partial embeddings $\phi, \psi : G \rightharpoonup H$ we say that:

$$\phi \sqsubseteq \psi \overset{\triangle}{\iff} \forall x \in \mathrm{dom}(\phi)\, \phi(x) \neq \bot \implies \psi(x) = \phi(x). \tag{1}$$

This definition extends, for any given pair of concrete bigraphs G and H, to a partial order over the set of partial bigraph embeddings of G into H. It is easy to check that the entirely undefined embedding $\varnothing$ is the bottom of this structure and that meets are always defined:

$$\phi \sqcap \psi \triangleq \lambda x. \begin{cases} \phi(x) & \text{if } \phi(x) = \psi(x) \\ \bot & \text{otherwise} \end{cases}$$

Likewise, joins, where they exist, are defined as follows:

$$\phi \sqcup \psi \triangleq \lambda x. \begin{cases} \phi(x) & \text{if } \phi(x) \neq \bot \\ \psi(x) & \text{if } \psi(x) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Clearly ϕ and ψ have to coincide where are both defined and their join $\phi \sqcup \psi$ is defined iff it meets every condition in Definition 5.

4 State, overlay and reactions

This section illustrates how a bigraph is distributed between a processes family and how it is maintained and updated. Firstly, we formalize the idea of a "bigraph being distributed" and show how a partition of the system global state defines a semantic overlay network. The rôle of this network is crucial for the embedding algorithm since communication will follow this semantic driven structure. Finally, we describe how reactions are carried out concurrently and consistently.

In the following, let Proc denote the family of processes forming the distributed machine under definition and let G be a generic concrete bigraph $(V_G, E_G, \mathrm{ctrl}_G, \mathrm{prnt}_G, \mathrm{link}_G) : \langle m_G, X_G \rangle \to \langle n_G, Y_G \rangle$ over a given signature Σ.

State partition Intuitively, a partition of the shared state G is a map assigning each component of the bigraph G to the process in charge of maintaining it.

Definition 6 (State partition). *A partition of (the shared state) G over* Proc *is a map $\mathbb{P} : G \to \mathsf{Proc}$ assigning each component of G to some process. In particular, $\mathbb{P}$ is given by the (sub)maps $\mathbb{P}^{\mathsf{v}}$, $\mathbb{P}^{\mathsf{e}}$, $\mathbb{P}^{\mathsf{s}}$, $\mathbb{P}^{\mathsf{r}}$, $\mathbb{P}^{\mathsf{i}}$, and $\mathbb{P}^{\mathsf{o}}$ on vertices, edges, sites, roots, inner names, and outer names respectively. Every component of G in the pre-image of a process is said to be* held *by that process. Ports are mapped into the process holding their node i.e. $\mathbb{P}((v, i)) \triangleq \mathbb{P}(v)$.*

Then, the notion of adjacency for bigraph components can be lifted to the family of processes along the given partition map. Here hyper-edges of the link graph are considered as trees where the root and leaves are the hyper-edge handle (i.e. edge or outer name) and all the points (i.e. ports or inner names) it connects.

Definition 7 (Adjacent processes). *Let $R, S \in \mathsf{Proc}$. The process R is said to be* adjacent (w.r.t. the partition $\mathbb{P}$) to S *whenever one of the following holds:*

(ADJ-P) *there exists a node or site* c *s.t.* $\mathbb{P}(c) = R$ *and* $\mathbb{P}(\mathrm{prnt}_G(c)) = S$*;*
(ADJ-L) *there exists a point* p *s.t.* $\mathbb{P}(p) = R$ *and* $\mathbb{P}(\mathrm{link}_G(p)) = S$*;*
(ADJ-R) *there exist two roots* r, r' *s.t.* $\mathbb{P}(r) = R$ *and* $\mathbb{P}(r') = S$*;*
(ADJ-H) *there exist two handles* h, h' *s.t.* $\mathbb{P}(h) = R$ *and* $\mathbb{P}(h') = S$*.*

In virtue of the adjacency being a symmetric relation, we will denote pairs of adjacent processes by $R \overset{\mathbb{P}}{\circ\!\!-\!\!\circ} S$ *and drop the partition when confusion seems unlikely. Two (partial) embeddings or a process and a (partial) embedding are said to be adjacent whenever their images are. The notation is extended accordingly.*

The adjacency relation defines an undirected graph with vertices in Proc and hence an *overlay network* $\mathsf{N}_{\mathbb{P}}$. The overlay network bares a specific semantic meaning since it reflects the adjacency of the bigraphical elements held by the processes forming the network: two processes are adjacent if, and only if, they hold components of the shared bigraphs G that are adjacent in G. Moreover, for any two components of G, say c_1 and c_2, the shortest path in the overlay $\mathsf{N}_{\mathbb{P}}$ between the processes $\mathbb{P}(c_1)$ and $\mathbb{P}(c_2)$ will never be greater than the shortest path between c_1 and c_2 in G. The last observation is crucial to our purposes since relates routing through the overlay $\mathsf{N}_{\mathbb{P}}$ with walks and visits of G used e.g. to compute embeddings into G in non-distributed settings. Notice that the restriction of $\mathsf{N}_{\mathbb{P}}$ to $\mathrm{img}(\mathbb{P})$ will always be connected.

Distributed reactions Let ϕ be an embedding of G into the bigraph shared by the process in the system and let $r : G \twoheadrightarrow G'$ be a parametric rewriting rule for the given BRS. Processes holding elements of G image through ϕ or in its parameters have to negotiate the firing of r and coordinate the update of their state. The negotiation phase is related to the specific execution policy and hence is left out from the present work. The update phase involves a distributed transaction and can be easily handled by established algorithms like *two-phase-commit* [5]. The embedding ϕ is selected among those published by the embedding engine however, the distributed transaction is still necessary since this collection of embeddings may be out of sync because of communication delays (cf. Section 5).

5 Distributed embedding

In this Section we present the main result of the paper: a decentralized algorithm for computing bigraphical embeddings in the distributed settings outlined in Section 4. Intuitively, each process running this algorithm maintains a collection of partial embeddings for the guests it has to look for and cooperates with its neighbouring processes (adjacency is lifted from bigraphs to processes) to complete of refute them. For the sake of simplicity we assume that all processes are given the same set of guests (e.g. the redexes of the rules of the underlying BRS) and that this set is fixed over the time; however, the algorithm can be readily adapted to work without these assumptions.

Process structure Each process maintain, for each guest G, a suitable structure Γ_G where it stores the all partial embeddings of G involving its partial view of the shared bigraph. Among these, there are all the total embeddings the process believes available at a current time and which are exposed to the outside system (e.g. the rewriting engine of the distributed bigraphical machine). Partial embeddings are decorated with some extra information to handle the non-monotonic changes of this structure over the life of the process.

Definition 8 (Γ_G). *A model Γ for the guest G is a set of triple (ϕ, B, ts) where:*

- *ϕ is a partial embedding from G to the shared bigraph H;*
- *ts is a logical timestamp composed by the values of the logical clocks of the processes involved in the making of ϕ i.e. those in* $\mathrm{img}(\mathbb{P} \circ \phi)$*;*
- *B: is a boolean value that states if ϕ holds. This is used to implement, together with ts, non-monotonic reasoning (with retracted embeddings).*

Each model Γ_G maintains only the last (according to the function ts) iteration of every partial embedding ϕ. For this reason, we will also sometimes use Γ_G as a partial function from partial embedding to pair $(Bool, \mathsf{Proc} \to \mathbb{N})$, s.t.:

$$\Gamma_G(\phi) = (B, ts) \overset{\triangle}{\iff} (\phi, B, ts) \in \Gamma_G.$$

Finally, we will say that: $\Gamma_G \models \phi \overset{\triangle}{\iff} \exists ts.(\phi, \mathtt{true}, ts) \in \Gamma_G(\phi)$.

The procedure onBigraphViewChanged of a process P is called whenever the portion of the global bigraph held by P is modified. Updates define ticks in the logical clock *P.time* held by each process. Moreover, updates may invalidate some of the (partial) embeddings computed so far by the process and render new embeddings available. The first have to be retracted and the seconds have to be suggested to the nearby processes. Clearly, processes can see directly only the side effects of updates on embeddings that are "local" to them.

Definition 9 (Local embedding). *Let $\phi : G \hookrightarrow H$ be a partial embedding and let $\mathbb{P} : H \to \mathsf{Proc}$ be a partition. The* owners *of ϕ are the processes in* $\mathrm{img}(\mathbb{P} \circ \phi)$*. If ϕ has exactly one owner then it is said to be* local *to it. We denote the restriction of ϕ to the portion of bigraph held by a set of processes S by $\phi\big|^{\mathbb{P}}_{S}$.*

Local embeddings can be easily computed by the algorithm proposed in [14] with minor modifications to relax the constraints ensuring totality.

Given a process Q, every partial embedding $\psi \sqsubseteq \phi\big|^{\mathbb{P}}_{\{Q\}}$ is local to Q except for the undefined embedding – since $\mathrm{img}(\mathbb{P} \circ \varnothing)$ will always be empty. Therefore, the restriction of ϕ to Q can be read as the largest embedding local to Q that *supports* ϕ and every change in the state held by Q that invalidates this local embedding invalidates also ϕ and hence have to be notified to every process owning ϕ.

It should be noted that, before sending a local embedding (suggest), the process will check for local embeddings that were found in a previous iteration of onBigraphViewChanged and do not appear in the current one: these embeddings

```
Procedure onBigraphViewChanged()
    time ← time + 1
    for G ∈ Guests do
        localEmbeddingsOfG ← getLocalEmbeddings(G)
        foreach (φ, true, ts) ∈ Γ_G s.t. |dom(ts)| = 1 and
        φ ∉ localEmbeddingsOfG do
            send ⟨φ, false, self ↦ time⟩ to self // self retraction
        end
        foreach φ ∈ localEmbeddingsOfG s.t. Γ_G ⊭ φ do
            send ⟨φ, true, self ↦ time⟩ to self // self suggestion
        end
    end
```

are now erroneous and hence retracted by the process. Notice that the process will never send the empty embedding.

In both cases, the message sent is an entry of a model Γ for a guest G, where the timestamp is the partial function defined only on the process holding the partial embedding (hence a pair $P \mapsto time$) and the boolean value included in the message is used to tell *retract* and *suggest* messages apart. These informations offer us a causal ordering between suggestions and retractions and hence the ability to handle non-monotonic reasoning in a system where messages can be received out of order. In our system, a process can compute an embedding ϕ that cannot be used in a reaction, for example if a retracting message about an embedding $\psi \sqsubseteq \phi$ has not been received yet. However, system consistency will be preserved because rewritings are performed inside distributed transitions and hence at least one of the processes that retracted ϕ will abort the transaction.

Retracts onBigraphViewChanged is the only procedure that generate retracted embeddings. Here we will explain why we only need to retract *local embeddings*. After a reaction, each process involved can decide that some embeddings no longer apply. Each process has only a limited knowledge about the system's bigraph, given by the portion assigned to it: for this reason, a process can only see the untruth of an embedding if it maps elements of the guest to elements assigned to that process. More formally, the set of embeddings that the process P can see as false can be written as $\mathcal{R}_P \subseteq \{\phi \mid P \in \mathrm{img}(\mathbb{P} \circ \phi)\}$.

For each embedding ϕ, if a process P is involved in its formation, then there exists at least one local embedding ψ computed by P such that $\psi \sqsubseteq \phi$. Given its local knowledge about the global bigraph, we can conclude that if $\phi \in \mathcal{R}_P$, then there exists $\psi \sqsubseteq \phi$ local to P and such that $\psi \in \mathcal{R}_P$. Therefore, if a process P wants to retract each embedding in $\mathcal{R}_P$, it only needs to retract the subset of its local embeddings $\{\phi \mid \phi \in \mathcal{R}_P \wedge \{P\} = \mathrm{img}(\mathbb{P} \circ \phi)\}$.

Γ-updates When a process P receives a *retraction message* $\langle \phi, \texttt{false}, ts \rangle$[1] such that ϕ occurs in Γ but the occurrence was generated earlier than ts it invali-

[1] Notice that the partial embedding being retracted is local.

```
Procedure retract(G,φ,ts)
    // ts involves exactly one process
    {P} ← dom(ts)
    t' ← 0
    if Γ_G(φ) ≠ ⊥ then // new embedding
        (B', ts') ← Γ_G(φ)
        t' ← ts'(P)
    end
    if ts(P) > t' then
        D ← ∅
        foreach (ψ, B'', ts'') ∈ Γ_G do
            if φ ⊑ ψ ∧ t > ts''(P) then // ⌢ relation
                ts''(P) ← t
                Γ_G(ψ) ← (false, ts'')
                D ← D ∪ {P | self ⊸ P ∧ P ∈ img(ℙ ∘ ψ)}
            end
        end
        send ⟨φ, false, ts⟩ to D
    end
```

dates every ψ in Γ made from ϕ and more recent than ts and then forwards the retraction message to every neighbour process involved by the retraction i.e. appearing in a partial embedding being removed from Γ. Formally, we define a *retract relation* $\frown$ between local and partial embedding:

$$(\phi, P \mapsto \mathrm{t}) \frown (\psi, ts) \overset{\triangle}{\iff} \phi \sqsubseteq \psi \wedge t > ts(P)$$

With this relation, we can define the first updating rule for Γ_G as follows:

$$\frac{(\psi, B, ts) \in \Gamma_G \qquad (\phi, P \mapsto \mathrm{t}) \frown (\psi, ts) \qquad (\psi, \texttt{false}, ts') \in \Gamma'_G}{\Gamma_G \xrightarrow{\langle \phi, \texttt{false}, P \mapsto t \rangle} \Gamma'_R} \quad (\Gamma\text{-UP1})$$

where

$$ts'(x) \triangleq \begin{cases} t, & \text{if } x = P. \\ ts, & \text{otherwise.} \end{cases}$$

and Γ'_G it's equal to Γ_G for each $(\psi, B, ts) \in \Gamma_G$ s.t. $(\phi, P \mapsto time) \not\prec (\psi, ts)$. The procedure retract implements what we have seen so far in this section, and it models rather closely the updating rule (Γ-UP1).

If, instead, P receives a *suggestion message* $\langle \phi, \texttt{true}, ts \rangle$, it needs to update a subset of its embeddings and derive new embeddings. These two actions are implemented respectively by the procedures suggest and combine. Embeddings that need to be updated are all ψ in Γ_G, $\Gamma_G(\psi) = (B', ts')$, such that $(\phi, ts) \smile (\psi, B', ts')$, where the *update relation* $\smile$ is defined as follows:

$$(\phi, ts) \smile (\psi, B', ts') \overset{\triangle}{\iff} \psi \sqsubseteq \phi \wedge \forall P. ts'(P) \leq ts(P) \wedge (B' \implies \exists Q. ts'(Q) < ts(Q))$$

Procedure `suggest`(G,ϕ,ts)
  if $\Gamma_G(\phi) = \bot$ **then** // new embedding
    $\Gamma_G(\phi) \leftarrow (\texttt{true}, ts)$
    `send` $\langle \phi, \texttt{true}, ts \rangle$ `to` $\{P \mid \phi \multimap P\}$
    `combine`(G,ϕ,ts)
  end
  foreach $(\psi, B', ts') \in \Gamma_G$ **do**
    // $\smile$ relation
    if $\psi \sqsubseteq \phi \wedge \forall (P,t) \in ts'\ t \leq ts(P) \ \wedge\ (B' \rightarrow \exists (P,t) \in ts'\ t < ts(P))$ **then**
      $ts' \leftarrow \{(P,t) | (P,t) \in ts \wedge \exists t' (P,t') \in ts'\}$
      $\Gamma_G(\psi) \leftarrow (\texttt{true}, ts')$
      `send` $\langle \psi, \texttt{true}, ts' \rangle$ `to` $\{P \mid \psi \multimap P\}$
      `combine`(G,ψ,ts')
    end
  end

An embedding $\psi \sqsubseteq \phi$ in Γ_G needs to be updated if the time associated to each process via the logical timestamp stored in $\Gamma_G(\psi)$ is lower or equal than its counterpart in the message's timestamp ts. Also, if $\Gamma_G \models \psi$, ψ will only be updated if exists a process P such that $ts(P)$ is strictly greater than the time of P associated with ψ in Γ_G. This last constraint ensures that if a process receives multiple instances of the same *suggesting* message, it will not update Γ_G and send that message to its neighbourhood more than once. We can now defines the Γ_G-update rule for suggesting messages:

$$\frac{(\psi, B', ts') \in \Gamma_G \quad (\phi, ts) \smile (\psi, B', ts') \quad (\psi, true, ts'') \in \Gamma'_R}{\Gamma_G \xrightarrow{(G,\phi,\texttt{true},ts)} \Gamma'_R} \qquad (\Gamma\text{-UP2})$$

where $ts'' = ts\big|_{\mathrm{dom}(ts')}$.

After this update step, each updated embedding will be used to derive new embeddings. Given an updated embedding ϕ from the previous step, processes will search for all ψ such that $\phi \sqcup_\downarrow \psi$, where the relation $\sqcup_\downarrow$, between partial embeddings, is defined as follows:

$$\phi \sqcup_\downarrow \psi \overset{\triangle}{\iff} \psi \not\sqsubseteq \phi \wedge \phi \not\sqsubseteq \psi \wedge \phi \sqcup \psi \text{ is a partial embedding}$$

We can now define the third updating rule for Γ_G, which describes how embeddings are derived:

$$\frac{(\phi, \texttt{true}, ts) \in \Gamma_G \qquad (\psi, \texttt{true}, ts') \in \Gamma_G \qquad \phi \sqcup_\downarrow \psi}{(\phi \sqcup \psi, \texttt{true}, ts'') \in \Gamma_G} \qquad (\Gamma\text{-UP3})$$

where

$$P''(x) = \begin{cases} \max(P(x), P'(x)), & \text{if } \overline{\Gamma}_R \not\models \phi \sqcup \psi; \\ \max(P(x), P'(x), \pi_2(\overline{\Gamma}_R(\phi \sqcup \psi))(x)), & \text{otherwise.} \end{cases}$$

```
Procedure combine(G,φ,ts)
    foreach (ψ, B′, ts′) ∈ Γ_G do
        ρ ← φ ⊔ ψ
        if B′ ∧ ψ ⋢ φ ∧ φ ⋢ ψ ∧ isConsistent(ρ) then // ⊔↓ relation
            ts″ ← {(P,t) | (ts(P) ≠ ⊥ ∨ ts′(P) ≠ ⊥) ∧ t = max(ts(P), ts′(P))}
            if Γ_G(ρ) = ⊥ then
                Γ_G(ρ) ← (true, ts″)
                send ⟨ρ, true, ts″⟩ to {P | ρ ⊸ P}
            else
                (oldB,oldts) ← Γ_G(ρ)
                if oldB then
                    ts″ ← {(P, max(t1, t2)) | (P,t1) ∈ ts″ ∧ (P,t2) ∈ oldts}
                end
                if ∀(P,t) ∈ oldts t ≤ ts″(P) then
                    Γ_G(ρ) ← (true, ts″)
                    send ⟨ρ, true, ts″⟩ to {P | ρ ⊸ P}
                end
            end
        end
    end
```

and $\overline{\Gamma}_R$ is Γ_G before the new derivation of $\phi \sqcup \psi$ (with the previous timestamp for $\phi \sqcup \psi$). This updating rule, if abstracted from the timestamps ts and ts', can be expressed the following cleaner form:

$$\frac{\Gamma_G \models \phi \qquad \Gamma_G \models \psi \qquad \phi \sqcup_{\downarrow} \psi}{\Gamma_G \models \psi \sqcup \phi}$$

Each embedding updated or derived from a suggestion message will update Γ and will be sent to the process neighbourhood restricted to those processes that are considered adjacent to the embedding i.e. those holding some component of the shared bigraph that is adjacent (in the sense of Definition 7) to the image of the partial embedding. Embeddings that are completed are exposed to the outer system contextually to the update of Γ.

To make our set of updating rules complete, we need two additional rules to manage incoming messages carrying embeddings that were never seen before (i.e. $\Gamma_G(\phi) = \bot$). This two rules are implemented by retract and suggest procedures with minor changes w.r.t. the case of (Γ-UP1) and (Γ-UP2).

6 Conclusions and future work

In this paper we have presented an algorithm for computing bigraph embeddings in a distributed environment where the host bigraph is spread across several cooperating processes. Differently from existing algorithms [8,14,21], this algorithm is completely decentralized and does not require any process in the system to have a complete view of the global state, hence it can scale to handle bigraphs too

large to reside in the memory of a single process/machine. Moreover, embeddings that are not affected by a reaction are not recomputed and in general the computation of an embedding requires a number of messages that is linearly bounded by the size of the embedded bigraph. However, the overall network impact is not negligible and, in the worst case, can be outperformed by the "semi-distributed" algorithm proposed in [12] where processes visit the shared bigraph (the visit is guaranteed to be minimal by the use of IPOs) and compute embeddings locally using the information gathered. Fortunately there is room for improvement for the algorithm proposed: suggestion and retraction messages can be grouped and compressed by suitable representations since the combinatoric explosion is due to symmetries and isomorphisms between local partial embeddings. Moreover, symbolic representations can be put in place to further reduce the communication footprint of the algorithm. We leave this developments for the extended version of the paper and future works.

The distributed embedding algorithm is the basic block of the *distributed bigraphical machine*, a distributed instance of the abstract bigraph machine. This machine inherits the benefits of the decentralized algorithm, e.g. its scalability.

A direct application of the distributed embedding algorithm is to simulate, or execute, multi-agent systems. In [12] the authors devise a methodology for design and prototype multi-agent systems with BRS. Intuitively, the application domain is modelled by a BRS and entities in its states are divided as "subjects" and "objects" depending on their ability to actively perform actions. Subjects are precisely the agents of the system and reactions are reconfigurations. This observation yields a coherent way to partition and distribute a bigraph among the agents, which can be assimilated to the processes of the distributed bigraphical machine (execution policies are defined by agents desires and goals). Therefore, these agents can find and perform bigraph rewritings in a truly concurrent, distributed fashion, by using the distributed embedding algorithm.

How the bigraph is partitioned and distributed can affect the performance of the system. For instance, it is easy to devise a situation in which even relatively small guests require the cooperation of several processes, say nearly one for each component of the guest. An interesting line of research would be to study the relation between guests, partitions, and performance in order to develop efficient distribution strategies. Moreover, structured partitions lend themselves to ad-hoc heuristics and optimizations. As an example, the way bigraphs are distributed among agents in [12] takes into account how they interact and reconfigure.

We considered adjacency as an undirected graph but some information is lost in this simplification. In fact, place and link graphs can be seen as forests suggesting the use of directed graphs. We intend to use this additional information to improve the routing through the induced semantic (directed) network.

References

1. G. Bacci, D. Grohmann, and M. Miculan. Bigraphical models for protein and membrane interactions. In G. Ciobanu, editor, *Proc. MeCBIC*, volume 11 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–18, 2009.

2. G. Bacci, M. Miculan, and R. Rizzi. Finding a forest in a tree. In *Proc. TGC*, *Lecture Notes in Computer Science*. Springer, 2014.
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. 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.
5. E. C. Cooper. Analysis of distributed commit protocols. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 175–183. ACM, 1982.
6. T. C. Damgaard, E. Højsgaard, and J. Krivine. Formal cellular machinery. *Electronic Notes in Theoretical Computer Science*, 284:55–74, 2012.
7. A. J. Faithfull, G. Perrone, and T. T. Hildebrandt. BigRed: A development environment for bigraphs. *ECEASST*, 61, 2013.
8. A. Glenstrup, T. Damgaard, L. Birkedal, and E. Højsgaard. An implementation of bigraph matching. *IT University of Copenhagen*, 2007.
9. E. Højsgaard. *Bigraphical Languages and their Simulation*. PhD thesis, IT University of Copenhagen, 2012.
10. O. H. Jensen and R. Milner. Bigraphs and transitions. In A. Aiken and G. Morrisett, editors, *POPL*, pages 38–49. ACM, 2003.
11. J. Krivine, R. Milner, and A. Troina. Stochastic bigraphs. In *Proc. MFPS*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 73–96, 2008.
12. A. Mansutti, M. Miculan, and M. Peressotti. Multi-agent systems design and prototyping with bigraphical reactive systems. In K. Magoutis and P. Pietzuch, editors, *DAIS*, volume 8460 of *Lecture Notes in Computer Science*, pages 201–208. Springer, 2014.
13. M. Miculan and M. Peressotti. Bigraphs reloaded. Technical Report UDMI/01/2013, Department of Mathematics and Computer Science, University of Udine, 2013.
14. M. Miculan and M. Peressotti. A CSP implementation of the bigraph embedding problem. In T. T. Hildebrandt, editor, *Proc. MeMo*, 2014.
15. R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
16. E. Pereira, C. M. Kirsch, J. B. de Sousa, and R. Sengupta. BigActors: a model for structure-aware computation. In C. Lu, P. R. Kumar, and R. Stoleru, editors, *ICCPS*, pages 199–208. ACM, 2013.
17. G. Perrone, S. Debois, and T. T. Hildebrandt. Bigraphical refinement. In J. Derrick, E. A. Boiten, and S. Reeves, editors, *Proc. REFINE*, volume 55 of *Electronic Proceedings in Theoretical Computer Science*, pages 20–36, 2011.
18. G. Perrone, S. Debois, and T. T. Hildebrandt. A model checker for bigraphs. In S. Ossowski and P. Lecca, editors, *Proc. SAC*, pages 1320–1325. ACM, 2012.
19. G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*, volume 1. World Scientific, River Edge, NJ, USA, 1997.
20. M. Sevegnani and E. Pereira. Towards a bigraphical encoding of actors. In T. T. Hildebrandt, editor, *Proc. MeMo*, 2014.
21. M. Sevegnani, C. Unsworth, and M. Calder. A SAT based algorithm for the matching problem in bigraphs with sharing. Technical Report TR-2010-311, Department of Computer Science, University of Glasgow, 2010.

A Unification Algorithm for GP

Ivaylo Hristakiev and Detlef Plump

The University of York, UK

Abstract. The graph programming language GP allows to apply sets of rule schemata (or "attributed" rules) nondeterministically. To analyse conflicts of programs statically, graphs labelled with expressisons are overlayed to construct critical pairs of rule applications. Each overlay induces a system of equations whose solutions represent different conflicts. We present a rule-based unification algorithm for GP expressions that is terminating and sound. Soundness means that every substitution generated by the algorithm solves the input system of equations. Since GP labels are lists constructed by concatenation, unification modulo associativity and unit laws is required. This problem, which is similar to *word unification*, is infinitary in general but becomes finitary due to GP's rule schema syntax.

1 Introduction

A common programming pattern in the graph programming language GP [7, 8] is to apply a set of graph transformation rules as long as possible. To execute such a loop $\{r_1, \ldots, r_n\}!$ on a host graph, in each iteration an applicable rule r_i is selected and applied. As rule selection and rule matching are nondeterministic, different graphs may result from the loop. Thus, if the programmer wants the loop to implement a function, a useful tool would be a static analysis that establishes or refutes functional behaviour.

The above loop is guaranteed to produce a unique result if the rule set $\{r_1, \ldots, r_n\}$ is terminating and confluent. However, conventional confluence analysis via critical pairs [6] assumes rules with constant labels whereas GP employs rule schemata (or "attributed" rules) whose graphs are labelled with expressions. Confluence of attributed graph transformation rules has been considered in [4, 2, 3], but we are not aware of *algorithms* that check confluence over non-trivial attribute algebras such as GP's which includes list concatenation and Peano arithmetic. The problem is that the equational theory of an attribute algebra needs to be taken into account when constructing critical pairs and checking their joinability.

For example, [4] presents a method of constructing critical pairs in the case where the equational theory of the attribute algebra is represented by a convergent term rewriting system. The algorithm first computes normal forms of the attributes of overlayed nodes and subsequently constructs the most general unifier of the normal forms. This has been shown to be incomplete [2, p.198] in that the constructed set of critical pairs need not represent all possible conflicts.

For, the most general unifier produces identical attributes—but it is necessary to find all substitutions that make attributes equivalent in the equational theory.

Graphs in GP rule schemata are labelled with lists of integer and string expressions, where lists are constructed by concatenation. In host graphs, list entries must be constant values. Integers and strings are subtypes of lists in that they represent lists of length one. As a simple example, consider the program in Figure 1 for calculating shortest distances. The program expects input graphs with non-negative integers as edge labels, and arbitrary lists as node labels. There must be a unique marked node (drawn shaded) whose shortest distance to each reachable node has to be calculated. The rule schemata `init` and `add`

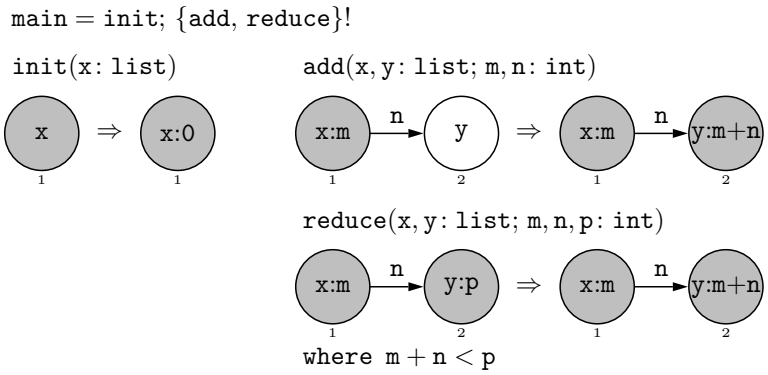
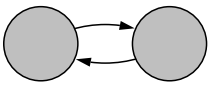


Fig. 1. A program calculating shortest distances

append distances to the labels of nodes that have not been visited before, while `reduce` decreases the distance of nodes that can be reached by a path that is shorter than the current distance.

To construct the conflicts of the rule schemata `add` and `reduce`, their left-hand sides are overlayed. For example, the structure of the left-hand graph of `reduce` can match the following structure in two different ways:

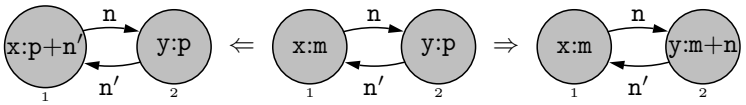


Consider a copy of `reduce` in which the variables have been renamed to $\mathtt{x}'$, $\mathtt{m}'$, etc. To match `reduce` and its copy differently requires solving the system of equations $\{\mathtt{x{:}m} =^? \mathtt{y}'\mathtt{{:}p}', \mathtt{y{:}p} =^? \mathtt{x}'\mathtt{{:}m}'\}$. Solutions to these equations should be as general as possible to represent all potential conflicts resulting from the above overlay. In this simple example, it is clear that the substitution

$$\sigma = \{\mathtt{x}' \mapsto \mathtt{y},\ \mathtt{m}' \mapsto \mathtt{p},\ \mathtt{y}' \mapsto \mathtt{x},\ \mathtt{p}' \mapsto \mathtt{m}\}$$

is a most general solution. It gives rise to the following critical pair:[1]

[1] For simplicity, we ignore the condition of `reduce`.



In general though, equations can arise that have several independent solutions. For example, the equation $\mathtt{n{:}x} =^{?} \mathtt{y{:}2}$ (with `n` of type `int` and `x`,`y` of type `list`) has the minimal solutions

$$\sigma_1 = \{\mathtt{x}, \mathtt{y} \mapsto \mathtt{empty},\, \mathtt{n} \mapsto 2\} \quad \text{and} \quad \sigma_2 = \{\mathtt{x} \mapsto \mathtt{z{:}2},\, \mathtt{y} \mapsto \mathtt{n{:}z}\}$$

where `empty` represents the empty list and `z` is a list variable.

Seen algebraically, we need to solve equations modulo the associativity and unit laws

$$\mathrm{AU} = \{x : (y : z) = (x : y) : z,\ \mathtt{empty} : x = x,\ x : \mathtt{empty} = x\}.$$

This problem is similar to *word unification* [1], which attempts to solve equations modulo associativity. Solvability of word unification is decidable, albeit in PSPACE [5], but there is not always a finite complete set of solutions. The same holds for AU-unification (see Subsection 3.3). Fortunately, GP's syntax for left-hand sides of rule schemata forbids labels with more than one list variable. We conjecture that this guarantees that left-hand overlays induce equation systems possessing finite complete sets of solutions.

This paper is the first step towards a static confluence analysis for GP programs. In Section 3, we present a rule-based unification algorithm for systems of equations with left-hand expressions of rule schemata. We show that the algorithm always terminates and that it is sound in that each substitution generated by the algorithm is an AU-unifier of the input problem.

2 Rule Schemata

We refer to [7, 8] for the definition of GP and more example programs. In this section, we define (unconditional) rule schemata which are the "building blocks" of graph programs.

A *graph* over a label set $\mathcal{C}$ is a system $G = (V, E, s, t, l, m)$, where V and E are finite sets of *nodes* (or *vertices*) and *edges*, $s, t\colon E \to V$ are the *source* and *target* functions for edges, $l\colon V \to \mathcal{C}$ is the node labelling function and $m\colon E \to \mathcal{C}$ is the edge labelling function. We write $\mathcal{G}(\mathcal{C})$ for the class of all graphs over $\mathcal{C}$.

Figure 2 shows an example for the declaration of a rule schema. The types `int` and `string` represent integers and character strings. Type `atom` is the union of `int` and `string`, and `list` represents lists of atoms. Given lists l_1 and l_2, we write $l_1 : l_2$ for the concatenation of l_1 and l_2. The empty list is denoted by `empty`. In pictures of graphs, nodes or edges without label (such as the dashed edge in Figure 2) are implicitly labelled with the empty list. We equate lists of length one with their entry to obtain the syntactic and semantic *subtype* relationships shown in Figure 3. Hence, for example, all labels in Figure 2 are list expressions.

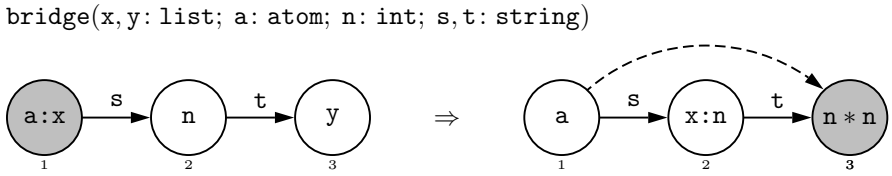


Fig. 2. Declaration of a rule schema

Also, GP 2 allows to *mark* nodes and edges. For example, the outermost nodes in Figure 2 are marked by a grey shading, and the dashed edge is a marked edge (labelled with the empty list). Figure 4 gives a grammar in Extended Backus-Naur Form defining the abstract syntax of labels. (In this paper, we omit string concatenation because it would inflate the unification algorithm without posing an extra challenge.) The functions `llength` and `slength` return the length of a list resp. string, while `indeg` and `outdeg` access the indegree resp. outdegree of a left-hand node in the host graph.

Figure 4 defines four syntactic categories of expressions: Integer, String, Atom and List, where Integer and String are subsets of Atom which in turn is a subset of List. Category Node is the set of node identifiers used in rule schemata. Moreover, IVar, SVar, AVar and LVar are the sets of variables of type `int`, `string`, `atom` and `list`. We assume that these sets are disjoint and define Var = IVar ∪ SVar ∪ AVar ∪ LVar. The mark components of labels are represented graphically rather than textually.

Each expression l has a unique smallest type, denoted by type(l), which can be read off the hierarchy in Figure 3 after l has been normalised with the rewrite rules shown at the beginning of Subsection 3.2. We write type(l_1) < type(l_2) or type(l_1) ≤ type(l_2) to compare types according to the subtype hierarchy. If the types of l_1 and l_2 are incomparable, we write type(l_1) ∥ type(l_2).

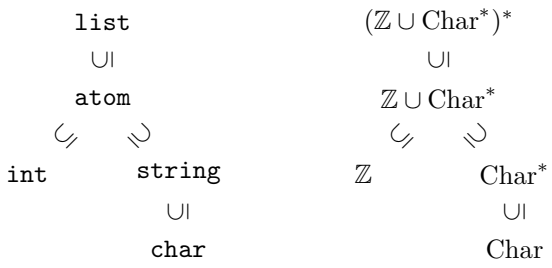


Fig. 3. Subtype hierarchy for labels

```
Integer   ::= Digit {Digit} | IVar
            | '−' Integer | Integer ArithOp Integer
            | llength '(' List ')' | slength '(' String ')'
            | (indeg | outdeg) '(' Node ')'
ArithOp   ::= '+' | '-' | '∗' | '/'
String    ::= ' " ' {Char} ' " ' | SVar
Atom      ::= Integer | String | AVar
List      ::= empty | Atom | LVar | List ':' List
Label     ::= List [Mark]
Mark      ::= red | green | blue | grey | dashed
```

Fig. 4. Abstract syntax of rule schema labels

The values of rule schema variables at execution time are determined by graph matching. To ensure that matches induce unique "actual parameters", expressions in the left graph of a rule schema must have a simple shape.

Definition 1 (Simple expression). A *simple* expression contains no arithmetic operators (with the possible exception of a unary minus preceding a sequence of digits), no length or degree operators, and at most one occurrence of a list variable.

For example, given the variable declarations of Figure 2, `a:x` and `y:n:n` are simple expressions whereas `n * 2` or `x:y` are not simple.

Definition 2 (Rule schema). A *rule schema* $\langle L, R, I\rangle$ consists of graphs L, R in $\mathcal{G}(\text{Label})$ and a set I, the *interface*, such that $I \subseteq V_L \cap V_R$. All labels in L must be simple and all variables occurring in R must also occur in L.

When a rule schema is graphically declared, as in Figure 2, the interface I is represented by the node numbers in L and R. Nodes without numbers in L are to be deleted and nodes without numbers in R are to be created. All variables in R have to occur in L so that for a given match of L in a host graph, applying the rule schema produces a graph that is unique up to isomorphism.

3 Unification

We start with introducing some technical notions such as substitutions, unification problems and complete sets of unifiers. Then, in Subsection 3.2, we present our unification algorithm. In Subsection 3.3, we prove that the algorithm terminates and is sound.

3.1 Preliminaries

A *substitution* is a family of mappings $\sigma = (\sigma_X)_{X \in \{I,S,A,L\}}$ where $\sigma_I \colon \mathrm{IVar} \to \mathrm{Integer}$, $\sigma_S \colon \mathrm{SVar} \to \mathrm{String}$, $\sigma_A \colon \mathrm{AVar} \to \mathrm{Atom}$, $\sigma_L \colon \mathrm{LVar} \to \mathrm{List}$. Here Integer, String, Atom and List are the sets of expressions defined by the GP label grammar of Figure 4. For example, if $\mathtt{z} \in \mathrm{LVar}$, $\mathtt{x} \in \mathrm{IVar}$ and $\mathtt{y} \in \mathrm{SVar}$, then we write $\sigma = \{\mathtt{x} \mapsto \mathtt{x} + \mathtt{1},\ \mathtt{z} \mapsto \mathtt{y} : -\mathtt{x} : \mathtt{y}\}$ for the substitution that maps $\mathtt{x}$ to $\mathtt{x} + \mathtt{1}$, $\mathtt{z}$ to $\mathtt{y} : -\mathtt{x} : \mathtt{y}$ and every other variable to itself.

Applying a substitution σ to an expression t, denoted by $t\sigma$, means to replace every variable x in t by $\sigma(x)$ simultaneously. In the above example, $\sigma(\mathtt{z} : -\mathtt{x}) = \mathtt{y} : -\mathtt{x} : \mathtt{y} : -(\mathtt{x} + \mathtt{1})$.

By $\mathrm{Dom}(\sigma)$ we denote the set $\{x \in \mathrm{Var} \mid \sigma(x) \neq x\}$ and by $\mathrm{VRan}(\sigma)$ the set of variables occurring in the expressions $\{\sigma(x) \mid x \in \mathrm{Var}\}$. A substitution σ is *idempotent* if $\mathrm{Dom}(\sigma) \cap \mathrm{VRan}(\sigma) = \emptyset$.

Definition 3 (Unification problem). A *unification problem* is a finite multiset of equations

$$P = \{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$$

between simple list expressions.

The symbol $=^?$ signifies that the equations must be *solved* rather than having to hold for all values of variables.

Consider the equational axioms for associativity and unity,

$$\mathrm{AU} = \{\mathtt{x} : (\mathtt{y} : \mathtt{z}) = (\mathtt{x} : \mathtt{y}) : \mathtt{z},\ \mathtt{empty} : \mathtt{x} = \mathtt{x},\ \mathtt{x} : \mathtt{empty} = \mathtt{x}\}$$

where $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are variables of type `list`, and let $=_{\mathrm{AU}}$ be the equivalence relation on expressions generated by these axioms.

Definition 4 (Unifier). A *unifier* of a problem $P = \{s_1 =^? t_1,\ \ldots,\ s_n =^? t_n\}$ is a substitution σ such that

$$s_1\sigma =_{\mathrm{AU}} t_1\sigma,\ \ldots,\ s_n\sigma =_{\mathrm{AU}} t_n\sigma.$$

The set of all unifiers of P is denoted by $\mathcal{U}(P)$. We say that P is *unifiable* if $\mathcal{U}(P) \neq \emptyset$.

A substitution σ is *more general* on a set of variables X than a substitution θ if there exists a substitution λ such that $x\theta =_{\mathrm{AU}} x\sigma\lambda$ for all $x \in X$. In this case we write $\sigma \leqq_X \theta$ and say that θ is an *instance* of σ on X. Substitutions σ and θ are *equivalent* on X, denoted by $\sigma =_X \theta$, if $\sigma \leqq_X \theta$ and $\theta \leqq_X \sigma$.

Definition 5 (Complete set of unifiers). A set $\mathcal{C}$ of substitutions is a *complete set of unifiers* of a unification problem P if

1. $\mathcal{C} \subseteq \mathcal{U}(P)$, that is, each substitution in $\mathcal{C}$ is a unifier of P, and
2. for each $\theta \in \mathcal{U}(P)$ there exists $\sigma \in \mathcal{C}$ such that $\sigma \leqq_X \theta$, where $X = \mathrm{Var}(P)$.

Set $\mathcal{C}$ is also *minimal* if it satisfies

3. each two substitutions in $\mathcal{C}$ are incomparable with respect to $\leqq_X$, that is, for all $\sigma, \sigma' \in \mathcal{C}$, $\sigma \leqq_X \sigma'$ implies $\sigma = \sigma'$.

If a unification problem P is not unifiable, then the empty set is a minimal complete set of unifiers of P.

We call a variable x *solved* in P if it occurs exactly once in P, namely on the left-hand side of an equation $x = L$ with $\text{type}(x) \geq \text{type}(L)$.

Definition 6 (Solved form). A unification problem $P = \{x_1 =^? t_1, \ldots, x_n =^? t_n\}$ is in *solved form* if the variables x_i are pairwise distinct and solved in P. In this case we define the substitution

$$\overrightarrow{P} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}.$$

For example, if `a` is an atom variable and `x` a list variable, then the problems $\{\mathtt{x} = \mathtt{a}\}$ and $\{\mathtt{x} = \mathtt{1} : \mathtt{a}\}$ are in solved form whereas $\{\mathtt{x} = \mathtt{a} : \mathtt{x}\}$, $\{\mathtt{a} : \mathtt{1} = \mathtt{2} : \mathtt{1}\}$ and $\{\mathtt{a} = \mathtt{x}\}$ are not solved. For simplicity, we replace $=^?$ with $=$ in unification problems from now on.

The minimal complete set of unifiers of the problem $\{\mathtt{a} : \mathtt{x} = \mathtt{y} : \mathtt{2}\}$ (where `a` is an atom variable and `x`,`y` are list variables) is $\{\sigma_1, \sigma_2\}$ with

$$\sigma_1 = \{\mathtt{a} \mapsto 2, \mathtt{x} \mapsto \mathtt{empty}, \mathtt{y} \mapsto \mathtt{empty}\} \quad \text{and} \quad \sigma_2 = \{\mathtt{x} \mapsto \mathtt{z} : 2, \mathtt{y} \mapsto \mathtt{a} : \mathtt{z}\}.$$

We have $\sigma_1(\mathtt{a} : \mathtt{x}) = \mathtt{2} : \mathtt{empty} =_{\text{AU}} \mathtt{2} =_{\text{AU}} \mathtt{empty} : \mathtt{2} = \sigma_1(\mathtt{y} : \mathtt{2})$ and $\sigma_2(\mathtt{a} : \mathtt{x}) = \mathtt{a} : \mathtt{z} : \mathtt{2} = \sigma_2(\mathtt{y} : \mathtt{2})$. Other unifiers such as $\sigma_3 = \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto \mathtt{a}\}$ are instances of σ_2.

3.2 Unification Algorithm

We start with some notational conventions for the rest of this section:

- L, M stand for simple expressions,
- x, y, z stand for variables of any type (unless otherwise specified),
- a, b stand for simple string or integer expressions, or atom variables,
- s, t stand for simple string or integer expressions, or atom variables, or list variables,
- the symbol $\cup$ denotes multiset union.

Preprocessing. Given a unification problem P, we rewrite the terms in P using the rules

$$L : \mathtt{empty} \to L \quad \text{and} \quad \mathtt{empty} : L \to L$$

where L ranges over list expressions. These reduction rules are applied exhaustively before any of the transformation rules. For example,

$$\mathtt{x} : \mathtt{empty} : \mathtt{1} : \mathtt{empty} \to \mathtt{x} : \mathtt{1} : \mathtt{empty} \to \mathtt{x} : \mathtt{1}.$$

We call this process *normalization*. In addition, the rules are applied to each instance of a transformation rule (that is, once the formal parameters have been replaced with actual parameters) before it is applied, and also after each transformation rule application.

Transformation rules. Figure 5 shows the transformation rules, the essence of our approach, in an inference system style where each rules consists of a premise and a conclusion.

Remove:	deletes trivial equations
Decomp:	replaces equations between list expressions by equations between their subexpressions
Subst1:	propagates a solved variable to the rest of the problem
Subst2:	assigns `empty` to a list variable
Subst3:	assigns an atom prefix and a fresh list variable to a list variable
Orient1/2:	move variables to left-hand side
Orient3:	moves variables of larger type to left-hand side

The rules induce a transformation relation $\Rightarrow$ on unification problems. In order to apply any of the rules to a problem P, the problem part of its premise needs to be *matched* onto P. Subsequently, the boolean condition of the premise is checked and the rule *instance* is normalized so that its premise is identical to P.

For example, the rule Orient3 can be matched to $P = \{\mathtt{a} : 2 = \mathtt{l}, \mathtt{a} = 3\}$ (where `a` and `l` are variables of type atom and list, respectively) by setting $y \mapsto \mathtt{a}$, $x \mapsto \mathtt{l}$, $L \mapsto \mathtt{l}$, $M \mapsto \mathtt{empty}$ and $P \mapsto \{\mathtt{a} = 3\}$. The rule instance is then

$$\frac{\{\mathtt{a} : 2 = \mathtt{l} : \mathtt{empty}\} \cup \{\mathtt{a} = 3\}}{\{\mathtt{l} : \mathtt{empty} = \mathtt{a} : 2\} \cup \{\mathtt{a} = 3\}}\ \mathsf{Orient3}$$

which gets normalized to

$$\frac{\{\mathtt{a} : 2 = \mathtt{l}\} \cup \{\mathtt{a} = 3\}}{\{\mathtt{l} = \mathtt{a} : 2\} \cup \{\mathtt{a} = 3\}}\ \mathsf{Orient3}$$

whose conclusion is the result of applying Orient3 to P.

Showing a unification problem cannot be unified can be a lengthy affair because we need to compute all normal forms with respect to $\Rightarrow$. Instead, the rules Occur and Clash1-4, shown in Figure 6, introduce *failure*. Failure cuts off parts of the search tree for a given problem P. This is because if $P \Rightarrow$ fail, then P has no unifiers and it is not necessary to compute a normal form. Effectively, the failure rules have precedence over the other rules. They are justified by the following lemmata.

Lemma 1. *A normalised equation $x = L$ has no solution if L is a simple expression, $x \in Var(L)$,* $\mathrm{type}(x) = \mathtt{list}$ *and $x \neq L$.*

Proof. Since $x \in Var(L)$ and $x \neq L$, L is of the form $s_1 : s_2 : \ldots : s_n$ with $n \geq 2$ and $x \in Var(s_i)$ for some $1 \leq i \leq n$. As L is normalised, none of the terms s_i contains the constant `empty`. Also, since L is simple, it contains no list variables other than x and x is not repeated. It follows $\sigma(x) \neq_{\mathrm{AU}} \sigma(L)$ for every substitution σ. □

$$\frac{\{L = L\} \cup P}{P}\ \textsf{Remove}$$

$$\frac{\{s : L = t : M\} \cup P \quad L \neq \texttt{empty}}{\{s = t,\, L = M\} \cup P}\ \textsf{Decomp}$$

$$\frac{\{x = L\} \cup P \quad x \in \mathrm{Var}(P) \quad x \notin \mathrm{Var}(L) \quad \mathrm{type}(x) \geq \mathrm{type}(L)}{\{x = L\} \cup P\{x \mapsto L\}}\ \textsf{Subst1}$$

$$\frac{\{x : L = M\} \cup P \quad L \neq \texttt{empty} \quad \mathrm{type}(x) = \texttt{list}}{\{x = \texttt{empty},\, L = M\} \cup P\{x \mapsto \texttt{empty}\}}\ \textsf{Subst2}$$

$$\frac{\{x : L = b : M\} \cup P \quad L \neq \texttt{empty} \quad z \text{ is a fresh list variable} \quad \mathrm{type}(x) = \texttt{list}}{\{x = b : z,\, z : L = M\} \cup P\{x \mapsto b : z\}}\ \textsf{Subst3}$$

$$\frac{\{a : L = x : M\} \cup P \quad a \text{ is not a variable}}{\{x : M = a : L\} \cup P}\ \textsf{Orient1}$$

$$\frac{\{y : L = x\} \cup P \quad L \neq \texttt{empty} \quad \mathrm{type}(x) = \mathrm{type}(y)}{\{x = y : L\} \cup P}\ \textsf{Orient2}$$

$$\frac{\{y : L = x : M\} \cup P \quad \mathrm{type}(y) < \mathrm{type}(x)}{\{x : M = y : L\} \cup P}\ \textsf{Orient3}$$

Fig. 5. Transformation rules

Lemma 2. *Equations of the form* $a : L =$ `empty` *or* `empty` $= a : L$ *have no solution if* a *is an atom expression.*

Lemma 3. *An equation* $a : L = b : M$ *has no solution if* $a \neq b$ *are atom expressions without variables.*

The algorithm. The algorithm in Figure 7 starts by normalizing the input problem, as explained above. It uses a queue of unification problems to search the derivation tree of P with respect to $\Rightarrow$ in a breadth-first manner. The first step is to put the normalized problem P on the queue.

The variable `next` holds the head of the queue. If `next` is in solved form, then $\overrightarrow{\texttt{next}}$ (see Definition 6) is a unifier of the original problem and is added to the set `U` of solutions. Otherwise, the next step is to construct all problems P' such

$$\frac{\{x = L\} \cup P \quad x \in \mathrm{Var}(L) \quad x \neq L \quad \mathrm{type}(x) = \texttt{list}}{\mathrm{fail}}\ \mathsf{Occur}$$

$$\frac{\{a : L = b : M\} \cup P \quad a \neq b \quad \mathrm{Var}(a) = \emptyset = \mathrm{Var}(b)}{\mathrm{fail}}\ \mathsf{Clash1}$$

$$\frac{\{a : L = \texttt{empty}\} \cup P}{\mathrm{fail}}\ \mathsf{Clash2}$$

$$\frac{\{\texttt{empty} = a : L\} \cup P}{\mathrm{fail}}\ \mathsf{Clash3}$$

$$\frac{\{x = L\} \cup P \quad \mathrm{type}(x) \parallel \mathrm{type}(L)}{\mathrm{fail}}\ \mathsf{Clash4}$$

Fig. 6. Failure rules

that $\texttt{next} \Rightarrow P'$. If P' is `fail`, then the derivation tree below `next` is ignored, otherwise P' gets normalized and enqueued.

An example tree traversed by the algorithm is shown in Figure 8. Nodes are labelled with unification problems and edges represent applications of transformation rules. The root of the tree is the problem $\{\texttt{a} : \texttt{x} = \texttt{y} : 2\}$ to which the rules Decomp and Orient3 can be applied. The two resulting problems form the second level of the search tree and are processed in turn. Eventually, the unifiers

$$\begin{aligned}\sigma_1 &= \{\texttt{x} \mapsto 2,\ \texttt{y} \mapsto \texttt{a}\}\\ \sigma_2 &= \{\texttt{x} \mapsto \texttt{z} : 2,\ \texttt{y} \mapsto \texttt{a} : \texttt{z}\}\\ \sigma_3 &= \{\texttt{a} \mapsto 2,\ \texttt{x} \mapsto \texttt{empty},\ \texttt{y} \mapsto \texttt{empty}\}\end{aligned}$$

are found, which represent a complete set of unifiers of the initial problem. Note that the set is not minimal because σ_1 is an instance of σ_2.

The algorithm is similar to the A-unification (word unification) algorithm presented in [9] which looks only at the head of an equation. That algorithm terminates for the special case that the input problem has no repeated variables, and is sound and complete. Our approach can be seen as an extension from A-unification to AU-unification, to handle the unit equations, and presented in the rule-based style of [1]. In addition, our algorithm deals with GP's subtype system.

```
Unify(P) :  U := ∅
            create empty queue Q of unification problems
            normalize P
            Q.enqueue(P)
            while Q is not empty
              next := Q.dequeue()
              if next is in solved form
                U := U ∪ {next→}
              else if next ↛ fail
                foreach P′ such that next ⇒ P′
                    normalize P′
                    Q.enqueue(P′)
                end foreach
                end if
              end if
            end while
            return U
```

Fig. 7. Unification algorithm

3.3 Termination and Soundness

We show that the unification algorithm terminates if the input problem contains no repeated list variables, where termination of the algorithm follows from termination of the relation $\Rightarrow$.

We first demonstrate that the algorithm need not terminate on unification problems with repeated list variables. A counterexample is the unification problem $\{\mathtt{x}{:}1 = 1{:}\mathtt{x}\}$ which initiates the following infinite sequence:

$$\begin{aligned}
\{\mathtt{x}{:}1 = 1{:}\mathtt{x}\} &\Rightarrow_{\mathsf{Subst3}} \{\mathtt{x} = 1{:}\mathtt{z}_1,\ \mathtt{z}_1{:}1 = \mathtt{x}\}\\
&\Rightarrow_{\mathsf{Subst1}} \{\mathtt{x} = 1{:}\mathtt{z}_1,\ \mathtt{z}_1{:}1 = 1{:}\mathtt{z}_1\}\\
&\Rightarrow_{\mathsf{Subst3}} \{\mathtt{x} = 1{:}\mathtt{z}_1,\ \mathtt{z}_1 = 1{:}\mathtt{z}_2,\ \mathtt{z}_2{:}1 = \mathtt{z}_1\}\\
&\Rightarrow_{\mathsf{Subst1}} \{\mathtt{x} = 1{:}1{:}\mathtt{z}_2,\ \mathtt{z}_1 = 1{:}\mathtt{z}_2,\ \mathtt{z}_2{:}1 = 1{:}\mathtt{z}_2\}\\
&\Rightarrow_{\mathsf{Subst3}} \{\mathtt{x} = 1{:}1{:}\mathtt{z}_2,\ \mathtt{z}_1 = 1{:}\mathtt{z}_2,\ \mathtt{z}_2 = 1{:}\mathtt{z}_3,\ \mathtt{z}_3{:}1 = \mathtt{z}_2\}\\
&\Rightarrow_{\mathsf{Subst1}} \{\mathtt{x} = 1{:}1{:}1{:}\mathtt{z}_3,\ \mathtt{z}_1 = 1{:}1 : \mathtt{z}_3,\ \mathtt{z}_2 = 1 : \mathtt{z}_3,\ \mathtt{z}_3{:}1 = 1 : \mathtt{z}_3\}\\
&\Rightarrow_{\mathsf{Subst3}} \cdots\\
&\quad\vdots
\end{aligned}$$

Note that $\{\mathtt{x}{:}1 = 1{:}\mathtt{x}\}$ has an infinite number of solutions that are mutually incomparable: $\{\mathtt{x} \mapsto \mathtt{empty}\}$, $\{\mathtt{x} \mapsto 1\}$, $\{\mathtt{x} \mapsto 1{:}1\}$, ... We remark that the A-unification algorithm of [9] also diverges on this problem.

To prove that the transformation relation $\Rightarrow$ terminates on problems without repeated list variables, we need to consider an invariant which is implied by this property. This is because rule Subst3 introduces a repeated list variable. We

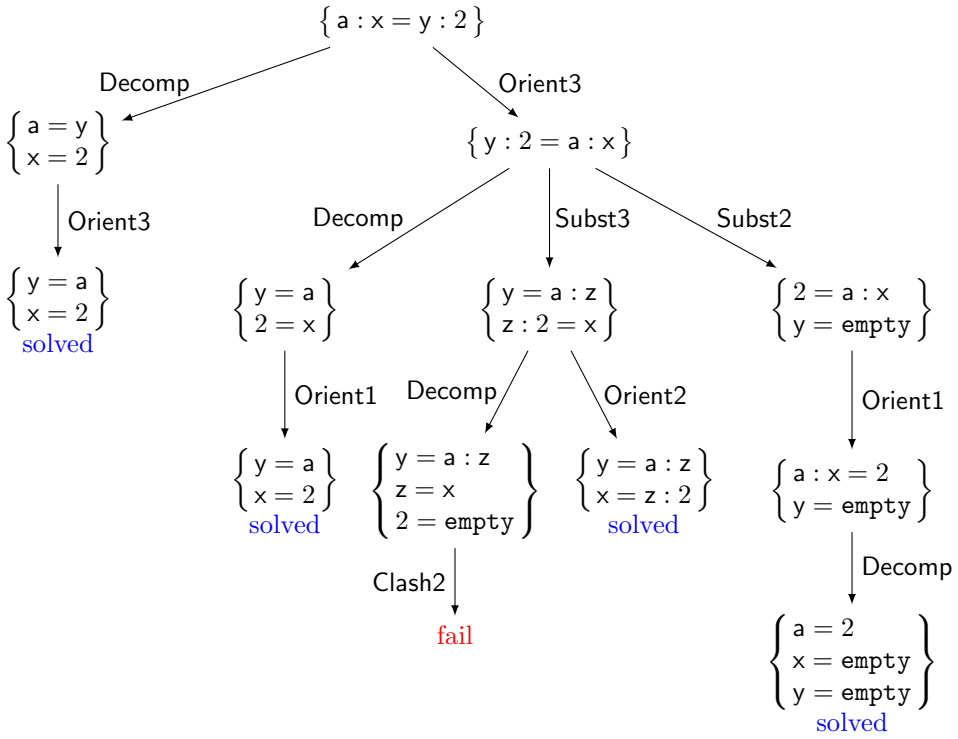


Fig. 8. Unification example

say that a unification problem P satisfies the *repeated variable condition* if for every list variable x, the subproblem $P \setminus \{y = L \mid y \neq x\}$ contains at most one occurrence of x.

Lemma 4 (Invariance). *For each transformation $P \Rightarrow P'$ where P satisfies the repeated variable condition, P' also satisfies this condition.*

The proof is by a careful but straightforward inspection of all rules in Figure 5. We are now ready to state our termination result.

Theorem 1 (Termination). *If P is a unification problem without repeated list variables, then there is no infinite sequence $P \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \ldots$*

Proof. Define the *size* $|L|$ of an expression L by

- 0 if $L = \texttt{empty}$,
- 1 if L is an expression of category Atom (see Figure 4) or a list variable,
- $|M| + |N| + 1$ if $L = M : N$.

We define a lexicographic termination order by assigning to a unification problem P the tuple $(n_1, n_2, n_3, n_4, n_5, n_6)$, where

- n_1 is the number of unsolved variables in P;
- n_2 is the size of $P \setminus Q$ where $Q = \{x = L \mid x \in \mathrm{Var}\}$, that is, $n_2 = \sum_{(L=R)\in P\setminus Q}(|L| + |R|)$;
- n_3 is the size of P, that is, $n_3 = \sum_{(L=R)\in P}(|L| + |R|)$;
- n_4 is the number of equations $L = x : M$ in P where L is not a variable;
- n_5 is the number of equations $y : L = x$ in P where $\mathrm{type}(x) = \mathrm{type}(y)$ and $L \neq \texttt{empty}$;
- n_6 is the number of equations $y : L = x : M$ in P where $\mathrm{type}(y) < \mathrm{type}(x)$.

The table in Figure 9 shows that for each transformation step $P \Rightarrow P'$, the tuple associated with P' is strictly smaller than the tuple associated with P in the lexicographic order induced by the components n_1 to n_6.

	n_1	n_2	n_3	n_4	n_5	n_6
Subst1	$>$					
Subst3	$\geq$	$>$				
Subst2	$\geq$	$\geq$	$>$			
Decomp	$\geq$	$\geq$	$>$			
Remove	$\geq$	$=$	$>$			
Orient1	$\geq$	$\geq$	$=$	$>$		
Orient2	$\geq$	$\geq$	$=$	$\geq$	$>$	
Orient3	$\geq$	$\geq$	$=$	$\geq$	$\geq$	$>$

Fig. 9. Lexicographic termination order

For most rules, the table entries are easy to check. The argument why Subst3 decreases n_2 is a bit more involved because the rule solves x and creates two copies of the fresh variable z. Since the repeated variable condition of Lemma 4 is an invariant, the instance of problem P in the premise of Subst3 can only contain occurrences of x that appear on the right-hand side of equations $y = L$ with $y \neq x$. It follows that the n_2-value of P is the same as that of $P\{x \mapsto b : z\}$. As a consequence, each application of Subst3 decreases n_2 by 2. □

In order to show that the unification algorithm is sound, we need some preliminary lemmata.

Lemma 5. *If $P = \{x_1 =^? t_1, \ldots, x_n =^? t_n\}$ is in solved form then for all $\sigma \in \mathcal{U}(P)$, $\sigma = \overrightarrow{P}\sigma$.*

Proof. We show that σ and $\overrightarrow{P}\sigma$ behave the same on all variables by considering the following cases:

1. $x \in \{x_1, \ldots, x_n\}$, i.e. $x = x_k$, then $x_k\overrightarrow{P} = t_k$ which implies $x_k\overrightarrow{P}\sigma = t_k\sigma$. Also since σ is a unifier of P, then $x_k\sigma = t_k\sigma$. Therefore $x_k\overrightarrow{P}\sigma = x_k\sigma$.

2. $x \notin \{x_1, \ldots, x_n\}$, then $x\overrightarrow{P} = x$ for variables outside of the domain and hence $x\overrightarrow{P}\sigma = x\sigma$

□

Lemma 6. *If P is in solved form then $\overrightarrow{P}$ is an idempotent most general unifier of P.*

Proof. Since none of the x_i occur in any of the t's, we get that $Dom(\overrightarrow{P}) \cap VRan(\overrightarrow{P}) = \varnothing$. Therefore, $\overrightarrow{P}$ is idempotent. Also, we have $x_i\overrightarrow{P} = t_i = t_i\overrightarrow{P}$ for the same reason, hence $\overrightarrow{P} \in \mathcal{U}(P)$. Finally, $\overrightarrow{P}$ is most general because $\overrightarrow{P} \leq \sigma$ for all $\sigma \in \mathcal{U}(P)$ by Lemma 5. □

Lemma 7. *If $P \Rightarrow P'$, then $\mathcal{U}(P) \supseteq \mathcal{U}(P')$*

Proof. For `Remove, Decomp` and `Orient`, this is obvious.

For `Subst1`, let $\theta = \{\mathrm{x} \mapsto L\}$. By applying Lemma 5 to $\{x = L\}$ which is in solved form, we get that $\sigma = \theta\sigma$ if $x\sigma = L\sigma$

$$\begin{aligned}
\sigma \in \mathcal{U}(\{\mathrm{x} = L\} \cup P\theta) &\iff x\sigma = L\sigma \wedge \sigma \in \mathcal{U}(P\theta) \\
&\iff x\sigma = L\sigma \wedge \theta\sigma \in \mathcal{U}(P) \\
&\iff x\sigma = L\sigma \wedge \sigma \in \mathcal{U}(P) \\
&\iff \sigma \in \mathcal{U}(\{\mathrm{x} = L\} \cup P)
\end{aligned}$$

For `Subst2`, the argument is similar. Let $\theta = \{\mathrm{x} \mapsto \texttt{empty}\}$. Then $\sigma = \theta\sigma$ if $\mathrm{x}\sigma = \texttt{empty}\ \sigma(= \texttt{empty})$

$$\begin{aligned}
\sigma \in \mathcal{U}(\{\mathrm{x} = \texttt{empty}, L = M\} \cup P\theta) &\iff L\sigma = M\sigma \wedge x\sigma = \texttt{empty}\ \sigma \wedge \sigma \in \mathcal{U}(P\theta) \\
&\iff L\sigma = M\sigma \wedge x\sigma = \texttt{empty}\ \sigma \wedge \theta\sigma \in \mathcal{U}(P) \\
&\iff L\sigma = M\sigma \wedge x\sigma = \texttt{empty}\ \sigma \wedge \sigma \in \mathcal{U}(P) \\
&\Rightarrow\ x\sigma : L\sigma = \texttt{empty}\ \sigma : M\sigma \wedge \sigma \in \mathcal{U}(P) \\
&\iff x\sigma : L\sigma = \texttt{empty} : M\sigma \wedge \sigma \in \mathcal{U}(P) \\
&\iff x\sigma : L\sigma = M\sigma \wedge \sigma \in \mathcal{U}(P) \\
&\iff \sigma(x : L) = \sigma M \wedge \sigma \in \mathcal{U}(P) \\
&\iff \sigma \in \mathcal{U}(\{\mathrm{x} : L = M\} \cup P)
\end{aligned}$$

For `Subst3`, let $\theta = \{\mathrm{x} \mapsto \mathtt{b} : \mathtt{z}\}$. Then $\sigma = \theta\sigma$ if $\mathrm{x}\sigma = \sigma(\mathtt{b} : \mathtt{z})$, again by Lemma 5

$$\begin{aligned}
&\sigma \in \mathcal{U}(\{\mathrm{x} = \mathtt{b} : \mathtt{z}, \mathtt{z} : L = M\} \cup P\theta) \\
\iff\ &\sigma(\mathtt{z} : L) = M\sigma \wedge \mathrm{x}\sigma = \sigma(\mathtt{b} : \mathtt{z}) \wedge \sigma \in \mathcal{U}(P\theta) \\
\iff\ &\sigma(\mathtt{z} : L) = M\sigma \wedge \mathrm{x}\sigma = \sigma(\mathtt{b} : \mathtt{z}) \wedge \sigma\theta \in \mathcal{U}(P) \\
\iff\ &\sigma(\mathtt{z} : L) = M\sigma \wedge \mathrm{x}\sigma = \sigma(\mathtt{b} : \mathtt{z}) \wedge \sigma \in \mathcal{U}(P) \\
\iff\ &\sigma(\mathtt{z} : L) = M\sigma \wedge \mathrm{x}\sigma : L\sigma = \sigma(\mathtt{b} : \mathtt{z}) : L\sigma \wedge \sigma \in \mathcal{U}(P) \\
\iff\ &\sigma(\mathtt{z} : L) = M\sigma \wedge \mathrm{x}\sigma : L\sigma = \mathtt{b}\sigma : \sigma(\mathtt{z} : L) \wedge \sigma \in \mathcal{U}(P) \\
\iff\ &\sigma(\mathtt{z} : L) = M\sigma \wedge \mathrm{x}\sigma : L\sigma = \mathtt{b}\sigma : M\sigma \wedge \sigma \in \mathcal{U}(P) \\
\Rightarrow\ &\mathrm{x}\sigma : L\sigma = \mathtt{b}\sigma : M\sigma \wedge \sigma \in \mathcal{U}(P) \\
\iff\ &\sigma(x : L) = \sigma(\mathtt{b} : \mathtt{M}) \wedge \sigma \in \mathcal{U}(P) \\
\iff\ &\sigma \in \mathcal{U}(\{\mathrm{x} : L = \mathtt{b} : \mathtt{M}\} \cup P)
\end{aligned}$$

□

Theorem 2 (Soundness). *If $P \Rightarrow^+ P'$ with P' in solved form, then $\overrightarrow{P'}$ is an idempotent unifier of P.*

Proof. Note that $\overrightarrow{P'}$ unifies P' because it is idempotent (by Lemma 6); a simple induction with Lemma 7 shows that $\overrightarrow{P'}$ must be a unifier of P. □

4 Conclusion

This paper presents groundwork for a static confluence analysis of GP programs. We have constructed a rule-based unification algorithm for systems of equations with left-hand expressions of rule schemata, and have shown that the algorithm always terminates and is sound.

Future work includes proving that our unification algorithm always delivers a *complete* set of solutions, that is, that every unifier of the input problem is an instance of some unifier in the computed set of solutions. Next, to establish a Critical Pair Lemma in the sense of [6], a notion of *independent* rule schema applications has to be developed, as well as restriction and embedding theorems for derivations with rule schemata. In addition, since critical pairs contain graphs labelled with expressions, checking joinability of critical pairs will require sufficient conditions under which equivalence of expressions can be decided. This is because the theory of GP's label algebra includes the undecidable theory of Peano arithmetic.

References

1. Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.
2. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer-Verlag, 2006.
3. Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. *Theoretical Computer Science*, 424:46–68, 2012.
4. Reiko Heckel, Jochen Malte Küster, and Gabi Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2002.
5. Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. In *Symposium on Foundations of Computer Science (FOCS 1999)*, pages 495–500. IEEE Computer Society, 1999.
6. Detlef Plump. Confluence of graph transformation revisited. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 280–308. Springer-Verlag, 2005.
7. Detlef Plump. The graph programming language GP. In *Proc. International Conference on Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009.

8. Detlef Plump. The design of GP 2. In *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2012.
9. Klaus U. Schulz. Makanin's algorithm for word equations: Two improvements and a generalization. In *Proc. Word Equations and Related Topics (IWWERT '90)*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150. Springer-Verlag, 1992.

Properties of Petri Nets with Context-Free Structure Changes

Nils Erik Flick* and Björn Engelmann*

Carl-von-Ossietzky-Universität, D-26111 Oldenburg
https://scare.informatik.uni-oldenburg.de

Abstract. Petri nets model systems with distributed state and synchronised state changes. Extending them with rewriting rules allows for evolving structure. In this paper, we investigate dynamic properties of simple structure-changing Petri nets. We show undecidability of checking a language-based notion of correctness even for very restricted classes of structure-changing nets. We also introduce a colour-based abstraction and use it to specify, and in special cases decide, reachability properties.

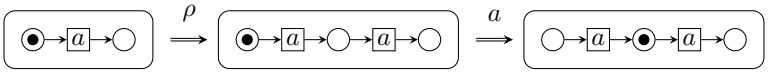
1 Introduction

Petri nets or place/transition nets [5] are system models where resource tokens are moved around on an immutable underlying structure. They originally lack a notion of structure change or reconfiguration, but several structure-changing extensions have been formulated. We shall use dynamic transition refinement.

Petri nets are sometimes employed in the context of workflow modeling, where tasks to be executed correspond to Petri net transitions labeled with the task type. A theory of workflow nets has been devised [11]. An important property of workflow nets is *soundness*, which is decidable (in the absence of extra features such as reset arcs), and intuitively means the workflow can always terminate correctly and there are no useless transitions.

Plain workflow nets lack the ability of representing dynamic evolution. This shortcoming had been recognized by the community [12, 14]. Structure-changing Petri nets offer a potential solution for dealing with dynamic change in workflows. In this paper, we will consider workflow nets augmented with replacement rules.

Example 1 (A structure-changing Petri net derivation).



Like most extensions to Petri nets, graph transformation systems are Turing complete, rendering any nontrivial property of all such systems undecidable.

* These authors' work is supported by the German Research Foundation (DFG), grant GRK 1765 (Research Training Group System Correctness under Adverse Conditions)

In this paper, structure-changing Petri nets will be regarded as a model of adverse influence, with structure-changing rules interfering with the intended system behaviour. The rules, which occur unpredictably, model the influence of an uncontrolled environment such as the dynamic addition of a component or the unexpected complication or iteration of a task. We introduce restricted classes of structure-changing Petri nets (where all nets involved are 1-safe, separable, even acyclic sound workflow nets and structure changes are of a particularly simple kind) and show the undecidability of inclusion of their language of net transition events in a regular language. We also investigate reachability problems.

The results presented here have been achieved by simple means. When suitably translated, they apply to many formalisms that add structure changes to Petri nets, for example those cited in the related work section at the end of this paper.

The paper is structured as follows: Section 2 gives the definitions of structure-changing Petri nets as well as structure-changing workflow nets and their dynamics, in Section 3 we state our decidability and undecidability results, Section 4 draws parallels to existing work and Section 5 concludes with an outlook.

2 Structure-changing Petri nets

In this section, we review Petri nets in the sense of [5] and introduce structure-changing Petri nets as special graph transformation systems.

Definition 1 (Petri net). *A marked labeled Place/Transition net with coloured places, short* marked net*, is a pair* (N, M)*, where* N *is a 7-tuple* $(P, T, F^-, F^+, \Sigma, l, c)$ *called the* net structure *or simply* net*. Its components are a finite set* P *of* places*, a finite set* T *of* transitions *and a finite alphabet* Σ *of* labels *(pairwise disjoint), two functions* $F^-, F^+ : T \times P \to \mathbb{N}$ *assigning* preset *and* postset arc multiplicities*, respectively, a* label *function* $l : T \to \Sigma$*, a* colouring *function* $c : P \to \mathbb{N}$*. The function* $M : P \to \mathbb{N}$ *is the* marking *of the marked net.*

When $F^-(t, p) = k$, one says there are k arcs from p to t. Likewise, when $F^+(t, p) = k$, there are k arcs from t to p. When $M(p) = k$, it means that p is marked with k tokens. When $c(p) = k$, we say that p has colour k. If a net is named N_x with sub- or superscript x, the components will likewise be named P_x, etc. by convention. If there is no subscript, they will be referenced as P and so on. If $M : P \to \mathbb{N}$, $k \cdot M$ is the function $P \to \mathbb{N}, p \mapsto k \cdot M(p)$. The pictorial representation of nets as graph-like diagrams, indeed the translation to graph transformations, is very well known and has first been formalized, to our knowledge, in [9]. A net is said to be *(strongly) connected*, or *acyclic*, if it is (strongly) connected, or acyclic, as a graph, regarding arcs as directed edges. A transition or place q of a net is said to be *graph-reachable* from another transition or place q' if (q', q) is in the reflexive and transitive closure of the relation $\{(x, y) \mid (y \in T \wedge y \in P \wedge F^-(y, x) \geq 1 \vee (x \in T \wedge y \in P \wedge F^+(x, y) \geq 1)\}$. *Isomorphism* ($N \cong N'$) of (marked) nets has the usual meaning.

Definition 2 (Transition firing). *In a net $N = (P, T, F^-, F^+, \Sigma, l, c)$, the transition $t \in T$ is said to be* enabled *in the marking M iff $\forall p \in P, n \cdot F^-(t,p) \leq M(p)$. The successor marking Mt to M via t (if N is understood) is then defined by $\forall p \in P, (Mt)(p) = M(p) - F^-(t,p) + F^+(t,p)$. Recursively define Mu for a sequence $u \in T^*$ as $M\epsilon := M$, $Mau := (Ma)u$. u is said to be* enabled *in M iff Mu is defined. The triple $((N,M), t, (N,Mt))$ is a* transition firing event*.*

Note that the place colouring does not affect the behaviour of the net at all. It will be used in Section 3 to specify abstract markings.

Definition 3 (Transition locality). *The* locality *of the transition $t \in T$ in the net $N = (P, T, F^-, F^+, \Sigma, l, c)$ is the net $loc_N(t) := (P_t = \{p \in P \mid F^-(t,p) > 0 \vee F^+(t,p) > 0\}, \{t\}, F^-|_{\{t\}}, F^+|_{\{t\}}, \Sigma, l|_{\{t\}}, c|_{P_t})$: each component is restricted to just the transition t and the places attached to it via arcs). Any connected net with precisely one transition is also called a locality.*

Throughout this paper, all replacement rules are of a simple "context-free" kind that never depend on, or even add, any tokens.

Definition 4 (Rule). *A context-free transition refinement rule, simply called* rule *in the following text, is a pair $\varrho = (N_l, N_r)$ where N_l is a locality containing only a transition t and the arcs and places attached to it, and $N_r = (P_l \uplus P_{r,new}, T_r, F_r^-, F_r^+, \Sigma, l_r, c_r)$ is a net structure which consists of the places of N_l and possibly some new transitions and places, subject to $\forall p \in P_l \; |F_l^{\pm}(p)| = |F_r^{\pm}(p)|$ for $\pm \in \{+,-\}$, and preservation of the place colours from P_l. N_l is called the* left hand side *and N_r the* right hand side *of the rule.*

Definition 5 (Rule match and application). *A* match *of the locality $N_l = (P_l, \{t_l\}, F_l^-, F_l^+, \Sigma, \{(t_l, a)\})$ in the net N is a mapping $m : P_l \cup \{t_l\} \to P \cup T$. m maps the transition t_l of N_l to a like-labeled transition in N: $l(m(t_l)) = l(t_l)$. m maps the places $p_l \in P_l$ injectively to places in P such that $F^{\pm}(m(t_l), m(p_l)) = m(F_l^{\pm}(t_l, p))$, and $F^{\pm}(m(t_l), p) = 0$ if $p \notin m(P_l)$, for $\pm \in \{+,-\}$. The notion is extended to matches in marked nets: a match of N_l in (N,M) is simply a match of N_l in N. A* match *of the rule $\varrho = (N_l, N_r)$ on a marked net (N_i, M_i) is a match of N_l in N_i. An* abstract application*, with match m, of ϱ to a marked net (N_i, M_i) is a pair $((N_i, M_i), (N_j, M_j))$ such that if t_l is the transition in N_l, $T_j = T_i - \{m(t_l)\} + T_r$, $P_j = P_i + (P_r - P_l)$ ($+$ and $-$ respectively denoting disjoint set union and set difference), M_j coincides with M_i on the places from P_i and has value 0 otherwise, the place colours are as in N_i and N_r, and*

$$F_j^{\pm}(t,p) = \begin{cases} F_i^{\pm}(t,p) & t \in T_i \wedge p \in P_i \\ F_r^{\pm}(t,p) & t \in T_r \wedge p \in (P_r - P_l) \\ F_r^{\pm}(t,p') & t \in T_r \wedge p = m(p') \\ 0 & \text{otherwise} \end{cases}$$

The triple $((N_i, M_i), m(t_l), (N_j, M_j))$ is a rule application event*.*

An *event* is either a transition firing event or a rule application event.

Example 2 (A rule match and application). In this example (Figure 1), there is precisely one possible match of the left hand side. The rule ϱ removes a transition and replaces it with a net. Match and rule induce an application.

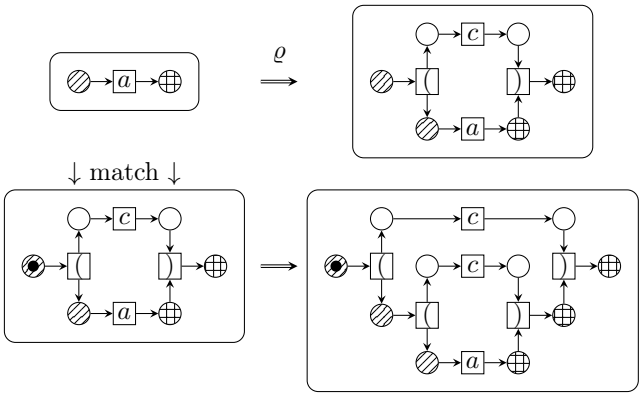


Fig. 1. An application induced by a rule ϱ and a match of its left hand side.

Definition 6 (Structure-changing Petri net). *A* structure-changing Petri net $\mathcal{S}$ *is a tuple* $(\mathcal{R}, \Sigma, R, \tau, s_0, \$)$*, where* $R \cap \Sigma = \emptyset$*,* $\mathcal{R}$ *is a finite set of rules,* R *is a finite alphabet of* rule names*,* s_0 *is a labeled Petri net (the* start state*),* $\$$ *is a set of Petri nets (the set of* terminal *states). Relation* $\tau \subseteq \mathcal{R} \times R$ *assigns to each rule one or more rule names. All nets (*s_0 *and those occurring in the rules) share the transition label alphabet* Σ*.*

Definition 7 (Step Relations and Derivations). *Every structure-changing Petri net* $\mathcal{S}$ *defines for each* $x \in R_\mathcal{S} \cup \Sigma_\mathcal{S}$ *a relation* $\overset{x}{\Rightarrow}_\mathcal{S}$ *between marked nets. If* $x \in \Sigma_\mathcal{S}$*, the transition relation* $\overset{x}{\Rightarrow}$ *is defined as the set of pairs* $((N, M), (N, M'))$ *such that* $M' = Mt$ *and* $l(t) = x$*. If* $x \in R_\mathcal{S}$*,* $\overset{x}{\Rightarrow}_\mathcal{S}$ *is defined as the set of pairs* $((N, M), (N', M'))$ *that are applications of rules* ϱ *with* $\tau(\varrho) = x$*. Let* $\Rightarrow_\mathcal{S} := \bigcup_{x \in R_\mathcal{S} \cup \Sigma_\mathcal{S}} \overset{x}{\Rightarrow}_\mathcal{S}$ *and* $\overset{*}{\Rightarrow}_\mathcal{S}$ *its reflexive and transitive closure.*
An (abstract) derivation (w, ω, σ) *of length* $n \in \mathbb{N}$ *in a structure-changing Petri net* $\mathcal{S} = (\mathcal{R}, \Sigma, R, \tau, s_0, \$)$ *consists of a sequence* $w = r_0 \cdot ... \cdot r_{n-1} \in (R \cup \Sigma)^*$ *of labels, a sequence* $\omega = e_0 \cdot ... \cdot e_{n-1}$ *of events, and a sequence* $\sigma = s_0 \cdot ... \cdot s_n = (N_0, M_0) \cdot ... \cdot (N_n, M_n)$ *of marked nets such that* $\forall i \in \{0, ..., n-1\}$*,* $s_i \overset{r_i}{\Rightarrow}_\mathcal{S} s_{i+1}$ *and, for some transitions* x_i*,* $e_i = (s_i, x_i, s_{i+1})$*. We write* $s_0 \overset{w}{\Rightarrow}_\mathcal{S} s_n$*.*

A marked net s is said to be reachable in $\mathcal{S}$ from s_0 iff there is a derivation in $\mathcal{S}$ that starts in s_0 and ends in s. The marked nets reachable in $\mathcal{S}$ are also called (reachable) *states* of $\mathcal{S}$. $\mathcal{RS}(\mathcal{S})$ denotes the set of all reachable states of $\mathcal{S}$.

Every marked net (N, M) with a specified set of terminal markings can also be seen as a structure-changing Petri net with empty rule set and start state (N, M), and will be called a *static* net. In a static net, instead of derivations one simply considers transition sequences, as they uniquely determine derivations.

A net, rule or structure-changing Petri net is *k-coloured* if the highest colour assigned to any place does not exceed $k - 1$. A marked net is *k-safe* if every reachable marking has at most k tokens per place. A marked net (N, M) is *separable* (strongly separable in [3]) if for any $k \in \mathbb{N}$ every transition sequence enabled in $(N, k \cdot M)$ is an interleaving of k transition sequences of (N, M).

Definition 8 (Terminal structure-changing Petri net language). *Let $\mathcal{S}$ be a structure-changing Petri net with start state s_0. Its* terminal language *is*

$$\mathcal{L}_{\$}(\mathcal{S}) := \{w \in (\Sigma \cup R)^* \mid \exists s \in \$ \; s_0 \overset{w}{\Rightarrow}_{\mathcal{S}} s\}$$

Also, let $\hat{h}$ be the homomorphism that deletes all letters of R and leaves transition labels Σ unchanged. $\hat{\mathcal{L}}_{\$}(\mathcal{S}) := \hat{h}(\mathcal{L}_{\$}(\mathcal{S}))$ is the terminal firing language *of $\mathcal{S}$.*

Given a structure-changing Petri net $\mathcal{S}$, we denote by $W(\mathcal{S})$ the set that comprises s_0 and the right hand sides of all rules in $\mathcal{R}$.

In the following, we introduce workflow nets [11] extended by structure-changing rules as a special case of structure-changing Petri nets.

Definition 9 (Workflow net structure). *A workflow net structure is a tuple $(N, (p_i, p_o))$ consisting of a Petri net structure N and a pair of distinguished places $p_i, p_o \in P$, the input and output place which have no input resp. no output arcs, subject to the requirement that adding an extra transition from p_o to p_i would render the net structure strongly connected.*

The data (p_i, p_o) need not be made explicit, since these places are easily seen to be uniquely determined in a workflow net. Thus we are justified in treating a workflow net as a special Petri net.

The *start marking* of a workflow net structure N, i.e. the marking where only the unique place with $\Sigma_{t \in T} F^+(t, p) = 0$ is marked with one token and all other places are not marked, is denoted by ${}^\bullet N$. The *end marking* where just the place with $\Sigma_{t \in T} F^-(t, p) = 0$ is marked with exactly one token is denoted by $N^\bullet$.

Definition 10 (Sound workflow net). *A workflow net N is* sound *iff from any marking reachable from ${}^\bullet N$, a marking is reachable where p_o is marked exactly once, and for each transition t there is some marking M reachable from ${}^\bullet N$ such that t is enabled in M.*

A *structure-changing workflow net* is a structure-changing Petri net whose every reachable state is a reachable state of some workflow net, and whose terminal

markings \$ are workflow nets $(N, N^\bullet)$ marked with their end marking, and whose initial marking s_0 is a workflow net $(N, {}^\bullet N)$ with start marking.

Note that soundness implies boundedness and, as remarked in [13], replacing a transition of a sound workflow net with a sound workflow net does not always result in a sound workflow net.

Definition 11 (Simple structure-changing workflow net). *A structure-changing workflow net $\mathcal{S}$ is said to be* simple *if*

(1) for every rule $(N_l, N_r) \in \mathcal{R}_\mathcal{S}$, $\forall p \in P_l(F^-(t_l, p) + F^+(t_l, p)) = 1$,

(2) for every rule $(N_l, N_r) \in \mathcal{R}_\mathcal{S}$, N_r is a sound workflow net structure, and its start place is the start place of N_l, and $(N_r, {}^\bullet N_r)$ is 1-safe and separable,

(3) s_0 is a 1-safe sound workflow net marked with its start marking.

(1) together with (2) implies that only single-input, single-output transition localities are permitted as left hand sides. Although the restrictions rule out markings with multiplicities, it is now possible to create more instances of a subnet just by replacing transitions. Simple structure-changing workflow nets can still capture situations such as workflows that undergo complications as they are executed. The net structures in Example 2 are actually sound workflow net structures; the nets of the lower row are marked with their start markings. As a structure-changing Petri net, it also meets the requirements of Definition 11.

3 Analysis of structure-changing Petri nets

In this section, we investigate some decision problems for structure-changing workflow nets. We show that language containment in a regular language is undecidable, but the word problem and abstract reachability problem are decidable for a restricted class (acyclic simple structure-changing workflow nets).

We prove a series of lemmata that allow us to equivalently re-arrange derivations, which will be convenient in later proofs.

Lemma 1 (Independence). *Given a structure-changing Petri net $\mathcal{S}$, if $x \in R_\mathcal{S}$, $a \in \Sigma_\mathcal{S}$, (1) $(N, M) \overset{x}{\Rightarrow}_\mathcal{S} (N', M')$ and (2) $(N, M) \overset{a}{\Rightarrow}_\mathcal{S} (N, M'')$ and the transition of the match in (1) is distinct from the transition of the firing in (2), then $(N', M') \overset{a}{\Rightarrow} (N', M''')$ and $(N, M'') \overset{x}{\Rightarrow} (N', M''')$ with*

$$M'''(p) = \begin{cases} M''(p) & p \in P \\ 0 & \text{otherwise} \end{cases}$$

Proof. Immediate from Definitions 5 and 2.

Lemma 2 (Permuting Independent Events). *If $\mathcal{S}$ is a structure-changing Petri net, for every derivation (w, ω, σ) of length n in $\mathcal{S}$, under the conditions*

given below there is a derivation $(w_\pi, \omega_\pi, \sigma_\pi)$ *of same length such that* $\sigma_n \cong (\sigma_\pi)_n$*, and* $w = uabv$ *and* $w_\pi = ubav$*. If* $a = w_i$ *corresponds to the event* $e_i = ((N_s, M_s), t_a, (N_{s'}, M_{s'}))$ *and* b *corresponds to* $e_{i+1} = ((N_{s'}, M_{s'}), t_b, (N_{s''}, M_{s''}))$*,*

Situation	*Sufficient condition for transposition*
$a \in \Sigma, b \in \Sigma$	$\forall p \in P, F^-(t_b, p) > 0 \Rightarrow F^+(t_a, p) = 0$
$a \in \Sigma, b \in R$	$t_a \notin T_s - T_{s'}$
$a \in R, b \in \Sigma$	$t_b \notin T_{s'} - T_s$
$a \in R, b \in R$	$t_b \notin T_{s'} - T_s$

Proof. $a, b \in \Sigma$: independent transitions can be transposed in an enabled sequence: in a net (N, M), if $t_1, t_2 \in T$ are enabled in M and the above condition holds, then $Mut_1t_2v = Mut_2t_1v$: t_1 is enabled in Mut_2, and by commutativity of addition $Mut_1t_2 = Mut_2t_1$.

$a \in \Sigma, b \in R$: we use Lemma 1 and notice that the b-labeled rule can still be applied, yielding the same result as ab. The argument for $a \in R, b \in \Sigma$ is similar.

$a, b \in R$: since only the matched transition is replaced and all others remain unchanged, the rules can be swapped without changing applicability or result.

3.1 Word Problem

We consider the word problem for structure-changing Petri nets.

Problem 1 (Word problem).

Given:	a structure-changing Petri net $\mathcal{S}$ and a word $w \in \Sigma^*$
Question:	$w \in \hat{\mathcal{L}}_\$(\mathcal{S})$?

It is a nontrivial question whether a certain word w occurs in the terminal firing language of $\mathcal{S}$. The question is decidable for simple structure-changing workflow nets by a search: while applying rules, one keeps track of the minimum length of any terminal word in which each transition occurs. Transitions that can only be used in words longer than w are not added. Rule applications are always postponed per Lemma 2 until they are needed. The creation of redundant transitions is limited by the definition.

Every word au of $\hat{\mathcal{L}}_\$(\mathcal{S})$ is generated by a derivation $(zau', e\omega, s\sigma)$, $z \in R^*$, $\hat{h}(u') = u$. It either starts with the firing of a transition a ($z = \epsilon$) or with a rule application. When exploring the first case, we will query for a derivation (u', ω, σ). Otherwise, rule applications must be tried. Since replacement can go on indefinitely, we again assume that the derivation has all rules postponed as far as possible so that in the sequence z each rule depends on the previous one.

Minimum Word Length In a static net with start state (N, M_0) and final states \$, let $\| \cdot \|_{(N,M_0)} : T \rightharpoonup \mathbb{N}$ be the partial function assigning to each transition t

of N the minimum length, if any, of a derivation using t and ending in a \$. It is undefined iff there is no such derivation, otherwise $\|t\|_{(N,M_0),\$} = min(\{|utu'| \in \Sigma^* \mid M_0utu' \in \$\})$, where $|w|$ is the length of a word w.

While calculating the minimum length might in general not be very efficient, it is certainly possible for all sound workflow nets. Note also that transition sequences in T^*, not label words in Σ^*, are interesting.

Lemma 3. *Let $\mathcal{S}$ be a simple structure-changing workflow net. If $s_0 \overset{*}{\Rightarrow}_{\mathcal{S}} (N, M)$, and $(N, M) \overset{r}{\Rightarrow}_{\mathcal{S}} (N', M')$ is an application of a rule (N_l, N_r) matching the locality of a transition t, then each newly added transition t' in $T' - T$ has $\|t'\|_{(N',M'),\$} = \|t\|_{(N,M),\$} - j + j \cdot \|t'\|_{(N_r,{}^\bullet N_r),N_r{}^\bullet} \geq \|t\|_{(N,M),\$}$, j being the minimum number of occurrences of t in any shortest word $utu' \in \hat{\mathcal{L}}_\$(\mathcal{S})$.*

Proof. We prove that a shortest word containing t' in (N', M') must be a word that is related by transpositions to a minimum length t-word that has a minimum-length t'-word of $(N_r, {}^\bullet N_r)$ substituted for every occurrence of t.

Fact (1): a transition sequence $v \in T_r^*$ is enabled in N' precisely when it is in N_r (changing the P_l places to their images under m) and has the same effect on the places of $P_r - P_l + m(P_l)$ and zero effect on all other places.

Each transition sequence u that is enabled in M' can be permuted by repeated applications of Lemma 2 to another equivalent sequence $\tilde{u}$ where all firings of $T' - T$ transitions are moved as far as possible towards the end of the sequence (by which we mean we chose an equivalent sequence whose image under the homomorphism $t \mapsto$ (if $t \in T$ then α else β) is minimal with respect to the lexicographic order induced by $\alpha < \beta$),

Using fact (1), and separability, the sequence $\tilde{u}$ can always be decomposed as $\tilde{u} = x_0 \cdot y_0 \cdot \ldots \cdot y_n \cdot x_{n+1}$, where $x_i \in T^*$ and $y_i \in (T' - T)^*$ and $(k \cdot ({}^\bullet N_r))y_i = k \cdot (N_l{}^\bullet)$, $|y_i| \geq \|t'\|_{(N_r,{}^\bullet N_r),\{N_r{}^\bullet\}}$. because events involving $(T' - T)$ may move to the right until hitting a transition firing event removing the token from the output place $m(p_o)$. Therefore, $\tilde{u}$ is not shorter than a word using t. namely, the word $x_0 \cdot t \cdot \ldots \cdot t \cdot x_{n+1}$, which is also a shortest word: otherwise, a shorter word using t' could be obtained by choosing the actual shortest t-word and replacing each t with the shortest N_r-t'-word.

Up to now, it is not clear whether the word problem for structure-changing workflow nets is decidable. For simple structure-changing workflow nets, annotating all transitions with the minimal word length limits the depth of the search for the R^*-labeled prefix of the derivation. The definition of a rule guarantees that repeated applications will not create unboundedly many parallel transitions.

Proposition 1 (Decidable word problem). *The word problem for simple structure-changing workflow nets is decidable.*

Proof. We modify $\mathcal{S}$ to obtain a new structure-changing Petri net $\mathcal{S}'$: in the start state (N, M), each a-labeled transition is relabeled to $(\|t\|_{(N,M),\$}, a)$. Replace

each rule $\rho \in \mathcal{R}$ by the set of rules $\{\rho_i \mid i \leq |w|\}$, where the left hand side transition's label is (i, a) instead of ρ's a, the transitions t' of the right hand side N_r are relabeled $(i - 1 + \|t'\|_{(N_r, \bullet N_r), N_r \bullet}, b)$ (where b is the old label of t'). All transitions with $i - 1 + \|t'\|_{(N_r, \bullet N_r), N_r \bullet} > |w|$, and places appearing only in the pre-set of such transitions, are removed and do not appear in the rule variant ρ_i. If this leaves no transitions, then ρ_i is discarded. This way, the first component of each transition's label always underapproximates the minimum length, and strictly increases the counter except if $\|t'\|_{(N_r, \bullet N_r), N_r \bullet} = 1$. This does, by design, not impact the language.

Explore the set of reachable states of $\mathcal{S}'$. Start searching for appropriate derivations by either firing a transition whose label is equal to the next letter of the word, if possible, or applying a rule to a transition whose pre-set place is marked. The number of rules is finite and in every state, every rule has a finite number of matches: the set of successor states is always finite.

To bound the depth of the search, until a transition is fired, each rule depends on the previous one, and only a transition created by the last rule is fired. This corresponds to moving all rules of a derivation as far as possible to the right, and this is almost enough to guarantee termination. The transition eventually fired could have the same label, pre- and postset as a transition replaced earlier, but then some rule applications would have been unnecessary.

It is easy to check for cycles, which can only happen if the right hand side of some rule of the modified system consists of a single transition locality.

The following algorithm may illustrate the procedure.

Algorithm 1 (Solving the word problem for $\hat{\mathcal{L}}_{\$}(\mathcal{S})$)

$u := \epsilon$
$St := \{s_0\}$
repeat
  repeat
    for all $s \in St$ ***do*** ; *gather all relevant* $s \overset{z}{\Rightarrow}_{\mathcal{S}'} s'$
      for all *pre-marked transitions* t *of* s ***do***
        if $\nexists s'' \in St. s' \cong s''$ ***then***
          add s' *to* St
  until *no new states found*
  $St := \{(N, Ma) \mid (N, M) \in St\}$
  $u := ua$ *(*$\exists u'$ *s.t.* $w = uau'$*)*
until $u = w$

3.2 Reachability Problems

Since system states can be arbitrarily large nets, to specify interesting state properties it is insufficient to specify the number of tokens on specific, concrete places. Instead, we want to state these constraints generically, for all places of a given colour. The place colours may carry a model-specific semantics, or just

encode structural information about the net structure. In this subsection, we therefore introduce the notion of an abstract marking: a vector that counts the total number of tokens on places of each colour in a structure-changing Petri net. First, we note that reachability of a given marked net is decidable.

Proposition 2 (Deciding the concrete reachability problem). *The problem whether a certain marked net (N, M) is a reachable state is decidable for simple structure-changing workflow nets.*

Proof. All reachable states are bounded nets and the rules of $\mathcal{S}$, finite in number, are monotonic with respect to the size of the net. A breadth-first search along the relation $\Rightarrow_{\mathcal{S}}$ hence decides the reachability problem.

We define $c^{\oplus}(N, M)$ as the function $\mathbb{N} \to \mathbb{N}$, $i \mapsto \sum_{p \in c^{-1}(i)} m(p)$. For k-coloured nets, we restrict the domain of $c^{\oplus}$ to $0, ..., k-1$. The set of images under $c^{\oplus}$ of the reachable states of $\mathcal{S}$ is denoted by $\mathcal{ARS}(\mathcal{S})$, for *abstract reachability set.*

Problem 2 (Abstract reachability).

Given	a finite number k of *colours* a vector $\check{c} : \{0, ..., k-1\} \to \mathbb{N}$ a structure-changing Petri net $\mathcal{S}$
Question	reachability of $\check{c}$ as an abstract marking

To obtain the following results, we introduce an auxiliary construction. To any structure-changing Petri net one assigns a fixed static net recording the number of times a certain rule has been applied, and how many tokens in total are present in the places of right hand side nets created during a derivation.

Construction 1 (Plan) *The* **plan** *of a k-coloured structure-changing Petri net $\mathcal{S} = (\mathcal{R}, \Sigma, R, \tau, s_0, \$)$ is a $k+1$-coloured net structure $\nu(\mathcal{S})$ where: the set of places P^{ν} is the disjoint union $\biguplus_{N_W \in W(\mathcal{S})} p \in P_W$ plus $\{p_\mu \mid \mu \in match(\mathcal{R}, W)\}$, the transitions T^{ν} are those of the nets of $W(\mathcal{S})$, $\biguplus_{N \in W(S)} t \in T_W$, plus $\{\mu^{\uparrow} \mid \mu \in match(\mathcal{R}, W)\} \cup \{\mu^{\downarrow} \mid \mu \in match(\mathcal{R}, W)\}$. The pre- and postset multiplicities are those of the individual nets in $W(\mathcal{S})$: $(F^{\pm})^{\nu}(t, p) = F^{\pm}(t, p)$ for $N \in W, \pm \in \{+, -\}$, extended for the match transitions: $F^{-^{\nu}}(\mu^{\uparrow}, p_\mu) = 1$, $F^{-^{\nu}}(\mu^{\uparrow}, p) = 1$ if p is the place in the preset of $\mu(t_l)$, $F^{+^{\nu}}(\mu^{\uparrow}, p) = 1$ if p is the start place of the right hand side of ρ, $F^{+^{\nu}}(\mu^{\uparrow}, p_{\mu'}) = 1$ if μ' is a match on the right hand side of the rule associated with μ. Σ and c are taken from the W nets and the places which are not from any net in W are assigned the new colour. All transitions keep their labels; the $\mu^{\uparrow}$ and $\mu^{\downarrow}$ have the corresponding rules as labels. Here,* $\mathrm{match}(\mathcal{R}, W)$ *is the set of all matches of rules $\mathcal{R}$ on nets of W. The start marking $\tilde{M}$ places exactly one token on every p_μ where μ is a match on s_0, the places from s_0 are marked with the start marking of s_0 in $\mathcal{S}$.*

Example 3 (A plan with start marking). Figure 2 shows $\nu(\mathcal{S})$ for a simple structure-changing workflow net $\mathcal{S}$.

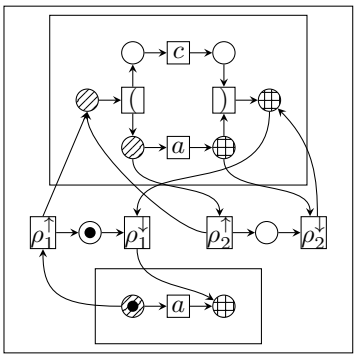


Fig. 2. Plan of a system $\mathcal{S}$ with one rule as in Example 2, and start state s_0 (below). The rectangular boxes highlight the nets of $W(\mathcal{S})$.

Definition 12 (Future Class). *Let (N, M) be an acyclic marked net. The future class $\mathcal{F}(N, M)$ is the set of all nets (N', M) that yield the same result under the operation F, defined as follows: F turns nets into nets, preserving only the places, transitions and arcs which are either places marked with a non-zero number of tokens, or graph-reachable from a marked place, or belonging to the locality of a transition graph-reachable from a marked place.*

Lemma 4 (Future classes share similar behaviour). *If two marked nets (N, M) and (N', M') have the same future class, then neither the set of reachable abstract markings nor the terminal firing language differ when comparing structure-changing Petri nets $(\mathcal{R}, \Sigma, R, \tau, (N, M), \$)$ and $(\mathcal{R}, \Sigma, R, \tau, (N', M'), \$)$.*

Proof. In any state s, any rule applied to the part of the state not in $F(s)$ does not lead into a different future class. In all other cases, if s, s' in the same future class, $\forall s \stackrel{x}{\Rightarrow}_{\mathcal{S}} s''; \exists s''' \; s' \stackrel{x}{\Rightarrow} s'''$ with $F(s'') = F(s''')$, because no new arcs are added to places or transitions not in $F(s)$ or $F(s')$.

Proposition 3. *The abstract reachability problem is decidable in a simple structure-changing workflow net $\mathcal{S}$ such that all nets in $W(\mathcal{S})$ are acyclic.*

Proof. By reduction to the Petri net reachability problem for $(\nu(\mathcal{S}), \tilde{M})$, whose decidability is well known. Under the above acyclicity condition, a k-coloured simple structure-changing workflow net $\mathcal{S}$ and $\nu(\mathcal{S})$ have similar sets of reachable abstract markings, $\mathcal{ARS}(\mathcal{S}) = \{M|_{\{0,\ldots,k-1\}} \mid M \in \mathcal{ARS}(\nu(\mathcal{S}), \tilde{M})\}$.

The inclusion from left to right is shown by induction over the length of a derivation (w, ω, σ), where all rule applications are moved as far as possible to the right by applying Lemma 2 (in the same sense as in the proof of Lemma 3) such that any rule application directly precedes the firing of a transition created by it. If none of its transitions or their descendants by further rule applications are

ever fired, the rule application has no effect on the reachable abstract markings and simply moves to the end of the derivation, where it can be ignored.

Induction hypothesis: reachability of the state (N, M) with $c^{\oplus}(N, M) = \check{c}$ in $\mathcal{S}$ implies reachability of a marking $\tilde{M}$ with $c^{\oplus}(\nu(\mathcal{S}), \tilde{M})|_{\leq k} = \check{c}$ in $\nu(S)$, and $\exists \alpha : P \cup T \to P^{\nu} \cup T^{\nu}$ mapping places of N to places of $\nu(\mathcal{S})$ and transitions to transitions such that (1) $\forall p \in P^{\nu}, \Sigma_{p \in \alpha^{-1}(p)} M(p) = M^{\nu}(p)$.

In the induction step one verifies that transitions corresponding to net transitions can always be simulated, $\mu^{\uparrow}$ transitions can be fired whenever the corresponding rule occurs, preserving (1), and $\mu^{\downarrow}$ transitions can be fired to get the number of colour k tokens back down and move the tokens back to where they would have been without the rule application, which results in the same future class. Transitions left over at the right end of the word do not change the abstract marking reached, which concludes the first half of the proof.

Conversely, to prove the inclusion from right to left we also require the existence of a suitable mapping α in the induction hypothesis and construct a derivation of $\mathcal{S}$ leading to the same abstract marking. In the induction step, if $(\nu(\mathcal{S}), M) \stackrel{x}{\Rightarrow} (\nu(\mathcal{S}), M')$, we must again distinguish 3 cases regarding the transitions: x corresponds to a firing of a net transition, or it is a $\mu^{\uparrow}$ or a $\mu^{\downarrow}$ event. If $x \in \Sigma$ corresponds to the firing of t in $\nu(\mathcal{S})$, there are two cases: since we can produce a state of $\nu(\mathcal{S})$ (as in the first half of the proof) by letting it simulate a derivation (w, ω, σ) of $\mathcal{S}$, the event in $\nu(\mathcal{S})$ can correspond to a transition firing event in the state s_i reached in $\mathcal{S}$ at a certain point.

However, depending on the distributions of the tokens on the places mapped by α to the places in the $\nu(\mathcal{S})$-preset of t, there may be no transition labeled x such that $\alpha(x) = t$ enabled in s_i. This happens when the rule whose right hand side contains t was used more than once and the tokens are scattered over different instances. Then our reverse simulation must backtrack, (w, ω, σ) must be permuted to another derivation that would enable a suitable x-labeled transition: first note that enabledness of such a transition in a reachable state s of $\mathcal{S}$ depends only on $\mathcal{F}(s)$, portions depending on replacing one of the transitions of $N_i \in W$ can be omitted from the derivation without changing the resulting state and we are left simply with a firing sequence in N_i from some marking. Using separability, this sequence is decomposed into an interleaving of firing sequences in $(N_i, {}^{\bullet}N_i)$ such that one of the individual firing sequences leads to a marking that activates the transition in question.

If t is a $\mu^{\uparrow}$ transition in $\nu(\mathcal{S})$, it corresponds to application of a rule to an enabled transition (since the derivation has all rule applications moved to the right, and also by Lemma 2 the rule application introducing a certain transition can be moved directly before the first, and in our case only, firing of that transition). Finally, if t is a $\mu^{\downarrow}$ transition in $\nu(\mathcal{S})$, it corresponds to forgetting a rule application that has no further effects on $\mathcal{F}$, concluding the proof.

Note how the first part of the proof still holds under more relaxed assumptions, but the reverse implication crucially depends on simplicity and acyclicity.

3.3 Containment Problems

In this subsection, we study the inclusion of the terminal firing language of a structure-changing workflow net in a given regular language. The motivation is that the regular language can specify all desirable net behaviour, and the problem is to check whether any undesirable firing sequences exist or not.

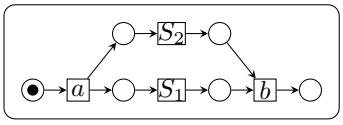
Problem 3 (Regular Specification).

Given	a regular language $L \subseteq \Sigma^*$ a simple structure-changing Petri net $\mathcal{S}$
Question	$\hat{\mathcal{L}}_\$(\mathcal{S}) \subseteq L$?

It is well known that the emptiness of the intersection of two context-free languages is undecidable. Knowledge of context-free languages is now assumed.

Proposition 4 (Undecidability of abstract language compliance). *Problem 3 is undecidable even when restricted to acyclic, simple structure-changing workflow nets with no more than* 2 *tokens in every reachable state.*

Proof. By reduction from the emptiness problem for the intersection of two context-free languages. Let $G_1 = (V_1, T, P_1, S_1)$ and $G_2 = (V_2, T, P_2, S_2)$ be two context-free grammars in Greibach normal form, w.l.o.g. with $V_1 \cap V_2 = \emptyset$. We construct the structure-changing workflow net $\mathcal{S}(G_1, G_2)$. s_0 is as shown:



Transition labels are used to encode the terminals and non-terminals of the grammars. For G_1, we use plain labels; for G_2, we use overlined labels from $\overline{T} := \{\overline{a} \mid a \in T\}$. For each production $p = (X, aX_1...X_n)$, the rule ρ_p replaces a transition labeled X in the obvious way with a chain of transitions labeled a (resp. $\overline{a}$), X_1 to X_n. Let the regular language L be $a\{a\overline{a} \mid a \in T\}^*b$.

Every rule application leads to a net $N(sf_1, sf_2)$ that is built exactly like s_0 except that the special a and b transitions are linked by two chains of transitions encoding a sentential form of G_1 and one of G_2. By disjointness of the non-terminal alphabets, rules corresponding to productions of G_1 apply only to the chain encoding the sentential form derived in G_1, same for G_2.

For every word $w \in L_1 \cap L_2$, there is a derivation (w', ω, σ) with $\hat{h}(w') = w$, by converting a (context-free grammar) derivation in G_1 to a derivation that starts by replacing the transition labeled S_1, and so on, likewise for G_2. Then, all the transitions can be fired, resulting in a word of $L \cap \hat{\mathcal{L}}_\$(\mathcal{S}(G_1, G_2))$.

Conversely, each word $w \in L \cap \hat{\mathcal{L}}_\$(\mathcal{S}(G_1, G_2))$ must be obtained by applying rules that correspond to productions in the two grammars, and a word of L (containing no non-terminals) can only be obtained by firing a, then stepping through two copies of the same word w (one overlined), then firing b.

$\mathcal{S}(G_1, G_2)$ is a simple structure-changing workflow net easily seen to have as reachable states only acyclic nets marked with one or two tokens.

4 Related Work

Petri nets can be extended with structure changes via graph transformations, as in [10]. Graph transformation systems [6] define derivation steps according to rules that serve to reconfigure the Petri net dynamically and can be mixed with transition firings. Their work, and ours, is thus closely related to graph transformation systems and results from graph transformation such as local Church-Rosser and concurrency theorems can be adapted, see [7, 10], but we have concentrated on Petri net specific aspects in this paper.

Further noteworthy work includes the box calculus [4], reconfigurable nets [1] and open nets [2]. The extensions cited here are much more general and allow structure changes beyond dynamic transition refinement, at the expense of decidability. Also, safe nets-in-nets [8] are somewhat similar to structure-changing ones. With arbitrarily deep net token nesting, it is possible to simulate a a 2-counter machine with counters and zero-tests. Finally, in [13], a notion of refinement for workflow net similar to ours was investigated. Indeed, the notion of separability was first introduced in that work.

5 Conclusion

Overview of the results ("+" means decidable, "−" undecidable, "?" unknown):

	general	simple	acyclic simple
word problem	?	+	+
concrete reach.	?	+	+
abstract reach.	?	?	+
regular spec.	−	−	−

The word problem turns out to be nontrivial (especially when the formalism is modified to allow creation of new parallel transitions, in which case it still seems to be decidable). Proposition 4 places severe limitations on the algorithmic analysis of structure-changing Petri nets. Even for systems with a simple structure and a globally bounded token number, language containment questions are undecidable due to the possibility of imposing synchronisation on concurrent context-free processes.

Structure-changing nets with arbitrary local replacement rules hardly seem to offer promising analysis methods. It seems most fruitful to investigate classes of nets that behave sufficiently like the "simple" ones presented here, and on the other hand to apply general graph transformation analysis methods. We conjecture that the word and concrete reachability problems, but not the abstract reachability problem, are decidable for general structure-changing workflow nets.

We plan further work on decidable problems for relatively simple subclasses of structure-changing Petri net. We hope that not all restrictions are really necessary to obtain the decidability results and hope to better determine the boundaries of decidability of these problems. On the other hand, we must fill in the left column of the above results table.

Acknowledgements

We would like to thank Annegret Habel and Martin Fränzle, as well as Thomas Strathmann for constructive discussions, and the reviewers for their comments.

References

1. Badouel, E., Llorens, M., Oliver, J.: Modeling concurrent systems: Reconfigurable nets. In: Proceedings of PDPTA'03 (2003)
2. Baldan, P., Corradini, A., Ehrig, H., Heckel, R., König, B.: Bisimilarity and behaviour-preserving reconfigurations of open Petri nets. In: Algebra and Coalgebra in Computer Science, LNCS, vol. 4624, pp. 126–142. Springer (2007)
3. Best, E., Darondeau, P.: Separability in persistent Petri nets. FI 113(3–4), 179–203 (2011)
4. Best, E., Devillers, R., Hall, J.: The box calculus: A new causal algebra with multi-label communication. Advances in Petri Nets 1992 pp. 21–69 (1992)
5. Desel, J., Reisig, W.: Place/transition Petri nets. In: Lectures on Petri Nets I: Basic Models, pp. 122–173. Springer (1998)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
7. Ehrig, H., Hoffmann, K., Padberg, J., Prange, U., Ermel, C.: Concurrency in reconfigurable P/T systems: Independence of net transformations and token firing in reconfigurable P/T systems. In: ICATPN 2007, pp. 104–123. Springer (2007)
8. Köhler-Bußmeier, M., Heitmann, F.: Safeness for object nets. FI 101(1), 29–43 (2010)
9. Kreowski, H.J.: A comparison between Petri-nets and graph grammars. In: Graph-theoretic Concepts in Computer Science, LNCS, vol. 100, pp. 306–317. Springer (1981)
10. Modica, T., Gabriel, K., Hoffmann, K.: Formalization of Petri nets with individual tokens as basis for DPO net transformations. In: PNGT 2010, ECEASST, vol. 40 (2011)
11. van der Aalst, W.M.P.: Verification of workflow nets. In: ATPN 1997, LNCS, vol. 1248, pp. 407–426. Springer (1997)
12. van der Aalst, W.M.P.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers 3(3), 297–317 (2001)
13. van Hee, K., Sidorova, N., Voorhoeve, M.: Soundness and separability of workflow nets in the stepwise refinement approach. In: ATPN 2003, LNCS, vol. 2679, pp. 337–356. Springer (2003)
14. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. DKE 66(3), 438–466 (September 2008)

Self-Developing Network : A simple and generic model for distributed graph grammars.

Frédéric Gruau, Luidnel Maignan

Laboratoire de Recherche en Informatique, Université Paris Sud

Résumé This work investigates what is the simplest way of rewriting graph in a distributed way. Nodes of the graph are identified as independent agents doing themselves the rewriting, hence the appellation "Self-Developing Network" (SDN).

We first study the most general way of rewriting a single independent agent-node. We define two complementary ways of selecting neighbors based on link labels : an individual random selection allowing to process neighbors one by one, and a collective selection allowing the creation of connections towards arbitrary many neighbors. The resulting node-rewriting rules can be applied in a distributed way, provided two neighbors are never simultaneously ready. This constitutes our first working definition of elementary SDNs.

This neighbor-exclusive requirement can be relaxed by using link orientation and adjusting the semantic of rule application. The resulting enhanced model can be programmed as a layer on top of elementary SDNs. We finally obtain a definition of SDN that is : 1- simple enough to be considered as canonical 2- not dependent on the particular scheme used to achieve distributed execution 3- generic enough to allow many other expressive notations, and a classification of existing well-known models as specific SDNs.

Keywords: Self-developing network, massive parallelism, simulation, formal system, graph grammar, distributed system

1 Introduction : motivation for a new model.

We believe there is a gap to be filled in the catalog of formal models describing parallelism : namely, systems where computing is *synonymous of creating exploitable parallelism.* Informally, a Self Developing Network (SDN) consists of a network of decentralized finite-state agents that update independently in parallel. They can modify their states, but also produce other agents and connections thereby "self-developing" the network.

Computing with SDN is *synonymous of creating parallelism.* An SDN not only specifies a network of agents operating in parallel, it also encodes a parallel development of that network which can grow arbitrary large out of a single ancestor agent, and can also shrink back. An agent is very fine grain : it holds

a single scalar data, so it cannot do anything meaningful by itself[1], therefore, doing computation is *synonymous* to developing a network. An arbitrary large set of n scalars can be processed, by developing a network of n agents. We have been studying sorting and matrix multiplication in [3]. Each agent also holds rules that will generate appropriate connections between agents, let them communicate their scalar, and do the necessary computation in order to solve the problem at hand.

Computing with SDN is synonymous of creating *exploitable* parallelism. There exists already many formal parallel models : process algebra, pi-calculus, population protocols...To our knowledge, they all use a *global name space* : a process can communicate with any other, using its name or id. The use of global identifier implicitly requires a shared memory for communication, which is not conducive to scalable parallel performance. In SDNs, communication is local : two agents can communicate only if there exists a connection between them. When generating new agent, connection inheritance is specified. In other words, graph rewriting is done. Since the communication pathways are known at every instant during execution, and are updated in a "continuous way", the whole network of agents can be dynamically mapped on hardware, or more precisely, continuously re-mapped during development. The parallelism will be exploitable, if the developed network match the processor network of the target parallel hardware. For example, it won't be efficient to develop a 3D grid on a 2D mesh. The problem of an efficient parallel implementation of self-development is our long-term project. In [3], we discuss the theoretical foundation, and in particular, how to dynamically map the developed network on hardware.

We have been using SDNs for quite a long time to program real tasks, and have become convinced of their generality. Several existing formalisms such as so-called "edNCE" graph grammars, L-system or self-assembly system can be considered as restrictions of SDNs. Special-purpose examples of SDNs exist in the literature, a nice example [7] uses SDN to simulate the Von-Neuman self-reproduction. The rules are devised so that while developing, each created agent always has exactly three neighbors, which in turn, allows to bound the possible rule application contexts. The goal of this paper is to give a formal definition of SDNs on which the community can agree. It should be simple and generic. We propose 3 steps :

1. Build Elementary SDNs which are simple to define but requires mutual neighbor exclusion.
2. Use directed link in order to relax the requirements of mutual exclusion. The resulting model can be programmed on top of elementary SDNs. It can therefore be considered as a higher-level SDNs.
3. Add more programming layers, to either improve the expressiveness, or classify existing approaches as specific instances of SDNs.

1. When an implementation is considered [3], agents will keep moving through hardware in order to accommodate development, therefore agents need to be lightweight.

2 Elementary Self Developing Networks

Self Developing Networks (SDNs) consider networks of agents which can communicate between local neighbors, and create new agents. Upon creation, connection are also inherited locally : a created agent is connected to neighbors of its parent agent. The formal tool to modify a graph in such a local way, is known and studied as Graph Rewriting rule Systems [6] (GRS). A configuration is a *labeled* graph (node and edge). The classic form of a graph rewriting rule includes a left member and a right member which are both graphs. A rule application involves two phases :

1. Matching the left member with a sub-graph of the graph being rewritten
2. Removing this sub-graph, adding the right member, and gluing it to the rest of the graph.

The third volume of [6] is entirely dedicated to *parallel GRS*, however it considers only how to formally construct a parallel composition of rewriting rules. By contrast, SDNs can be informally defined as *distributed GRS* which consider a distribution of the network onto several processing elements and would like to execute concurrently several rewrites concerning different parts of the network. In general, distributed rule application involves a difficult *partition problem* : the graph has to be partitioned into disjoint sub-graphs forming valid left members for different rules. First, matching a sub-graph is itself a difficult operation since it is a graph homomorphism, which is known to be NP complete. Second, there may be many possible ways of partitioning.

Section Outline. We will defined SDNs rewriting rules as the minimum number of restrictions to add to generic graph rewriting rules in order to obtain a distributed-GRS. We are interested by the computation that can be done by the developing network, we therefore have to define the inputs and outputs. Finally we introduce elementary SDNs which just need an additional constraint of mutual exclusion ensuring a decentralized execution.

2.1 Node rewriting Rule

As a first requirement, we impose that the replaced subgraph is reduced to a single node, such rules are usually called *node-rewriting rule*. node rewriting greatly simplifies the partition problem, we know that each partition contains exactly one node, what is left to do is partitioning the edges. We will present two equivalent method for that purpose : by mutual exclusion, and by link orientation. More importantly, with node rewriting, the network being rewritten becomes like a "growable parallel hardware". We can imagine that the nodes are independent agents doing locally the matching, adding, and gluing in a distributed way ; hence the appellation "self developing network" Some examples of Node-rewriting rule such as EdNCE graph grammar [6] have been proposed in the literature of graph grammar. In this paper, by node-rewriting we mean the most generic rewriting that can be done by a node, while still allowing for a

distributed execution between different nodes. Consider an agent a in the process of creating n new agents indexed by $i = 1 \dots n$. We now need to answer the following question : how can a specify new connections between those n new agents, and also between the former neighbors of a, in the most generic way ? Let $q \in Q$ be the node's label also called the agent *state*, and let $l \in L$ be the connection's labels. Because we want agents to be independent, our agent a cannot access the state of its neighbors. Its local view is therefore the set $C(a)$ of its connections labels. Since a given label can appear several times for distinct connections, $C(a)$ is in fact a multi-set of labels. An agent cannot distinguish between two neighbors connected via links carrying the same label l, which we will call l-neighbors. Let $|a|_l$ be the number of l-neighbors for a given label l. In order to be able to bind neighbors individually, whenever $|a|_l > 1$, a needs to do a prior non-deterministic indexing of labels $l_i, i = 1 \dots |a|_l$ of its l-neighbor.

New connections are specified using a *connecting triplet* $(l_{\text{new}}, v_1, v_2,)$, with the new connection label l_{new} and the extremities v_1, v_2 which can be either of :

1. a newly created agents, specified by its index in $\{1 \dots n\}$;
2. an *individually bound* neighbor specified by an indexed label $l_i, i \in 1..|a|_l$;
3. all neighbors having a given label $l \in L$, and not individually bound, in which case, up to $|a|_l$ connections can be created simultaneously.

The third connecting mode is called *collective binding*, and allows to handle arbitrary large contexts, with a finite list of connecting triplets. Such bindings are necessary, because the number of neighbors for a given agent can grow arbitrary large through development. Collective bindings can be used only for one extremity v_1 exclusive v_2 in a connecting triplet $(l_{\text{new}}, v_1, v_2,)$. If it was used for both v_1 and v_2, the number of created connections would be quadratic with respect to the degree of a node. For example, an agent with 10 connections labeled l_1 and 10 other connections labeled l_2 would create 100 connections labeled l_{new} with a connecting triplet $(l_{\text{new}}, l_1, l_2,)$. In the following definition, $\widehat{C}$ is the set obtained from C by attributing a unique index to the different occurrences of a given label. For example $\widehat{\{x, x, y\}} = \{x_1, x_2, y_1\}$.

Definition 1. *A node rewriting rule is given by* $(q_0, C) \to (q_1, \dots q_n, c_1, \dots c_m)$ *where* $C \in \mathbb{N}^L, q_i \in Q, c_j \in L \times (\{1 \dots n\} \cup \widehat{C} \cup L)^2$.

An agent can fire this rule, if its state is q_0 and its context contains C. Moreover, its context should not contain labels l, if l-neighbors are never bound by any triplet, neither individually, not collectively. This is usefull to implement synchronisation mechanisms. The agent applies the rule by indexing its neighbors, deleting its connections, creating n new agents with state $q_1, \dots, q_n$ and creating connections according to each connecting triplet $c_1, \dots, c_m$. The goal of this definition is just to give a precise formalization, at one point in this paper. This concept of node-rewriting rule can be applied to different network structure. In this paper we consider undirected networks and then directed networks. For designing examples of rules, we will always use an intuitive graphical convention, that avoid the use of indexes. We will consider two kinds of collective binding :

one in which at least one neighbor must be present, and the other one where there can be zero neighbors, represented using respectively the symbol '+' and '*', a common notation of formal language theory. The '+' form is a syntactic sugar, easily encoded using the '*' and an individual binding.

2.2 Undirected Node rewriting Rule

As we search for the simplest model of self development, it makes sense to consider first the simplest network structure : undirected labeled graph, we will adjust the definition to consider directed network in subsection 3.1.

Definition 2. *An undirected node rewriting rule is a node rewriting rule acting on undirected networks*

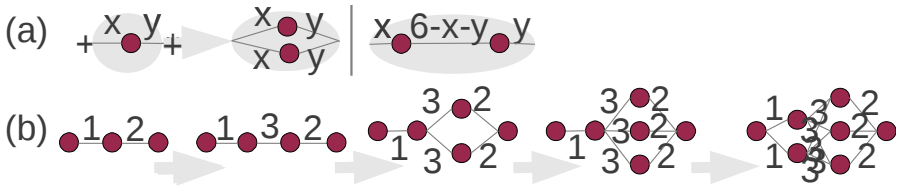


Figure 1. node rewriting rules developing all serie-parallel graphs : (a) Rules for parallel and serial creation. The symbol '+' codes for collective binding, with at least one link present. (b) Example of execution.

For such rules, in a connecting triplet $(l_{\text{new}}, v_1, v_2,)$, the order between v_1 and v_2 does not matter. Figure 1 describes an undirected rule which develops any so-called *s*erie-parallel graph. The rules are described with the following graphical convention : The left member locates bound neighbors by placing them around a light-gray disc showing the label of bound connections, the right member reproduces the same disk, and assumes the bound neighbors conserve the same location on that disc. The rule in fig. 1 (a) uses three integer labels $x, y \in \{1, 2, 3\}$ which are used to distinguish two sets of labels between left and right. In the left member, x and y can be any of these three numbers, with $x \neq y$. An agent rewrites into two agents, in two possible ways [2].

1. In the "parallel" rewriting, all the links are duplicated, one copy for each new agent.
2. In the "serial" rewriting, one agent get x-links, and the other y-links. A link with label $6 - x - y$ is added to connect them.

2. If an agent match two rules or more, one can either define a priority on rules or choose non deterministically.

The label $6-x-y$ is the third possible label $z \in \{1,2,3\} \setminus \{x,y\}$. It ensures that any generated nodes will always have exactly two of the three possible labels within their neighbors, except the extremity nodes which have only one, and cannot rewrite. All the agents have the same state, which can be ignored. The SDN is *ever-growing* : development never stops, network size always augment. The figure shows a development starting from an initial network of three nodes. The central node does one serial division , one offspring does two parallel divisions, and then the other does one last parallel division during which the three links labeled 3 are collectively bound and get duplicated.

2.3 Providing input-output with grounded node-GRS.

We call *node-GRS* a Graph Rewriting System, including a set of node-rewriting rules plus an initial network. The initial agents are called the "*ancestors*", all the other generated agents being the *descendants*. A directed (resp. undirected) node-GRS is a node GRS whose rules are directed (resp. undirected). In order to define a computation, we need to "ground" the node-GRS with ports carrying the inputs and the outputs. We define *hosts* as designated ancestors which remain present during the whole execution. Connections to the hosts are called *port* and also persist, thus the number of hosts and ports is a constant of a given node-GRS. The port's labels is used as a memory shared by the host, and the developing network. Reading and writing this label amounts to input and output values as shown in fig. 2 (c_1,c_2). The host program is specified externally. In the following example, we assume that the left (resp. right) host only writes (resp. reads).

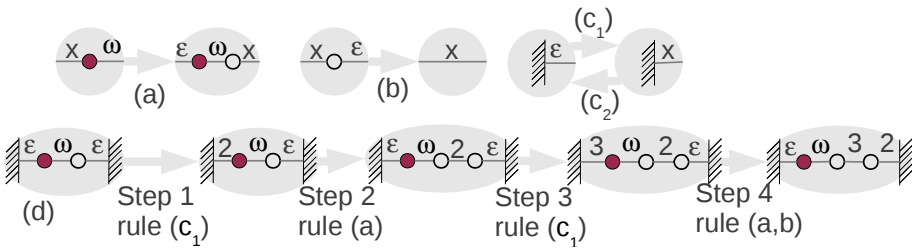


Figure 2. A grounded node-GRS implementing a buffer ; Hosts are drawn as an *electrical ground*. $x \in \mathbb{N}$, ϵ and ω are special labels. (a) Root agent (disc) (b) data agent (circle) (c) Host read, and write, which rule is applied is under external control. (d) Example of execution.

Fig. 2 (a)(b) represents a grounded node-GRS implementing a buffer. The buffer uses only individual binding. Buffer agents have two states : root and data. Fig. 2 (d) shows an execution. The agents are organized in a line starting with a writing host, followed by one root, several data-agents, and a reading host. The buffer initial configuration needs to contain one data-agent.

An agent is called *ready* if at least one of the rules is matched. The root is ready when it has an integer on one edge and the markup ω on the other edge, which distinguishes right from left. A data-agent is ready when it has an integer on one edge, and the empty label ϵ on the other. The root-agent updates by inserting a data-agent which stores an integer data item on its left connection. Data-agents update by suppressing themselves and make the next data item available for reading. The buffer has no capacity limit, since the creation of new data-agents augments the available memory on the fly. At step 2 and 4 the number 2 and 3 are stored by data-agent into to the buffer, the number 2 is retrieved at step 4, and can be read.

The buffer example illustrates that a developed network has an inner state, and can be reused and modified indefinitely, depending on the hosts' program : The hosts can push and pop values indefinitely and generate infinite derivations. Alternatively, if all the hosts remain idle at some point, the execution may reach an idle configuration, where no further rewriting is possible. Such a system can be used to compute a function : the hosts will first input values through the ports, and then retrieve values computed from the developed circuit. The buffer uses a single input and output port. In general, one should provide many ports for parallel inputs and outputs, otherwise the parallelism inherent in the developed network cannot be exploited.

2.4 Simple Definition of Elementary SDNs

In a decentralized distributed execution, at a given time t, any agent which is ready (i.e. which context matches a given rule's left member) may decide to rewrite. Thus, a network configuration c_1 develops into c_2, by rewriting a subset A of the ready agents. This parallel rewriting step is noted $c_1 \xrightarrow{A} c_2$.

A node-GRS for which two neighbor agents can never be simultaneously ready is called *neighbor-exclusive*. Such a system can be executed in a decentralized distributed way, because any agent that rewrites is guaranteed that its neighbors will not, and can safely be used as stable anchors for receiving new connections.

Definition 3. *An elementary Self Developing Network (SDN) is a neighbor-exclusive undirected node-GRS.*

The serie-parallel GRS is not neighbor exclusive, the two agents doing the parallel division are adjacent, and simultaneously ready. Should the two agents decide to divide simultaneously, we would not know what to do. So decentralized executions are not defined.

The buffer GRS is neighbor exclusive, : only the left writing host and either the right reading host or an already read data agent can be simultaneously ready, in which case they are separated by the data agent having a label ω on the left. In other words, reading and a writing the buffer can be done simultaneously, while still enforcing neighbor-exclusion. This proves that the buffer is an elementary SDN. Note that a node-GRS can always be made neighbor exclusive

using randomness, by adding a rule implementing a random local tournament between simultaneously ready neighbors, and blocking one of the two (or both in case of equality). Proposition 1 illustrates the simplicity of elementary SDNs :

Proposition 1. *Deterministic elementary SDNs are strongly confluent.*

Proof : Consider two distinct parallel rewriting step, and let E (resp. F) be the set of agents involved. Because of determinism, the agent in $E \cap F$ apply the same rule. Because of neighbor-exclusion, two agents which update simultaneously, are not neighbor, and do not influence each other. Rewriting agents in $F \setminus E$ (resp. $E \setminus F$) will therefore lead to the same configuration, obtained by rewriting agent in $E \cup F$.

3 Higher Level Self Developing Network.

In our quest for reaching the simplest possible definition of SDNs, we gave the requirement of mutual exclusion. While this is probably a fundamental key for the simplest definition, it is also a bit awkward, since it is the task of the person who designs the GRS to ensure mutual exclusion. There exists other ways of specifying development, that does allow simultaneous rewriting of adjacent nodes.

3.1 Directed node-rewriting rules

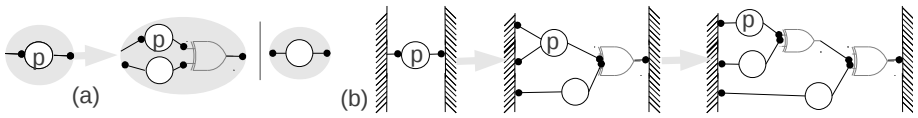


Figure 3. A directed rule developing a p-agent into a parity network. (a) Two rewriting rules. The empty disc represent idle agents. (b) Execution with two recursive rewriting. Each of the two host is a single node.

Using oriented connections allows to partition the network naturally : one simply decides that each connection belongs to the node which is at the *source*, while the target agent is considered to be *pointed by the connection*. Outgoing connection are owned, incoming are not. Having partitioned the network into disjoint left members, one can now perform simultaneous rewriting of adjacent nodes.

We need to modify the semantic of rule application as follows : an agent is not deleted upon rewriting. We must specify connections for it, using connecting triplets, just like other created nodes. The preserved agent can then be used as a stable gluing point for a given input neighbor, updating simultaneously. Bounded connections are deleted, and new connections specified by the right member of

the rule are added. Connections that are not bound, implicitly remain on the preserved agent.

Definition 4. *A directed node-rewriting rule is a node-rewriting rule acting on directed networks. Link orientation defines ownership, agents are preserved, and serve as gluing points.*

In comparison with definition 2, the link's label need to be coupled with a boolean encoding whether it is an incoming or outgoing link. Moreover, in a connecting triplet $(l_{\text{new}}, v_1, v_2)$, the order between v_1 and v_2 now matters. It indicates the orientation of the created link. Furthermore, the semantic of ownership imposes some constraints on the rule : Since an agent can modify only the owned connections, only those can be used in connecting triplet. Not owned link can be bound in the left member, only for testing their presence or absence. Not-owned link remain on the preserved agent. This is used in our graphical notation, to identify the preserved agent in the right member without having to introduce another markup : the preserved agent is the one who gets all not-owned links, for example in fig. 3 it is the XOR.

If an agent owns all its connections, it does not need to be preserved and can be deleted. Such a rule is called *owner-all*. If all rules are owner-all, the system itself is called *owner-all* and is neighbor-exclusive. A rule which deletes its active agent is implicitly owner-all. A rule can also be owner-all for synchronization.

3.2 Examples of Directed node-GRS

Fig. 3 (a) represents a directed rule acting on a node labeled p. It develops a network computing the parity function, i.e. it inputs booleans, and returns true if the number of true input is even. This development uses only individual binding. Ownership is represented using a tiny black disk at the owner extremity. The owned links correspond to the inputs of the parity function. Rewriting a node labeled p is either recursive, or gives an idle node, which is used just for duplicating signals and can be removed in a second step.

Since a node labeled p has two possible rewritings, the parity rule is not deterministic, it can generate an infinite family of parity networks. Fig. 3 (b) shows an execution with two recursive steps, starting from an initial configuration with one writing, and one reading host represented as electrical ground. The execution generates a new port for each recursive rewriging. Two such recursive rewriting generates a network which can compute the parity of three inputs using two chained XORs. If we want to generate a network for exactly n inputs, we would need to include a loop counter in the agent labeled p, initialize to n, decrement at each recursion and do the non-recursive rewrite when it reaches 0.

Consider now the generic problem of simulating boolean circuits. Fig. 4 (a) shows how a XOR gate rewrites like a synchronous data-flow gate : it consumes two tokens on its incoming links, and generates one token on the outgoing link. The link labels encodes a single token encoded by a label $0, 1$ or empty. The synchronisation is implemented by an owner all rule : A gate needs to own all its

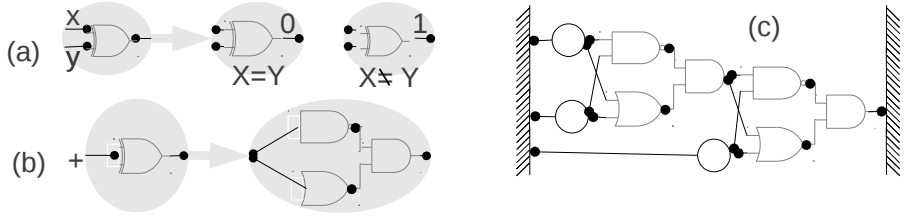


Figure 4. (a) Rule for computing a XOR (b) Rule for developing a XOR (c) Simultaneous development of the two adjacent XORs of fig. 3 (b).

links, before it can fire. It then gives back ownership to the neighbors. Fig. 4 (b), shows how a XOR gate can be rewritten using only OR, AND, and NAND gates. Here collective binding is used : the unique incoming links of the rule will bind two connections. This reflects the symmetry between the two inputs which are both sent to the NAND gate and to the OR gate.

The parity-GRS shows how a data-flow graph can be developed, using rewriting rules which either create new nodes for development or modify labels and orientation for communication and computation. This "dynamic data-flow graph" still has two problems :

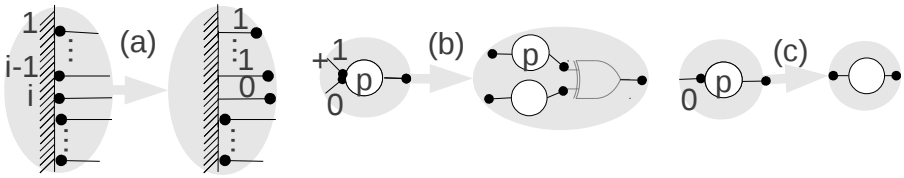


Figure 5. Grounding the parity self-develoment : (a) Bits communicated by hosts 1 $\ldots i$, at step i. (b) Parity Rule preserving ports (c) non recursive case.

1. An agent cannot statically distinguish between boolean inputs and outputs [3]. We do have a link orientation, but it is used for synchronization, and is constantly flipping back and forth. Thus it cannot serve a second purpose.
2. The parity-GRS is not grounded, since ports are created non-deterministically.

The first problem is solved by representing a *p*rogrammed orientation ; using two "opposite" label : for example L and R for right and left. The label is systematically negated when ownership is flipped ; The second problem is solved by having all the ports exists in the ancestor, and rewrite the p-agent as many times as needed to take them into account. For this purpose, the ports must be distinguishable, which is done sending on each ports a distinguishing bit shown

3. The figure makes falsely believe that a gate can do that, because of the gate pictorial representation which is not a symmetric circle.

fig. 5 (a). Using the rule fig. 5 (b,c), after the first iteration , the first port is distinguished, then the second, and so on...

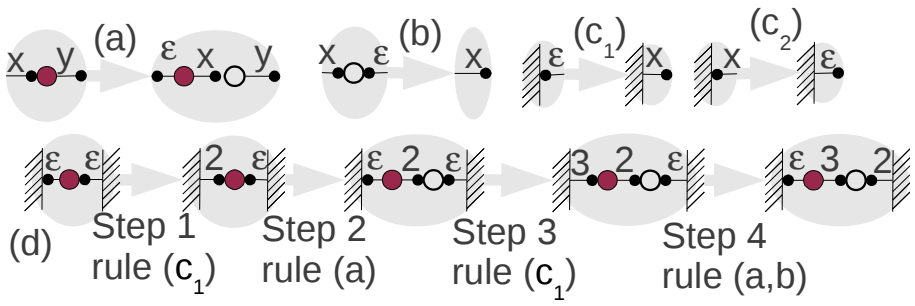


Figure 6. Buffer implemented as a directed node-GRS. (a) Root agent's rule(b) Data agent's rule (c) Host's rule (d) Execution.

The link orientation leads to a more concise higher level representation : In fig. 6 the buffer now needs a single ancestor and no ω markup.

3.3 Simulation of directed GRS by elementary SDN

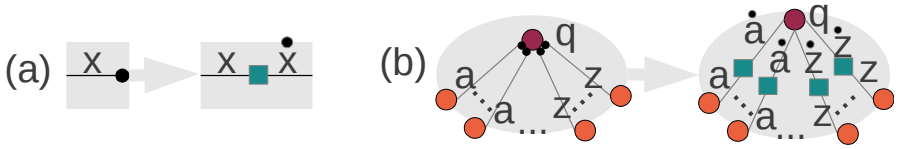


Figure 7. The bipartite transformation (a) Insertion of an edge-agent on each edge. (b) Representation as a directed node-GRS.

The main result of this paper states that directed node-GRS can be considered as higher-level SDNs, where higher level than means "programmed on top with no performance loss" [4]. The converse is shown in [2], therefore the two formalisms are really equivalent, and the definition of self-development does not depend on the particular choice we make, to enable distributed rewriting.

Theorem 1. *Directed node-GRS are higher-level SDNs.*

4. More precisely, the encoding does not augment the amount of information needed to build the network, up to a constant factor, and each rewriting step is simulated using a constant number of parallel steps (3 steps).

Proof sketch : Let S be a directed node-GRS. We "compile" it as an elementary SDN $\phi(S)$. We illustrate the compilation for $S =$ the directed buffer of fig 6. We first compile rules using only individual binding, For each label l of S, $\phi(S)$ needs four labels noted $l, \dot{l}, \underline{l}, \underline{\dot{l}}$ called *scalar labels*, plus three fixed labels denoted by Greek letters α, β, δ. We encode a directed network as an undirected one, by inserting an agent called *edge-agent* on each directed link labeled l, as shown in figure 7 (a) An edge-agent has two connections : one to the owner, and one to the output agent. The original label l is copied on both connections, but with a dot above it ($\dot{l}$) on the connection towards the owner, in order to encode the orientation. The original agent itself remains untouched and is called *node-agent*. Note that this transformation amounts to do a single rewriting step of a directed node-GRS, as shown in fig. 7 (b) [5].

One step of parallel rewriting in S $c_1 \xrightarrow{A} c_2$ is simulated in three steps of parallel rewriting in $\phi(S)$, illustrated in fig. 8.

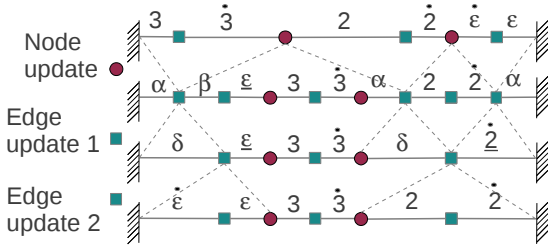


Figure 8. Execution of the directed buffer's simulation, for step 4 of figure 6 (d)

1. Node-agents add one edge agent on each created connections.
2. Edge-agents simplify and restore one edge-agent per connections
3. Edge-agents restore the labeling corresponding to the bipartite encoding.

Step 1 in $\phi(S)$: A node-agent is called a *master*, because its rules are compiled from the simulated system S, as an example, fig. 9 (a) (resp. (b, c_1 and c_2)) shows the compilation of the buffer's root (resp. data agent, host read and write). The links to edge-agents corresponding to not-owned connection, are preserved and labeled by α. For each created connections labeled l, a new edge-agent is inserted with labels $(\beta, \underline{l})$ or $(\beta, \underline{\dot{l}})$ [6] (resp . $(l, \dot{l})$) if l connects a neighbor to a new agent, (resp. two neighbors or two new agents). The label β is inserted towards the neighbors, so that it will always pair with an α link. On the other hand, an α link can be paired with an underlined scalar label.

Step 2 in $\phi(S)$: Edge-agents are called *slaves* because they execute a fixed rule shown in fig. 9 (d,e,f). They receive orders from the master who owns the

5. Encoding is a special case of development, though, since the domain and codomain of the rewriting are distinct, and each node is rewriten once, and exactly once.
6. The label l is dotted, if ownership is kept, which is never the case for the buffer.

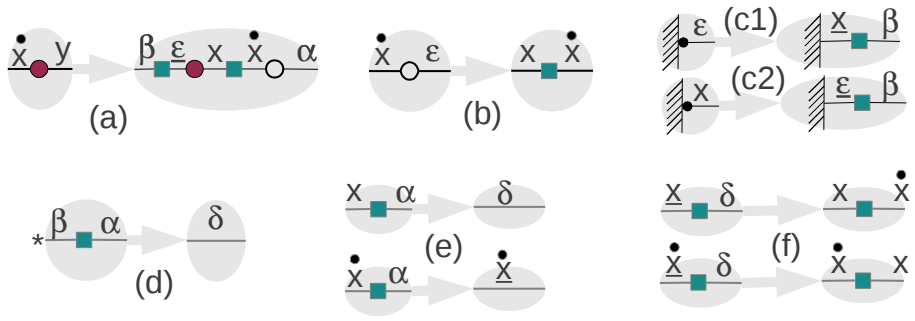


Figure 9. Simulation of the directed buffer. (a,b,c) compiled rule for a node agent including root agent, data agent, host , idle agent. (d,e,f) edge-agent's fixed rule. In rule (d), the symbol '$*$' denotes a collective binding of possibly zero neighbors

edge. which let them replace themselves by a link. An order is encoded as a link labels : The β means "let you simplify by pairing with an α and generate δ" (rule(d)). Here, the δ is a forwarded order meaning "final label is not known yet, wait and see !". The owner of a connection to a neighbor n can create arbitrary many connections to n or none at all. As a result, the edge-agent of rule (d), can get arbitrary many β connection or zero.

The α means "let you simplify with either β or a scalar label (rule (e)), in which case preserve the scalar if it is an owned link, and underline it" or else generate δ. This δ will be next to an underlined scalars.

Step 3 in $\phi(S)$: With rule (f), δ is replaced by the complement of the other edge agent's link label. where complement$(l) = \dot{l}$, and complement$(\dot{l})= l$. The effect is to restore the encoding of the orientation. The role of underlined label $\underline{l}, \underline{\dot{l}}$ is to prevent a node-agent firing before all its edge-agents are done. In other words, it ensures the neighbor-exclusive execution.

The simulation of one complete step needs that all the node-agents update, including those representing idle agents [7]. This is done by imagining that idle agents apply the idle rule shown in fig. 10 (a), and simulate that. The compilation of the idle rule is shown in fig. 10 (b). It needs the simulation of collective binding, which we will do now using an additional fixed label χ. Consider the compilation of a collective binding creating multiple connections carrying label x to all y-neighbors. In step 1, the node-agent insert an edge-agent with link label (x ,χ), where x will be doted or not, depending on orientation. Before beginning step 2, the edge-agent does an extra iterative processing that will create one edge-agent for each x-neighbor, using repetitively rule fig. 10 (c_1). When no link labeled $\underline{\dot{x}}$ is left, then, rule (c_2) ends the processing [8]. The combined effect of collective binding and individual binding in rule (c_1) illustrates well their complementarity,

7. An agent is idle, either because it is not ready, or because it is not in the set A of the considered transition $c_1 \xrightarrow{A} c_2$.

8. Here, it is compulsory that rule c_2 has a lower priority than rule c_1

and the resulting expressiveness for node rewriting rule : it enables an iterative processing, link by link.

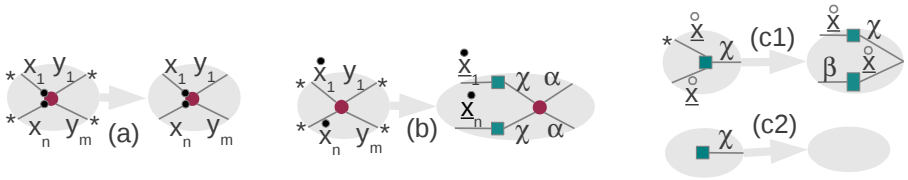


Figure 10. Simulation of Collective binding. (a) the idle rule (b) compilation of the idle rule (c1,c2) edge-agent rule inserting iteratively one edge-agent for each link labeled $\underline{\dot{x}}$. The circle above the labels means that it can be either a dotted, or not dotted label.

4 Yet Higher level Self-Developing Network.

4.1 Other Syntax for programming Higher level SDNs

Encoding a link orientation is just one way of enriching the network data-structure, and then exploit this added information with an appropriate rule syntax. In [1], we explore many others possible ways, and proved that they also are higher level SDNs, by programming them using directed node-GRS.

1. *Transitive SDN* redirect the input connections to distinct created agents, instead of simply preserving them on the same persistent agent.
2. *Programmed orientations* can be defined on top of the orientation defining ownership. It can be used to distinguish left from right in a sequence of agents, or up from down in a tree.
3. *Flags on connection extremities* provide a local memory per connections, and make it easier for an agent to manage its connections without interfering with the neighbors. For example, one can index the connections locally.
4. *Combining directed and undirected link* ; An undirected link cannot be modified except for acquiring ownership by setting an "ownership extremity flag", and thus orienting the link. If both extremities try to acquire ownership, a random choice is made. This break symmetry at system level.

4.2 Classification of existing SDNs.

Well known systems can be also programmeed on top of directed node-GRS, and classified as specific type of SDNs.

1. *SDN with ever-growing Network :* EdNCE graph grammar in [6] describes a sub class of directed systems using only collective binding, forbidding creation of connections between neighbors. Thus if two nodes are not connected,

they will never be in the future. EdNCE Network do not remove node, indeed, when removing nodes, connecting neighbors together is indispensable to maintain the connectedness.

2. *SDN with acyclic network :* L-systems are grammars introduced by Lindemeyer [5] to model the development of algae and plants. The object being rewritten is a string using brackets representing a compact encoding of a tree. It can be programmed as a directed SDN, where the network is acyclic : in other words it is a tree. A simulation of context-free L-systems need edge-agents to synchronize the rewriting of all node-agent.
3. *SDN with a constant number of agents :* Self assembly system focus on maintaining a specific subset of connections for persistent pairwise communication, or for progressive assembly of a structure, between robots or molecules. As a result, a dynamic network is built and maintained. Klavins [4] use node-GRS to move interacting robots so as to cover a given region. Rules can create or delete connections, and trigger agent movement.
4. *SDN with a fixed network :* If the rules neither creates nor deletes agents or connections, the network is preserved. Such a degenerated SDN is called "*static*" and models a fixed network of finite state automata. There are two widely studied families of automata network : Artificial Neural Networks (ANNs) and Cellular Automata (CA). Moreover, static SDN can model real hardware. The same vocabulary, principles and methods can be used for simulating SDN on real parallel hardware, which is the long term goal of our project.

Références

1. F. Gruau. Self developing networks, part 1 : the formal system. Technical Report 1549, LRI, 2012. http ://www.lri.fr/~bibli/Rapports-internes/2012/RR1549.pdf.
2. F. Gruau. Self developing networks, part 2 : Universal machines. Technical Report 1550, LRI, 2012. http ://www.lri.fr/~bibli/Rapports-internes/2012/RR1550.pdf.
3. F. Gruau, C. Eisenbeis, and L. Maignan. The foundation of self-developing blob machines for spatial computing. *physica D :Nonlinear Phenomena*, 237, 2008.
4. E. Klavins. Programmable self-assembly. *Control Systems Magazine*, 24(4) :43–56, August 2007. See the COVER !
5. A. Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *jtb*, 18 :280–299, 1968.
6. Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*, volume 1,2,3. WSP, 1997.
7. Tomita, Murata, Kamimura, and Kurokawa. Self-description for construction and execution in graph rewriting automata. In *European Conference on (Advances in) Artificial Life, LNCS*, volume 8, 2005.

More on Graph Rewriting With Contextual Refinement

Berthold Hoffmann

Fachbereich Mathematik und Informatik, Universität Bremen, Germany

Abstract. In GRGEN, a graph rewrite generator tool, rules have the outstanding feature that variables in their pattern and replacement graphs may be refined with meta-rules based on contextual hyperedge replacement grammars. A refined rule may delete, copy, and transform subgraphs of unbounded size and of variable shape. In this paper, we show that rules with contextual refinement can be transformed to standard graph rewrite rules that perform the refinement incrementally, and are applied according to a strategy called residual rewriting. With this transformation, it is possible to state precisely whether refinements can be determined in finitely many steps or not, and whether refinements are unique for every form of refined pattern or not.

1 Introduction

Everywhere in computer science and beyond, one finds systems with a structure represented by graph-like diagrams, whose behavior is described by incremental transformation. Model-driven software engineering is a prominent example for an area where this way of system description is very popular. Graph rewriting is a natural formalism for specifying such systems in an abstract way, ever since this branch of theoretical computer science emerged in the seventies of the last century [8]. Graph rewriting has a well developed theory [4] that gives a precise meaning to such specifications. It also allows to study fundamental properties, such as termination and confluence. Over the last decades, various tools have been developed that generate (prototype) implementations for graph rewriting specifications. Some of them do also support the analysis of specifications: AGG [9] allows to determine confluence of a set of rules by the analysis of finitely many critical pairs [17], and GROOVE [18] allows to explore the state space of specifications.

This work relates to GRGEN, an efficient *graph rewrite generator* [1]. Edgar Jakumeit has drastically extended the rules of this tool, by introducing recursive refinement for sub-rules and application conditions [15]. A single refined rule can match, delete, replicate, and transform subgraphs of unbounded size and variable shape. These rules have motivated the research presented in this paper. Because, the standard theory [4] does not cover recursive refinement, so that such rules cannot be analyzed for properties like termination and confluence, and tool support concerning these questions cannot be provided.

Our ultimate goal is to lift results concerning confluence to rules with recursive refinement. So we formalize refinement by combining concepts of the existing theory, on two levels: We define a GRGEN rule to be a schema – a plain rule containing variables. On the meta-level, a schema is refined by replacing variables by sub-rules, using meta-rules based on contextual hyperedge replacement [3]. Refined rules then perform the rewriting on the object level. This mechanism is simple enough for formal investigation. For instance, properties of refined rules can be studied by using induction over the meta-rules. Earlier work [14] has already laid the fundaments for modeling refinement. Here we study conditions under which the refinement behaves well. We translate these rules into standard rules that perform the refinement in an incremental fashion, using a specific ("residual") rewriting strategy, and show the correctness of this translation.

The examples shown in this paper arise in the area of model-driven software engineering. *Refactoring* shall improve the structure of object-oriented software without changing its behavior. Graphs are a straight-forward representation for the syntax and semantic relationships of object-oriented programs (and models). Many of the basic refactoring operations proposed by Fowler [10] do require to match, delete, copy, or restructure program fragments of unbounded size and variable shape. Several plain rules are needed to specify such an operation, and they have to be controlled in a rather delicate way in order to perform it correctly. In contrast, we shall see that a single rule schema with appropriate contextual meta-rules suffices to specify it, in a completely declarative way.

The paper is organized as follows. The next section defines graphs, plain rules for graph rewriting, and contextual rules for deriving languages of graphs. In Sect. 3 we define schemata, meta-rules, and the refinement of schemata by applying meta-rules to them, and state under which conditions refinements can be determined in finitely many steps, and the replacements of refined rules are uniquely determined by their patterns. In Sect. 4, we translate schemata and meta-rules to standard graph rewrite rules,and show that the translation is correct. We conclude by indicating future work, in Sect. 5. The appendix recalls some facts about graph rewriting.

2 Graphs, Rewriting, and Contextual Grammars

We define graphs wherein edges may not just connect two nodes – a source to a target – but any number of nodes. Such graphs are known as hypergraphs in the literature [11].

Definition 2.1 (Graph). Let $\Sigma = (\dot{\Sigma}, \bar{\Sigma})$ be a pair of finite *label sets*.

A *graph* $G = (\dot{G}, \bar{G}, att, \ell)$ consists of two disjoint finite sets $\dot{G}$ of *nodes* and $\bar{G}$ of *edges*, a function $att\colon \bar{G} \to \dot{G}^*$ that *attaches* sequences of nodes to edges, and of a pair $\ell = (\dot{\ell}, \bar{\ell})$ of *labeling functions* $\dot{\ell}\colon \dot{G} \to \dot{\Sigma}$ for nodes and $\bar{\ell}\colon \bar{G} \to \bar{\Sigma}$ for edges.[1] We will often refer to the component functions of a graph G by att_G and ℓ_G.

[1] A^* denotes *finite sequences* over a set A; the empty sequence is denoted by ε.

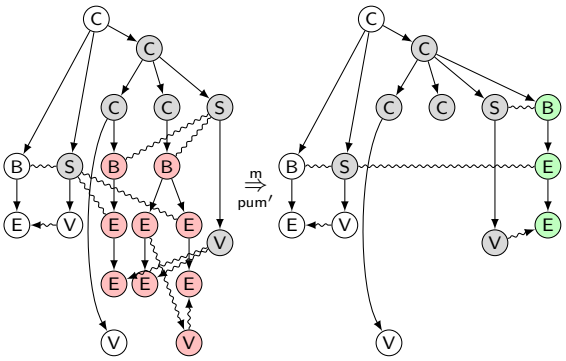


Fig. 1. Two program graphs

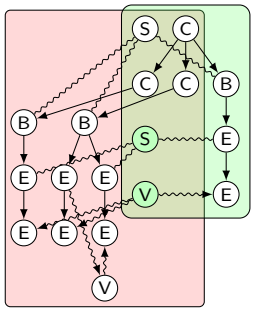


Fig. 2. A refactoring rule

A (*graph*) *morphism* $m\colon G \to H$ is a pair $m = (\dot{m}, \bar{m})$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\bar{m}\colon \bar{G} \to \bar{H}$ that preserve attachments and labels: $att_H \circ \bar{m} = \dot{m}^* \circ att_G$, $\dot{\ell}_H = \dot{\ell}_G \circ \dot{m}$, and $\bar{\ell}_H = \bar{\ell}_G \circ \bar{m}$.[2] The morphism m is *injective*, *surjective*, and *bijective* if its component functions have the respective property. If m is bijective, we call G and H *isomorphic*, and write $G \cong H$. If m maps nodes and edges of G onto themselves, it defines the *inclusion* of G as a subgraph in H, written $G \hookrightarrow H$.

Example 1 (Program Graphs). Figure 1 shows two graphs G and H representing object-oriented programs. Circles represent nodes, and have their labels inscribed. In these particular graphs, edges are always attached to exactly two nodes, and are drawn as straight or wave-like arrows from their source node to their target node. (The filling of nodes, and the colors of edges will be explained in Example 2.)

Program graphs have been proposed in [19] for representing key concepts of object-oriented programs in a language-independent way. In the simplified version that is used here, nodes labeled with C, V, E, S, and B represent program entities: classes, variables, expressions, signatures and bodies of methods, respectively. Straight arrows represent the syntactical composition of programs, whereas wave-like arrows relate the use of entities to their declaration in the context.

We use the standard definition of graph rewriting [4], and insist on injective matching of rules; this is no restriction, see [12]. We choose an alternative representation of rules proposed in [7] so that the rewriting of rules in Sect. 3 can be easier defined, see also in Appendix A.

Definition 2.2 (Graph Rewriting). A *graph rewrite rule* (*rule* for short) $r = (P \hookrightarrow B \hookleftarrow R)$ consists of graph inclusions, of a *pattern* P and a *replacement*

[2] For a function $f\colon A \to B$, its extension $f^*\colon A^* \to B^*$ to sequences A^* is defined by $f^*(a_1 \dots a_n) = f(a_1) \dots f(a_n)$, for all $a_i \in A$, $1 \leqslant i \leqslant n$, $n \geqslant 0$; $f \circ g$ denotes the composition of functions or morphisms f and g.

R in a common *body* B. A rule is *concise* if the inclusions are jointly surjective. By default, we refer to the components of a rule r by P_r, B_r, and R_r.

The rule r *rewrites* a *source graph* G into a *target graph* H if there is an injective morphism $B \to U$ to a *united graph* U so that the squares in the following diagram are pushouts:

$$\begin{array}{ccccc} r\colon P & \hookrightarrow & B & \hookleftarrow & R \\ {\scriptstyle m}\downarrow & & \downarrow & & \downarrow{\scriptstyle \tilde{m}} \\ G & \longrightarrow & U & \longleftarrow & H \end{array}$$

The diagram exists if the morphism $m\colon P \to G$ is injective, and satisfies the following *gluing condition*: Every edge of G that is attached to a node in $m(P \setminus R)$ is in $m(P)$. Then m is a *match* of r in G, and H can be constructed by (i) uniting G disjointly with a fresh copy of the body B, and gluing its pattern subgraph P to its match $m(P)$, giving U, and (ii) removing the nodes and edges $m(P \setminus R)$ from U, yielding H with the *embedding* morphism $\tilde{\mathsf{m}}\colon R \to H$.[3] The construction is unique up to isomorphism, and yields a *rewrite step*, which is denoted as $G \Rightarrow_r^m H$.

Example 2 (A Refactoring Rule). Figure 2 shows a rule pum'. Rounded shaded boxes enclose its pattern and replacement, where the pattern is the box extending farther to the left. Together they designate the body. (Rule pum' is concise.) We use the convention that an edge belongs only to those boxes that contain it entirely; so the "waves" connecting the top-most S-node to nodes in the pattern belong only to the pattern, but not to the replacement of pum'.

The pattern of pum' specifies a class with two subclasses that contain method implementations for the same signature. The replacement specifies that one of these methods shall be moved to the superclass, and the other one shall be deleted. In other words, pum' *pulls up methods*. However, it only applies if the class has exactly two subclasses, and if the method bodies have the particular shape specified in the pattern.

The graphs in Figure 1 constitute a rewrite step $\mathsf{G} \Rightarrow^{\mathsf{m}}_{\mathsf{pum}'} \mathsf{H}$. The shaded nodes in the source graph G distinguish the match m of pum', and the shaded nodes in the target graph H distinguish the embedding $\tilde{m}$ of its replacement. (The red nodes in G are removed, and the green nodes in H are inserted, with their incident edges, respectively.)

The general *Pull-up Method* refactoring of Fowler [10] works for classes with any positive number of subclasses, and for method bodies of varying shape and size. This cannot be specified with a plain rule. The general refactoring will be specified in Example 4 further below.

Graph rewriting can be used for computations on graphs by applying a set of rules to some input graph as long as possible. Let $\mathcal{R}$ be a set of graph rewrite rules. We write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_r^m H$ for some match m of a rule $r \in \mathcal{R}$, and

[3] If r is not consise, the nodes and edges of B that are not in the subgraph $(P \cup R)$ are not relevant for the construction.

denote the transitive-reflexive closure of this relation by $\Rightarrow^*_{\mathcal{R}}$. A graph G is *in normalform* wrt. $\mathcal{R}$ if there is no graph H so that $G \Rightarrow_{\mathcal{R}} H$. A set $\mathcal{R}$ of graph rewrite rules *reduces* a graph G to some graph H, written $G \Rightarrow^!_{\mathcal{R}} H$, if $G \Rightarrow^*_{\mathcal{R}} H$ and H is in normalform. $\mathcal{R}$ (and $\Rightarrow_{\mathcal{R}}$) is *terminating* if it does not admit an infinite rewrite sequence $G_o \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots$, and *confluent* if for all rewrite sequences $H_1 \overset{*}{\underset{\mathcal{R}}{\Leftarrow}} G \overset{*}{\underset{\mathcal{R}}{\Rightarrow}} H_2$, there exists a graph K with $H_1 \overset{*}{\underset{\mathcal{R}}{\Rightarrow}} K \overset{*}{\underset{\mathcal{R}}{\Leftarrow}} H_2$.

So, $\mathcal{R}$ defines a partial nondeterministic function from graphs to sets of their normalforms. This function is deterministic if $\mathcal{R}$ is confluent, and total if $\mathcal{R}$ is terminating.

Graph rewrite rules can also be used to derive sets of graphs, which are called *languages*, as for string grammars. A restricted form of rules has turned out to be useful for that purpose: they replace a variable by gluing a graph to its attached nodes and to some nodes in the context [3].

We assume that the labels contain a set $X \subseteq \bar{\Sigma}$ of *variable names* that are used to label placeholders for subgraphs. $X(G) = \{e \in \bar{G} \mid \ell_G(e) \in X\}$ is the set of *variables* of a graph G, and $\underline{G}$ is its *kernel*, i.e., G without $X(G)$. For a variable $e \in X(G)$, the *variable subgraph* $G(e)$ consists of e and its attached nodes.

Graphs with variables are required to be *typed* in the following way: Variable names $x \in X$ come with a *variable graph* G_x, which consists of a single edge labeled with x, to which all nodes are attached exactly once; in every graph G, the variable subgraph $G(e)$ must be isomorphic to $\mathsf{G}_{\ell_G(e)}$, for every variable $e \in X(G)$.

Definition 2.3 (Contextual Grammar). A rule $r\colon (P \hookrightarrow B \hookleftarrow R)$ is *contextual* if the only edge e in its pattern P is a variable, and if R equals B without e.

With some *start graph* Z, a finite set $\mathcal{R}$ of contextual rules forms a *contextual grammar* $\Gamma = (\Sigma, \mathcal{R}, Z)$ over the labels Σ, which derives the *language*

$$\mathcal{L}(\Gamma) = \{G \mid Z \Rightarrow^*_{\mathcal{R}} G, X(G) = \emptyset\}\,.$$

The pattern P of a contextual rule r is the disjoint union of a variable graph G_x with a discrete *context graph*, which is denoted as C_r. We call r *context-free* if C_r is empty. (Grammars with only such rules have been studied in the theory of hyperedge replacement [11].)

Example 3 (Contextual Rules for Method Bodies). Figure 3 shows contextual rules. Variables are represented as boxes with their variable names inscribed; they are connected with their attached nodes by lines and arrows, ordered from left to right. When drawing contextual rules like those in Fig. 3, we omit the box around the pattern. The variable outside the replacement box is the unique edge in the pattern, and green filling (appearing grey in B/W print) designates the contextual nodes within the box representing the replacement graph.

The set $\mathsf{M} = \{\mathsf{body}_n, \mathsf{use}, \mathsf{call}_n, \mathsf{ass}\}$ of contextual rules derives the data flow of method bodies in program graphs. A method body consists of expressions, which in turn either *use* the value of a variable, or *call* a method signature with expressions that are their actual parameters, or *assign* the value of an expression

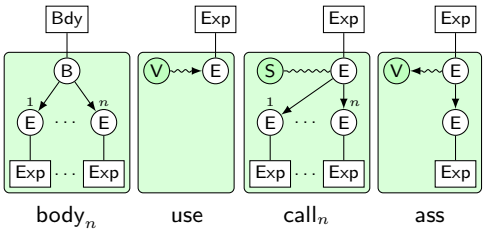


Fig. 3. Contextual rules M for method bodies

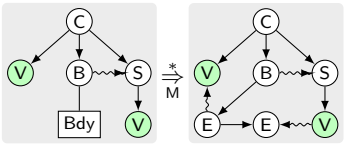


Fig. 4. Deriving a method body

to it. Actually, body_n and call_n abbreviate sets of (context-free) replicative rules that generate graphs with $n \geqslant 0$ copies of variables named Exp. The rules body_n are context-free; in the rules for Exp, variable and signature nodes are contextual.

Figure 4 shows a derivation of a method body with M. Note that the body can only be derived if the start graph contains appropriate nodes representing variables and signatures. The missing rules of the complete grammar for program graphs are given in [3]; they do derive appropriate contextual nodes. (The language of program graphs cannot be derived with context-free rules [3].)

As for context-free string grammars, ambiguity is an important issue if the graphs derived by a contextual grammar shall be transformed. This property will be used in Lemma 3.6 further below.

Definition 2.4 (Ambiguity). Let $\Gamma = (\Sigma, \mathcal{R}, Z)$ be a contextual grammar.

Consider two rewrite steps $G \Rightarrow^{m}_{r} H \Rightarrow^{m'}_{r'} K$ where $\tilde{m}\colon R \to H$ is the embedding of r in H. The steps may be *swapped* if $m'(P') \hookrightarrow \tilde{m}(P \cap R)$, yielding steps $G \Rightarrow^{m'}_{r'} H' \Rightarrow^{m}_{r} K$. Two rewrite sequences are *equivalent* if they can be made equal by repeatedly swapping their steps.

Then Γ is *unambiguous* if all rewrite sequences $Z \Rightarrow^{*}_{\mathcal{R}} G$ for a graph G are equivalent to each other; if some graph G has at least two rewrite sequences that are not equivalent, Γ is *ambiguous*.

3 Schema Refinement with Contextual Meta-Rules

The graph rewriting tool GRGEN [1] supports object-oriented graph models with subtyping and attributes, named and parameterized rewrite rules with negative application conditions, and translates them to code that is highly optimized. Edgar Jakumeit [16,15] has extended the rules drastically, by introducing recursive refinement:

- Rules may contain variables; we call them schemata.
- The substitution of variables can be defined by meta-rules that are based on contextual rules as in Def. 2.3.
- A variable may be attached to nodes in the pattern and the replacement of a rule. Then its substitution refines pattern and the replacement of a schema

at the same time. This does not only allow to match, delete, or replicate subgraphs of unbounded size and arbitrary shape: the rules that derive recursive sub-rules *transform* such subgraphs in a single rule application.

We started to study this way of *rewriting with contextual refinement* in [14]; this work shall be continued in this paper.

We lift morphisms from graphs to rules, for defining the rewriting of rules by meta-rules. For (graph rewrite) rules r and s, a graph morphism $m\colon B_r \to B_s$ on their bodies is a *rule morphism*, and denoted as $m\colon r \to s$, if $m(P_r) \hookrightarrow P_s$ and $m(R_r) \hookrightarrow R_s$. Graph rewrite rules and rule morphisms form a category. This category has pushouts, pullbacks, and unique pushout complements along injective rule morphisms, just as graphs. As with graphs, we write rule inclusions as "$\hookrightarrow$", and let $\underline{r}$ be the *kernel* of a rule r wherein all variables are removed.

Definition 3.1 (Rule Rewriting). A pair $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ of rule inclusions is a *rule rewrite rule*, or *meta-rule* for short. With δ_B we denote its *body rule*, which is a graph rewrite rule consisting of the bodies of p, b, and r.

Consider a rule s, a meta-rule δ as above, and a rule morphism $m\colon p \to s$. The meta-rule δ *rewrites* the source rule s at m to the target rule t, written $s \Downarrow_\delta^m t$, if there is a pair of pushouts

$$\begin{array}{ccccc} \delta\colon p & \hookrightarrow & b & \hookleftarrow & r \\ {\scriptstyle m}\downarrow & & \downarrow & & \downarrow{\scriptstyle \tilde{m}} \\ s & \hookrightarrow & u & \hookleftarrow & t \end{array}$$

The pushouts above exist if the underlying body morphism of m satisfies the graph gluing condition wrt. the body rule δ_B and the body graph B_s.

We use meta-rules with contextual body rules, and apply them to rules that contain variables in their body (but neither in their pattern, nor in their replacement graphs).

Definition 3.2 (Schema Refinement). A *schema* $s\colon (P \hookrightarrow B \hookleftarrow R)$ is a graph rewrite rule with $P \cup R = \underline{B}$.

Every schema s is required to be *typed* in the following sense: every variable name $x \in X$ comes with a *schema* s_x with body G_x so that for every variable $e \in X(B)$, the variable subgraph $B(e)$ is the body of a subschema that is isomorphic to s_x.

A meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ is *contextual* if p, b, and r are schemata, and if its body rule $\delta_B\colon (B_p \hookrightarrow B_b \hookleftarrow B_r)$ is a contextual rule so that the contextual nodes C_{δ_B} are in $P_p \cap R_p$.

A *less contextual variation* δ' of a meta-rule δ equals δ up to the fact that in its body rule δ'_B, some nodes of C_{δ_B} are removed from P_{δ_B}, but kept in R_{δ_B}. Let Δ be a finite set of meta-rules that is closed under less contextual variations.[4] Then $\Downarrow_\Delta$ denotes refinement steps with one of its meta-rules, and $\Downarrow_\Delta^*$ denotes

[4] We explain in Example 7 why less contextual variations are needed.

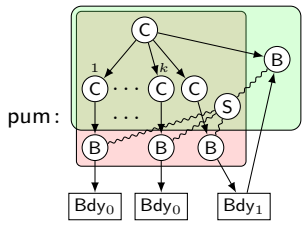


Fig. 5. Pull-up Method: schema

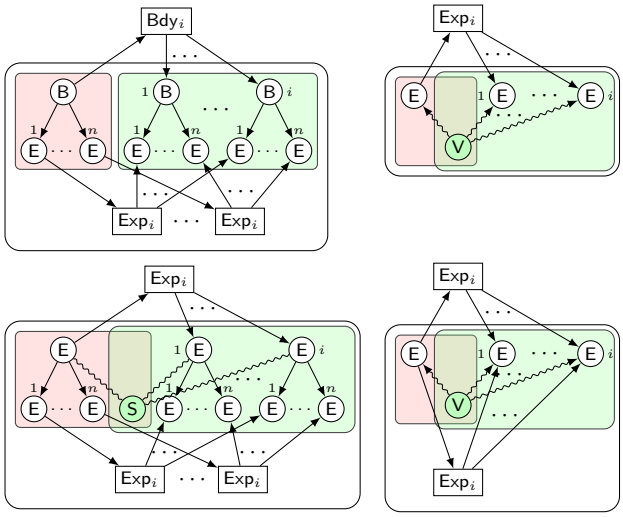


Fig. 6. Replicating meta-rules $\Delta_{\mathsf{M}} = \{\mathsf{body}_{n,i}, \mathsf{use}_i, \mathsf{call}_{n,i}, \mathsf{ass}_i\}$ for the rules M in Fig. 3

repeated refinement, its reflexive-transitive closure. $\Delta(s)$ denotes the *refinements* of a schema $s\colon (P \hookrightarrow B \hookleftarrow R)$, containing its refinements without variables:

$$\Delta(s) = \{r \mid s \Downarrow^*_\Delta r, X(B_r) = \emptyset\}$$

We write $G \Rightarrow_{\Delta(s)} H$ if $G \Rightarrow_r H$ for some $r \in \Delta(s)$, and say that the refinements $\Delta(s)$ *rewrite* G to H.

Example 4 (Pull-Up Method). Fowler's refactoring operation *Pull-up Method* [10] applies to a class c where all direct subclasses contain bodies for the same method signature that are semantically equivalent.[5] It pulls one of these bodies up to c, and removes all others.

The meta-rules $\Delta_{\mathsf{M}} = \{\mathsf{body}_{n,i}, \mathsf{use}_i, \mathsf{call}_{n,i}, \mathsf{ass}_i\}$ for generic meta-variables Bdy_i and Exp_i in Fig. 6 replicate method bodies as defined by the contextual

[5] This condition cannot be decided mechanically; it has to be confirmed by the user when s/he applies the operation, by *a priori* verification or *a-posteriori* testing.

rules M in Fig. 3: they remove one method body from a pattern and insert $i \geqslant 0$ copies of this body in the replacement of a schema. In the less contextual variations $\overline{\mathsf{use}}_i$, $\overline{\mathsf{call}}_{n,i}$, and $\overline{\mathsf{ass}}_i$ of these meta-rules (which are not shown here) the S- and V-nodes are no longer contextual. The schema pum in Fig. 5 uses several meta-variables Bdy_0 that just remove one method body from a subclass in the pattern, and one variable Bdy_1 that moves a method body from one subclass in the pattern to the superclass in the replacement.[6]

In schemata and meta-rules, the lines between a variable e and a node v attached to e get arrow tips (i) at e if v occurs in the pattern, and (ii) at v if v occurs in the replacement, and (iii) both at e and v if v occurs in both, pattern and replacement. (The last case does not occur in our example.)

The rule pum' in Fig. 2 is a refinement of pum with Δ_{M}, i.e., $\mathsf{pum}' \in \Delta_{\mathsf{M}}(\mathsf{pum})$. The upper row in Fig. 10 on page 12 shows a step in the refinement sequence $\mathsf{pum} \Downarrow^*_{\Delta_{\mathsf{M}}} \mathsf{pum}'$; it applies the context-free variation $\overline{\mathsf{ass}}_i$ of the meta-rule ass_i in Fig. 6.

A single rewriting step with some refinement of pum copies one method body of arbitrary shape and size, and deletes an arbitrarily number of other bodies, which are also of variable shape and size. This goes beyond the expressiveness of plain rewrite rules, which may only match, delete, and replicate subgraphs of constant size. Note that the application of a refinement $r \in \Delta(s)$, although it is the result of a compound meta-derivation, is a single rewriting step $G \Rightarrow_r H$ on the source graph G, similar to a transaction in a data base. Note also that the refinement process is completely rule-based.

Operationally, we cannot construct all refinements of a schema s first, and apply one of them later, because the set $\Delta(s)$ is infinite in general. Rather, we interleave matching and refinement, in the next section.

The following assumption excludes useless sets of meta-rules.

Assumption. The set $\Delta(s)$ of refinements of a schema s shall be non-empty.

This property is decidable for contextual grammars [3, Corollary 2].

We need a mild condition to show that residual rewriting terminates.

Definition 3.3 (Pattern-Refining Meta-Rules). A meta-rule $(p \hookrightarrow b \hookleftarrow r)$ *refines its pattern* if $X(R_r) = \emptyset$ or if $P_r \ncong P_p$. A set Δ of meta-rules that refine their patterns is called *pattern-refining*.

Theorem 3.4. *For a schema s and a set Δ of pattern-refining meta-rules, it is decidable whether some refinement $r \in \Delta(s)$ applies to a graph G, or not.*

Proof. By Algorithm 1 in [14], the claim holds under the condition that meta-rules "do not loop on patterns". It is easy to see that pattern-refining meta-rules are of this kind. □

[6] The ellipses "…" allows any number $k \geqslant 0$ subclasses to be matched for removing a body. In Fowler's operation, no further subclasses should exist. However, this could only be scpecified with a negative application condition for the schema, in future work.

We now turn to the question whether the patterns of refinements uniquely define the replacement they perform.

Definition 3.5 (Right-Unique Meta-Rules). A set $\mathcal{R}$ of graph rewrite rules is *right-unique* if different meta-rules $r_1, r_2 \in \Delta$ have different patterns, i.e., $P_1 \cong P_2$ implies that $r_1 \cong r_2$.

We have to define an auxiliary notion first. The *pattern rule* δ_P of a meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ is a contextual rule obtained from the body rule δ_B by removing all nodes and edges in $B_b \setminus R_b$, and by detaching all variables in δ_B from the removed nodes. Let Δ_P denote the set of (contextual) *pattern rules* of a set Δ of meta-rules.[7]

Lemma 3.6 (Right-Uniqueness of Refinements). *A set $\Delta(s)$ of refinements is right-unique if the pattern grammar (Σ, Δ_P, P_s) is unambiguous.*

Proof Sketch. Consider rules $r_1, r_2 \in \Delta(s)$ with $P_1 \cong P_2$. Then $P_s \Rightarrow_{\Delta_P} P_1$ and $P_s \Rightarrow_{\Delta_P} P_2$. The rewrite sequences can be made equal since Δ_P is unambiguous. This rewriting sequence has a unique extension to a meta-rewrite sequence so that $r_1 \cong r_2$. □

Example 5 (Pattern-Refining, Right-Unique Meta-Rules). The meta-rules Δ_{M} in Fig. 6 are pattern-refining. The contextual rules M for method bodies in Fig. 3 are unambiguous. They correspond to the pattern rules of the meta-rules Δ_{M} in Fig. 6, so that these are right-unique. (The meta-rules for the *encapsulate Field* refactoring schema in [14, Ex. 5] are pattern-refining and right-unique as well.)

4 Modeling Refinement by Residual Rewriting

As a first step to analyzing further properties of schemata and meta-rules, we translate them into standard graph rewrite rules:

- We turn every schema into an ordinary rule that *delays refinement*, by adding the meta-variables to its replacement, with all their attached nodes.
- We turn every contextual meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$ into an graph rewrite rule that *refines* the delaying rule *incrementally*, by adding the pattern of r to that of p, and the variable graphs of B_r to the replacement R_r.

Definition 4.1 (Incremental Refinement Rules). Let $s\colon (P \hookrightarrow B \hookleftarrow R)$ be a schema for meta-rules Δ.

The *delaying rule* $\tilde{s}\colon (P \hookrightarrow B \hookleftarrow R_{\tilde{s}})$ of s has the same pattern P and body B as s, and its replacement $R_{\tilde{s}} = R \cup \{B(e) \mid e \in X(B)\}$ is obtained by uniting R with the graphs of all variables in B.

For a meta-rule $\delta = (p \hookrightarrow b \hookleftarrow r)$, the *incremental rule* $\tilde{\delta}\colon (\tilde{P} \hookrightarrow \tilde{B} \hookleftarrow \tilde{R})$ has the pattern $\tilde{P} = B_p \cup P_r$, a replacement $\tilde{R} = R_r \cup \{B_r(e) \mid e \in X(B_r)\}$, and the body $\tilde{B} = B_b$. $\tilde{\Delta}$ denotes the incremental rules of Δ.

[7] The graphs and in Δ_P are also typed, but in the type graph G_x of a variable name x, all nodes that do not belong to the pattern of the schema s_x are removed.

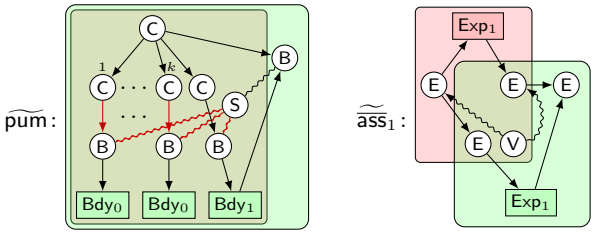


Fig. 7. Delaying and incremental rules for *Pull-up Method* in Fig. 5 and Fig. 6

Example 6 (Incremental Refinement). Figure 7 shows how the schema pum for the *Pull-up Method* refactoring in Fig. 5 is translated into a delaying rule $\widetilde{\mathsf{pum}}$, and how the context-free variation $\overline{\mathsf{ass}}_1$ of the meta-rule ass_1 in Fig. 6 is translated into an incremental rule $\widetilde{\overline{\mathsf{ass}}}_1$. (In the delayed rule $\widetilde{\mathsf{pum}}$, red arrow and waves (appearing grey in B/W print) indicate edges that do not belong to the replacement.)

If a schema s is refined with a metarule δ to a schema t, the composition $\tilde{s} \circ_d \tilde{\delta}$ of its delayed and incremental rules (defined in Def. A.1) equals the delayed rule $\tilde{t}$ (for a particular dependency d).

Lemma 4.2. *Consider a schema $s = (P \hookrightarrow B \hookleftarrow R)$ and a meta-rule $\delta\colon (p \hookrightarrow b \hookleftarrow r)$.*

Then $s \Downarrow_{\delta,m} t$ for some schema t iff there is a composition $r^d = \tilde{s} \circ_d \tilde{\delta}$ for a dependency $d\colon (R \xleftarrow[m]{} B_p \to (B_p \cup R_p)$ so that $r^d = \tilde{t}$.

Proof Sketch. Let s, δ be as above, $t\colon (P' \hookrightarrow B' \hookleftarrow R')$, $\tilde{s}\colon (P \hookrightarrow B \hookleftarrow R_{\tilde{s}})$ with $R_{\tilde{s}} = R \cup \{B(e) \mid e \in X(B)\}$, and $\tilde{\delta}\colon (\tilde{P} \hookrightarrow \tilde{B} \hookleftarrow \tilde{R})$ with $\tilde{P} = B_p \cup P_r$, $\tilde{R} = R_r \cup \{B_r(e) \mid e \in X(B_r)\}$, and $\tilde{B} = B_b$, see Def. 4.1. Their composition according to the dependency $d\colon (R \xleftarrow[m]{} B_p \to (B_p \cup R_p)$ is constructed as in Def. A.1, and shown in Fig. 9.

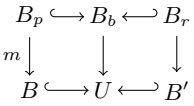


Fig. 8. $B \Rightarrow^{m}_{\delta_B} B'$

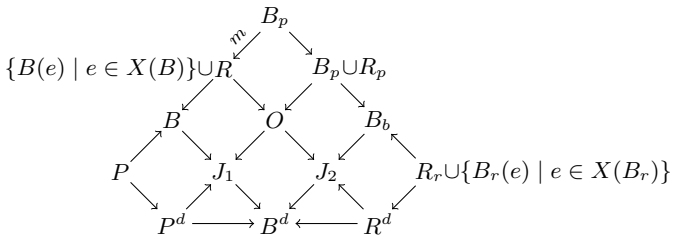


Fig. 9. $r^d = \tilde{s} \circ_d \tilde{\delta}$

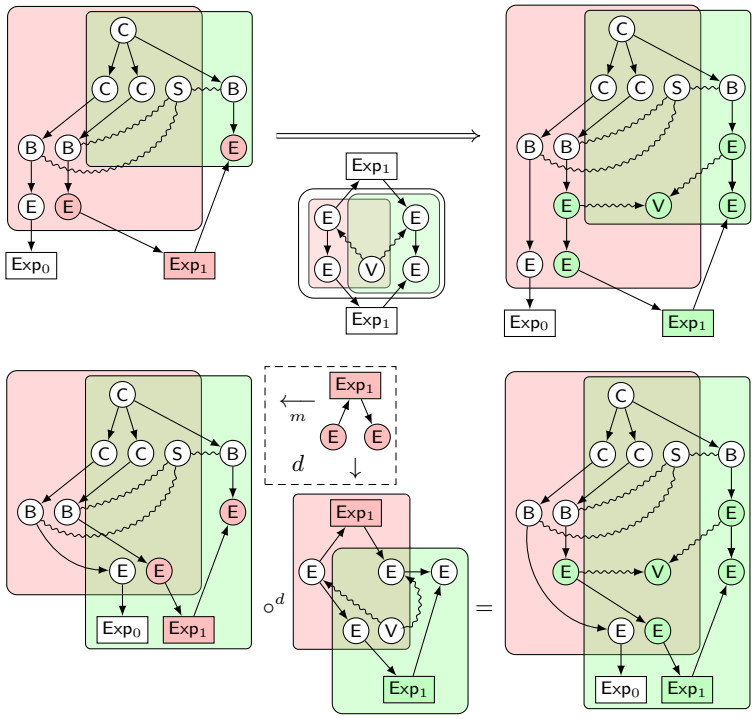


Fig. 10. Schema refinement and incremental composition

Consider the underlying body refinement $B \Rightarrow^{m}_{\delta_B} B'$. (See Fig. 8, where we assume that the lower horizontal morphisms are inclusions.) By uniqueness of pushouts, $U \cong B_d$. Then $(B_b \setminus B_p) = X(B_p)$ since δ_B is contextual, and $B' = U \setminus \bar{m}(X(B_p))$.

It is then easy to show that the body $B'_{\tilde{\delta}}$ equals the body B^d of the composed delaying rule, and an easy argument concerning the whereabouts of variables shows that $\tilde{t} = r^d$. □

Example 7 (Schema Refinement and Incremental Rules). Figure 10 illustrates the relation between schema refinement and the composition of their incremental rules established in Lemma 4.2. As already mentioned in Example 4, the upper row shows a step in the refinement sequence $\mathsf{pum} \Downarrow^{*}_{\Delta_{\mathsf{M}}} \mathsf{pum}'$ that applies the context-free variation $\overline{\mathsf{ass}}_1$ of the meta-rule ass_1 in Fig. 6. This step shows why we need less contextual variations of meta-rules: The original meta-rule does not apply to the source schema, as it does not contain a node labeled V. The less contextual rule does apply; the refined rule is constructed so that The V-node will be matched in the context when it is applied to a source graph.

The lower row shows the composition of the corresponding delaying rule with the corresponding incremental refinement rule $\widetilde{\overline{\mathsf{ass}}}_1$, where the dashed box

specifies the dependency d for the composition. The composed rule equals the delaying rule for the refined schema.

Using a refined schema has the same effect as applying its delaying rule, and the incremental rules of the corresponding meta-rules. This must follow a strategy that applies incremental rules as long as possible, matching the residuals of the source graphs, before another delaying rule is applied.

We define the subgraph that is left unchanged in refinement steps and sequences. To ease the following definitions, we assume wlog. that a rewrite step $G \Rightarrow_r^m H$ with a diagram as in Def. 2.2 is constructed so that the lower horizontal morphisms are inclusions $G \hookrightarrow U \hookleftarrow H$. The *track* of G in H (via the match m of the rule r) is then defined as $tr_r^m(G) = (G \cap H)$. For a rewrite sequence $d = G_0 \Rightarrow_{r_1}^{m_1} G_1 \Rightarrow_{r_2}^{m_2} \ldots \Rightarrow_{r_n}^{m_n} G_n$, the track of G in H is given by intersecting the tracks of its steps:

$$tr_d(G) = tr_{r_1}^{m_1}(G_0) \cap \cdots \cap tr_{r_n}^{m_n}(G_{n-1})$$

The incremental rules have to be applied so that the patterns of the refinements of the original meta-rules do not overlap.

Definition 4.3 (Residual Incremental Refinement). Consider an incremental refinement sequence

$$G_0 \Rightarrow_{\tilde{\delta}_1}^{m_1} G_1 \Rightarrow_{\tilde{\delta}_2}^{m_2} \ldots \Rightarrow_{\tilde{\delta}_n}^{m_n} G_n$$

with incremental rules $\tilde{\delta}_i$ for meta-rules $\delta_i \colon (p_i \hookrightarrow b_i \hookleftarrow r_i)$ (for $1 \leqslant i \leqslant n$).

The step $G_{i-1} \Rightarrow_{\tilde{\delta}_i}^{m_i} G_i$ is *residual* if $m_i(P_{r_i}) \subseteq tr_{r_1 \ldots r_{i-1}}^{m_1 \ldots m_{i-1}}(G)$. The sequence is residual if every of its steps is residual. Residual steps and sequences are denoted as $\Rrightarrow$ and $\Rrightarrow^*$, respectively.

Lemma 4.4. *Consider a schema s for meta-rules Δ with delaying rule $\tilde{s}$ and incremental rules $\tilde{\Delta}$.*

Then a rule $r \colon (P \hookrightarrow B \hookleftarrow R)$ is a refinement in $\Delta(s)$ if and only if $P \Rightarrow_{\tilde{s}} P' \Rrightarrow_{\tilde{\Delta}}^{!} R$.

Proof. By induction over the length of meta-derivations, using Lemma 4.2 and the fact that compositions correspond to residual rewrite steps. □

Theorem 4.5. *Consider a schema s with meta-rules Δ as above. Then, for graphs G, H, and K, $G \Rightarrow_{\Delta(s)} H$ if and only if $G \Rightarrow_{\tilde{s}} K \Rrightarrow_{\tilde{\Delta}}^{!} H$.*

Proof. Combine Lemma 4.4 with the embedding theorem [4, Sect. 6.2]. □

5 Conclusions

In this paper we have defined how the refinement of schemata of plain graph rewrite rules according to contextual meta-rules can be translated to standard

rules that perform the refinement incrementally. We have also investigated conditions under which the refinement behaves well, i.e., terminates, and yields unique refinements.

Our ultimate goal is to analyze confluence of systems of schemata and meta-rules with the critical pair lemma [17]. The negative result shown in [14, Thm. 3] indicates that considerable restrictions have to be made to reach this aim. A possible way could be to restrict the rewriting with refinements to graphs that are *shaped*, e.g., according to contextual grammars like the program graphs shown in this paper.

Until now, we have not considered attributed graphs and subtyping. As they are included in the foundation [4], we expect that this can be added in a rather orthogonal way. We also restricted ourselves to unconditional rules. Rules with nested application conditions have been added to the theory in [6,5]; recently, Hendrik Radke has studied recursive refinement of such conditions [13]. We plan to add these concepts to our definition in the future.

Acknowledgments. I am indebted to Annegret Habel and Rachid Echahed for their patience and their encouragement.

References

1. J. Blomer, R. Geiß, and E. Jakumeit. GrGen.net: A generative system for graph-rewriting, user manual. www.grgen.net, 2006. Version 4.3bms2 (30. 04. 2014).
2. F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. Van Eetvelde. Shaped generic graph transformation. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, number 5088 in Lecture Notes in Computer Science, pages 201–216. Springer, 2008.
3. F. Drewes, B. Hoffmann, and M. Minas. Contextual hyperedge replacement. In A. Schürr, D. Varró, and G. Varró, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'11)*, number 7233 in Lecture Notes in Computer Science, pages 182–197. Springer, 2012. Long version as UMINF report 14.04, Institutionen för datavetenskap,Umeå universitet.
4. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs. Springer, 2006.
5. H. Ehrig, U. Golas, A. Habel, L. Lambers, and F. Orejas. Adhesive transformation systems with nested application conditions. part 2: Embedding, critical pairs and local confluence. *Fundam. Inform.*, 118(1-2):35–63, 2012.
6. H. Ehrig, A. Habel, L. Lambers, F. Orejas, and U. Golas. Local confluence for rules with nested application conditions. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors, *ICGT*, volume 6372 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2010.
7. H. Ehrig, F. Hermann, and U. Prange. Cospan DPO approach: An alternative for DPO graph transformations. *Bulletin of the EATCS*, 98:139–149, 2009.
8. H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: An algebraic approach. In *IEEE Conf. on Automata and Switching Theory*, pages 167–180, Iowa City, 1973.

9. C. Ermel, M. Rudolf, and G. Taentzer. The AGG 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*, chapter 14, pages 551–603. World Scientific, Singapore, 1999.
10. M. Fowler. *Refactoring—Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, Reading, MA, 1999.
11. A. Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.
12. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
13. A. Habel and H. Radke. Expressiveness of graph conditions with variables. *Elect. Comm. of the EASST*, 30, 2010. International Colloquium on Graph and Model Transformation (GraMoT'10).
14. B. Hoffmann. Graph rewriting with contextual refinement. *Electr. Comm. of the EASST*, 61:20 pages, 2013.
15. B. Hoffmann, E. Jakumeit, and R. Geiß. Graph rewrite rules with structural recursion. In M. Mosbah and A. Habel, editors, *2nd Intl. Workshop on Graph Computational Models (GCM 2008)*, pages 5–16, 2008.
16. E. Jakumeit. *Mit* GrGen *zu den Sternen.* Diplomarbeit (in German), Universität Karlsruhe, 2008.
17. D. Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In M. R. Sleep, R. Plasmeijer, and M. v. Eekelen, editors, *Term Graph Rewriting, Theory and Practice*, pages 201–213. Wiley & Sons, Chichester, 1993.
18. A. Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl, J. Pfaltz, and B. Böhlen, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'03)*, number 3062 in Lecture Notes in Computer Science, pages 479–485. Springer, 2004.
19. N. Van Eetvelde and D. Janssens. A hierarchical program representation for refactoring. *Electronic Notes in Theoretical Computer Science*, 82(7), 2003.

A Double-Pushout Graph Rewriting

The standard theory of graph rewriting is based on so-called *spans* of (injective) graph morphisms [4], where a rule consists of two morphisms from a common interface I to a pattern P and a replacement R. An alternative proposed in [7] uses so-called co-spans (or joins) of morphisms where the pattern and the replacement are both included in a common supergraph, which we call the body of the rule.

Rewriting is defined by double pushouts as below:

$$\begin{array}{ccccc} \hat{r}\colon & P & \hookleftarrow & I & \hookrightarrow & R \\ & {\scriptstyle m}\downarrow & & \downarrow & & \downarrow \\ & G & \hookleftarrow & C & \hookrightarrow & H \end{array} \qquad \begin{array}{ccccc} \check{r}\colon & P & \hookrightarrow & B & \hookleftarrow & R \\ & {\scriptstyle m}\downarrow & & \downarrow & & \downarrow \\ & G & \hookrightarrow & U & \hookleftarrow & H \end{array}$$

Intuitively, rewrites are constructed via a match morphism $m\colon P \to G$ in a source graph G; for a span rule $\hat{r}$, removing the match of obsolete pattern items

$P \setminus I$ yields a context graph C to which the new items $R \setminus I$ of the replacement are then added; for a cospan rule $\check{r}$, the new items $B \setminus P$ are added first, yielding the united graph U before the obsolete pattern items $B \setminus P$ are removed. The constructions work if the matches m satisfy certain *gluing conditions*.

The main result of [4] says that $\check{r}$ is the pushout of $\hat{r}$, making these rules, their rewrite steps, and gluing conditions dual to each other. Therefore we feel free to use the more intuitive gluing condition for $\hat{r}$ together with a rule $\check{r}$.

The following definition and theorem adapt well-known concepts of [4] to our notion of rules.

Definition A.1 (Sequential Rules Composition). Let $r_1 \colon (P_1 \hookrightarrow B_1 \hookleftarrow R_1)$ and $r_2 \colon (P_2 \hookrightarrow B_2 \hookleftarrow R_2)$ be rules, and consider a graph D with a pair $d \colon (R_1 \leftarrow D \rightarrow P_2)$ of injective morphisms.

1. Then d is a *sequential dependency* of r_1 and r_2 if $D \not\to P_1$ (which implies that $D \neq \langle\rangle$).
2. The *sequential composition* $r_1 \circ_d r_2 \colon (P^d \hookrightarrow B^d \hookleftarrow R^d)$ of r_1 and r_2 along d is the rule constructed as in the commutative diagram of Fig. 11, where all squares are pushouts.
3. Two rewrite steps $G \Rightarrow_{r_1} H \Rightarrow_{r_2'} K$ are *d-related* if d is the pullback of the embedding $R_1 \to H$ and of the match $P_2 \to H$.[8]

Proposition A.2. *Let r_1 and r_2 be rules with a dependency d and a sequential composition r^d as in Def. A.1.*

Then there exist d-related rewrite steps $G \Rightarrow_{r_1} H \Rightarrow_{r_2} K$ if and only if $G \Rightarrow_{r^d} K$.

Proof. Straightforward use of the corresponding result for "span rules" [4, Thm. 5.23] and of the duality to "cospan rules" [7]. □

[8] A *pullback* of a pair of morphisms $B \to D \leftarrow C$ with the same codomain is a pair of morphisms $B \leftarrow A \to C$ that is *commutative*, i.e., $A \to B \to D = B \to C \to D$, and *universal*, i.e., for every pair of morphisms $B \to A' \leftarrow C$ so that $A' \to B \to D = A' \to C \to D$, there is a unique morphism $A' \to A$ so that $A \to A' \to B = A' \to B$ and $A' \to A \to C = A' \to C$. See [4, Def. 2.2]

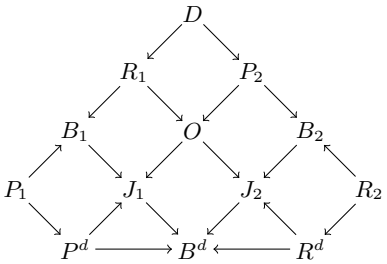


Fig. 11. Sequential composition of graph rewrite rules