GCM 2012

The Fourth International Workshop on Graph Computation Models

Proceedings

Bremen, Germany, 28-29 September 2012

Editors:
Rachid Echahed, Annegret Habel and Mohamed Mosbah

Contents

Preface

This volume contains the proceedings of the Fourth International Workshop on *Graph Computation Models (GCM 2012*[1]*)*. The workshop took place in Bremen, Germany, on 28-29 September, 2012, as part of the sixth edition of the International Conference on Graph Transformation (ICGT 2012).

The aim of GCM[2] workshop series is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation techniques. It promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications, and implementations of graph computation models and related areas. Previous editions of GCM series were held in Natal, Brazil (GCM2006), in Leicester, UK (GCM2008) and in Enschede, The Netherlands (GCM 2010).

These proceedings contain 11 accepted papers. All submissions were subject to careful refereeing. The topics of accepted papers range over a wide spectrum, including theoretical aspects of graph transformation, proof methods, formal languages 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 2012, 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 2012 Conference Organizing Committee for their help in organizing GCM 2012 in Bremen, Germany.

September, 2012 Rachid Echahed, Annegret Habel and Mohamed Mosbah
Programme co-chairs of GCM 2012

[1] GCM2012 web site: http://gcm2012.imag.fr
[2] GCM web site : http://gcm-events.org

Programme committee of GCM 2012

Paolo Baldan	University of Padova, Italy
Frank Drewes	Umea University, Sweden
Rachid Echahed	University of Grenoble, France (co-chair)
Stefan Gruner	University of Pretoria, South Africa
Annegret Habel	University of Oldenburg, Germany (co-chair)
Dirk Janssens	University of Antwerp, Belgium
Hans-Jörg Kreowski	University of Bremen, Germany
Pascale Le Gall	University of Evry-Val d'Essonne, France
Mohamed Mosbah	University of Bordeaux 1, France (co-chair)
Detlef Plump	University of York, UK

Additional Reviewers

Christopher Bak
Sabine Kuske
Christopher Poskitt
Caroline von Totth

Verifying Total Correctness of Graph Programs

Christopher M. Poskitt and Detlef Plump

Department of Computer Science
The University of York, UK

Abstract. GP (for Graph Programs) is an experimental nondeterministic programming language for solving problems on graphs and graph-like structures. The language is based on graph transformation rules, allowing visual programming at a high level of abstraction. Previous work has demonstrated how to verify such programs using a Hoare-style proof system, but only partial correctness was considered. In this paper, we extend our calculus with new rules and termination functions, allowing proofs that program executions always terminate (weak total correctness) and that programs always terminate without failing program runs (total correctness). We show that the new proof system is sound with respect to GP's operational semantics, complete for termination, and demonstrate how it can be used.

1 Introduction

The verification of graph transformation systems is an area of active and growing interest, motivated by the many applications of graph transformation to specification and programming. While much of the research in this area (see e.g. [2,3,8,4]) has focused on sets of rules or graph grammars, the challenge of verifying graph-based programming languages is also beginning to be addressed. In particular, Habel, Pennemann, and Rensink [6,5] contributed a weakest precondition based verification framework for a simple graph transformation language, using nested conditions as the assertions. The language however, does not support important practical features such as computations on labels.

In [12] we consider the verification of GP [11], a nondeterministic graph programming language whose states are directed labelled graphs. These are manipulated directly via the application of (conditional) rule schemata, which generalise double-pushout rules with expressions over labels and relabelling. The framework of [12] is a Hoare-style proof calculus for partial correctness. However, the calculus cannot be used to prove that programs eventually terminate if their preconditions are satisfied, nor that their executions are absent of failure states. This paper aims to address these issues.

We define two notions of total correctness: a weaker one accounting for termination, and a stronger one accounting for that as well as for absence of failures. We define two calculi for these notions of total correctness by modifying our previous proof rules, addressing divergence via the use of termination functions

that map graphs to natural numbers. We demonstrate the proof calculi on programs that have loops and failure points, before proving them to be sound, and proving that the proof rule for loops is complete for termination.

Section 2 reviews some technical preliminaries; Section 3 is an informal refresher on graph programs; Section 4 reviews our assertion language and the partial correctness proof rules of our previous calculus; Section 5 formalises the notion of (weak) total correctness and presents new proof rules which allow one to prove these properties; Section 6 demonstrates the use of the new calculi; Section 7 presents a proof that the new calculi are sound for (weak) total correctness, and also a proof that the calculi are complete for termination; and finally, Section 8 concludes.

2 Preliminaries

Graph transformation in GP is based on the double-pushout approach with relabelling [7]. This framework deals with partially labelled graphs, whose definition we recall below. We consider two classes of graphs, "syntactic" graphs labelled with expressions and "semantic" graphs labelled with (sequences of) integers and strings. We also introduce assignments which translate syntactic graphs into semantic graphs, and substitutions which operate on syntactic graphs.

A *graph* over a label alphabet $\mathcal{C}$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$, where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G \colon E_G \to V_G$ are the *source* and *target* functions for edges, $l_G \colon V_G \to \mathcal{C}$ is the partial node labelling function and $m_G \colon E_G \to \mathcal{C}$ is the (total) edge labelling function. Given a node v, we write $l_G(v) = \bot$ to express that $l_G(v)$ is undefined. Graph G is *totally labelled* if l_G is a total function. We write $\mathcal{G}(\mathcal{C})$ for the set of all totally labelled graphs over $\mathcal{C}$, and $\mathcal{G}(\mathcal{C}_\bot)$ for the set of all graphs over $\mathcal{C}$.

A *graph morphism* $g \colon G \to H$ between graphs G and H consists of two functions $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that preserve sources, targets and labels; that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, and $l_H(g(v)) = l_G(v)$ for all v such that $l_G(v) \neq \bot$. Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x. It is *injective* (*surjective*) if g_V and g_E are injective (surjective). It is an *isomorphism* if it is injective, surjective and satisfies $l_H(g_V(v)) = \bot$ for all nodes v with $l_G(v) = \bot$. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

We consider graphs over two distinct label alphabets. Graph programs and our assertion language contain graphs labelled with expressions, while the graphs on which programs operate are labelled with (sequences of) integers and character strings. We consider graphs of the first type as syntactic objects and graphs of the second type as semantic objects, and aim to clearly separate these levels of syntax and semantics.

Let $\mathbb{Z}$ be the set of integers and Char be a finite set of characters. We fix the label alphabet $\mathcal{L} = (\mathbb{Z} \cup \mathrm{Char}^*)^+$ of all non-empty sequences over integers and character strings. The other label alphabet we are using, Exp, consists of underscore delimited sequences of arithmetical expressions and strings. These

may contain (untyped) variable identifiers, the class of which we denote VarId. For example, `x*5` and "`root`"_`y` are both elements of Exp with `x`, `y` in VarId. (See [12] for a formal grammar defining Exp.) We write $\mathcal{G}(\text{Exp})$ for the set of all graphs over the syntactic class Exp.

Each graph in $\mathcal{G}(\text{Exp})$ represents a possibly infinite set of graphs in $\mathcal{G}(\mathcal{L})$. The latter are obtained by instantiating variables with values from $\mathcal{L}$ and evaluating expressions. An *assignment* is a partial function $\alpha\colon \text{VarId} \to \mathcal{L}$. Given an expression e, α is *well-typed* for e if it is defined for all variables occurring in e and if all variables within arithmetical (resp. string) expressions are mapped to integers (resp. strings). In this case we inductively define the value $e^\alpha \in \mathcal{L}$ as follows. If e is a numeral or a sequence of characters, then e^α is the integer or character string represented by e. If e is a variable identifier, then $e^\alpha = \alpha(e)$. For arithmetical and string expressions, e^α is defined inductively in the usual way. Finally, if e has the form t_e_1 with t a string or arithmetical expression and $e_1 \in \text{Exp}$, then $e^\alpha = t^\alpha e_1^\alpha$ (the concatenation of the sequences t^α and e_1^α). Given a graph G in $\mathcal{G}(\text{Exp})$ and an assignment α that is well-typed for all expressions in G, we write G^α for the graph in $\mathcal{G}(\mathcal{L})$ that is obtained from G by replacing each label e with e^α (note that G^α has the same nodes, edges, source and target functions as G). If $g\colon G \to H$ is a graph morphism with $G, H \in \mathcal{G}(\text{Exp})$, then g^α denotes the morphism $\langle g_V, g_E\rangle\colon G^\alpha \to H^\alpha$.

A *substitution* is a partial function $\sigma\colon \text{VarId} \to \text{Exp}$. Given an expression e, σ is *well-typed* for e if it does not replace variables in arithmetical expressions with strings (and similarly for string expressions). In this case, the expression e^σ is obtained from e by replacing every variable `x` for which σ is defined with $\sigma(\texttt{x})$ (if σ is not defined for a variable `x`, then $\texttt{x}^\sigma = \texttt{x}$). Given a graph G in $\mathcal{G}(\text{Exp})$ such that σ is well-typed for all labels in G, we write G^σ for the graph in $\mathcal{G}(\text{Exp})$ that is obtained by replacing each label e with e^σ. If $g\colon G \to H$ is a graph morphism between graphs in $\mathcal{G}(\text{Exp})$, then g^σ denotes the morphism $\langle g_V, g_E\rangle\colon G^\sigma \to H^\sigma$. Given an assignment $\alpha\colon \text{VarId} \to \mathcal{L}$, the substitution $\sigma_\alpha\colon \text{VarId} \to \text{Exp}$ *induced* by α maps every variable `x` to the expression that is obtained from $\alpha(\texttt{x})$ by replacing integers and strings with their syntactic counterparts. For example, if $\alpha(\texttt{x})$ is the integer 23, then $\sigma_\alpha(\texttt{x})$ is `23` (the syntactic digits). Consider another example: if $\alpha(\texttt{x})$ is the sequence $56, a, bc$, where 56 is an integer and a and bc are strings, then $\sigma_\alpha(\texttt{x}) =$ `56_"a"_"bc"`.

3 Graph Programs

We introduce graph programs informally and by example in this section. For technical details, further examples, and the operational semantics, refer to [11].

The "building blocks" of graph programs are conditional rule schemata: a program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling their application. Rule schemata generalise graph transformation rules, in that labels can contain (sequences of) expressions over parameters of type integer or string. Labels in the left graph comprise only variables and constants (no composite expressions) because their

values at execution time are determined by graph matching. The condition of a rule schema is a simple Boolean expression over the variables.

The program `colouring` in Figure 1 produces a colouring (assignment of integers to nodes such that adjacent nodes have different colours) for every (untagged) integer-labelled input graph, recording colours as so-called tags. In general, a tagged label is a sequence of expressions separated by underscores.

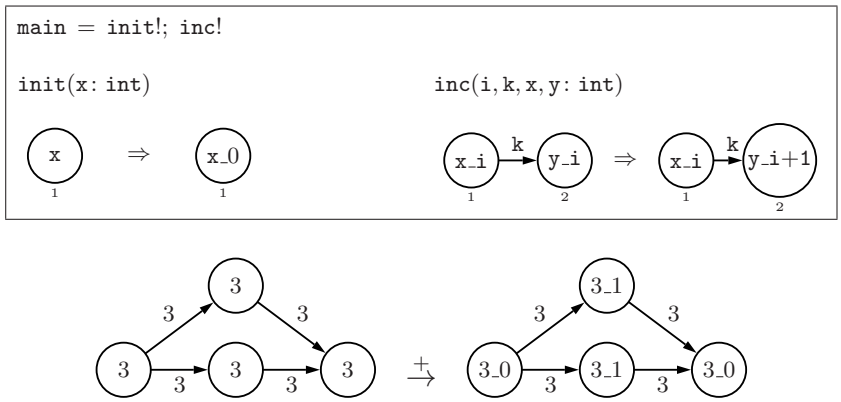


Fig. 1. The program `colouring` and one of its executions

The program initially colours each node with 0 by applying the rule schema `init` as long as possible, using the iteration operator '`!`'. It then repeatedly increments the target colour of edges with the same colour at both ends. Note that this process is nondeterministic: Figure 1 shows one possible execution; there is another execution resulting in a graph with three colours.

The program `reachable`? in Figure 2 checks if there is a path from one distinguished node (tagged with 1, i.e. x_1) to another (tagged with 2, i.e. y_2), returning the input graph if there is one, otherwise returning the same graph but with a new direct link between them. It repeatedly propagates 0-tagged nodes from the 1-tagged node (and subsequent 0-tagged nodes) for as long as possible via `prop`!. It then tests via `reachable` whether there is a direct link between the distinguished nodes, or a link from a 0-tagged node to the 2-tagged node (indicating a path). If so, nothing happens; otherwise, a direct link is added via `addlink`. In both cases, the 0-tags are removed by the iteration of `undo`.

GP's formal semantics is given in the style of structural operational semantics. Inference rules (omitted here, but given in [12]) inductively define a small-step transition relation $\rightarrow$ on *configurations*. In our setting, a configuration is either a command sequence (ComSeq) together with a graph (i.e. an unfinished computation), just a graph, or the special element fail (representing a failure state). The meaning of graph programs is summarised by a semantic

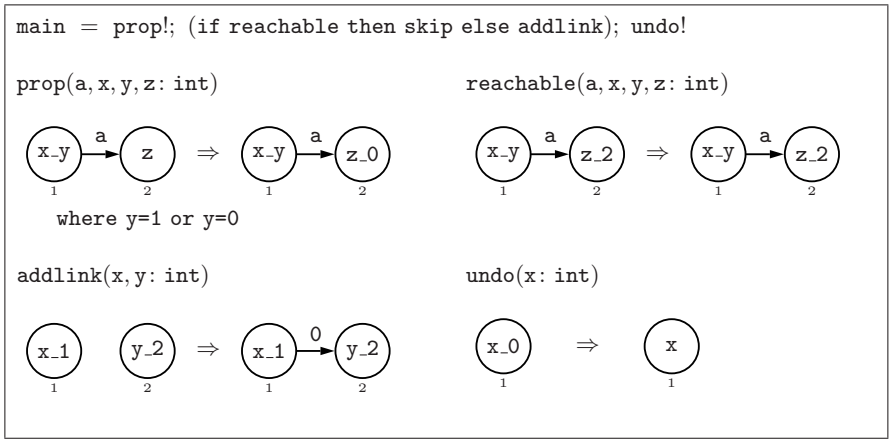


Fig. 2. The program `reachable?`

function $[\![\_]\!]$, which assigns to every program P the function $[\![P]\!]$ mapping an input graph G to the set of all possible results of running P on G. The result set may contain, besides proper results in the form of graphs, the special value $\perp$ which indicates a non-terminating or stuck computation. The *semantic function* $[\![\_]\!]\colon \mathrm{ComSeq} \to (\mathcal{G}(\mathcal{L}) \to 2^{\mathcal{G}(\mathcal{L})\cup\{\perp\}})$ is defined by:

$$[\![P]\!]G = \{H \in \mathcal{G}(\mathcal{L}) \mid \langle P, G\rangle \xrightarrow{+} H\} \cup \{\perp \mid P \text{ can diverge or get stuck from } G\}$$

where P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$, and P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$ (where the rest program Q cannot be executed because no inference rule is applicable).

4 Proving Partial Correctness

In this section we first review E-conditions, the assertion language of our proof calculus. Then, we review the partial correctness proof calculus presented in previous work.

Nested graph conditions with expressions (or E-conditions) are a morphism-based formalism for expressing graph properties. E-conditions [12] extend the nested conditions of [6] with expressions for labels, and *assignment constraints*, which are simple Boolean expressions that restrict the types of – and relations between – values that instantiate variables. An assignment constraint γ is evaluated with respect to an assignment α, denoted γ^α, by instantiating variables with the values given by α then replacing function and relation symbols with the obvious functions and relations. Because of space limitations, we do not give a formal syntax or semantics, but refer the reader to [12].

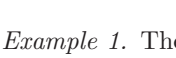
A substitution $\sigma\colon \mathrm{VarId} \to \mathrm{Exp}$ can be applied to an assignment constraint γ, if it is well-typed for all expressions in γ. The resulting assignment constraint, denoted by γ^σ, is simply γ with each expression e replaced by e^σ.

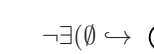
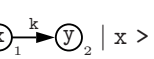
Definition 1 (E-condition). An *E-condition* c over a graph P is of the form true or $\exists(a \mid \gamma, c')$, where $a\colon P \hookrightarrow C$ is an injective graph morphism with $P, C \in \mathcal{G}(\mathrm{Exp})$, γ is an assignment constraint, and c' is an E-condition over C. Boolean formulae over E-conditions over P yield E-conditions over P, that is, $\neg c$ and $c_1 \wedge c_2$ are E-conditions over P if c, c_1, c_2 are E-conditions over P. □

The *satisfaction* of E-conditions by injective graph morphisms between graphs in $\mathcal{G}(\mathcal{L})$ is defined inductively. Every such morphism satisfies the E-condition true. An injective graph morphism $s\colon S \hookrightarrow G$ with $S, G \in \mathcal{G}(\mathcal{L})$ *satisfies* the E-condition $c = \exists(a\colon P \hookrightarrow C \mid \gamma, c')$, denoted $s \models c$, if there exists an assignment α that is well-typed for all expressions in P, C, γ and is undefined for variables present only in c', such that $S = P^\alpha$, and such that there is an injective graph morphism $q\colon C^\alpha \hookrightarrow G$ with $q \circ a^\alpha = s$, $\gamma^\alpha = \mathrm{true}$, and $q \models (c')^{\sigma_\alpha}$. Here, σ_α is the substitution induced by α, which we require to be well-typed for all expressions in c'. If such an assignment α and morphism q exist, we say that s *satisfies* c *by* α, and write $s \models_\alpha c$.

For brevity, we write false for $\neg\mathrm{true}$, $\exists(a \mid \gamma)$ for $\exists(a \mid \gamma, \mathrm{true})$, $\exists(a, c')$ for $\exists(a \mid \texttt{true}, c')$, and $\forall(a \mid \gamma, c')$ for $\neg\exists(a \mid \gamma, \neg c')$. In our examples, when the domain of morphism $a\colon P \hookrightarrow C$ can unambiguously be inferred, we write only the codomain C. For instance, an E-condition $\exists(\emptyset \hookrightarrow C, \exists(C \hookrightarrow C'))$ can be written as $\exists(C, \exists(C'))$, where the domain of the outermost morphism is the empty graph, and the domain of the nested morphism is the codomain of the encapsulating E-condition's morphism.

An E-condition over a graph morphism whose domain is the empty graph is referred to as an *E-constraint*.

Example 1. The E-constraint $\forall($  $\mid \mathtt{x} > \mathtt{y}, \exists($  $))$ expresses that every pair of adjacent integer-labelled nodes with the source label greater than the target label has a loop incident to the source node. The unabbreviated version of the condition is as follows:

$\neg\exists(\emptyset \hookrightarrow$  $\mid \mathtt{x} > \mathtt{y}, \neg\exists($ $\hookrightarrow$  $\mid \mathtt{true}, \mathrm{true}))$.

□

A graph G in $\mathcal{G}(\mathcal{L})$ satisfies an E-constraint c, denoted $G \models c$, if the morphism $i_G\colon \emptyset \hookrightarrow G$ satisfies c.

The satisfaction of (resp. application of well-typed substitutions to) Boolean formulae over E-conditions is defined inductively, in the usual way.

Definition 2 (Partial correctness). A graph program P is *partially correct* with respect to a precondition c and postcondition d (both of which are E-constraints), denoted $\models_{\mathrm{par}} \{c\}\ P\ \{d\}$, if for every graph $G \in \mathcal{G}(\mathcal{L})$, $G \models c$ implies $H \models d$ for every graph H in $[\![P]\!]G$. □

In [12] we defined axioms and rules for deriving Hoare triples from graph programs. These are given in Figure 3, where r (resp. $\mathcal{R}$) ranges over conditional rule schemata (resp. sets of conditional rule schemata), c, c', d, d', e, inv over E-constraints, and P, Q over graph programs. Together, the axioms and rules define a proof system for partial correctness. If a Hoare triple $\{c\}\ P\ \{d\}$ can be derived from the proof system, we write $\vdash_{\text{par}} \{c\}\ P\ \{d\}$. The proof system is sound in the sense of partial correctness, that is, $\vdash_{\text{par}} \{c\}\ P\ \{d\}$ implies $\models_{\text{par}} \{c\}\ P\ \{d\}$ (see [12]).

$$[\text{ruleapp}]\ \frac{}{\{\text{Pre}(r, c)\}\ r\ \{c\}} \qquad [\text{nonapp}]\ \frac{}{\{\neg\text{App}(\mathcal{R})\}\ \mathcal{R}\ \{\text{false}\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}} \qquad [!]\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\text{App}(\mathcal{R})\}}$$

$$[\text{comp}]\ \frac{\{c\}\ P\ \{e\}, \quad \{e\}\ Q\ \{d\}}{\{c\}\ P; Q\ \{d\}} \qquad [\text{cons}]\ \frac{c \Rightarrow c', \quad \{c'\}\ P\ \{d'\}, \quad d' \Rightarrow d}{\{c\}\ P\ \{d\}}$$

$$[\text{if}]\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\}, \quad \{c \wedge \neg\text{App}(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

Fig. 3. Partial correctness proof rules for GP's core commands

Two transformations – App and Pre – appear in the axioms and rules. Intuitively, App takes as input a set $\mathcal{R}$ of conditional rule schemata, and transforms it into an E-condition satisfied only by graphs for which at least one rule schema in $\mathcal{R}$ is applicable. Pre on the other hand constructs an E-condition such that if $G \models \text{Pre}(r, c)$, and the application of r to G results in a graph H, then $H \models c$. Formal constructions of the transformations are omitted from this paper, but can be found in [12].

We note that the proof system is for a strict subset of graph programs. Specifically, as-long-as-possible iteration can only be applied to sets of rule schemata, and the guards of conditionals are restricted to sets of rule schemata (in both cases the semantics of GP allows arbitrary programs). Without this restriction, the proof rules would require an assertion language able to express that an arbitrary program will not fail.

5 Proving Total Correctness

If $\vdash_{\text{par}} \{c\}\ P\ \{d\}$, then should P be executed on a graph G satisfying c, we can be sure that any graph resulting will satisfy d. What we cannot be sure about is whether an execution of P will ever terminate (i.e. whether an execution might

diverge or not). Moreover, if an execution of P does in fact terminate, we cannot be sure that it does so without failure. When referring to *total correctness*, we follow [1] in meaning both absence of divergence and failure; and when referring to *weak total correctness*, we mean only absence of divergence.

Definition 3 (Weak total correctness). A graph program P is *weakly totally correct* with respect to a precondition c and postcondition d (both of which are E-constraints), denoted $\models_{\text{wtot}} \{c\}\ P\ \{d\}$, if $\models_{\text{par}} \{c\}\ P\ \{d\}$ and if for every graph $G \in \mathcal{G}(\mathcal{L})$ such that $G \models c$, there is no infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \cdots$. □

Definition 4 (Total correctness). A graph program P is *totally correct* with respect to a precondition c and postcondition d (both of which are E-constraints), denoted $\models_{\text{tot}} \{c\}\ P\ \{d\}$, if $\models_{\text{wtot}} \{c\}\ P\ \{d\}$, and if for every graph $G \in \mathcal{G}(\mathcal{L})$ such that $G \models c$, there is no derivation $\langle P, G\rangle \to^* \text{fail}$. □

Our proof system for weak total correctness is formed from the proof rules of Figure 3, but with $[!]_{\text{tot}}$ in Figure 4 substituted for $[!]$. If a triple $\{c\}\ P\ \{d\}$ can be derived from this proof system, we write $\vdash_{\text{wtot}} \{c\}\ P\ \{d\}$. The issue of termination is localised to the proof rule for as-long-as-possible iteration: $[!]_{\text{tot}}$ has an additional premise to $[!]$ which handles this. It requires, for a particular rule schemata set, that there is a function assigning naturals to graphs such that these naturals are decreasing along derivation steps. Such a function $\#$ is called a *termination function*. If $\#$ decreases along derivation steps yielded from applying $\mathcal{R}$ to graphs satisfying *inv*, we say that $\mathcal{R}$ is *$\#$-decreasing under inv*. These definitions are given more precisely below.

$$[!]_{\text{tot}}\ \frac{\vdash_{\text{par}} \{inv\}\ \mathcal{R}\ \{inv\}, \quad \mathcal{R} \text{ is } \#\text{-decreasing under } inv}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg \text{App}(\mathcal{R})\}}$$

$$[\text{ruleset}]_{\text{tot}}\ \frac{c \Rightarrow \text{App}(\mathcal{R}), \quad \vdash_{\text{par}} \{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

Fig. 4. Total correctness proof rules for two core GP commands

Definition 5 (Termination function; $\#$-decreasing). A *termination function* is a mapping $\# : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ from (semantic) graphs to natural numbers. Given an E-constraint c, a set of conditional rule schemata $\mathcal{R}$ is *$\#$-decreasing under c* if for all graphs G, H in $\mathcal{G}(\mathcal{L})$ such that $G \models c$ and $H \models c$,

$$G \Rightarrow_{\mathcal{R}} H \text{ implies } \#G > \#H.$$

□

In an application of $[!]_{\text{tot}}$, one must find a suitable termination function # that returns smaller natural numbers along the graphs of direct derivations. A simple, intuitive termination function would be one that maps a graph to its size (e.g. total number of nodes and edges). If a rule schemata set is reducing the size of a graph upon each application, then clearly the iteration cannot continue indefinitely, and this is reflected by the output of # tending towards zero. However, in cases when rule schemata are not necessarily decreasing the size of the graph, much less trivial termination functions may be needed. We also mention that the problem to decide whether a set of rule schemata is terminating or not, is undecidable in general [10]. Note that the rule $[!]_{\text{tot}}$ requires only that # is decreasing for graphs that satisfy the invariant inv, i.e. it need not be decreasing for graphs outside of the particular context.

Our proof system for total correctness is formed of [comp], [cons], [if], and the proof rules of Figure 4. If a triple $\{c\}\ P\ \{d\}$ can be derived from this proof system, we write $\vdash_{\text{tot}} \{c\}\ P\ \{d\}$. (We do not include a proof rule for a program that is just a single rule schema r, because this case is captured by proving $\vdash_{\text{tot}} \{c\}\ \{r\}\ \{d\}$.) This proof system allows one to prove that all program executions terminate without failure. Essentially, this is achieved by ensuring that the preconditions of rule schemata sets imply their applicability. Hence if graphs satisfy the preconditions, by implication the rule schemata sets are applicable to those graphs, and thus we can be certain that no execution will fail.

The proof rule $[\text{ruleset}]_{\text{tot}}$ separates the issues of failure and partial correctness. In using the proof rule, one must show (outside the calculus) that the applicability of $\mathcal{R}$ is logically implied by the precondition c. In showing that this premise holds, we can be sure that at least one rule schema in $\mathcal{R}$ can be applied to a graph satisfying c, hence no execution on that graph will fail. Separately, it must be shown that $\vdash_{\text{par}} \{c\}\ r\ \{d\}$ for each $r \in \mathcal{R}$, that is, each rule schema in the set is partially correct with respect to the pre- and postcondition. Together, we derive that every execution of $\mathcal{R}$ will yield a graph, and that the graph will satisfy the postcondition.

The axiom [nonapp] is excluded from our proof system for total correctness, as $\{\neg\text{App}(\mathcal{R})\}\ \mathcal{R}\ \{\text{false}\}$ does not hold in the sense of total correctness. Suppose that it did. Then $\mathcal{R}$ would never fail on graphs satisfying the precondition. But satisfying $\neg\text{App}(\mathcal{R})$ implies that $\mathcal{R}$ fails on that graph – a contradiction.

6 Example Proofs

In this section, we return to the example graph programs from Section 3, and demonstrate how to prove (weak) total correctness properties using our revised proof calculus.

First, we revisit the program `colouring` of Figure 1. Though the program contains no failure points (since if a rule schema under as-long-as-possible iteration cannot be applied, the execution simply moves on), the iteration operator can introduce non-termination. In [12] we proved that $\vdash_{\text{par}} \{c\}$ `colouring` $\{d \wedge$

$\neg\mathrm{App}(\{\texttt{inc}\})\}$, where c expresses that every node is integer-labelled, and $d \wedge \neg\mathrm{App}(\{\texttt{inc}\})$ expresses that the graph is properly coloured. In Figure 5, we strengthen this to $\vdash_{\mathrm{tot}} \{c\}\ \texttt{colouring}\ \{d \wedge \neg\mathrm{App}(\{\texttt{inc}\})\}$, i.e. if the program is executed on a graph containing only integer-labelled nodes, then a graph will eventually be returned and that graph will be properly coloured. Note that the E-conditions resulting from Pre, implications in instances of [cons], and their justifications, are omitted to preserve space – but can be found in [12].

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{init}, e)\}\ \texttt{init}\ \{e\}}[\text{ruleapp}]}{\vdash_{\mathrm{par}} \{e\}\ \texttt{init}\ \{e\}}[\text{cons}]}{\{e\}\ \texttt{init!}\ \{e \wedge \neg\mathrm{App}(\{\texttt{init}\})\}}[!]_{\mathrm{tot}} \quad X}{\{c\}\ \texttt{init!}\ \{d\}}[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{inc}, d)\}\ \texttt{inc}\ \{d\}}[\text{ruleapp}]}{\vdash_{\mathrm{par}} \{d\}\ \texttt{inc}\ \{d\}}[\text{cons}]}{\{d\}\ \texttt{inc!}\ \{d \wedge \neg\mathrm{App}(\{\texttt{inc}\})\}}[!]_{\mathrm{tot}} \quad Y}}{\vdash_{\mathrm{tot}} \{c\}\ \texttt{init!};\ \texttt{inc!}\ \{d \wedge \neg\mathrm{App}(\{\texttt{inc}\})\}}[\text{comp}]$$

X : `init` is $\#_1$-decreasing under e; Y : `inc` is $\#_2$-decreasing under d

$$\begin{aligned}
c &= \neg\exists(\ \textcircled{\texttt{a}}\ \mid \texttt{not type(a) = int})\\
d &= \forall(\ \textcircled{\texttt{a}}_1, \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{a = b_c and type(b, c) = int}))\\
e &= \forall(\ \textcircled{\texttt{a}}_1, \exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{type(a) = int})\\
&\qquad \vee\exists(\ \textcircled{\texttt{a}}_1 \mid \texttt{a = b_c and type(b, c) = int}))\\
\neg\mathrm{App}(\{\texttt{init}\}) &= \neg\exists(\ \textcircled{\texttt{x}}\ \mid \texttt{type(x) = int})\\
\neg\mathrm{App}(\{\texttt{inc}\}) &= \neg\exists(\ \textcircled{\texttt{x_i}} \xrightarrow{\texttt{k}} \textcircled{\texttt{y_i}}\ \mid \texttt{type(i, k, x, y) = int})
\end{aligned}$$

Fig. 5. A proof tree for the program `colouring` of Figure 1

The key revision in the proof tree is in the two uses of $[!]_{\mathrm{tot}}$, which unlike its partial correctness counterpart requires the definition of termination functions. For `init`, we define $\#_1 \colon \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ to map graphs to the number of their nodes labelled by a single integer. The rule schema is clearly #-decreasing under e, since every application of `init` reduces by one the number of nodes with such labels. The rule schema `inc` however requires a less obvious termination function $\#_2 \colon \mathcal{G}(\mathcal{L}) \to \mathbb{N}$. For a graph $G \in \mathcal{G}(\mathcal{L})$, we define:

$$\#_2 G = \sum_{i=0}^{|V_G|-1} i - \sum_{v \in V_G} \mathrm{tag}(v)$$

where $\mathrm{tag}(v)$ for a node v returns the tag of its label (that is, the second element of a sequence x_i). We show that `inc` is $\#_2$-decreasing under d (rather, under

any E-condition). Observe that if G is a graph with $\mathrm{tag}(v) = 0$ for every node v, then for every derivation $G \Rightarrow^*_{\mathtt{inc}} H$ there is some $0 \leq k \leq |V_H| - 1$ such that k is the largest tag in V_H. We obtain an upper bound for the second summation:

$$\sum_{v \in V_H} \mathrm{tag}(v) < 1 + 2 + \cdots + |V_H| = 1 + 2 + \cdots + |V_G|.$$

Since this summation equals the number of rule schema applications in $G \Rightarrow^*_{\mathtt{inc}} H$, by subtracting it from the upper bound, we have a termination function that decreases towards 0 after every application of `inc`, hence is suitable for our proof tree.

Now, we return to the program `reachable`? of Figure 2, which unlike earlier, can fail on some input graphs (in particular, those graphs omitting the pair of 1- and 2-tagged nodes). We give a proof tree[1] for the program in Figure 7, where the E-conditions are as in Figure 6, showing $\vdash_{\mathrm{tot}} \{c \wedge d\}$ `reachable`? $\{c \wedge d\}$. This means that if the program is executed on a graph that contains only integer-labelled nodes but with one tagged 1 and another tagged 2, then (1) the program is guaranteed to return a graph eventually, and (2) that graph will satisfy the same condition (i.e. an invariant). Again, due to space limitations, we have omitted the implications in instances of [cons] and their justifications. We have also omitted from Figure 6 the E-conditions $\mathrm{Pre}(\mathtt{addlink}, d \wedge e)$ and $\mathrm{Pre}(\mathtt{undo}, d \wedge e)$. Rather, we note that these are very similar to $\mathrm{Pre}(\mathtt{prop}, d \wedge e)$ which is given.

In this proof tree, there are simple suitable termination functions $\#_p, \#_u$. We define the termination function $\#_p \colon \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ (resp. $\#_u$) to return the number of nodes in a graph that are labelled by an integer (resp. number of integer-labelled nodes tagged with a 0). That is, both termination functions exploit that each application of their respective rule schemata reduces the number of remaining matches.

The rule schema `addlink` is the program's only potential failure point, and is addressed in the proof tree by the application of $[\mathrm{ruleset}]_{\mathrm{tot}}$. It must be shown that the precondition at that point implies the applicability of `addlink`. From Figure 6, it is clear that satisfying e is sufficient to deduce the applicability of `addlink`.

7 Soundness and Completeness for Termination

In this section we revise our soundness proof from [12] to account for (weak) total correctness, before showing that any iterating rule schemata set that terminates can be proven to terminate by the rule $[!]_{\mathrm{tot}}$. The proofs use GP's semantic inference rules which are given in [12].

Theorem 1 (Soundness of $\vdash_{\mathtt{wtot}}$). For all graph programs P and E-conditions c, d, we have that $\vdash_{\mathrm{wtot}} \{c\}\ P\ \{d\}$ implies $\models_{\mathrm{wtot}} \{c\}\ P\ \{d\}$. □

[1] For simplicity we use an obvious additional axiom [skip]: $\vdash_{\mathrm{tot}} \{c\}$ `skip` $\{c\}$. We could have used the core proof rules since `skip` is syntax for the rule schema $\emptyset \Rightarrow \emptyset$.

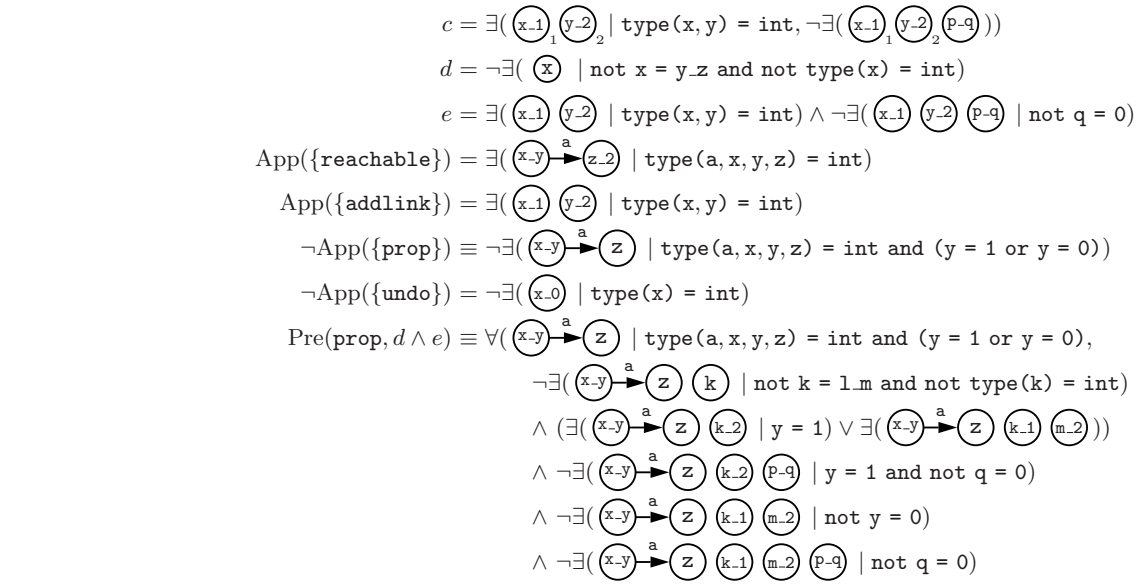


Fig. 6. Partial list of E-conditions for Figure 7

Proof. For all weak total correctness proof rules except $[!]_{\text{tot}}$, this follows from (1) the soundness result for partial correctness in [12], and (2) the semantics of graph programs, from which it is clear that only as-long-as-possible iteration can introduce divergence.

Let $\mathcal{R}$ be a set of (conditional) rule schemata, inv an E-constraint, and $\#$ a termination function. Assume $\vdash_{\text{par}} \{inv\}\ \mathcal{R}\ \{inv\}$. By soundness for partial correctness, we have $\models_{\text{par}} \{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\text{App}(\mathcal{R})\}$. Assume also that $\mathcal{R}$ is $\#$-decreasing under inv. By Definition 5, for all graphs $G, H \in \mathcal{G}(\mathcal{L})$ with $G \models inv$ and $H \models inv$, $G \Rightarrow_{\mathcal{R}} H$ implies $\#G > \#H$. Assume that $\mathcal{R}$ diverges for any such G. Since $\mathcal{R}$ is $\#$-decreasing under inv, every derivation step yields a graph for which $\#$ returns a smaller natural number. Since $\mathcal{R}$ diverges, there are infinitely many derivation steps. But from any natural n, there are only finitely many smaller numbers. A contradiction. It cannot be the case that $\mathcal{R}$ diverges from any such G. Hence $\models_{\text{wtot}} \{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\text{App}(\mathcal{R})\}$. □

Theorem 2 (Soundness of $\vdash_{\text{tot}}$). For all graph programs P and E-conditions c, d, we have that $\vdash_{\text{tot}} \{c\}\ P\ \{d\}$ implies $\models_{\text{tot}} \{c\}\ P\ \{d\}$. □

Proof. For the proof rules [comp], [cons], [if], $[!]_{\text{tot}}$, this follows from (1) the soundness of $\vdash_{\text{wtot}}$ (see Theorem 1), and (2) the semantics of graph programs, from which it is clear that these proof rules are sound in the sense of total correctness. What remains to be shown is the soundness of $[\text{ruleset}]_{\text{tot}}$ in the sense of total correctness.

Let $\mathcal{R}$ denote a set of (conditional) rule schemata and c, d denote E-constraints. Assume that $\vdash_{\text{par}} \{c\}\ r\ \{d\}$ for each $r \in \mathcal{R}$. Then by soundness for partial cor-

Let $P = \texttt{if reachable then skip else addlink}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{prop}, d \wedge e)\}\ \texttt{prop}\ \{d \wedge e\}}[\text{ruleapp}]}{\vdash_{\mathrm{par}} \{d \wedge e\}\ \texttt{prop}\ \{d \wedge e\}}[\text{cons}] \qquad \texttt{prop is } \#_p\text{-decreasing under } d \wedge e}{\{d \wedge e\}\ \texttt{prop!}\ \{d \wedge e \wedge \neg \mathrm{App}(\{\texttt{prop}\})\}}[!]_{\mathrm{tot}}}{\dfrac{\{c \wedge d\}\ \texttt{prop!}\ \{d \wedge e\} \qquad X}{\vdash_{\mathrm{tot}} \{c \wedge d\}\ \texttt{prop!};\ P;\ \texttt{undo!}\ \{c \wedge d\}}[\text{comp}]}[\text{cons}]$$

Subtree X:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{d \wedge e\}\ \texttt{skip}\ \{d \wedge e\}}[\text{skip}]}{\{d \wedge e \wedge \mathrm{App}(\{\texttt{reachable}\})\}\ \texttt{skip}\ \{d \wedge e\}}[\text{cons}] \qquad Y}{\{d \wedge e\}\ P\ \{d \wedge e\}}[\text{if}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{undo}, d \wedge e)\}\ \texttt{undo}\ \{d \wedge e\}}[\text{ruleapp]}}{\vdash_{\mathrm{par}} \{d \wedge e\}\ \texttt{undo}\ \{d \wedge e\}}[\text{cons}] \qquad \texttt{undo is } \#_u\text{-decreasing under } d \wedge e}{\{d \wedge e\}\ \texttt{undo!}\ \{d \wedge e \wedge \neg \mathrm{App}(\{\texttt{undo}\})\}}[!]_{\mathrm{tot}}}{\{d \wedge e\}\ \texttt{undo!}\ \{c \wedge d\}}[\text{cons}]}{\{d \wedge e\}\ P;\ \texttt{undo!}\ \{c \wedge d\}}[\text{comp}]$$

Subtree Y:

$$[\text{ruleset}]_{\mathrm{tot}}\ \dfrac{d \wedge e \wedge \neg \mathrm{App}(\{\texttt{reachable}\}) \Rightarrow \mathrm{App}(\{\texttt{addlink}\}) \qquad \dfrac{\dfrac{\dfrac{}{\{\mathrm{Pre}(\texttt{addlink}, d \wedge e)\}\ \texttt{addlink}\ \{d \wedge e\}}[\text{ruleapp}]}{\vdash_{\mathrm{par}} \{d \wedge e \wedge \neg \mathrm{App}(\{\texttt{reachable}\})\}\ \texttt{addlink}\ \{d \wedge e\}}[\text{cons}]}{\{d \wedge e \wedge \neg \mathrm{App}(\{\texttt{reachable}\})\}\ \texttt{addlink}\ \{d \wedge e\}}$$

Fig. 7. Total correctness proof tree for the program `reachable`? of Figure 2

rectness, we have $\models_{\text{par}} \{c\}\ \mathcal{R}\ \{d\}$. Now assume the validity of $c \Rightarrow \text{App}(\mathcal{R})$. Then if a graph $G \in \mathcal{G}(\mathcal{L})$ satisfies c, by assumption it will satisfy $\text{App}(\mathcal{R})$. By Proposition 7.1 of [12], there is a graph H such that $G \Rightarrow_{\mathcal{R}} H$. Then the semantic rule $[\text{Call}_1]_{\text{SOS}}$ will be applied (and in particular, $[\text{Call}_2]_{\text{SOS}}$ will not be), hence a graph is guaranteed from the execution and failure is avoided. We yield $\models_{\text{tot}} \{c\}\ \mathcal{R}\ \{d\}$. □

Now, we show that every iterating set of rule schemata that terminates can be proven to terminate using $[!]_{\text{tot}}$, by showing that there always exists a termination function for which the rule schemata set is decreasing under its invariant.

Theorem 3 (Completeness of $[!]_{\text{tot}}$ for termination). Let $\mathcal{R}$ be a set of conditional rule schemata and c be an E-constraint such that for every graph G in $\mathcal{G}(\mathcal{L})$, $G \models c$ implies that $\mathcal{R}!$ cannot diverge from G. Then there exists a termination function $\#$ such that $\mathcal{R}$ is $\#$-decreasing under c. □

Proof. Let G be a graph such that $G \models c$. Then there cannot exist an infinite sequence $G \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} G_2 \Rightarrow_{\mathcal{R}} \dots$ as otherwise, by the semantics of GP, there would be an infinite sequence $\langle \mathcal{R}!, G\rangle \rightarrow \langle \mathcal{R}!, G_1\rangle \rightarrow \langle \mathcal{R}!, G_2\rangle \dots$ To define the termination function $\#$, we show that the length of $\Rightarrow_{\mathcal{R}}$-derivations starting from G is bounded. (Note that, in general, a terminating relation need not be bounded.)

We exploit that $\Rightarrow_{\mathcal{R}}$ is closed under isomorphism in the following sense: given graphs M, M', N and N' such that $M \cong M'$ and $N \cong N'$, then $M \Rightarrow_{\mathcal{R}} N$ implies $M' \Rightarrow_{\mathcal{R}} N'$. Hence we can lift $\Rightarrow_{\mathcal{R}}$ to a relation on isomorphism classes of graphs by defining: $[M] \Rightarrow_{\mathcal{R}} [N]$ if $M \Rightarrow_{\mathcal{R}} N$. Then, since $\mathcal{R}$ is finite, for every isomorphism class $[M]$ the set $\{[N] \mid [M] \Rightarrow_{\mathcal{R}} [N]\}$ is finite.

Now, since there is no infinite sequence of $\Rightarrow_{\mathcal{R}}$-steps starting from $[G]$, it follows from König's lemma [9] that the length of $\Rightarrow_{\mathcal{R}}$-derivations starting from $[G]$ is bounded. (In the tree of all derivations starting from $[G]$, all nodes have a finite degree. Hence the tree cannot be infinite, as otherwise it would contain an infinite derivation.) Hence the length of $\Rightarrow_{\mathcal{R}}$-derivations starting from G is bounded as well. In general, given any graph M in $\mathcal{G}(\mathcal{L})$, let $\#M$ be the length of a longest $\Rightarrow_{\mathcal{R}}$-derivation starting from M if $M \models c$, and $\#M = 0$ otherwise. Then if $M, N \models c$ and $M \Rightarrow_{\mathcal{R}} N$, we have $\#M > \#N$. Thus $\mathcal{R}$ is $\#$-decreasing under c. □

8 Conclusion

In this paper we have presented two Hoare calculi which allow one to prove (weak) total correctness. Both proof systems have been shown to be sound. We have shown how to reason about termination via termination functions, and shown that the proof rule for termination is complete in the sense that all terminating loops (having a set of conditional rule schemata as the body) can be proven to be terminating. Finally, we have demonstrated the use of the proof

systems on two non-trivial graph programs, showing how to prove the absence of divergence and failure.

Future work will explore how to implement the proof calculi in an interactive proof system. A first step towards this was made in [13], where translations from E-conditions to many-sorted formulae (and back) were defined, providing a suitable front-end logic for an implemented verification system. Future work will also address the question of whether or not the calculi are (relatively) complete. It would also be worthwhile to integrate a stronger assertion language into the calculi that can express non-local properties.

Acknowledgements. We are grateful to the anonymous referees for their helpful comments.

References

1. Krzysztof R. Apt. Ten years of Hoare's logic: A survey part II: Nondeterminism. *Theor. Comput. Sci.*, 28:83–109, 1984.
2. Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206(7):869–907, 2008.
3. Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositional verification of architectural refactorings. In *Proc. Architecting Dependable Systems VI (WADS 2008)*, volume 5835, pages 308–333. Springer-Verlag, 2009.
4. Simone André da Costa and Leila Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming*, 77(4):480–504, 2012.
5. Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.
6. Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, pages 445–460. Springer-Verlag, 2006.
7. Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505, pages 135–147. Springer-Verlag, 2002.
8. Barbara König and Javier Esparza. Verification of graph transformation systems with context-free specifications. In *Proc. Graph Transformations (ICGT 2010)*, volume 6372, pages 107–122. Springer-Verlag, 2010.
9. Dénes König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1936.
10. Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.
11. Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725, pages 99–122. Springer-Verlag, 2009.
12. Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
13. Christopher M. Poskitt, Detlef Plump, and Annegret Habel. A many-sorted logic for graph programs, 2012. Submitted for publication.

HR* graph conditions between counting monadic second-order and second-order graph formulas

Hendrik Radke

Universität Oldenburg hendrik.radke@uni-oldenburg.de

Abstract. Graph conditions are a means to express structural properties for graph transformation systems and graph programs in a large variety of application areas. With HR* graph conditions, non-local graph properties like "there exists a path of arbitrary length" or "the graph is circle-free" can be expressed. We show, by induction over the structure of formulas and conditions, that (1) any node-counting monadic second-order condition can be expressed by an HR* condition and (2) any HR* condition can be expressed by a second-order graph formula.

1 Introduction

In order to develop trustworthy systems, formal methods play an important role. Visual modeling techniques help to understand complex systems. It is therefore desirable to combine visual modeling with formal verification. The approach taken here is to use graphs and graph transformation rules [5] to model states and state changes, respectively. Structural properties of the system are described by graph conditions.

In [7, 9], nested graph conditions have been discussed as a formalism to describe structural properties in a visual and intuitive way.

Nested graph conditions are expressively equivalent to first-order graph formulas and can express local properties in the sense of Gaifman [6]. However, there are many interesting non-local graph properties like the existence of an arbitrary-length path between two nodes, connectedness or circle-freeness of the graph. Several logics and languages have been developed to express such non-local properties. In [1], a modal logic is described which uses monadic second-order formulas to describe state properties and temporal modalities to describe behavioural properties. A linear temporal logic is used in [11], including the monadic second-order quantification over sets of nodes. In [2], a logic is introduced that can quantify over subobjects of a categorical object. For the category of graphs, this logic is as expressive as monadic second-order logic on graphs. The idea of enhancing nested conditions with variables which are later substituted is also used for the E-conditions in [10].

We showed in [8] that a variant of HR* graph conditions is more expressive than monadic second-order graph formulas. However, an upper bound on the expressiveness of HR* conditions remained an open question. This paper gives both a tighter lower and an upper bound for HR* conditions. We will show

that HR* conditions are at least as strong as node-counting monadic second order formulas, and that every HR* condition can be expressed as a formula in second-order logic on graphs.

The paper is organized as follows: In section 2, we will give the necessary definitions for HR* graph conditions, along with some examples. In section 3, we introduce second-order graph formulas. We then show that HR* conditions can express every node-counting monadic second-order formula in section 4. The construction of a second-order graph formula from an HR* graph condition is then given step-by-step, along with some examples, in section 5. We discuss the results in the concluding section 6.

2 HR* conditions

HR* conditions combine the first-order logical framework and graph morphisms from nested conditions [7] with hyperedge replacement to represent context-free structures of arbitrary size.

Hyperedges relate an arbitrary, fixed number of nodes and are used in HR* conditions as variables, which are later substituted by graphs. We extend the concept of directed, labeled graphs with hyperedge variables, which can be seen as placeholders for graphs to be added later.

Definition 1 (graph with variables). *Let* C *be a fixed, finite alphabet of set and edge labels and* $\mathcal{X}$ *a set of variables with a mapping* $\mathrm{rank}\colon \mathcal{X} \to \mathbb{N}$[1] *defining the rank of each variable.*
A graph with variables *over* C *is a system* $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{Y}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{att}_G, \mathrm{l}_G, \mathrm{ly}_G)$ *consisting of finite sets* V_G, E_G, *and* Y_G *of* nodes *(or* vertices*),* edges*, and* hyperedges*,* source *and* target functions $\mathrm{s}_G, \mathrm{t}_G\colon \mathrm{E}_G \to \mathrm{V}_G$*, an* attachment function $\mathrm{att}_G\colon \mathrm{Y}_G \to \mathrm{V}_G^*$[2]*, and* labeling functions $\mathrm{l}_G\colon \mathrm{V}_G \cup \mathrm{E}_G \to \mathrm{C}$, $\mathrm{ly}\colon \mathrm{Y}_G \to \mathcal{X}$ *such that, for all* $y \in \mathrm{Y}_G$, $|\,\mathrm{att}_G(y)| = \mathrm{rank}(\mathrm{ly}_G(y))$*. We call the set of all graphs with variables* $\mathcal{G}_\mathcal{X}$*. A graph* G *is empty iff* $\mathrm{V}_G = \emptyset$ *and* $\mathrm{Y}_G = \emptyset$*, denoted* $\emptyset$*. Let* $G_{\overline{H}}$ *be the graph* G *with subgraph* H *removed.*

Example 1. Consider the graphs G, H in Figure 1 over the label alphabet $\mathrm{C} = \{A, B, \square\}$ and $\mathcal{X} = \{u, v\}$ where the symbol $\square$ stands for the invisible edge label and is not drawn and $\mathcal{X} = \{u, v\}$ is a set of variables that have rank 4 and 2, respectively. The graph G contains five nodes with the labels A and B, respectively, seven edges with (invisible) label $\square$, and one hyperedge of rank 4 with label u. Additionally, the graph H contains a node, an edge, and a hyperedge of rank 2 with label v.

Nodes are drawn as circles carrying the node label inside, edges are drawn by arrows pointing from the source to the target node and the edge label is placed next to the arrow, and hyperedges are drawn as boxes with attachment

[1] $\mathbb{N}$ denotes the set of natural numbers, including 0.
[2] This also includes hyperedges with zero tentacles.

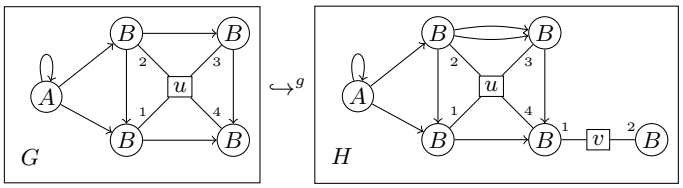


Fig. 1. Graph morphisms with variables

nodes where the i-th tentacle has its number i written next to it and is attached to the i^{th} attachment node and the label of the hyperedge is inscribed in the box. Nodes with the invisible $\square$ label are drawn as points ($\bullet$). For visibility reasons, we may abbreviate hyperedges of rank 2 by writing $\bullet \xrightarrow{x} \bullet$ instead of $\bullet \overset{1}{—} \boxed{x} \overset{2}{—} \bullet$.

Graph morphisms consist of structure-preserving mappings between the sets of nodes, edges and hyperedges of graphs.

Definition 2 (graph morphism with variables). *A* (graph) morphism (with variables) $g\colon G \to H$ *consists of functions* $g_{\mathrm{V}}\colon \mathrm{V}_G \to \mathrm{V}_H$, $g_{\mathrm{E}}\colon \mathrm{E}_G \to \mathrm{E}_H$, *and an injective mapping* $g_{\mathrm{Y}}\colon \mathrm{Y}_G \hookrightarrow \mathrm{Y}_H$ *that preserve sources, targets, attachment nodes and labels, i.e.* $\mathrm{s}_H \circ g_{\mathrm{E}} = g_{\mathrm{V}} \circ \mathrm{s}_G$, $\mathrm{t}_H \circ g_{\mathrm{E}} = g_{\mathrm{V}} \circ \mathrm{t}_G$, $\mathrm{att}_H = g_{\mathrm{V}}^* \circ \mathrm{att}_G$, $\mathrm{l}_H \circ g_{\mathrm{V}} = \mathrm{l}_G$, $\mathrm{l}_H \circ g_{\mathrm{E}} = \mathrm{l}_G$, *and* $\mathrm{ly}_H \circ g_{\mathrm{Y}} = \mathrm{ly}_G$ [3].

The composition $h \circ g$ *of* g *with a graph morphism* $h\colon H \to M$ *consists of the composed functions* $h_{\mathrm{V}} \circ g_{\mathrm{V}}$, $h_{\mathrm{E}} \circ g_{\mathrm{E}}$, *and* $h_{\mathrm{Y}} \circ g_{\mathrm{Y}}$. *A morphism* g *is* injective *(*surjective*) if* g_{V}, g_{E}, *and* g_{Y} *are injective (surjective), and an* isomorphism *if it is both injective and surjective. In the latter case* G *and* H *are* isomorphic*, which is denoted by* $G \cong H$. *An injective graph morphism* $m\colon G \hookrightarrow H$ *is an* inclusion*, written* $G \subseteq H$, *if* $\mathrm{V}_G \subseteq \mathrm{V}_H$, $\mathrm{E}_G \subseteq \mathrm{E}_H$ *and* $\mathrm{Y}_G \subseteq \mathrm{Y}_H$. *For a graph* G, *the* identity $\mathrm{id}_G\colon G \to G$ *consists of the identities* $\mathrm{id}_{G\mathrm{V}}$, $\mathrm{id}_{G\mathrm{E}}$, *and* $\mathrm{id}_{G\mathrm{Y}}$ *on* G_{V}, G_{E}, *and* G_{Y}, *respectively.*

Arbitrary graph morphisms are drawn by the usual arrows "$\to$*"; the use of "*$\hookrightarrow$*" indicates an injective graph morphism. The actual mapping of elements is conveyed by indices, if necessary.*

The hyperedges are replaced by graphs according to a hyperedge replacement system. To describe how the original graph and the graph which replaces a hyperedge are connected, we need to map each tentacle of the hyperedge to a node in the latter graph.

Definition 3 (pointed graph with variables). *A* pointed *graph with variables* $\langle G, \mathrm{pin}_G \rangle$ *is a graph with variables* G *together with a sequence* $\mathrm{pin}_G = v_1 \dots v_n$ *of pairwise disjoint nodes from* G. *We write* $\mathrm{rank}(\langle G, \mathrm{pin}_G \rangle)$ *for the number* n *of nodes. For* $x \in \mathcal{X}$ *with* $\mathrm{rank}(x) = n$, $x^\bullet$ *denotes the pointed*

[3] For a mapping $g\colon A \to B$, the free symbolwise extension $g^*\colon A^* \to B^*$ is defined by $g^*(a_1 \dots a_k) = g(a_1) \dots g(a_k)$ for all $k \in \mathbb{N}$ and $a_i \in A$ $(i = 1, \dots, k)$.

graph with the nodes $v_1, \ldots, v_n$, one hyperedge attached to $v_1 \ldots v_n$, and sequence $v_1 \ldots v_n$. $\mathrm{Pin}(G)$ denotes the set of pinpoints of $\langle G, \mathrm{pin}_G \rangle$.

Definition 4 (hyperedge replacement system). *A* hyperedge replacement (HR) system *$\mathcal{R}$ is a finite set of replacement pairs of the form x/R where x is a variable and R a pointed graph with* $\mathrm{rank}(x) = \mathrm{rank}(R)$.

Given a graph G, the application *of the replacement pair $x/R \in \mathcal{R}$ to a hyperedge y with label x proceeds in two steps (see Figure 2): For a set X, let $G - X$ be the graph G with all elements in X removed, and for a graph H, let $G + H$ be the disjoint union of G and H.*

1. *Remove the hyperedge y from G, yielding the graph $G - \{y\}$.*
2. *Construct the disjoint union $(G - \{y\}) + R$ and fuse the i^{th} node in $\mathrm{att}_G(y)$ with the i^{th} attachment point of R, for $i = 1, \ldots, \mathrm{rank}(y)$, yielding the graph H.*

G directly derives *H by $x/R \in \mathcal{R}$ applied to y, denoted by $G \Rightarrow_{x/R,y} H$ or $G \Rightarrow_{\mathcal{R}} H$. A sequence of direct derivations $G \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} H$ is called a* derivation *from G to H, denoted by $G \Rightarrow^*_{\mathcal{R}} H$. For every variable x, $\mathcal{R}(x) = \{G \in \mathcal{G}_{\mathcal{X}} \mid x^\bullet \Rightarrow^*_{\mathcal{R}} G\}$ denotes the set of all graphs derivable from $x^\bullet$ by $\mathcal{R}$.*

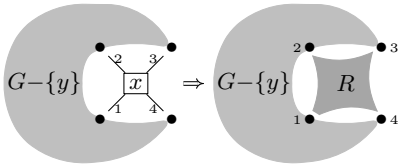
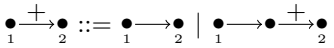


Fig. 2. Application of replacement pair x/R.

Example 2. The hyperedge replacement system $\mathcal{R}$ with the rules given in Backus-Naur form



generates the set of all directed paths from node 1 to node 2.

In HR* conditions, we simultaneously substitute all hyperedges by graphs, which are generated according to a hyperedge replacement system.

Definition 5 (substitution). *A* substitution *induced by a hyperedge replacement system $\mathcal{R}$ is a mapping $\sigma \colon \mathcal{X} \to \mathcal{G}$ with $\sigma(x) \in \mathcal{R}(x)$ for all $x \in \mathcal{X}$. The set of all substitutions induced by $\mathcal{R}$ is denoted by Σ. The substitution of a hyperedge y with label x in a graph G by $\sigma(x)$ is obtained from G by removing y from G, constructing the disjoint union $(G - \{y\} + \sigma(x)$ and then fusing the i^{th} node in*

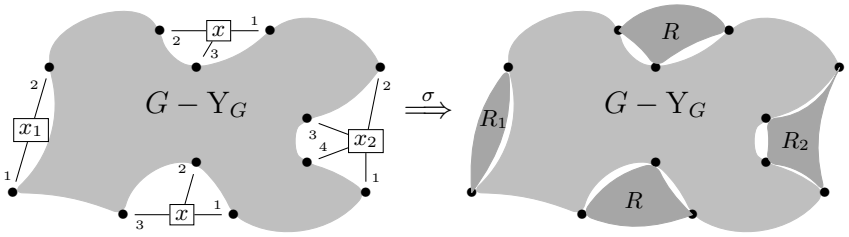


Fig. 3. Substitution of hyperedges.

$\mathrm{att}_G(y)$ *with the* i^{th} *point of* $\sigma(x)$ *for* $i \in [\mathrm{rank}(y)]$[4]*. Application of* σ *to a graph* G*, denoted* $G \Rightarrow G^\sigma$*, is obtained by simultaneous substitution of all hyperedges in* $y \in Y_G$ *by* $\sigma(\mathrm{ly}_G(y))$ *(Figure 3).*

With the preliminaries done, we can now define HR* conditions. They allow one to use variables for structures of arbitrary size, and to "peek into" such variables and formulate properties of the graphs that the variable is substituted by.

Definition 6 (HR* graph condition). *A* HR* (graph) conditions (over $\mathcal{R}$) *consists of a condition with variables and a replacement system* $\mathcal{R}$*. Conditions are inductively defined as follows.*

1. *For a graph* P, `true` *is a condition over* P*.*
2. *For an injective morphism* $a\colon P \hookrightarrow C$ *and a condition* c *over* C, $\exists(a, c)$ *is a condition over* P*.*
3. *For graphs* P, C *and a condition* c *over* C, $\exists(P \sqsupseteq C, c)$ *is a condition over* P*.*
4. *For an index set* J *and conditions* $(c_j)_{j\in J}$ *over* P, $\neg c_1$ *and* $\wedge_{j\in J} c_j$ *are conditions over* P [5]*.*

HR* *conditions* c *over* $\mathcal{R}$ *are denoted by* $\langle c, \mathcal{R}\rangle$*, or* c *if* $\mathcal{R}$ *is clear from the context. A* HR* *condition is* finite *if every index set* J *in the condition is finite; we will assume finite conditions in the following if not explicitly stated otherwise.*

The following abbreviations are used: $\exists a$ abbreviates $\exists(a, \texttt{true})$, $\forall(\_, c)$ abbreviates $\neg\exists(\_, \neg c)$, `false` abbreviates $\neg$`true`, and $\vee_{j\in J} c_j$ abbreviates $\neg \wedge_{j\in J} \neg c_j$. The domain of a morphism may be omitted if no confusion arises: $\exists C$ can replace $\exists(P \hookrightarrow C)$ in this case.

Example 3. The following HR* condition intuitively expresses "There exists a path from the image of node 1 to the image of node 2".

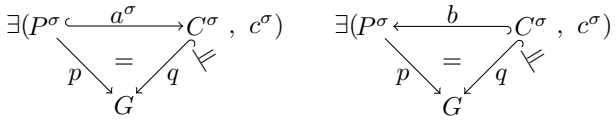
$$\exists(\bullet_1 \xrightarrow{+} \bullet_2) \quad \text{with} \quad \bullet_1 \xrightarrow{+} \bullet_2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \longrightarrow \bullet \xrightarrow{+} \bullet_2$$

[4] $[k]$ denotes the set $\{1, \ldots, k\}$ of natural numbers up to k.
[5] Usually, J is a set of natural numbers from 1 to some number k.

We now give the formal semantics for HR* conditions.

Definition 7 (satisfaction of HR* conditions). *Given a hyperedge replacement system $\mathcal{R}$, a morphism $p\colon P^\sigma \hookrightarrow G$, the* satisfaction *of a condition by a substitution $\sigma \in \Sigma$ is inductively defined as follows.*

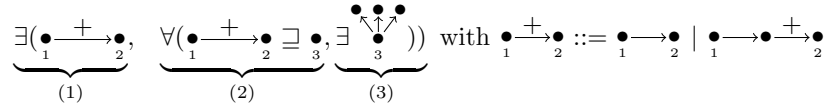
(1) p satisfies `true`.
(2) p satisfies $\exists(a, c)$ for a morphism $a\colon P \hookrightarrow C$ if there is an injective morphism $q\colon C^\sigma \hookrightarrow G$ such that $q \circ a^\sigma = p$ and q satisfies c^σ[6] (left figure).



(3) p satisfies $\exists(P \sqsupseteq C, c)$ if there are an inclusion $b\colon C^\sigma \hookrightarrow P^\sigma$ and an injective morphism $q\colon C^\sigma \hookrightarrow G$ such that $p \circ b = q$ and q satisfies c by σ (right figure).
(4) p satisfies $\neg c$ if p does not satisfy c. p satisfies $\wedge_{i\in I} c_i$ if p satisfies all c_i $(i \in I)$.

A graph G satisfies *a condition c over $\emptyset$ if the morphism $\emptyset \hookrightarrow G$ satisfies c. We write $G \models_\sigma c$ to denote that a graph G satisfies c by σ and $G \models c$ if there is a $\sigma \in \Sigma$ such that $G \models_\sigma c$.*

Example 4. The following example shows a HR* condition intuitively expressing "There is a path from a node to another, and all nodes on this path have at least three outgoing edges to different nodes".



In subformula (1), the existence of the path is established. Part (2) quantifies over every node that is contained in the path, while part (3) ensures that each such node has three outgoing edges to different nodes.

In logical formulas, distinct variables do not necessarily mean distinct objects: one has to explicitly state that two variables x, y stand for distinct objects with a formula $\neg x \doteq y$. Nodes and edges in HR* conditions, on the other hand, are distinct by default. This can also be done with a variant on the semantics of HR* conditions, $\mathcal{A}$-satisfaction.

Definition 8 ($\mathcal{A}$-satisfiability). *A morphism $p\colon P \to G$ $\mathcal{A}$-*satisfies *a HR* formula c, short $p \models_\mathcal{A} c$, as per definition 7, except for $c = \exists(a\colon P \to C, c)$: $p \models \exists(a, c)$ if there exists a (possibly non-injective) morphism $q\colon C \to G$ such that $p = q \circ a$ and $q \models c$.*

[6] $a^\sigma\colon P^\sigma \hookrightarrow C^\sigma$ is the morphism induced by σ from a. c^σ is σ applied to c in a similar fashion.

The consequence of this definition is that nodes and edges in $\mathcal{A}$-satisfiable HR^* conditions are no longer disjoint by default, but may be unified.

Lemma 1 ($\mathcal{A}$-satisfaction). *For every* HR^* *condition* c*, there is a* HR^* *condition* $\mathrm{CondA}(c)$ *such that for every graph* G*,*

$$G \models c \iff G \models_{\mathcal{A}} \mathrm{CondA}(c).$$

The construction of CondA is quite straightforward: To any HR^* condition $\exists(P \hookrightarrow C, c)$, a subcondition is added that forbids the unification of distinct nodes and edges in C.

Construction 1. For a condition over P, CondA is inductively defined:

1. $\mathrm{CondA}(\texttt{true}) = \texttt{true}$.
2. $\mathrm{CondA}(\exists(P{\to}C, c)) = \exists(P{\to}C, c \wedge \forall(C \sqsupseteq \bullet\ \bullet, \nexists(\bullet\to\bullet)) \wedge \forall(C \sqsupseteq \bullet\!\downarrow\!\bullet, \nexists(\bullet\!\downarrow\!\bullet\to\bullet\ \bullet)))$.
3. $\mathrm{CondA}(\exists(P \sqsupseteq C, c)) = \exists(P \sqsupseteq C, c)$.
4. $\mathrm{CondA}(\neg c) = \neg\,\mathrm{CondA}(c)$ and $\mathrm{CondA}(\bigwedge_{i\in I} c_i) = \bigwedge_{i\in I} \mathrm{CondA}(c_i)$.

Proof. For conditions $\texttt{true}$, $\exists(P \sqsupseteq C, c)$, $\neg c$ and $\bigwedge_{i\in I} c_i$, the proof is trivial as CondA does not change the condition. For a condition $\exists(P \hookrightarrow C, c)$ and graph $G \in \mathcal{G}$, we can directly transform the statement that two objects d, d' must be injective into a condition that fits our construction:

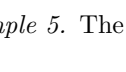
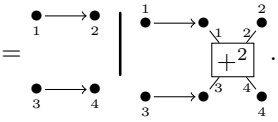
$$\begin{aligned}
& G \models \exists(P \hookrightarrow C, c) \\
\Leftrightarrow\ & \exists\sigma, q\colon C^\sigma \hookrightarrow G.p = q \circ a^\sigma \wedge q \models c^\sigma && \text{(Def. 7)} \\
\Leftrightarrow\ & \exists\sigma, q\colon C^\sigma \to G.p = q \circ a^\sigma \wedge q \models c^\sigma \wedge \\
& \quad \nexists d, d' \in \mathrm{D}_C.d \neq d' \wedge q(d) = q(d') && (q \text{ injective}) \\
\Leftrightarrow\ & \exists\sigma, q\colon C^\sigma \to G.p = q \circ a^\sigma \wedge q \models c^\sigma \wedge \\
& \quad \nexists C' \subseteq C.\nexists d, d' \in \mathrm{D}_{C'}.d \neq d' \wedge q(d) = q(d') \\
\Leftrightarrow\ & \exists\sigma, q\colon C^\sigma \to G.p = q \circ a^\sigma \wedge q \models c^\sigma \wedge \\
& \quad q \models \nexists(\bullet \to \bullet) \wedge \forall(C \sqsupseteq \bullet\!\downarrow\!\bullet, \nexists(\bullet\!\downarrow\!\bullet \to \bullet\ \bullet)) && \text{(Constr. 1)} \\
\Leftrightarrow\ & G \models_{\mathcal{A}} \mathrm{CondA}(c). && \text{(Def. 8)}
\end{aligned}$$

□

Furthermore, for $\mathcal{A}$-satisfiability, substitution of hyperedges (i.e. all edges with the same label are replaced by isomorphic graphs) is equivalent to replacement of hyperedges (i.e. edges with the same label may be replaced by different graphs). It is easy to see that, for every HR^* condition using replacement, an equivalent HR^* condition can be constructed using substitution, simply by giving each hyperedge a unique label and cloning the rules.

On the other hand, it is possible to simulate substitution with replacement, using the following construction idea: For every HR^* condition $\exists(P + \boxed{x} \to P + \boxed{x}\boxed{x}, c)$ (the x-labeled hyperedges may have arbitrary many tentacles), we add $\exists(P + \boxed{x} \to P + \boxed{x}\boxed{x}, \exists(P + \boxed{x}\boxed{x} \sqsupseteq P + \boxed{x^2} \wedge c)$, where x^2 has both hyperedges combined into a single one, and modify the grammar to perform

rules in parallel. For conditions $\exists(P + \boxed{x}, c_1) \wedge \exists(P + \boxed{x}, c_2)$, we modify to $\exists(P + \boxed{x^2}, \exists P + \boxed{x^2} \sqsupseteq P + \boxed{x}, c_1) \wedge \exists P + \boxed{x^2} \sqsupseteq P + \boxed{x}, c_2)$ and modify the grammar as in the first case.

Example 5. The HR* condition with $\mathcal{A}$ substitution semantics $\exists($  $, \nexists($ 
with ::= is equivalent to the HR* condition with $\mathcal{A}$ replacement semantics
$\exists($$, \nexists($$, \exists$  $\sqsupseteq$  with  ::=  .

Remark 1. The idea of enhancing nested conditions with variables is also used in [10] for E-conditions. In contrast to HR* conditions, the variables in E-conditions are not substituted by graphs, but by labels or attributes, so E-conditions cannot express non-local conditions, but can work with infinite label alphabets (e.g. natural numbers) and perform calculations on attributes.

3 Graph formulas

A classic approach to express properties of a graph is to use logical formulas over graphs. The expressiveness of such formulas depends on the underlying logic. We begin with the definition of second-order graph formulas, following [4]. Second-order formulas can quantify over individual objects in the underlying universe, as well as over arbitrary relations over the underlying universe, allowing one to formulate many interesting graph properties. For a comparative overview on the power of several graph logics, see [3]; our definition of second-order logic is equivalent to that in [3], except that we also consider node and edge labels.

Definition 9 (second-order graph formulas). *Let C be a set of labels, $\mathcal{V}_1$ be a (denumerable) set of* individual *(or* first-order*) variables $x_0, x_1, \ldots$ and $\mathcal{V}_2$ a (denumerable) set of* relational *(or* second-order*) variables $X_0, X_1, \ldots$, together with a function* rank$\colon \mathcal{V}_2 \to \mathbb{N}-\{0\}$ *that maps to each variable in $\mathcal{V}_2$ a positive natural number, its* rank*. We let $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$ be the set of all variables.*

Second-order graph formulas*, short* SO formulas*, are defined inductively:* $\mathrm{inc}(x, y, z)$*,* $\mathrm{lab}_b(x)$ *and $x \doteq y$ are SO graph formulas for individual variables $x, y, z \in \mathcal{V}_1$ and labels $b \in C$. For any variable $x \in \mathcal{V}_1$ and SO formula F, $\exists x.F$ is an SO formula. Also, for any variable $X \in \mathcal{V}_2$ with* rank$(X) = k$ *and SO formulas $F_1, \ldots, F_k$, $X(F_1, \ldots, F_k)$ is an SO formula. Finally, Boolean expressions over SO formulas c, d are SO formulas:* `true`*, $\neg F$, $F_1 \wedge F_2$.*

For a non-empty graph G, let $\mathrm{D}_G^{\times}$ be the set of all relations over $\mathrm{D}_G = \mathrm{V}_G \cup \mathrm{E}_G$. The semantics $G[\![F]\!](\theta)$ of a SO formula F under assignment $\theta\colon \mathcal{V} \to \mathrm{D}_G \cup \mathrm{D}_G^{\times}$ is inductively defined as follows:

1. $G[\![\mathrm{lab}_b(x)]\!](\theta)$=`true` *iff* $\theta(x)=\mathrm{lv}_G(b)$ *or* $\theta(x)=\mathrm{le}_G(b)$,
   $G[\![\mathrm{inc}(e,x,y)]\!](\theta)$ = `true` *iff* $\theta(e) \in \mathrm{E}_G$, $\mathrm{s}_G(\theta(e)) = \theta(x)$, *and* $\mathrm{t}_G(\theta(e)) = \theta(y)$, *and*
   $G[\![x \doteq y]\!](\theta)$ = `true` *iff* $\theta(x) = \theta(y)$.
2. $G[\![\mathtt{true}]\!](\theta)$ = `true`, $G[\![\neg F]\!](\theta) = \neg G[\![F]\!](\theta)$, $G[\![F \wedge F']\!](\theta) = G[\![F]\!](\theta) \wedge G[\![F']\!](\theta)$, *and*
   $G[\![\exists x.F]\!](\theta)$ = `true` *iff* $G[\![F]\!](\theta\{x/d\})$ = `true` *for some* $d \in \mathrm{D}_G$, *where* $\theta\{x/d\}$ *is the modified assignment with* $\theta\{x/d\}(y) = d$ *if* $x = y$ *and* $\theta\{x/d\}(y) = \theta(y)$ *otherwise.*
3. $G[\![\exists X.F]\!](\theta)$ = `true` *iff* $G[\![F]\!](\theta\{X/D\})$ = `true` *for some* $d \in \mathrm{D}_G^{\times}$.
   $G[\![X(F_1,\ldots,F_k)]\!](\theta)$ = `true` *iff* $(G[\![F_1]\!](\theta),\ldots,G[\![F_k]\!](\theta)) \in \theta(X)$.
4. $G[\![\neg F]\!](\theta) = \neg G[\![F]\!](\theta)$ *and* $G[\![F \wedge F']\!](\theta) = G[\![F]\!](\theta) \wedge G[\![F']\!](\theta)$.

A non-empty graph G satisfies *a SO formula* F, *denoted by* $G \models F$, *iff, for all assignments* $\theta\colon \mathcal{V} \to \mathrm{D}_G \cup \mathrm{D}_G^{\times}$, $G[\![F]\!](\theta)$ = `true`.

Example 6. The SO formula below is true for every graph which has a non-trivial automorphism[7]:

$$\exists X.[\beta_{\mathrm{inj}}(X) \wedge \beta_{\mathrm{total}}(X) \wedge \beta_{\mathrm{surj}}(X) \wedge \beta_{\mathrm{ntriv}}(X) \wedge \beta_{\mathrm{predg}}(X)]$$

where the subformulas are defined as follows:

- $\beta_{\mathrm{inj}}(X) = \forall x,y,z.(X(x,y) \wedge X(x,z)) \Rightarrow y \doteq z \wedge (X(x,z) \wedge X(y,z)) \Rightarrow x \doteq y$ expresses that relation X is injective,
- $\beta_{\mathrm{total}}(X) = \forall x \exists y.X(x,y)$ expresses that X is total,
- $\beta_{\mathrm{surj}}(X) = \forall x \exists y.X(y,x)$ expresses that X is surjective,
- $\beta_{\mathrm{ntriv}}(X) = \exists x,y.x \neq y \wedge X(x,y)$ expresses that X is non-trivial,
- $\beta_{\mathrm{predg}}(X) = \forall e,x,y,e,x',y'.(\mathrm{inc}(e,x,y) \wedge (X(e,e') \wedge X(x,x') \wedge X(y,y')) \Rightarrow \mathrm{inc}(e',x',y')$ expresses that X preserves edges, i.e. for every pair of nodes x,y connected by an edge and related to nodes x',y' by relation X, x' and y' are connected by an edge.

Counting monadic second-order graph formulas are a subclass of second-order graph formulas and an extension of monadic second-order graph formulas [3]. Like monadic second-order graph formulas, they allow quantification over individual nodes and edges as well as quantification over unary relations, i.e. sets of nodes and edges. Furthermore, they have a special quantifier that allows one to count modulo natural numbers.

Definition 10 (counting monadic second-order graph formulas). *A* counting monadic second-order graph formula*, short* CMSO formula*, are inductively defined as follows. Every SO formula where every relational variable* X *has* $\mathrm{rank}(X) = 1$ *is a CMSO formula, and for every natural number* $m \in \mathbb{N}$ *and every CMSO formula* F, $\exists^{(m)}x.F(x)$ *is a CMSO formula. For a non-empty graph* G,

$$G[\![\exists^{(m)}x.F(x)]\!](\theta) = \mathtt{true} \text{ iff } |\{u \in V_G \cup E_G : G[\![F(u)]\!](\theta)\}| \equiv 0 \pmod m.$$

[7] i.e. an automorphism which is not the identity

A CMSO formula is a node-CMSO formula *if counting is only allowed over nodes, i.e. every subformula* $\exists^{(m)}x.F$ *is equivalent to* $\exists^{(m)}x.\,\mathrm{node}(x) \wedge F'$*, where* $\mathrm{node}(x) = \nexists y, z.\,\mathrm{inc}(x, y, z)$ *states that* x *is a node. A CMSO formula is a* monadic second-order formula*, short* MSO formula*, if it contains no subformulas of the form* $\exists^{(m)}x.F$*.*

Example 7. The node-CMSO formula $\exists^{(2)}x.\,\mathrm{node}(x)$ expresses "The graph has an even number of nodes".

4 Expressing node-CMSO formulas with HR* conditions

In [8], a variant of HR* conditions was introduced and shown to be at least as strong as MSO formulas. We now go one step further and show that HR* formulas can also express arbitrary node-CMSO formulas.

Theorem 1 (node-CMSO formulas to HR* conditions). *For every node-CMSO formula* F*, there is a* HR* *graph condition* $\mathrm{Cond}(F)$ *such that for all graphs* $G \in \mathcal{G}$*,* $G \models F$ *iff* $G \models \mathrm{Cond}(F)$*.*

We use hyperedge replacement to count the nodes: It is easy to construct a grammar which generates all discrete graphs (i.e. with no edges) with $k * m$ nodes, where m is a fixed number and $k \in \mathbb{N}$ is variable. For all nodes inside the generated subgraph, the property F to be counted is checked. Also, F must not hold for any node outside of the generated subgraph.

Construction 2. For a graph P and a formula F, $\mathrm{Cond}(P, F)$ is defined as follows. For any MSO formula, Cond is defined as in [8]. Otherwise, i.e. for formulas of the form $\exists^{(m)}v.F$,

$$\mathrm{Cond}(P, \exists^{(m)}v.F) = \exists(\boxed{Y}, \forall(\boxed{Y} \sqsupseteq \underset{v}{\bullet}, \mathrm{Cond}(F(v))) \wedge \nexists(\boxed{Y}\,\underset{v}{\bullet}, \mathrm{Cond}(F(v))))$$
with $\boxed{Y} ::= \emptyset \mid \boxed{Y}\; D_m$, where D_m is a discrete graph with m nodes.

Example 8. Take as an example the node-CMSO formula expressing "There is an even number of nodes": $\exists^{(2)}x.\mathrm{node}(x)$. Using the construction above, this is transformed into $\exists(\boxed{Y}, \forall(\boxed{Y} \sqsupseteq \underset{1}{\bullet}, \exists(\underset{1}{\bullet})) \wedge \nexists(\boxed{Y}\,\underset{2}{\bullet}, \exists(\underset{2}{\bullet})))$ with $\boxed{Y} ::= \emptyset \mid \boxed{Y}\;\begin{smallmatrix}\bullet\\\bullet\end{smallmatrix}$

Simplification of the HR* condition yields $\exists(\boxed{Y}, \nexists(\boxed{Y}\,\underset{2}{\bullet}))$ with $\boxed{Y} ::= \emptyset \mid \boxed{Y}\;\begin{smallmatrix}\bullet\\\bullet\end{smallmatrix}$.

Proof. For MSO formulas, see the proof in [8]. For formulas $\exists^{(m)}x.\phi(x)$ and every graph G, assume that $G \models \phi(x) \iff G \models \mathrm{Cond}(\phi(x))$ and let $p\colon \emptyset \hookrightarrow G$.

By the definition of HR* satisfaction (Def. 7) and construction 2,

$G \models \mathrm{Cond}(\exists^{(m)}x.\phi(x)) \Leftrightarrow p \models \mathrm{Cond}(\exists^{(m)}x.\phi(x))$
$\Leftrightarrow p \models \exists(\emptyset \hookrightarrow \boxed{Y}, \forall(\boxed{Y} \sqsupseteq \underset{x}{\bullet}, \mathrm{Cond}(\phi(x))) \wedge \nexists(\boxed{Y} \hookrightarrow \boxed{Y}\,\underset{x}{\bullet}, \mathrm{Cond}(\phi(x)))).$
Performing the substitution of hyperedge $\boxed{Y}$ yields
$\Leftrightarrow \exists n \in \mathbb{N}.p \models \exists(\emptyset \hookrightarrow D_{n*m}, \forall(D_{n*m} \sqsupseteq \underset{x}{\bullet}, \mathrm{Cond}(\phi(x)))$
$\wedge \nexists(D_{n*m} \hookrightarrow D_{n*m}\,\underset{x}{\bullet}, \mathrm{Cond}(\phi(x)))).$

We use the semantics of HR* conditions again to get

$$\begin{aligned}
&\Leftrightarrow \exists n \in \mathbb{N}.\exists D_{n*m} \overset{q_a}{\hookrightarrow} G.p = q_a \circ a \wedge \\
&\quad \forall \emptyset \overset{q_b}{\hookrightarrow} \underset{x}{\bullet}.q_b(\underset{1}{\bullet}) \subseteq q_a(D_{n*m}) \wedge q_b \models \mathrm{Cond}(\phi(x))^\sigma \wedge \\
&\quad \nexists (D_{n*m} + \underset{x}{\bullet} \overset{q_c}{\hookrightarrow} G.q_a = q_c \circ c \wedge q_c \models \mathrm{Cond}(\phi(x))^\sigma) \\
&\text{and, by the definition of graph morphisms,} \\
&\exists n \in \mathbb{N}.\exists D_{n*m} \subseteq G.\forall (\underset{1}{\bullet} \subseteq D_{n*m}.\underset{x}{\bullet} \models \mathrm{Cond}(\phi(x))^\sigma) \\
&\qquad\qquad \wedge \nexists (D_{n*m} + \underset{x}{\bullet} \subseteq G.\underset{x}{\bullet} \models \mathrm{Cond}(\phi(x))^\sigma). \\
&\text{Using simple arithmetics and set theory, it is easy to see that} \\
&\Leftrightarrow \exists n \in \mathbb{N}.|\{\underset{1}{\bullet} \subseteq V_G \mid \underset{x}{\bullet} \models \mathrm{Cond}(\phi(x))^\sigma\}| \leq n * m \\
&\qquad \wedge \neg|\{\underset{1}{\bullet} \subseteq V_G \mid \underset{x}{\bullet} \models \mathrm{Cond}(\phi(x))^\sigma\}| \leq n * m + 1 \\
&\Leftrightarrow \exists n \in \mathbb{N}.|\{v \in V_G : G \models \mathrm{Cond}(\phi(v))\}| = n * m \\
&\Leftrightarrow \exists n \in \mathbb{N}.|\{v \in V_G : G \models \phi(v)\}| = n * m. \\
&\text{Using the initial assumption and Definition 10, we get} \\
&\Leftrightarrow |\{v \in V_G : G \models \phi(v)\}| \equiv 0 \pmod{m} \\
&\Leftrightarrow G[\![\exists^{(m)} x.F(x)]\!](\theta) = \texttt{true} \\
&\Leftrightarrow G \models \exists^{(m)} x.\phi(x).
\end{aligned}$$

This completes the proof. □

5 Expressing HR* conditions with SO formulas

With a lower bound for the expressiveness of HR* conditions established, we now turn to the upper bound and show that every HR* condition can be expressed as a SO formula. The main difficulty here lies in the representation of the replacement process within the formula. The transformation is made somewhat easier by using a slightly changed semantics for HR* conditions. We use replacement instead of substitution and $\mathcal{A}$-satisfiability.

Since every HR* condition can be transformed into an equivalent $\mathcal{A}$-satisfiable one, we can prove that every HR* condition can be expressed as a SO graph formula, which we will now do step by step.

Theorem 2 (HR* conditions to SO formulas). *For every* HR* *graph condition c, there is a second-order graph formula* $\mathrm{SO}(c)$ *such that for all graphs $G \in \mathcal{G}$,*

$$G \models c \iff G \models \mathrm{SO}(c).$$

We use several helper constructions that we will present and prove individually later. $\mathrm{SOgra}(G, F)$ is used to represent a graph G as a formula, where F is some nested subformula. SOsys represents the replacement system of the HR* condition. Finally, $\mathrm{SOset}(G, X)$ is used to collect all nodes and edges in graph G^σ (i.e. after replacement) inside a set variable X. This is needed to check whether there is an inclusion $C^{\sigma'} \hookrightarrow P^\sigma$ for a HR* condition $\exists(P \sqsupseteq C, c)$.

Construction 3. Without loss of generality, $P \hookrightarrow C$ is an inclusion. For a condition c with HR system $\mathcal{R}$, we let $\mathrm{SO}(\langle c, \mathcal{R}\rangle) = \mathrm{SOsys}(\mathcal{R}) \wedge \mathrm{SO}(c)$ and define

1. $\mathrm{SO}(\mathtt{true}) = \mathtt{true}$.
2. $\mathrm{SO}(\exists(P \hookrightarrow C, c)) = \mathrm{SOgra}(C{-}P, \mathrm{SO}(c))$.
3. $\mathrm{SO}(\exists(P \sqsupseteq C, c)) = \mathrm{SOgra}(C, \exists X_P, X_C.\, \mathrm{SOset}(P, X_P) \wedge \mathrm{SOset}(C, X_C) \wedge X_C \subseteq X_P \wedge \mathrm{SO}(c))$, where X_P, X_C are fresh second-order variables of rank 1 (i.e. set variables) and the relation $\subseteq$ is constructed in SO logic as usual: $X_C \subseteq X_P = \forall x.x \in X_C \Rightarrow \exists y \in X_P.x \doteq y$.
4. $\mathrm{SO}(\neg c) = \neg\,\mathrm{SO}(c)$ and $\mathrm{SO}(\bigwedge_{i\in I} c_i) = \bigwedge_{i\in I} \mathrm{SO}(c_i)$.

The transformation SOgra represents a graph with variables as a formula. The construction is quite straightforward: we state the existence of every node, edge and hyperedge and then express each label and the attachment of edges and hyperedges to the nodes.

Lemma 2. *For graphs $R \in \mathcal{G}_\mathcal{X}$ and $G \in \mathcal{G}$,*

$$G \models_\mathcal{A} \exists(\emptyset \hookrightarrow R{-}\mathrm{Y}_R) \iff G \models \mathrm{SOgra}(R{-}\mathrm{Y}_R, \mathtt{true}).$$

Construction 4. For a set A and SO formula F, let $\exists F$ be the existential closure of F and $\dot\exists F = \exists F \wedge \bigwedge_{a\neq b}^{a,b\in A}(\neg a \doteq b)$ be the existential closure of F with disjointness check. Define the universal closure $\forall F$ analogously. For a graph with variables G and a SO formula F, we define

$$\begin{aligned}
\mathrm{SOnod}(G, F) &= \exists \textstyle\bigwedge_{v\in \mathrm{V}_R} \mathrm{lab}_{\mathrm{l}_G(v)}(v) \wedge F \\
\mathrm{SOedg}(G) &= \exists \textstyle\bigwedge_{e\in \mathrm{E}_R} \mathrm{lab}_{\mathrm{l}_G(e)}(e) \wedge \mathrm{inc}(e, \mathrm{s}_G(e), \mathrm{t}_G(e)) \\
\mathrm{SOhyp}(G) &= \dot\exists \textstyle\bigwedge_{y\in \mathrm{Y}_R} (\exists d.\, \mathrm{ly}_G(y)(\mathrm{att}_G(y)_{1,\ldots,k}, d)) \\
\mathrm{SOgra}(G, F) &= \mathrm{SOnod}(G, \mathrm{SOedg}(G) \wedge \mathrm{SOhyp}(G) \wedge F)
\end{aligned}$$

Proof. Assume $G \models_\mathcal{A} \exists(\emptyset \hookrightarrow^a R_{\overline{\mathrm{Y}}_R})$.
By the semantics of HR^* conditions, for $p\colon \emptyset \to G$, this is equivalent to
$\Leftrightarrow p \models_\mathcal{A} \exists(\emptyset \to^a R_{\overline{\mathrm{Y}}_R})$
$\Leftrightarrow \exists q\colon R_{\overline{\mathrm{Y}}_R} \to G.p = q \circ a \wedge q \models_\mathcal{A} \mathtt{true}$.
By the definition of morphisms, this equals
$\Leftrightarrow \exists q\colon R_{\overline{\mathrm{Y}}_R} \to G.\forall o \in \mathrm{D}_R.p(o) = q(a(o))$
$\Leftrightarrow \exists R' \in \mathcal{G}.\exists q'\colon R_{\overline{\mathrm{Y}}_R} \to R' \wedge R' \subseteq G$
which can be expressed as a SO formula
$\Leftrightarrow \exists R' \in \mathcal{G}.\exists_{v\in \mathrm{V}'_R}.(\bigwedge_{v\in \mathrm{V}'_R}(\mathrm{lab}_{\mathrm{l}(v)}(v)) \wedge \exists(\bigwedge_{e\in \mathrm{E}'_R}(\mathrm{lab}_{\mathrm{l}(e)}(e)) \wedge \mathrm{inc}(e, \mathrm{s}(e), \mathrm{t}(e))))$
$\Leftrightarrow \exists R' \in \mathcal{G}.\,\mathrm{SOnod}(R', \mathrm{SOedg}(R') \wedge \mathrm{SOhyp}(R'))$
which equals the definition of SOgra:
$\Leftrightarrow G \models \mathrm{SOgra}(R', \mathtt{true})$
$\Leftrightarrow G \models \mathrm{SOgra}(R_{\overline{\mathrm{Y}}_R}, \mathtt{true})$. □

In order to translate HR* conditions of the form $\exists(P \sqsupseteq C, c)$, we need sets of every object in P and C after the replacement of the hyperedges. This is achieved by the transformation $\mathrm{SOset}(G, X)$, which ensures that every node and edge in G^σ (after replacement) is member of the set variable X.

Construction 5. For any graph G and unary variable X,

$\mathrm{SOset}(G, X) = \bigwedge_{o \in \mathrm{D}_G} o \in X \wedge \bigwedge_{y \in \mathrm{Y}_G} \forall d.\, \mathrm{ly}_G(y)(\mathrm{att}_G(y)_{1,\ldots,k}, d) \Rightarrow d \in X$,
where $k = \mathrm{rank}(y)$ is the rank of hyperedge y.

We finally turn to the simulation of the hyperedge replacement process itself.

Lemma 3. *For a graph $S \in \mathcal{G}_\mathcal{X}$, hyperedge replacement system $\mathcal{R}$ and graph G,*

$$G \models_\mathcal{A} \langle \exists(\emptyset \hookrightarrow S), \mathcal{R} \rangle \iff G \models \mathrm{SOgra}(S) \wedge \mathrm{SOsys}(\mathcal{R}).$$

The main idea is to represent hyperedges as relations over nodes. A hyperedge with k nodes is represented as a $(k+1)$-ary relation, where the first k elements represent the nodes attached to the hyperedge by its k tentacles. The last element, d, is used as an adjoint set to capture all nodes and edges that the hyperedge is replaced by. The latter is needed to collect the set of all elements in a graph after replacement.

Construction 6. For any replacement pair x/R with $\mathrm{rank}(x) = k$ and hyperedge replacement system $\mathcal{R}$, let

$$\begin{aligned} \mathrm{SOrule}(x/R) &= \forall v_1, \ldots, v_k. \forall d. x(v_1, \ldots, v_k, d) \Rightarrow \mathrm{SOgra}(R, \mathtt{true}) \\ \mathrm{SOsys}(\mathcal{R}) &= \textstyle\bigwedge_{x \in \mathcal{X}} \bigvee_{x/R \in \mathcal{R}} \forall v_i.\, \mathrm{SOrule}(x/R) \end{aligned}$$

Proof. We begin by showing $\exists S^\sigma, q\colon S^\sigma \to G.S \Rightarrow^*_\mathcal{R} S^\sigma \iff G \models \mathrm{SOgra}(S) \wedge \mathrm{SOsys}(\mathcal{R})$, using induction over the structure of HR* conditions.

Base case. By the definition of derivations,
$\exists S^\sigma \in \mathcal{G}, q\colon S^\sigma \to G.S \Rightarrow_\mathcal{R} S^\sigma$
$\Leftrightarrow \exists S^\sigma, q\colon S^\sigma \to G. \exists x/R \in \mathcal{R}. S \Rightarrow_{x/R} S^\sigma$
$\Leftrightarrow \exists S^\sigma, q\colon S^\sigma \to G. \exists y \in \mathrm{Y}_S.\, \mathrm{ly}(y) = x \wedge S^\sigma \cong S_{\overline{y}} \cup R \wedge \forall i \in [k]. \mathrm{pin}_{Ri} = \mathrm{att}_S(y)_i$
By Lemma 2, we can reduce this to
$\Leftrightarrow \exists y \in \mathrm{Y}_S.\, \mathrm{ly}(y) = x \wedge G \models \mathrm{SOgra}(S_{\overline{y}} \cup R_{\overline{\mathrm{Pin}(R)}}, \bigwedge_{i \in [k]} \mathrm{pin}_{Ri} \doteq \mathrm{att}_S(y)_i)$.
Since $k \geq 1$ and $v_i = \mathrm{pin}_{Ri}$ for $i \in [k]$, we include the formula for $\mathrm{SOgra}(y)$:
$\Leftrightarrow G \models \mathrm{SOgra}(S) \wedge \forall_{i \in [k]} v_i. (v_1, \ldots, v_k) \Rightarrow \mathrm{SOgra}(R_{\overline{\mathrm{Pin}(R)}}, \bigwedge_{i \in [k]} v_i \doteq \mathrm{att}_S(y)_i))$.
and by the definition of SOrule, we get
$\Leftrightarrow G \models \mathrm{SOgra}(S) \wedge \forall_{i \in [k]} v_i.\, \mathrm{SOrule}(x/R)$.
Since S has only a single hyperedge, $x'(v_1, \ldots, v_{\mathrm{rank}\, x'})$ is false for every $x' \neq x$,
$\Leftrightarrow G \models \mathrm{SOgra}(S, \mathrm{SOsys}(\mathcal{R}))$.

Induction hypothesis. For some $S' \in \mathcal{G}_\mathcal{X}$ with $S \Rightarrow_\mathcal{R} S'$, assume
$\exists S', q'\colon S' \to G.S' \Rightarrow^*_\mathcal{R} S^\sigma \iff G \models \mathrm{SOgra}(S') \wedge \mathrm{SOsys}(\mathcal{R})$.

Induction step. Then
$\exists S^\sigma, q\colon S^\sigma \to G.S \Rightarrow^*_\mathcal{R} S^\sigma$

$\Leftrightarrow \exists S^\sigma, q\colon S^\sigma \to G.\exists S'.S \Rightarrow_\mathcal{R} S' \Rightarrow^*_\mathcal{R} S^\sigma$
By Lemma 2, we can express S' as a SO formula
$\Leftrightarrow \exists S'.G \models \mathrm{SOgra}(S') \wedge \bigwedge_{x\in\mathcal{X}} \bigvee_{x/R\in\mathcal{R}} \forall v_i.\, \mathrm{SOrule}(x/R)$
$\Leftrightarrow \exists S'.G \models \mathrm{SOgra}(S') \wedge \mathrm{SOsys}(\mathcal{R}) \wedge S \Rightarrow_\mathcal{R} S' \Leftrightarrow G \models \mathrm{SOgra}(S) \wedge \mathrm{SOsys}(\mathcal{R})$. □

We can now prove Theorem 2 for $\mathcal{A}$-satisfiable HR* conditions: For every HR* condition c and for all graphs $G \in \mathcal{G}$, $G \models_\mathcal{A} c \iff G \models \mathrm{SO}(c)$.

Proof (of Theorem 2). We proceed by induction over the structure of HR* conditions. The proofs for conditions `true`, $\neg c$ and $\bigwedge_{i\in I} c_i$ are straightforward. For conditions $\exists(a, c)$, we use the Lemmata 2 and 3 to show that graph morphisms and substitution can be simulated by our construction. For conditions $\exists(P \sqsupseteq C, c)$, Lemma 2 is used to show that the inclusion of C^σ in P^σ is simulated by the constructed formula.
Base case. $c =$ `true`. Then $\mathrm{SO}(c) =$ `true` $\Rightarrow G \models_\mathcal{A} c \Leftrightarrow$ `true` $\Leftrightarrow \mathrm{SO}(c) \models$ `true`.
Induction hypothesis. Assume that for HR* conditions $c_i, i \in J$ for an index set J, $G \models_\mathcal{A} c_i \Leftrightarrow G \models \mathrm{SO}(c_i)$ holds.
Induction step. Let $a = P \hookrightarrow C$.

1. $c = \exists(a, c_1)$. By Lemma 3, we have

$$\begin{aligned}
& G \models_\mathcal{A} \exists(a, c_1) && \text{Def. HR}^* \\
\Leftrightarrow\ & \exists\sigma, p\colon P \hookrightarrow G, q\colon C^\sigma \to G.q \circ a^\sigma = p \wedge q \models_{\mathcal{A},\sigma} c_1 && \text{Lemma 2} \\
\Leftrightarrow\ & G \models \mathrm{SOgra}(C{-}P) \wedge \mathrm{SOsys}(\mathcal{R}) \wedge \mathrm{SO}(c_1) && \text{Construction SO}^8 \\
\Leftrightarrow\ & G \models \mathrm{SO}(\exists(a, c_1)).
\end{aligned}$$

2. $c = \exists(P \sqsupseteq C, c_1)$. Then we have

$$\begin{aligned}
& G \models_\mathcal{A} \exists(P \sqsupseteq C, c_1) && \text{Def. HR}^* \\
\Leftrightarrow\ & \exists p\colon P \to G, \sigma, b\colon C^\sigma \to P^\sigma, q\colon C^\sigma \to G.p \circ b = q \\
& \quad \wedge q \models_{\mathcal{A},\sigma} c_1 && \text{Ind. hypothesis} \\
\Leftrightarrow\ & \exists p\colon P \to G, \sigma, b\colon C^\sigma \to P^\sigma, q\colon C^\sigma \to G.p \circ b = q \\
& \quad \wedge \exists C^\sigma \to G \wedge G \models \mathrm{SO}(c_1) \\
\Leftrightarrow\ & \exists p\colon P \to G, \sigma, b\colon C^\sigma \to P^\sigma, q\colon C^\sigma \to G.p \circ b = q \\
& \quad \wedge \exists C^\sigma \to G \wedge G \models \mathrm{SO}(c_1) \\
\Leftrightarrow\ & \exists \sigma \in \mathcal{R}, P^\sigma, C^\sigma, p\colon P^\sigma \to G.P \Rightarrow^*_\sigma P^\sigma \wedge C \Rightarrow^*_\sigma C^\sigma \\
& \quad \wedge P^\sigma \supseteq C^\sigma \wedge C \models_{\mathcal{A},\sigma} c_1 \\
\Leftrightarrow\ & \exists \sigma \in \mathcal{R}, P^\sigma, C^\sigma, p\colon P^\sigma \to G.P \Rightarrow^*_\sigma P^\sigma \wedge C \Rightarrow^*_\sigma C^\sigma \\
& \quad \wedge P^\sigma \supseteq C^\sigma \wedge \mathrm{SO}(c_1) \\
\Leftrightarrow\ & G \models \mathrm{SOgra}(C, \exists X_P, X_C.\, \textstyle\bigwedge_{x\in \mathrm{D}_P}(x \in X_P) \\
& \quad \wedge \textstyle\bigwedge_{y\in \mathrm{D}_C}(y \in X_C) \wedge X_C \subseteq X_P \wedge \mathrm{SO}(c)) && \text{Constr. SOgra} \\
\Leftrightarrow\ & G \models \mathrm{SOgra}(C, \exists X_P, X_C.\, \mathrm{SOset}(P, X_P) \wedge \mathrm{SOset}(C, X_C) \\
& \quad \wedge X_C \subseteq X_P \wedge \mathrm{SO}(c)) && \text{Constr. SO} \\
\Leftrightarrow\ & G \models \mathrm{SO}(\exists(P \sqsupseteq C, c_1))
\end{aligned}$$

3. For $c = \neg c_1$, $\mathrm{SO}(c) = \neg\,\mathrm{SO}(c_1)$. By the induction hypothesis, we have $G \models_\mathcal{A} c \Leftrightarrow G \not\models_\mathcal{A} c_1 \Leftrightarrow G \not\models \mathrm{SO}(c_1) \Leftrightarrow G \models \mathrm{SO}(c)$.
For $c = \bigwedge_{i\in J} c_j$, $\mathrm{SO}(c) = \mathrm{SO}(\bigwedge_{i\in J} c_j)$. Using the induction hypothesis, we get: $G \models_\mathcal{A} \bigwedge_{i\in J} c_j \Leftrightarrow G \models \bigwedge_{i\in J} \mathrm{SO}(c_j) \Leftrightarrow G \models \mathrm{SO}(\bigwedge_{i\in J} c_j)$.

□

It follows that every ($\mathcal{A}$-satisfiable) HR* condition can be transformed into an equivalent second-order formula. Since, by Lemma 1, every HR* condition can be transformed into an $\mathcal{A}$-satisfiable one with replacement, this is also true for HR* conditions.

Example 9. We convert the HR* condition from Example 3 into an equivalent SO formula.

$$
\begin{aligned}
&\mathrm{SO}\left(\left\langle \exists(\bullet_1 \xrightarrow{X} \bullet_2), \bullet_1 \xrightarrow{X} \bullet_2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \longrightarrow \bullet \xrightarrow{X} \bullet_2 \right\rangle\right) \\
&\equiv \mathrm{SOsys}(\bullet_1 \xrightarrow{X} \bullet_2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \longrightarrow \bullet \xrightarrow{X} \bullet_2) \\
&\quad \wedge \mathrm{SO}(\exists(\bullet_1 \xrightarrow{X} \bullet_2)) \\
&\equiv \exists X. \forall v_1, v_2. (X(v_1, v_2) \Rightarrow \mathrm{SOgra}(\bullet_1 \longrightarrow \bullet_2) \vee \mathrm{SOgra}(\bullet_1 \longrightarrow \bullet \xrightarrow{X} \bullet_2)) \\
&\quad \wedge \mathrm{SOgra}(\bullet_1 \xrightarrow{X} \bullet_2) \\
&\equiv \exists X. \forall v_1, v_2. (X(v_1, v_2) \Rightarrow \exists e. \mathrm{inc}(e, v_1, v_2) \vee \exists v_3, e. \mathrm{inc}(e, v_1, v_3) \wedge X(v_3, v_2)) \\
&\quad \wedge \mathrm{SOgra}(\bullet_1 \xrightarrow{X} \bullet_2) \\
&\equiv \exists X. \forall v_1, v_2. X(v_1, v_2) \Rightarrow \exists e. \mathrm{inc}(e, v_1, v_2) \vee \exists v_3, e. \mathrm{inc}(e, v_1, v_3) \wedge X(v_3, v_2) \\
&\quad \wedge \exists v_1, v_2. X(v_1, v_2)
\end{aligned}
$$

The resulting formula expresses "There is a relation X such that for every pair v_1, v_2 in relation X, there is either an edge from v_1 to v_2 or an edge from v_1 to some node v_3, which is in turn in relation X with v_2 (i.e. there is a path of arbitrary length from v_1 to v_2); and the graph has two nodes v_1, v_2 in relation X."

6 Conclusion

In this paper, we established a lower and an upper bound on the expressiveness of HR* conditions. The relation of HR* conditions to other formalisms is shown in Figure 4: HR* conditions extend nested conditions and are situated between node-counting monadic second-order logic and second-order logic.

Several questions regarding the expressiveness of HR* conditions remain open, as indicated by question marks in Figure 4. It is unclear how HR* conditions relate to counting monadic second-order formulas, which may count over nodes *and* edges. Furthermore, the question remains open whether any second-order formula can be expressed as a HR* condition. The author suspects that this is not the case, as quantification over arbitrary relations seems to be more powerful than the hyperedge replacement used in HR* conditions.

References

1. Baldan, P., Corradini, A., König, B., Lluch-Lafuente, A.: A temporal graph logic for verification of graph transformation systems. In: WADT 2006. pp. 1–20 (2006)

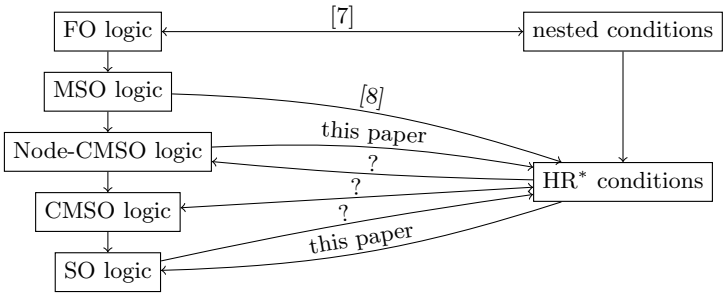


Fig. 4. Comparison of the expressiveness of several types of logics and conditions.

2. Bruggink, H.S., König, B.: A logic on subobjects and recognizability. In: IFIP Conference on Theoretical Computer Science. pp. 197–212 (2010)
3. Courcelle, B.: On the expression of graph properties in some fragments of monadic second-order logic. In: Descriptive Complexity and Finite Models: Proceedings of a DIMACS Workshop (1997)
4. van Dalen, D.: Logic and Structure. Springer-Verlag Berlin, 4th edn. (2004)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science, Springer (2006)
6. Gaifman, H.: On local and non-local properties. In: Stern, J. (ed.) Proceedings of the Herbrand Symposium: Logic Colloquium'81. pp. 105–135. North Holland Pub. Co. (1982)
7. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19, 245–296 (2009)
8. Habel, A., Radke, H.: Expressiveness of graph conditions with variables. Electronic Communications of the EASST 30 (2010)
9. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Universität Oldenburg (2009)
10. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
11. Rensink, A.: Towards model checking graph grammars. In: Workshop on Automated Verification of Critical Systems (AVoCS). pp. 150–160 (2003)

A Graph Transformational View on Reductions in NP

Marcus Ermler, Sabine Kuske, Melanie Luderer and Caroline von Totth

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
{maermler,kuske,melu,caro}@informatik.uni-bremen.de

Abstract. Many decision problems in the famous and challenging complexity class NP are graph problems and can be adequately specified by polynomial graph transformation units. In this paper, we propose to model the reductions in NP by means of a special type of polynomial graph transformation units, too. Moreover, we present some first ideas how the semantic requirements of reductions including their correctness can be proved in a systematic way.

1 Introduction

Many famous NP-complete problems involve graphs. Examples of this kind are the problems of finding a clique, a Hamiltonian cycle, a vertex cover, an independent set, etc. Whereas the complexity class NP as well as the notion of reductions between NP problems are usually defined by means of polynomial Turing machines on the general level, explicit problems and reductions are described on some higher level for easier reading and understanding.

Since the algorithms solving decision problems or modeling reductions are often composed of graph transformation steps, polynomial graph transformation units [9, 8] serve as visual, rule-based and formal descriptions for these algorithms. Consequently, graph transformation units may be helpful not only for specifying and understanding decision problems and reductions but also for obtaining correctness proofs in a systematic way. Graph transformation units contain graph transformation rules for modeling the graph transformation steps in an elegant, formal, visual and intuitive way. Moreover, the control conditions of graph transformation units restrict the set of all derivations induced by the rules to those which solve the problem. Finally, the initial and terminal graph class expressions of graph transformation units allow to specify the input and output types of the algorithms.

In [7], it has already been shown that polynomial graph transformation units are a formal computational model for decision problems in NP. To underline the usefulness of this result, we model in this paper the problems of finding a clique, an independent set, a vertex cover and a Hamiltonian cycle as graph transformation units. Moreover, we extend [7] by considering also reductions in NP. A reduction from a graph transformation unit to a graph transformation

unit transforms the initial graphs of the first unit to the initial graphs of the second unit. Polynomial graph transformation units with stepwise control serve as a computational model for such reductions in NP if they are deadlock-free and correct. The correctness can be split into forward and backward correctness. To stress this, we present reduction units from the clique problem to the independent set problem and - more sophisticated - from the vertex cover problem to the Hamiltonian cycle problem.

Moreover, we make a first step towards a proof scheme for the correctness of reductions. We show that forward correctness is obtained by induction on the length of computations provided that there are certain auxiliary reductions compatible with the computation steps. We illustrate the principle for the presented examples.

The aim of this paper is to show that graph transformation units provide a uniform, systematic, and high-level framework for the specification of decision problems as well as of reductions between them which is more intuitive than but as formal as Turing machines. Moreover, the paper illustrates how the presented approach may serve as a foundation of a proof scheme for correctness of reductions.

The paper is organized as follows. In Section 2, polynomial graph transformation units with stepwise control are presented. Section 3 shows how graph transformation units can be used as a computational model for the decision problems in NP. Section 4 proposes graph transformation units for modeling reductions in NP. Section 5 deals with correctness of reductions. The paper ends with the conclusion.

2 Polynomial Graph Transformation Units with Stepwise Control

In this section, we briefly recall graph transformation units as far as they are needed in the following sections. We emphasize especially the use of stepwise control conditions in polynomial graph transformation units as they are essential for our approach to reductions in NP. We assume that the reader is familiar with graph transformation (see [12] for an overview).

In the following, we use *edge-labeled directed graphs* with multiple edges in which the edge labels are taken from a finite alphabet A. The class of all such graphs is denoted by $\mathcal{G}$. A *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathcal{G}$ such that K is a subgraph of L and R. The application of r to a graph G yields a directly derived graph H and is performed according to the double pushout approach with injective matches (cf. e.g. [1]). Roughly spoken, an (injective) match of L in G (i.e. a subgraph of G isomorphic to L) is replaced by (a copy of) R such that the part of the match which corresponds to K is identified with the subgraph K of R. A rule with a *negative application condition* is composed of a rule $r = (L \supseteq K \subseteq R)$ and a graph N of which L is a subgraph. Its application consists of applying r only if the corresponding match of L cannot be extended to a match of N. By $\mathcal{R}$ we denote the class of rules consisting of

rules with a negative application condition. Negative application conditions are studied in [5]. Please note that polynomial time is needed to find a match and to construct a direct derivation if the rule set is fixed (which is the case throughout this paper). Moreover, the difference of the size of the resulting graph and the host graph is bounded by a constant (cf. [8]).

A *graph class expression* may be any syntactic entity X that specifies a class of graphs $SEM(X) \subseteq \mathcal{G}$. A typical example is a forbidden structure. Let $\mathcal{F}$ be a set of graphs; then $SEM(forbidden(\mathcal{F}))$ consists of all graphs G such that for each $F \in \mathcal{F}$ there is no injective match of F in G. In the following the class of graph class expressions is denoted by $\mathcal{E}$.

A stepwise control condition directly guides the derivation process, i.e. it provides for each derivation step the next permitted rule application steps. More formally, a *stepwise control condition* $C = (S, J, F, choice)$ consists of a finite set of *control states* S, two subsets $J, F \subseteq S$ of *initial and final control states* resp. and a *choice function choice* with $choice(G, s) \subseteq \mathcal{G} \times S$ for $G \in \mathcal{G}$ and $s \in S$. We denote the class of stepwise control conditions by $\mathcal{C}$.

Stepwise control conditions can be often defined w.r.t. control conditions. A *control condition* is any expression that specifies a binary relation on graphs. Examples of control conditions are the *basic control conditions* $r!$ and $try(r)$ where r is a rule. The expression $r!$ means to apply r as long as possible. The expression $try(r)$ means that if r is applicable to the current graph apply r once. In the following, *Try&Alap* denotes the class in which each control condition is either a basic control condition or it is the sequential composition of basic control conditions, i.e., it has the form $c_1; \cdots ; c_n$ $(n \geq 1)$ where for $i = 1, \ldots, n$ c_i is a basic control condition.

For each control condition $c \in$ *Try&Alap* the corresponding stepwise control condition $stw(c)$ is equal to $(S^c, J^c, F^c, choice^c)$ where S^c is defined recursively as $S^c = \{b; lambda, lambda\}$ if $c = b$ and $S^c = \{c; lambda\} \cup S^d$ if $c = b; d$ for some basic control condition b and $d \in$ *Try&Alap*. (Hence, every state is either equal to *lambda* or has the form $b; s'$ where s' is a state and b is a basic control condition.) Moreover, $J^c = \{c; lambda\}$ and $F^c = \{lambda\}$. For each $G \in \mathcal{G}$, each $s \in S^c$ and each rule r occurring in c the choice function is given by $choice^c(G, lambda) = \emptyset$ and

$$choice^c(G, r!; s) = \begin{cases} \{(G', r!; s) \mid G \underset{r}{\Longrightarrow} G'\} & \text{if } \exists G' \in \mathcal{G} : G \underset{r}{\Longrightarrow} G' \\ \{(G, s)\} & \text{otherwise} \end{cases}$$

$$choice^c(G, try(r); s) = \begin{cases} \{(G', s) \mid G \underset{r}{\Longrightarrow} G'\} & \text{if } \exists G' \in \mathcal{G} : G \underset{r}{\Longrightarrow} G' \\ \{(G, s)\} & \text{otherwise} \end{cases}$$

A *configuration* of a stepwise control condition $C = (S, J, F, choice)$ is a pair (G, s) with $G \in \mathcal{G}$ and $s \in S$. $(G, s) \vdash (G', s')$ is a *computational step* if $(G', s') \in choice(G, s)$. A *computation* is a sequence of computational steps $(G_0, s_0) \vdash (G_1, s_1) \vdash \cdots \vdash (G_n, s_n)$ (also denoted by $(G_0, s_0) \vdash^n (G_n, s_n)$). Obviously, each computation induces an underlying derivation. The *semantics* $SEM(C)$ is given by the set of derivations induced by all computations $(G_0, s_0) \vdash^n (G_n, s_n)$ with $s_0 \in J$ and $s_n \in F$.

A *graph transformation unit* is a system $gtu = (I, P, C, T)$, where $I, T \in \mathcal{E}$ are graph class expressions to specify the *initial* and the *terminal* graphs respectively, $P \subseteq \mathcal{R}$ is a finite set of rules, and $C \in \mathcal{C}$ is a stepwise control condition. In examples of graph transformation units, each stepwise control condition $stw(c)$ is abbreviated by c. Every graph transformation unit gtu specifies a binary relation $SEM(gtu) \subseteq SEM(I) \times SEM(T)$ that contains a pair (G, H) of graphs if and only if there is a derivation $G \overset{*}{\underset{P}{\Longrightarrow}} H \in SEM(C)$. For each $G \in SEM(I)$ $gtu(G)$ denotes the set $\{H \in \mathcal{G} \mid (G, H) \in SEM(gtu)\}$.

Let $gtu = (I, P, C, T)$ be a transformation unit with a stepwise control condition $C = (S, J, F, choice)$. A configuration (G, s) is *initial* if $G \in SEM(I)$ and $s \in J$. It is *terminal* if $G \in SEM(T)$ and $s \in F$. A *permitted computation* is a sequence of computational steps $(G_0, s_0) \vdash (G_1, s_1) \vdash \cdots \vdash (G_n, s_n)$ if (G_0, s_0) is an initial configuration. The induced derivation of a permitted computation is called *permitted derivation*. Note that the 0-derivation $G \overset{0}{\Longrightarrow} G$ is always permitted if $G \in SEM(I)$. A permitted computation is *successful* if (G_n, s_n) is a terminal configuration. The induced derivation of a successful computation is called *successful derivation*.

A *gtu* is *polynomial* if the following holds: (1) there is a polynomial p such that for each initial graph $G \in SEM(I)$ and each permitted derivation $G \overset{n}{\Longrightarrow} G'$, $n \leq p(size(G))$, where $size(G)$ is the sum of the number of nodes and the number of edges of G. (2) the membership problems of $SEM(I)$ and $SEM(T)$ are polynomial. (3) to compute a next configuration via the choice function takes polynomial time. Please note that for all graph class expressions and all stepwise control conditions used in this paper, the second and the third condition are satisfied (each next configuration is picked up nondeterministically based on a fixed rule set).

3 Decision Problems of NP as Graph Transformation Units

In the following, it is recalled how polynomial graph transformation units can be used as a computation model for decision problems in the complexity class NP (cf. [7]).

A decision problem is a mapping $D\colon \Sigma^* \to BOOL$, where Σ is some finite alphabet. D is in the complexity class NP if there exists a nondeterministic Turing machine TM and a polynomial p such that for each input $w \in \Sigma^*$ the following holds. (1) there is a computation of TM starting in the initial state with input w and ending in an accepting state if and only if $D(w) = true$ and (2) no computation of TM starting with input w is longer than $p(|w|)$ (cf., e.g., [6]).

Very often, inputs of decision problems are not strings but they are composed of different data types such as graphs and natural numbers. Describing these instances as words in Σ^* would be very hard to read for human beings. Hence, in the literature, they are usually defined directly. For the same reason,

the algorithms solving the decision problems are generally not given as Turing machines but as some higher level algorithmic description, and — based on the Church-Turing thesis — it is assumed that they can be computed by some Turing machine.

Whenever inputs of decision problems are graphs, polynomial graph transformation units serve as an intuitive computational model for NP-problems. More explicitly, a polynomial graph transformation unit $gtu = (I, P, C, T)$ *solves* a *graph transformational decision problem* $D\colon SEM(I) \to BOOL$ in the following way. Whenever there is a successful derivation $G \stackrel{*}{\Longrightarrow} G'$ in gtu, the result of the decision problem applied to G is *true*. Otherwise, it is *false*. Let $\mathrm{NP}_{\mathrm{GT}}$ denote the class of all graph transformational decision problems solvable by some polynomial graph transformation unit. The following statement is shown in [7].

Observation 1 $\mathrm{NP}_{\mathrm{GT}} = \mathrm{NP}$.

Example. To illustrate how decision problems can be modeled as graph transformation units, we present the decision problem *clique* of NP as a graph transformation unit. The decision problem *clique* has as input an undirected unlabeled graph G and a natural number k. The result of $clique(G, k)$ returns *true* if G contains a clique of size k, i.e., a complete subgraph with k nodes. Otherwise it returns *false*. For technical simplicity we assume that k is less than or equal to the number of nodes in G.

The problem *clique* can be modeled by the transformation unit $CLIQUE$ in Fig. 1. Each initial graph of $CLIQUE$ is the disjoint union of two graphs. The first is an undirected simple unlabeled graph G in which each node is equipped with a (directed) loop. (An undirected edge can be represented by two parallel opposite edges; an unlabeled edge is labeled with a special symbol not shown in drawings.) The second one is the graph $gr(k)$ for some $k \in \mathbb{N}$ consisting of a single node with k *succ*-loops (i.e., k loops each labeled with *succ*). It is worth noting that the unary encoding of k is possible because *clique* is strongly NP-complete.

The rule *select* is applied at first as long as possible selecting in each application a node of G while removing a *succ*-loop. Afterwards the rule $test(CLIQUE)$ is applied once if possible. Its application inserts a *bad*-edge if the selected nodes do not form a clique. The resulting graph is accepted if it does not contain a *bad*-edge.

Each permitted derivation consists of at most $k+1$ rule applications. Hence, according to the considerations of Section 2 concerning polynomial graph transformation units, we get that $CLIQUE$ is polynomial.

The semantic relation $SEM(CLIQUE)$ consists of all pairs $(G + gr(k), H + gr(0))$ such that G is simple, unlabeled and looped, and H is obtained from G by inserting an s-loop at each node of a k-clique. Hence, $(G+gr(k), H+gr(0)) \in SEM(CLIQUE)$ if $clique(G, k) = true$. Otherwise, there is no $\hat{H} \in \mathcal{G}$ such that $(G + gr(k), \hat{H}) \in SEM(CLIQUE)$. This means that $CLIQUE$ is correct.

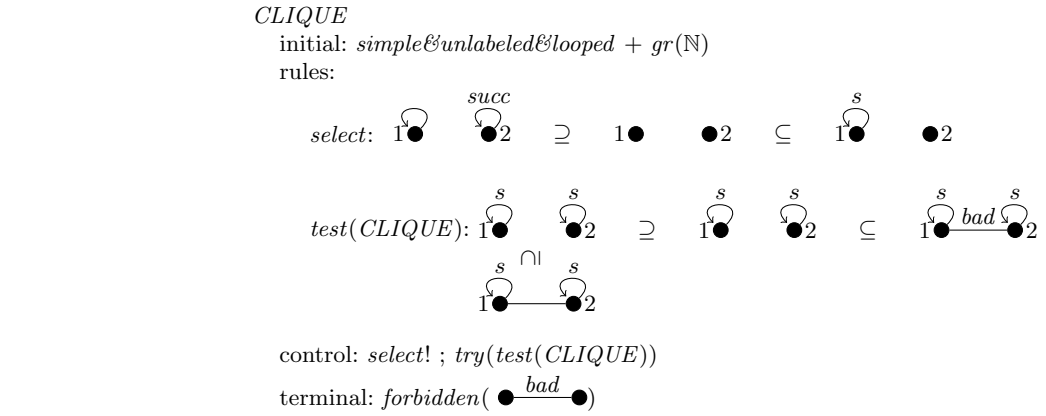


Fig. 1. A graph transformation unit for *clique*

4 Reductions in NP as Graph Transformation Units

In this section, we show how reductions in NP can be modeled by polynomial graph transformation units in a systematic way.

For $i = 1, 2$, let $D_i \colon \Sigma^* \to BOOL$ be decision problems. A (polynomial) reduction from D_1 to D_2 is a function $translate \colon \Sigma^* \to \Sigma^*$ such that (1) for each $w \in \Sigma^*$, $D_1(w) = D_2(translate(w))$ and (2) *translate* can be computed by a polynomial Turing machine. Strictly speaking, translate does not need to be a function from $\Sigma^* \to \Sigma^*$. It suffices to require that it associates a nonempty set with each string w provided that $D_1(w) = D_2(w')$ for each w' in the set associated with w. Hence, the polynomial Turing machine does not need to be deterministic. The set of all reductions is denoted by RED. As mentioned before, Turing machines are hard to read and that is why reductions are usually described on a higher level.

Whenever reductions involve graphs, polynomial graph transformation units are a natural means to specify them. More precisely, a polynomial graph transformation unit $red = (I_1, P, C, I_2)$ models a reduction from $D_1 \colon SEM(I_1) \to BOOL$ to $D_2 \colon SEM(I_2) \to BOOL$ if *red* is *deadlock-free* and *correct*. Deadlock-freeness means that every permitted derivation that is not prolongable is successful. Correctness means that $D_1(G) = D_2(H)$ for each $G \in SEM(I_1)$ and each $H \in red(G)$. In this case, *red* is called a *reduction unit*. Please note that since graph transformation is highly nondeterministic reduction units are not required to be functional, i.e., for every $G \in SEM(I_1)$, there may be more than one H in $red(G)$. If D_1 and D_2 are given as graph transformation units $gtu_1 = (I_1, P_1, C_1, T_1)$ and $gtu_2 = (I_2, P_2, C_2, T_2)$, then the correctness of *red* is implied by its forward and backward correctness defined as follows.

1. *Forward correctness*: If there is a successful derivation from $G \in SEM(I_1)$ in gtu_1, then there is a successful derivation from H in gtu_2, for all $H \in red(G)$.

2. *Backward correctness*: If there is a successful derivation from H in gtu_2, where $H \in red(G)$ for some $G \in SEM(I_1)$, then there is a successful derivation from G in gtu_1.

Let $\mathrm{RED}_{\mathrm{GT}}$ be the set of all reductions given as graph transformation units. Then it can be shown that $\mathrm{RED}_{\mathrm{GT}}$ corresponds to RED. This means that for every reduction $translate\colon \Sigma^* \to \Sigma^*$ from D to D' in RED, there is a reduction unit red from $D_{\mathcal{G}}$ to $D'_{\mathcal{G}}$ in $\mathrm{RED}_{\mathrm{GT}}$ where $D_{\mathcal{G}}$ and $D'_{\mathcal{G}}$ are the decision problems in $\mathrm{NP}_{\mathrm{GT}}$ corresponding to D and D', respectively. Conversely, let red be a reduction unit from a graph transformational decision problem D to a graph transformational decision problem D'. Then there is a reduction $translate$ from D_{Str} to D'_{Str} in RED where D_{Str} and D'_{Str} are the decision problems in NP corresponding to D and D', respectively. The proof is very similar to the proof of the correspondence of $\mathrm{NP}_{\mathrm{GT}}$ and NP (cf. [7]) and hence omitted.

Observation 2 $\mathrm{RED}_{\mathrm{GT}} = \mathrm{RED}$.

The first point of the following proposition presents a sufficient condition for deadlock-freeness. The second point relates deadlock-freeness to the example class of stepwise control conditions presented in Section 2. It makes use of the fact that every permitted computation that cannot be prolonged ends in a final state (which is not true in general). Hence to show deadlock-freeness, it is sufficient to show that all those computations end in a terminal graph.

Proposition 1. Let $gtu = (I, P, C, T)$ be a polynomial graph transformation unit with $C = (S, J, F, choice)$.

1. Then gtu is deadlock-free, if for each permitted computation $(G_0, s_0) \vdash \cdots \vdash (G_n, s_n)$ of gtu with $(G_n, s_n) \notin SEM(T) \times F$, $choice(G_n, s_n) \neq \emptyset$.
2. If $C = stw(c)$ for some $c \in Try\&Alap$, then gtu is deadlock-free if for each permitted computation $(G_0, s_0) \vdash \cdots \vdash (G_n, s_n)$ of gtu $s_n = lambda$ implies $G_n \in SEM(T)$.

A deadlock-free graph transformation unit is *functional* if for every initial graph G every successful derivation from G yields the same terminal graph (up to isomorphism). The next proposition relates the functionality to control conditions. It states that if the control condition of a deadlock-free unit is based on expressions of the form $r!$ only and the rule applications of each r are locally confluent, then the unit is functional. Clearly, in forward correctness proofs of functional reduction units only one derivation has to be checked for each initial graph of the first unit.

Proposition 2. Let $gtu = (I, P, C, T)$ be a deadlock-free polynomial graph transformation unit such that $C = stw(c)$ where c is of the form $r_1!; \cdots ; r_n!$ with $\{r_1, \ldots, r_n\} \subseteq P$. Then gtu is functional if for all $G, G_1, G_2 \in \mathcal{G}$ where G_1 and G_2 are not isomorphic: $G \underset{r}{\Longrightarrow} G_1$ and $G \underset{r}{\Longrightarrow} G_2$ implies that there is a graph $G_3 \in \mathcal{G}$ such that $G_1 \underset{r}{\Longrightarrow} G_3$ and $G_2 \underset{r}{\Longrightarrow} G_3$.

4.1 From Cliques to Independent Sets

The graph transformation unit *CLIQUE*-to-*INDEP* in Fig. 2 models a reduction from *clique* to *indep*. The decision problem *indep* gets as inputs a graph G and a natural number k. It returns *true* if and only if G has an independent set of size k, i.e., a set M of k nodes such that no two nodes of M are adjacent in G. The decision problem *indep* can be modeled by the graph transformation unit

$$INDEP = (I_{CLIQUE}, \{select, test(INDEP)\}, select!; try(test(INDEP)), T_{CLIQUE})$$

where $test(INDEP)$ is the rule 

.

The rule *complement* of the unit *CLIQUE*-to-*INDEP* inserts a d-edge between all pairs of distinct nodes provided that they are not connected via an undirected edge in the initial graph. The rule *remove* deletes all original undirected edges and, finally, the rule *relabel* turns each d-edge into an unlabeled edge.

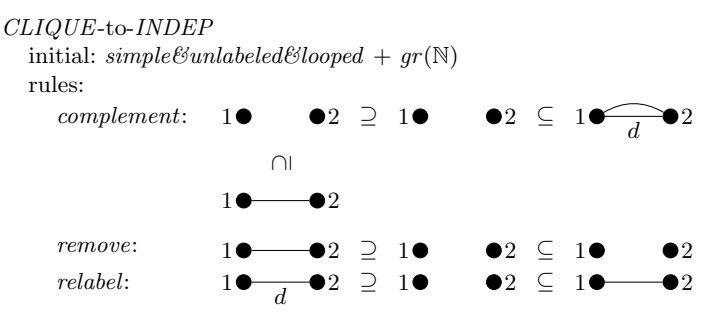


Fig. 2. A graph transformation unit for the reduction from *clique* to *indep*

Since every derivation that cannot be prolonged reaches a terminal graph and since the rule applications of each rule are locally confluent, we get by Propositions 1 and 2 that *CLIQUE-to-INDEP* is functional. The following observation concerns the successful derivations of *CLIQUE*-to-*INDEP* and can be shown by induction. It states that *CLIQUE-to-INDEP* generates for each initial graph $G + gr(k)$ the graph $H + gr(k)$ where H is the complement graph of G.

Observation 3 Let $G \in \mathcal{G}$. Then $G \stackrel{*}{\Longrightarrow} H$ is a successful derivation of *CLIQUE*-to-*INDEP* if and only if H is obtained from G by inserting an unlabeled undirected edge between each pair of nodes that is not connected via an unlabeled undirected edge and by deleting all original unlabeled undirected edges.

In every successful derivation, each of the three rules is applied at most n^2 times where n is the number of nodes of the initial graph. Hence, taking

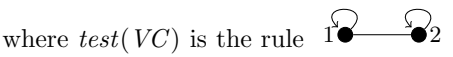
into account the considerations of Section 2 we get that *CLIQUE*-to-*INDEP* is polynomial.

4.2 From Vertex Covers to Hamiltonian Cycles

In the following, a more sophisticated example of a reduction is presented.

Let *VC* be the graph transformation unit

$$(I_{CLIQUE}, \{select, test(VC)\}, select!; try(test(VC)), T_{CLIQUE})$$

where $test(VC)$ is the rule 

. This unit *VC* is the graph transformational version of the decision problem *vc* with a graph G and a natural number k as inputs. It returns *true* if and only if G has a vertex cover of size k, i.e., a set of k nodes so that every edge is incident to at least one of these nodes.

The graph transformation unit *HC* in Fig. 3 models the decision problem *hc* the input of which is a graph G. It returns *true* if and only if G has a Hamiltonian cycle, i.e., a cycle that visits every node exactly once.

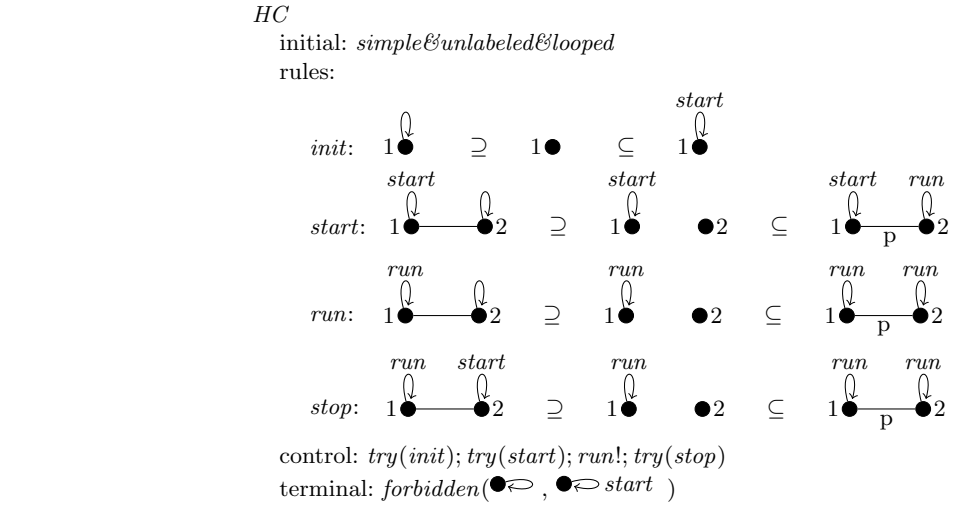


Fig. 3. A graph transformation unit for Hamiltonian cycles

The following unit *VC*-to-*HC* models a reduction from *VC* to *HC*. It is based on the construction presented in [4].

VC-to-*HC*
initial: *simple&unlabeled&looped* + $gr(\mathbb{N})$
rules: $\{r_1, \ldots, r_{11}\}$
control: $r_1!$; $try(r_2)$; $r_3!; \cdots ; r_8!; r_9(n)!; r_9(b)!; r_{10}! r_{11}!$
terminal: *simple&unlabeled&looped*

The rules of the unit *VC*-to-*HC* are depicted in Fig. 4, 5, and 6.

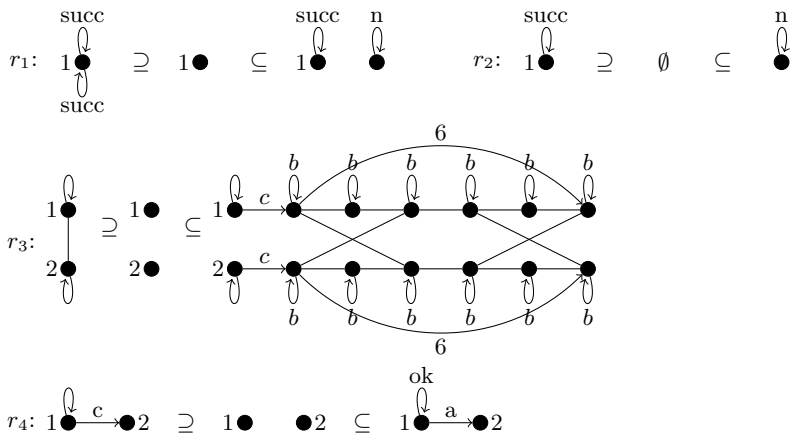


Fig. 4. The rules $r_1, \ldots, r_4$ of the unit *VC*-to-*HC*

According to the control condition the first rule is applied as long as possible before trying to apply the second rule once. This converts the graph $gr(k)$ into k n-nodes. The third rule generates a $\{u, v\}$-*edge-ladder*, two 6-edges and two c-edges for each subset $\{u, v\}$ of distinct nodes that are adjacent in the initial graph. After applying it as long as possible each initial node v is connected by a c-edge to l different ladders where l is the number of undirected edges incident to v. The target of a c-edge originating from v is called a v-*entry* and the target of the 6-edge starting from a v-entry is a v-*exit*. (The 6-edges are for remembering in further rules which exits belong to which entries.) A ladder with a v-entry is also called a v-*ladder*. The fourth rule, applied as long as possible, chooses one c-edge for each (non-isolated) initial node v replacing it by an a-edge; hence, this rule selects one $\{u, v\}$-ladder for each initial node v. The rule r_5 connects the v-entry of each selected ladder to each n-node. The rule r_6 selects for each initial node v a not yet selected v-ladder and connects the v-exit of the previously selected ladder to the v-entry of this ladder. This is repeated as long as possible so that finally all v-ladders are selected.

Rule r_7 connects for each initial node v the v-exit of the last selected ladder to each n-node. The rule r_8 removes all initial nodes together with the attached loops and the incident a-edges; with the rule r_9 every n-loop and every b-loop is turned into an unlabeled loop; r_{10} deletes all 6-edges and r_{11} all isolated initial nodes.

The next observation concerns the successful derivations of *VC*-to-*HC*.

Observation 4 Let $(G + gr(k)) \in SEM(I_{VC})$ and let $H \in \mathcal{G}$. Then $H \in$ *VC*-to-$HC(G + gr(k))$ if and only if H consists of the following components:

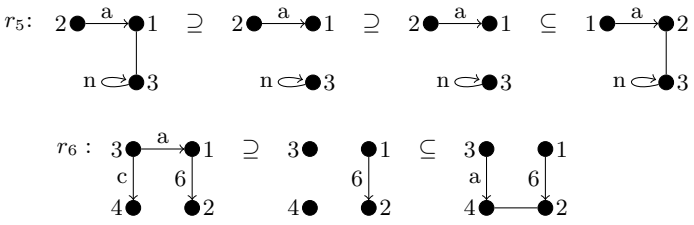


Fig. 5. The rules r_5 and r_6 of the unit *VC*-to-*HC*

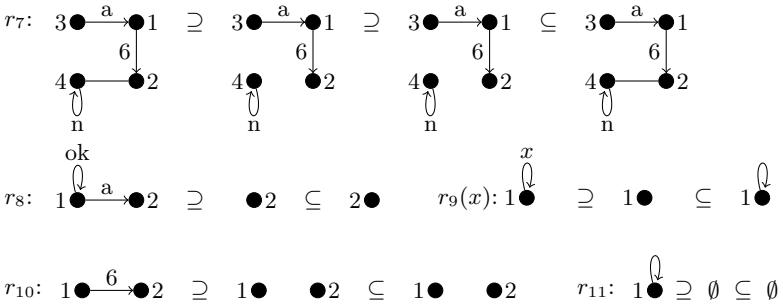


Fig. 6. The rules $r_7, \ldots, r_{11}$ of the unit *VC*-to-*HC*

- A $\{u, v\}$-ladder for each set $\{u, v\}$ of distinct adjacent nodes in G,
- k nodes, say $1, \ldots, k$ (not being part of a ladder),
- for each $v \in V_G$ there is some ordering $l_1^v, \ldots, l_{m(v)}^v$ of the set of v-ladders such that for $j = 1, \ldots, m(v) - 1$ the v-exit of l_j^v is adjacent to the v-entry of l_{j+1}^v and every node in $[k]$ is adjacent to the v-entry of l_1^v as well as to the v-exit of $l_{m(v)}^v$. [1]

The nodes $1, \ldots, k$ will also be called *clip nodes*.

Since every permitted derivation that cannot be prolonged ends in a terminal graph we get by Proposition 1 that *VC*-to-*HC* is deadlock-free. By Observation 4, it is not functional. Moreover, since each successful derivation consists of a polynomial number of steps, we get together with the considerations in Section 2 that *VC*-to-*HC* is polynomial.

5 Correctness

The following proposition is useful for proving forward correctness of reductions and provides a first step towards a proof scheme for correctness of reductions. Roughly spoken, it states that the existence of a set Aux of deadlock-free graph transformation units leads to the forward correctness of a reduction from gtu_1

[1] $[k]$ denotes the set $\{1, \ldots, k\}$.

to gtu_2 if the units in Aux satisfy certain compatibility conditions that may be seen as stepwise correctness.

Proposition 3. For $i = 1, 2$, let $gtu_i = (I_i, P_i, C_i, T_i)$ be polynomial graph transformation units. Let $red = (I_1, P, C, I_2)$ be a deadlock-free polynomial graph transformation unit. Then red is a forward correct unit from gtu_1 to gtu_2, if there is a set Aux of deadlock-free polynomial graph transformation units such that for each successful derivation $G_0 \Longrightarrow \cdots \Longrightarrow G_n$ in gtu_1 and each $H_0 \in red(G_0)$ there are units $red_1, \ldots, red_n \in Aux$ and graphs $H_1, \ldots, H_n \in \mathcal{G}$ such that the following hold:

1. For $i = 1, \ldots, n$, the graph H_i is in $red_i(G_i)$, and there is a derivation $der_i = (H_{i-1} \overset{*}{\Longrightarrow} H_i)$ such that the sequential composition $der_1 \cdots der_n$ is permitted in gtu_2.
2. $H_n \in SEM(T_2)$.

Remarks.

1. This proposition allows to prove forward correctness by induction on the length of the permitted derivations that can be prolonged to successful derivations. To this aim, stepwise control is essential.
2. Let $\widehat{red}$ be a deadlock-free polynomial unit from gtu_2 to gtu_1 such that for each $G \in SEM(I_1)$ and each $H \in red(G)$ with a successful derivation in gtu_2, $G \in \widehat{red}(H)$. Then forward correctness of $\widehat{red}$ implies backward correctness of red.

5.1 Correctness of *CLIQUE*-to-*INDEP*

Let $der = (G_0 \overset{n}{\Longrightarrow} G_n)$ be a successful derivation of *CLIQUE*. Then in every derivation step the rule *select* is applied. Let $H_0 \in$ *CLIQUE-to-INDEP*(G_0). Then due to the functionality of *CLIQUE-to-INDEP*, H_0 is the unique graph in *CLIQUE-to-INDEP*(G_0). Let red' be the unique auxiliary unit defined as

$$red' = (all, P_{\textit{CLIQUE-to-INDEP}}, C_{\textit{CLIQUE-to-INDEP}}, all),$$

where $SEM(all) = \mathcal{G}$. Please note that red' is used for transforming the graphs $G_1, \ldots, G_n$ which are not in $SEM(I_{CLIQUE})$. Hence, the initial graph class expression of red' is more general than that of *CLIQUE*. For similar reasons the terminal expression of red' is more general than the initial expression of *INDEP*. By Propositions 1 and 2, red' is functional. Moreover, for $i = 1, \ldots, n$, $red(G_i)$ consists of the complement of the underlying simple graph of G_i plus the loops of G_i. We assume without loss of generality that the node set of the graph in $red'(G_i)$ is equal to V_{G_i}.

If $n = 0$ then *CLIQUE-to-INDEP*$(G_n) = \{H_0\}$ and $H_0 \overset{*}{\Longrightarrow} H_0$ is permitted in *INDEP*. Now assume that for some $n \in \mathbb{N}$, $H_0 \underset{select}{\overset{*}{\Longrightarrow}} H_n$ is a derivation in *INDEP* where $H_n \in red'(G_n)$ if $n > 0$ and $H_n \in$ *CLIQUE-to-INDEP*(G_n) if

$n = 0$. Let $G_n \underset{select}{\Longrightarrow} G_{n+1}$. Then G_n contains a node v, an unlabeled loop of which is replaced by an s-loop and a *succ*-loop which is deleted. Then v has also an unlabeled loop in H_n and the *succ*-loop is also present in H_n. Hence there is a derivation step $H_n \underset{select}{\Longrightarrow} H_{n+1}$ in which the rule *select* is applied to v and the *succ*-loop. Since *select* only affects loops and *red*$'$ does not affect loops, $H_{n+1} \in red'(G_{n+1})$. Since $H_0 \overset{*}{\underset{select}{\Longrightarrow}} H_{n+1}$ is permitted, Point 1 of Proposition 3 is satisfied.

If $G_n \in SEM(T_{CLIQUE})$, then $test(INDEP))$ is not applicable to H_n because otherwise, there would be an edge between s-nodes in H_n which is not present in G_n. This means that $test(CLIQUE)$ would be applicable to G_n, i.e., $G_n \notin SEM(T_{CLIQUE})$ which is a contradiction. Hence, $H_n \in SEM(T_{INDEP})$, i.e., the second condition of the proposition is also satisfied.

Hence, *CLIQUE-to-INDEP* is forward correct. Since *CLIQUE-to-INDEP* models a bijective function, it is also backward correct. This leads to the following observation.

Observation 5 The unit *CLIQUE-to-INDEP* is correct.

5.2 Forward Correctness of *VC*-to-*HC*

Let $G_0 = (G_0' + gr(k)) \in SEM(I_{VC})$ and let $G_0 \overset{n}{\underset{select}{\Longrightarrow}} G_n$. Let $H_0 \in VC\text{-to-}HC(G_0)$ and for $v \in V_{G_0'}$ let $l_1^v, \ldots, l_{m(v)}^v$ be the ordering of the v-ladders in H_0 and let $c_1, \ldots, c_n$ be clip nodes (cf. Observation 4).[2] Then for each ordering $v_1, \ldots, v_n$ of the s-nodes in G_n the sequence $(c_1, l_1^{v_1}, \ldots, l_{m(v_1)}^{v_1}, \ldots, c_n, l_1^{v_n}, \ldots, l_{m(v_n)}^{v_n})$ induces a set $Paths_n$ consisting of all paths in H_0 that visit nodes $c_1, \ldots, c_n$ in this order so that after each c_i the ladders $l_1^{v_i}, \ldots, l_{m(v_i)}^{v_i}$ are visited in this order. In more detail, for $j = 1, \ldots, m(v_i)$ the ladder $l_j^{v_i}$ is passed straight from its v_i-entry to its v_i-exit if the other entry of the ladder (i.e., the entry not equal to the v_i-entry) is an s-node; otherwise it is passed straight or zigzag from its v_i-entry to its v_i-exit. The ladders depicted in Fig. 7 illustrate the courses of the straight and zigzag paths. The top nodes of the ladders represent their entries and the bottom nodes their exits.

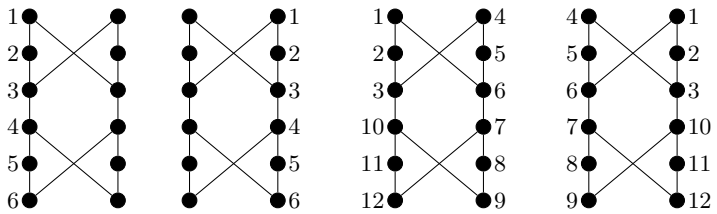


Fig. 7. Straight and zigzag paths through ladders

[2] Since $n \leq k$ these clip nodes exist.

We can construct a polynomial deadlock-free graph transformation unit *red* such that for each graph G derivable from some $G_0 \in SEM(I_{VC})$ by successive applications of the rule *select*, $red(G)$ consists of all graphs H which can be obtained from some $H_0 \in VC\text{-to-}HC(G_0)$ by highlighting one of the described paths in such a way that all edges on the path are labeled with p the first clip node has a *start*-loop and all other nodes on the path have a *run*-loop. For reasons of space limitations *red* is not presented here; but it is worth noting that it is quite similar to the unit VC-to-HC although needing more complicated control conditions. By induction on n it can be shown that for every $H_0 \in VC\text{-to-}HC(G_0)$ there is a derivation $H_0 \stackrel{*}{\Longrightarrow} H_n$ with application sequence $init\ start\ run^{n-1}$ and $H_n \in red(G_n)$ if $n > 0$. If $n = 0$ the application sequence is equal to λ.

Moreover, if G_n is a terminal graph, then $test(VC)$ is not applicable to G_n i.e., the path in H_n contains all ladders and all clip nodes of H_0 (remember that k cannot be larger than the number of nodes in G'_0). Hence the application of *stop* to the last and the first node of the path yields a terminal graph of HC.

Altogether we get that VC-to-HC satisfies the conditions for the forward correctness and hence the following observation is holds.

Observation 6 The unit VC-to-HC is forward correct.

6 Conclusion

In this paper, we have shown how reductions in NP can be modeled by graph transformation units in a visual and formal way. In particular, we have presented a first step towards a proof scheme for the correctness of reduction units. It turned out that the presented approach is suitable to specify and prove the (forward) correctness of two well-known reductions where the latter is rather complex.

In the future, we want to undertake further steps in the following directions. (1) The presented proof scheme for forward correctness is based on the correctness of a set of auxiliary units which in turn can be shown by induction. Hence, an interesting question is how the induction proof of the forward correctness can be interwoven with the induction proofs of the auxiliary units in a systematic way. (2) The first of our two reduction examples is backward correct because the reduction can be done the other way round. However, the proof of backward correctness should be integrated in the presented proof scheme in a systematic way. (cf. also [2]). (3) Since reductions are a special kind of model transformations we would like to investigate how the presented ideas can be used to prove correctness of model transformations, in general. To this aim, the presented ideas should be related to other approaches to correctness proofs of model transformations based of graph transformations. In particular, we will compare our results with those obtained in the field of model transformations using triple grammars. (4) In addition to the considered class of stepwise control conditions based on *try* and *as-long-as-possible*, we want to find out whether more general control conditions are suitable for our purposes (see, e.g., [3, 10, 11]).

Acknowledgment. We are grateful to Hans-Jörg Kreowski and to the anonymous reviewers for their helpful comments.

References

1. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In: Rozenberg [12], pp. 163–245
2. Ehrig, H., Ermel, C., Hermann, F., Prange, U.: On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In: Schürr, A., Selic, B. (eds.) 12th International Conference on Model Driven Engineering Languages and Systems(MoDELS 2009). Lecture Notes in Computer Science, vol. 5795, pp. 241–255 (2009)
3. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language and java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Proc. 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98), Selected Papers. Lecture Notes in Computer Science, vol. 1764, pp. 296–309. Springer (2000)
4. Garey, M. R., Johnson, D. S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
5. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae 26(3,4), 287–313 (1996)
6. Hopcroft, J. E., Motwani, R., Ullman, J. D.: Introduction to automata theory, languages, and computation. Pearson/Addison Wesley (2007)
7. Kreowski, H.-J., Kuske, S.: Graph multiset transformation – a new framework for massively parallel computation inspired by DNA computing. Natural Computing 10(2), 961–986 (2011)
8. Kreowski, H.-J., Kuske, S.: Polynomial graph transformability. Theoretical Computer Science 429, 193–201 (2012)
9. Kreowski, H.-J., Kuske, S., Rozenberg, G.: Graph transformation units – an overview. In: Degano, P., Nicola, R. D., Meseguer, J. (eds.) Concurrency, Graphs and Models, Lecture Notes in Computer Science, vol. 5065, pp. 57–75. Springer (2008)
10. Kuske, S.: More about control conditions for transformation units. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Proc. 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98), Selected Papers. Lecture Notes in Computer Science, vol. 1764, pp. 323–337 (2000)
11. Plump, D.: The graph programming language GP. In: Proc. Algebraic Informatics, Third International Conference (CAI 2009). Lecture Notes in Computer Science, vol. 5725, pp. 99–122. Springer-Verlag (2009)
12. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations. World Scientific, Singapore (1997)

Co-Transformation of Type and Instance Graphs Supporting Merging of Types with Retyping *

Florian Mantz[1], Yngve Lamo[1], Gabriele Taentzer[1,2]

[1] Bergen University College, Norway
{fma,yla}@hib.no
[2] Philipps-Universität Marburg, Germany
{taentzer}@informatik.uni-marburg.de

Abstract. Algebraic graph transformation is a well known rule-based approach to manipulate graphs that can be applied in many contexts. In this paper we use it in the context of model-driven engineering (MDE). Graph transformation rules usually only describing changes of one graph, however there are use cases such as model co-evolution where not only a single graph should be manipulated but related ones. The co-transformation of type graphs together with their instance graphs has shown to be a promising approach to formalize model and meta-model co-evolution. In this paper, we extend our earlier work on co-evolution by allowing transformation rules that have less restrictions so that graph manipulations such as merging and retyping of graph elements become possible.

1 Introduction

Model-driven engineering (MDE) is a software engineering discipline that uses models as the primary artifacts throughout software development processes and adopt model transformation both for their optimization as well as for model and code generation. Models in MDE describe application-specific system designs which are automatically translated into code. A commonly used technique to define modeling languages is meta-modeling. In contrast to traditional software development where programming languages rarely change, domain-specific modeling languages, and therefore meta-models, often change frequently: modeling language elements may be, e.g., renamed, extended by additional attributes, merged or refined by a hierarchy of sub-elements. The evolution of a meta-model requires the consistent migration of its models (See Fig. 1) which is a considerable research challenge in MDE [22].

Previous work e.g. [20,10,8] has mainly focused on the development of usable tools to define and execute model migrations. The relation between meta-model changes and model migrations has not been studied much on a formal level. In our work, we focus on a formal setting of meta-model and model co-evolution to study their relations and introduce correctness criteria of co-evolutions. Models are specified as graphs while model relations are defined by graph morphisms.

* This work was partially funded by NFR project 194521 (FORMGRID)

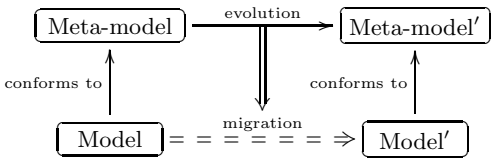


Fig. 1: Meta-model evolution and model migration

Especially the type conformance of models to their meta-models are specified by graph morphisms. Manipulations of these graphs are described by graph transformation [6] and category theoretical constructs [2]. In particular the proposed approach is based on the co-span double pushout [7] approach which is a variant of the double pushout (DPO) approach where transformation rules are co-spans. Equivalence to the "traditional" DPO approach has been shown in [7]. We prefer this alternative to the "traditional" DPO approach because we consider this approach more practical since migrations can be easier synchronized over an joint-type graph instead of a difference type graph [23]). The aim is to understand the co-evolution problem better so that future tool support can profit from this work. In [23] we proposed a framework that relates co-span DPO transformations [7] on type graphs and their instance graphs that can be applied in several categories of graphs. The framework clarifies the conditions model migrations need to satisfy as well as explain how the fully determined part of migration transformations such as deleting instances of deleted types can be derived. Variants of instance graph migrations are still possible and can be specified choosing a proper migration strategy. While our earlier work only considered injective rules with injective matching, we show in this paper that these conditions can be relaxed so that one morphism of the rule can be non-injective. In contrast to [23] it is now possible to merge elements in graphs and type graphs and retype instance nodes and edges accordingly. This is a quite useful feature when it comes to model co-evolution since the context of merged elements is preserved. To motivate the approach we consider a running example using the extended theory. Note, in the following we call a co-span transformation on the type graph "evolution", while co-span transformations on instance graphs are considered to be "migrations". An "evolution" transformation together with its related "migration" transformation are considered as "co-transformation". The following results can be summarized as:

- We revise the definition of co-span transformation so that we do not require that the right morphism of the rule has to be injective. Then we show that for a given type graph evolution a graph migration exists which is well-typed by the evolution under this relaxed condition.
- We also give conditions that ensure the uniqueness of the typing relation between the type graph evolution and the graph migration.

2 Merging Metamodel Elements

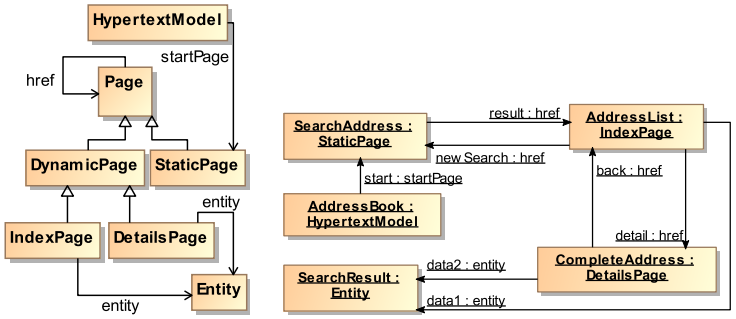
In the following we illustrate an application of the proposed co-transformation framework in the context of meta-model evolution with model co-adaptation by a running example. The example is given in the category of attributed, typed graphs with inheritance *AIGraph* as introduced in [9]. First we explain an evolution step using injective rules before we revise the same step by using non-injective rules that merge types. In addition the example illustrates how to use co-span transformations and co-transformations.

Figure 2a shows the meta-model of SWAL[3], a simple domain-specific modeling language for development of interconnected web pages. The meta-model mainly distinguishes between different kinds of web pages which are connected by hyper references. There are two types of web pages, static and dynamic ones. While instances of static pages result in plain HTML pages after code generation i.e. in pages that contain HTML tags only, instances of dynamic pages result in Java Server Pages (JSP). Note that static pages may contain static HTML forms. In addition, the meta-model defines two special types of dynamic pages, both referring to data entities being further specified in a data model not presented here, due to space limitations. An instance of the type *IndexPage* is translated to a JSP handling a list index needed to manage a data entity containing a list of values in a browsable table widget. Each instance of type *DetailsPage* is translated to a JSP showing the full content of a data entity as well as offering the typically CRUD operations like insert and update. Figure 2b shows a SWAL model of a simple address book application. The application starts with a page for address search and prints its results to an address list page. Furthermore, the user can navigate to a page where address details may be updated and stored in a corresponding data entity.

The meta-model evolution step: Initially, it was decided that the start page should be a plain HTML page. However, we might realize that the distinction between dynamic pages and static ones is unnecessary and even annoying. For example, in the address book application the search page cannot show previous search parameters because of this decision. The search page is the start page and therefore must be a *StaticPage* which cannot save any information in the session environment. Therefore, we decide to merge class *StaticPage* and *DynamicPage* in the meta-model (see Fig. 3). For the address book application, this means that class *SearchAddress* must be retyped.

First we consider this meta-model change by adding and deleting elements only, following the approach of [23]. Hence, we have to replace elements in the type graph as well as in the instance graph. In particular in the instance graph

[3] The development of SWAL was initiated by Manuel Wimmer and Philip Langer at the Technische Universität Wien and reimplemented for its use in modeling courses at Philipps-Universität Marburg.



(a) SWAL meta-model (b) AddressBook model

Fig. 2: Meta-model with instance model

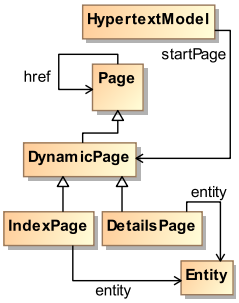


Fig. 3: Evolved meta-model

we have to recreate nodes together with their context edges considering their new types. By using our new approach we will later give a simplified version of this particular transformation.

In Fig. 4, the co-span graph transformation is shown describing the meta-model evolution step from Fig. 2a to Fig. 3 using an injective rule. Mappings are indicated by numbers. This is done in two steps: first, a second reference *startPage* is added to the meta-model (see TU). Formally, this is done by a pushout construction. Second, the former reference *startPage* and class *StaticPage* are deleted from the meta-model (see TH). This is done by substracting $TI - TR$ from TU. Formally, we do a pushout complement construction which results in a unique (up to isomorphism) graph TH for the case of co-span transformation rules.

Figure 5 shows the new version of the meta-model evolution "Merge class". The evolution rule is more compact since the context of the classes does not need to be considered by the rule. The merge is achieved by the non-injective morphisms tl and tg. The right pushout is the id morphism i.e. nothing is deleted here. Note that both outgoing inheritance arrows of the merged classes *DynamicPage* and *StaticPage* are merged. This merge of inheritance arrows results from

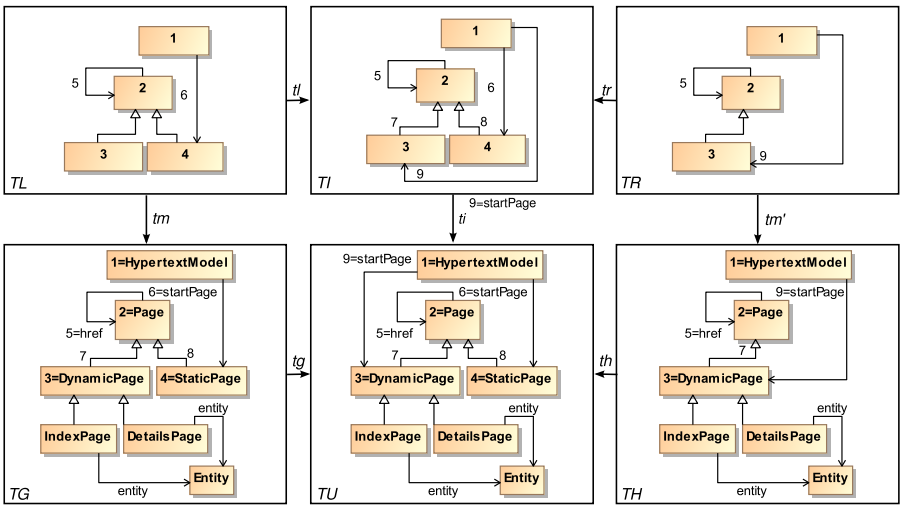


Fig. 4: Evolution co-span transformation *Delete sub-class*

the property of the used category *AIGraph* (see [9]). Using the revised approach instance elements only need to be retyped so that we do not need to consider their context during their migration.

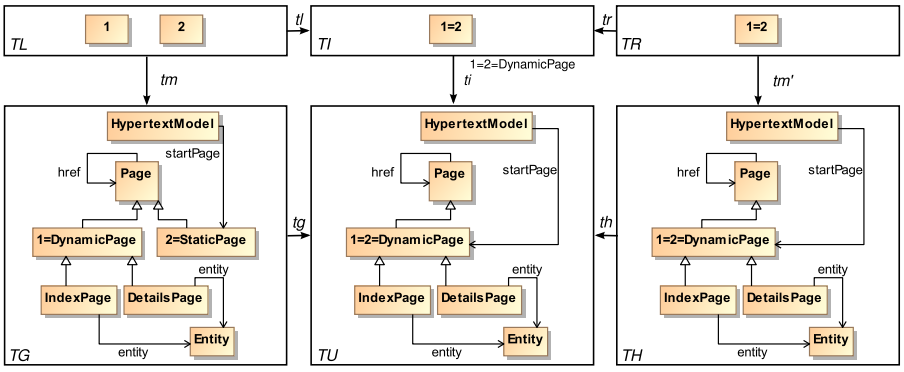


Fig. 5: Evolution co-span transformation *Merge class*

In Fig. 6 a model migration for the new meta-model evolution "Merge class" is shown. The migration rule can be deduced in a similar way as discussed in [23]. The model migration basically does nothing except that class *SearchAddress* is retyped. Retyping in the instance graph is performed in the first pushout. Hence the graphs in the middle and the right side of the rule are the same. To save space we do not show these graphs in Fig. 6 twice. If we do not have retyping facilities available, page *SearchAddress* has to be recreated having a new type. In

addition, its context references have to be recreated. Note that the corresponding migration rule would be a classical graph transformation rule, while the rule in Fig. 6 is not: in Fig. 6 the graph is not changed but the typing morphisms. Note further that the migration rule contains two pages not being changed. These are included if we want that the migration shall contain all instances of types being changed, this will be formalized as match completeness in Section 3.

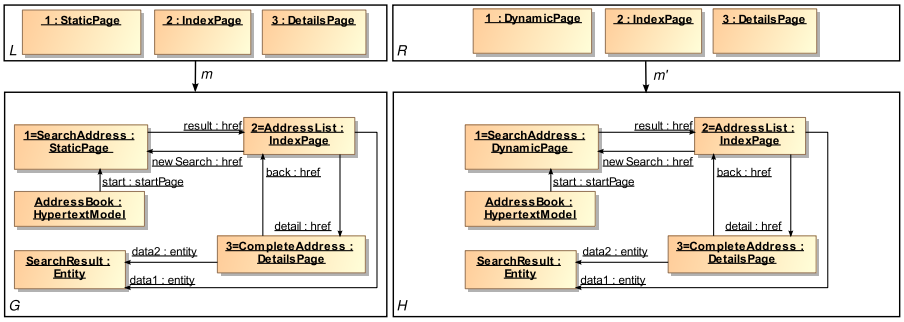


Fig. 6: Migration co-span transformation *Merge class*

3 Extending the Theory

In this section we extend the theory on co-transformations. First we relax the definition of co-span transformation and co-transformation given in [23]. Afterwards we generalize the theory. In contrast to the definition of co-span transformation in [23] the left morphism of the rule is no longer required to be injective. Since, the theory should be applicable to different kinds of graphs, the theory is formulated and proven in the context of (weak) adhesive High-Level-Replacement (HLR) categories [6,13] in the long version of this paper [16]. In the following we present the results for (typed) graphs.

Definition 1 (Graph). *A graph* $G = (G_V, G_E, src^G, trg^G)$ *consists of a set* G_V *of vertices (or nodes), a set* G_E *of edges (or arrows) and two maps* $src^G, trg^G : G_E \to G_V$ *assigning the source and target to each edge, respectively.* $e : x \to y$ *denotes that* $src^G(e) = x$ *and* $trg^G(e) = y$.

Definition 2 (Graph morphism). *A graph morphism* $g : G \to H$ *consists of a pair of maps* $g_V : G_V \to H_V$, $g_E : G_E \to H_E$ *which preserve the graph structure, i.e., for each edge* $e : x \to y$ *in* G *we have* $g_E(e) : g_V(x) \to g_V(y)$ *in* H, *i.e.,* $g_V \circ src^G = src^H \circ g_E$ *and* $g_V \circ trg^G = trg^H \circ g_E$.

$$\begin{array}{ccc} G_V & \xrightarrow{g_V} & H_V \\ {\scriptstyle src^G}\uparrow & = & \uparrow{\scriptstyle src^H} \\ G_E & \xrightarrow{g_E} & V_E \end{array} \qquad \begin{array}{ccc} G_V & \xrightarrow{g_V} & H_V \\ {\scriptstyle trg^G}\uparrow & = & \uparrow{\scriptstyle trg^H} \\ G_E & \xrightarrow{g_E} & V_E \end{array}$$

Definition 3 (Co-span transformation rule). *A* co-span transformation rule *$p = L \xrightarrow{l} I \xleftarrow{r} R$ consists of graphs L, I and R and two jointly surjective graph morphisms l and r where r is injective.*

Definition 4 (Co-span transformation). *Given a co-span transformation rule $p = L \xrightarrow{l} I \xleftarrow{r} R$ together with an injective graph morphism m, called* match*, rule p can be applied to G if a co-span double-pushout exists as shown in the diagram on the right. $t : G \overset{p,m}{\Longrightarrow} H$ is called a* co-span transformation.

$$\begin{array}{ccccc} L & \xrightarrow{l} & I & \xleftarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO1) & \downarrow{\scriptstyle i} & (PO2) & \downarrow{\scriptstyle m'} \\ G & \xrightarrow[g]{} & U & \xleftarrow[h]{} & H \end{array}$$

A co-span transformation rule is applied to a match m by first constructing the pushout $PO1$ before constructing the pushout $PO2$ as pushout complement of $i \circ r$. A co-span transformation rule is only applicable iff the gluing condition is satisfied.

Definition 5 ((Co-span) gluing condition). *Given a (typed) co-span transformation rule $p = L \xrightarrow{l} I \xleftarrow{r} R$, a (typed) graph G, and a match $m\colon L \to G$ with $X = (X_V, X_E, src^X, trg^X)$ for all $X \in \{L, I; R, G\}$, we can state the following defintions:*

- *The gluing points GP are those nodes and edges in L that are not deleted by p, i.e. $GP = (l(L_V) \cup (l(L_E)) \cap (r(R_V) \cup r(R_E))$.*
- *The dangling points DP are those nodes in L whose images under m are the source or target of an edge in G that does not belong to $m(L)$, i.e. $DP = \{v \in L_V | \exists e \in G_E \backslash m_E(L_E) : src^G(e) = m_V(v) \text{ or } trg^G(e) = m_V(v)\}$.*

p and m satisfy the gluing condition if all dangling points are also gluing points, i.e. $DP \subseteq GP$.

Based on co-span transformations (rules), we now define co-transformation (rules).

Definition 6 (Co-transformation rule). *A type graph rule $tp = TL \xrightarrow{tl} TI \xleftarrow{tr} TR$ together with an instance graph rule $p = L \xrightarrow{l} I \xleftarrow{r} R$ form a* co-transformation rule *(tp, p), if there are graph morphisms $t_L\colon L \to TL, t_I\colon I \to TI$ and $t_R\colon R \to TR$ such that both squares in the diagram on the right commute. In such a co-transformation rule (tp,p), the type graph transformation rule tp is called an* evolution rule *while the instance graph transformation p is called a* migration rule *wrt. tp. We also say that migration rule p is well-typed wrt. tp.*

$$\begin{array}{ccccc} TL & \xrightarrow{tl} & TI & \xleftarrow{tr} & TR \\ {\scriptstyle t_L}\uparrow & (1) & \uparrow{\scriptstyle t_I} & (2) & \uparrow{\scriptstyle t_R} \\ L & \xrightarrow[l]{} & I & \xleftarrow[r]{} & R \end{array}$$

Corresponding addition and deletions can be reflected in a co-transformation rule.

Definition 7 (Reflecting migration rules). *Consider a* co-transformation rule (tp, p) *like in the figure above. A co-transformation rule* (tp, p) *is called*

1. addition-reflecting *if* $TL \xrightarrow{tl} TI \xleftarrow{t_I} I$ *is a pushout (left square).*
2. deletion-reflecting *if* $I \xleftarrow{r} R \xrightarrow{t_R} TR$ *is a pullback (right square).*

We also say that migration rule p is addition-reflecting or deletion-reflecting wrt. tp.

A co-transformation rule applied to meta-model and model forms a co-transformation.

Definition 8 (Co-transformation). *Two co-span graph transformations* $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ *and* $t : G \overset{p,m}{\Longrightarrow} H$ *that apply co-transformation rule* (tp, p) *to instance graph* G *being typed over graph* TG *by* $t_G \colon G \to TG$ *form a* co-transformation (tt, t)*, if there are graph morphisms* $t_U \colon U \to TU$*, and* $t_H \colon H \to TH$ *such that all faces of Fig. 7 commute. In such a co-transformation* (tt, t)*, the type graph transformation* $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ *is called an* evolution *while the instance graph transformation* $t : G \overset{p,m}{\Longrightarrow} H$ *is called a* migration *wrt. tt.* (tm, m) *with* $tm \colon TL \to TG$ *and* $m \colon L \to G$ *is called the match of the co-transformation rule* (tp, p)*. If* $G \xleftarrow{m} L \xrightarrow{t_L} TL$ *is a pullback (left face) then the co-transformation* (tt, t) *is called match-complete.*

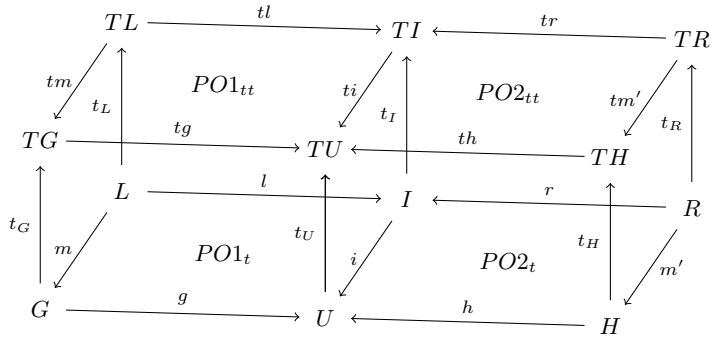


Fig. 7: Co-transformation

Note that $PO2_t$ exists only if the gluing condition is satisfied for p (see Definition 5). The gluing condition has to hold for both transformations of the co-transformation. In specific cases, however, a satisfied gluing condition on the meta-model level implies a satisfied gluing condition on the model level [16].

Proposition 1. *Let (tt, t) with $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ and $t : G \overset{p,m}{\Longrightarrow} H$ be a co-transformation with migration rule p deduced by the following construction:*

1. *Construct $G \xleftarrow{m} L \xrightarrow{t_L} TL$ as pullback of $G \xrightarrow{t_G} TG \xleftarrow{tm} TL$*
2. *Complete $L \xrightarrow{t_L} TL \xrightarrow{tl} TI$ by $L \xrightarrow{l} I \xrightarrow{t_I} TI$ to a commuting square*
3. *Construct $I \xleftarrow{r} R \xrightarrow{t_R} TR$ as pullback of $I \xrightarrow{t_I} TI \xleftarrow{t_r} TR$*

If tm satisfied the gluing condition wrt. to rule tp, then m satisfies the gluing condition wrt. rule p.

Proof can be found in [16].

Theorem 1 (existence of match-complete co-transformation). *Given a co-span graph transformation $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ (describing the evolution of type graph TG, see Fig. 7) and a graph G typed by TG with typing morphism $t_G : G \to TG$ then there exists a co-span graph transformation $t : G \overset{p,m}{\Longrightarrow} H$ (describing a corresponding model migration) such that:*

1. *migration rule p is deletion-reflecting wrt. evolution rule tp*
2. *(tt, t) forms a match-complete co-transformation*

Note that in the example in Section 2, the evolution and migration shown form a match-complete, deletion-reflecting co-transformation. See Fig. 5 together with Fig. 6.

Corollary 1 (unique typing of co-transformation). *Given a deletion-reflecting co-transformation rule (tp, p) with an applicable complete match (tm, m) on a type graph TG with instance graph G: then migration t is uniquely typed by evolution tt (up to isomorphism), i.e. t_U and t_H are uniquely determined (see Fig. 7).*

Proof can be found in [16].

4 Related Work

Co-evolution of structures has been considered in several areas of computer science such as for database schemes, grammars, and meta-models [15,14,18,21]. Especially database schema evolution has been a subject of research in the last decades. However, the challenge of schema evolution differs from meta-model evolution with model migration for various reasons. To mention only a few: while models usually are hold in the main memory, database tables often cannot. Hence schema evolution is often considered on the level of implementation focusing on the efficiency of the migration. Moreover data should be retrieved via queries. It means that the structure of stored information has to be considered when formulating a query, but the query response is one joint relation even containing duplicate entries. Hence, research considering schema evolution is often concerned with the relation between database queries and the database

schemes [5,4]. In MDE, structural information usually has to be reflected in generated artifacts. Furthermore the basic constraints of relational databases are only a few, i.e. primary key constraint, foreign key constraint and typing [4]. In modelling one need more complex constraints to specify complex business rules. In addition, in relational models, retyping of tuples i.e. tables is slightly different since there is no concept of inheritance.

Recently, research activities have started to consider meta-model evolution and to investigate the transfer of schema evolution concepts to meta-model evolution (see e.g. [10]). Our work differs from other work on model co-evolution such as e.g.[11,20] since we are more concerned with the correctness of model migrations with respect to checkable criteria rather than tooling. In the following, we focus on related work that considers merging of types while we refer to a general discussion of related work in paper [23]. [11] provides an overview of approaches that considers the coupled evolution of meta-models and models. Merging is considered in three papers, one describing the meta-model evolution tool COPE [10] and two papers considering merging of types in object-oriented databases [3,19]. The mentioned approaches provide basic merge operators with implementation descriptions based on retyping, addition and deletion of elements.

König et al. [12] also consider model evolution correctness criteria based on a categorical framework. However, König et al. work with an indexed view of models and employ functor-based folding and unfolding constructions instead of algebraic graph transformation. In addition, we tackle the problem of model co-evolution while König et al. target the migration of large data.

In object-oriented database schema evolution, merging of types is considered on the level of set theory [1]. Furthermore, merging of types has also been considered in the context of relational databases [5] and ontology evolution [17]. To the best of our knowledge merging of types has not been considered in the context of graphs and graph transformation before.

5 Conclusion

In [23] we introduce co-span co-transformations that allow adding and deleting of elements in type and instance graphs. In this paper we extend the theory of co-transformations by allowing co-evolution rules that enable us to specify merging of types with corresponding retyping of model elements. By applying this feature, meta-model evolutions as well as migrations can be specified easier since merging of elements is naturally supported and has not to be emulated by adding and deleting actions. In Theorem 1 we show that there is always a match-complete and deletion-reflecting co-transformation given an evolution and a typed graph. Corollary 1 states that the application of a deletion reflecting co-transformation always results in a unique typing. The example in Section 2 shows a typical co-evolution case with merging. While the presented approach considers co-transformations on a theoretical level, providing correctness criteria for model migrations, we plan to use these criteria in future tool support. Furthermore in the future we plan to consider evolution rules including constraints.

References

1. Alhajj, R., Polat, F.: Rule-based schema evolution in object-oriented databases. Knowledge-Based Systems 16(1), 47–57 (2003)
2. Barr, M., Wells, C.: Category Theory for Computing Science (2^{nd} Edition). Prentice Hall (1995)
3. Bréche, P.: Advanced Primitives for Changing Schemas of Object Databases. In: CAiSE'96. LNCS, vol. 1080, pp. 476–495 (1996)
4. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. Proceedings of VLDB 2010: 36^{th} International Conference on Very Large Database Endowment 4(2), 117–128 (2010)
5. Curino, C., Moon, H.J., Ham, M., Zaniolo, C.: The PRISM Workwench: Database Schema Evolution without Tears. In: Ioannidis, Y.E., Lee, D.L., Ng, R.T. (eds.) Proceedings of ICDE 1999: 25^{th} International Conference on Data Engineering. pp. 1523–1526. Proceedings of ICDE 1999: 25^{th} International Conference on Data Engineering (2009)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (March 2006)
7. Ehrig, H., Hermann, F., Prange, U.: Cospan DPO Approach: An Alternative for DPO Graph Transformation. EATCS Bulletin 98, 139–149 (2009), http://tfs.cs.tu-berlin.de/publikationen/Papers09/EHP09.pdf
8. EMF Migrate: Project Web Site, http://www.emfmigrate.org
9. Hermann, F., Ehrig, H., Ermel, C.: Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In: Fundamental Approaches to Software Engineering, 12th Int. Conference, FASE 2009. vol. LNCS 5503, pp. 325–339. Springer (2009), long version as TR 2008-07 at TU Berlin
10. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Drossopoulou, S. (ed.) Proceedings of ECOOP 2009: 23^{rd} European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 5653, pp. 52–76. Springer (2009)
11. Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G.: An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In: Malloy, B.A., Staab, S., van den Brand, M. (eds.) SLE: Proceedings of the 3^{nd} International Conference of Software Language Engineering. LNCS, vol. 6563, pp. 163–182 (2010)
12. König, H., Löwe, M., Schulz, C.: Model Transformation and Induced Instance Migration: A Universal Framework. In: da Silva Simão, A., Morgan, C. (eds.) Proceedings of SBMF 2011: 14^{th} Brazilian Symposium on Formal Methods, Foundations and Applications. Lecture Notes in Computer Science, vol. 7021, pp. 1–15. Springer (2011)
13. Lack, S., Sobocinski, P.: Adhesive Categories. In: Walukiewicz, I. (ed.) Proceedings of FoSSaCS 2004: 7^{th} Foundations of Software Science and Computation Structures. Lecture Notes in Computer Science, vol. 2987, pp. 273–288 (2004)
14. Lämmel, R.: Grammar Adaptation. In: FME. pp. 550–570 (2001)
15. Li, X.: A Survey of Schema Evolution in Object-Oriented Databases. In: TOOLS. pp. 362–371. IEEE Computer Society (1999)
16. Mantz, F., Taentzer, G., Lamo, Y.: Co-Transformation of Type and Instance Graphs Supporting Merging of Types with Retyping: Long Version. Technical report, Department of Mathematics and Computer Science, University of Marburg, Germany (September 2012), www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk

17. Noy, N.F., Klein, M.C.A.: Ontology Evolution: Not the Same as Schema Evolution. Knowledge-Based Systems 6(4), 428–440 (2004)
18. Pizka, M., Juergens, E.: Automating Language Evolution. In: TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. pp. 305–315. IEEE Computer Society, Washington, DC, USA (2007)
19. Pons, A., Keller, R.K.: Schema Evolution in Object Databases by Catalogs. In: IDEAS'97. pp. 368–378 (1997)
20. Rose, L., Kolovos, D., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (eds.) Proceedings of ICMT 2010: 3^{rd} International Conference on Theory and Practice of Model Transformation. Lecture Notes in Computer Science, vol. 6142, pp. 184–198. Springer (2010)
21. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. J. Vis. Lang. Comput. 15(3-4), 291–307 (2004)
22. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: Metamodelling - State of the Art and Research Challenges. In: Model-Based Engineering of Embedded Real-Time Systems. LNCS, vol. 6100, pp. 57–76. Springer (2010)
23. Taentzer, G., Mantz, F., Lamo, Y.: Co-Transformation of Graphs and Type Graphs With Application to Model Co-Evolution. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Proceedings of ICGT 2012: 6^{nd} International Conference on Graph Transformations. Lecture Notes in Computer Science, vol. 7562. Springer (2012)

Big Red: A Development Environment for Bigraphs

Alexander Faithfull, Gian Perrone, and Thomas T. Hildebrandt

IT University of Copenhagen, Denmark
{alef,gdpe,hilde}@itu.dk

Abstract. We present Big Red, a visual editor for bigraphs and bigraphical reactive systems, based upon Eclipse. The editor integrates with several existing bigraph tools to permit simulation and model-checking of bigraphical models. We give a brief introduction to the bigraphs formalism, and show how these concepts manifest within the tool using a small motivating example bigraphical model developed in Big Red.

Keywords: bigraphs, editor, reactive systems

1 Introduction

Bigraphical reactive systems are a class of graph-rewriting systems designed to capture orthogonal notions of *connectivity* and *locality* through the use of two graph structures — a *place* graph, and a *link* graph. They were first proposed by Robin Milner [5] to address the challenges associated with modelling of ubiquitous computing applications. Bigraphs have been successful in capturing the syntax and semantics of a number of well-known formalisms and real-world applications[1].

The Big Red tool is a prototype editor to support the development of bigraphs and bigraphical reactive systems in a visual manner. It interfaces with existing bigraph tools such as the BigMC bigraphical model checker [6] to permit the execution of models. Big Red aims to make bigraphs more accessible to novice users, as well as providing development support to more experienced bigraph users. Bigraphs have a visual presentation that is formal and unambiguous, and one of the major benefits is the ability to present a relatively complex bigraphical model in a way that is comprehensible by non-experts. This is the motivation for the development of Big Red: making it easier to create and interact with bigraphs increases the applicability and utility of the formalism in more diverse application areas.

Below we first briefly in Section 2 describe the previous efforts to implement bigraphical reactive systems. We then proceed in Section 3 describing how bigraphs are expressed in Big Red, using a small example of a context-aware printing system, inspired by that given in [1]. Finally, Section 4 briefly describes the implementation of the tool, and suggests ways in which it may be extended using additional modules.

[1] Some examples are available from http://bigraph.org/papers/gcm2012/.

2 Related Work

One of the first attempts at creating a graphical editor for bigraphs was within the Bigraphspace [3] project in 2009, during which a prototype bigraph editor based upon eclipse was developed; however, this work was never completed, and no usable editor currently exists. Bigraphspace used the correspondence between the structure of bigraphs and XML documents [4] to provide a tuplespace-like API with which to manipulate bigraphs. Big Red differs from these efforts in that it implements editing of Milner's bigraphs in such a way as to enable external tools to perform further analysis of these models, rather than imposing a particular (extra-bigraphical) semantics upon the models.

To date, bigraphical reactive systems have been largely implemented using term representations of bigraphs, such as that used in the BPLtool [2], or in the BigMC model checker [6]. While this is appropriate for bigraph experts, it presents a significant learning curve for novice users, and ignores the benefits provided by the formal graphical syntax provided by bigraphs.

3 Bigraphs in Big Red by Example

A bigraph is a forest of node labelled trees called the *place graph*, the roots of which are indexed by integers, and referred to as *regions*. The place graph parent relationship is usually drawn by nesting in the graphical syntax. The label of a non-root place graph node is referred to as its *control*, and is drawn from the bigraph *signature*. We will use "`X` node" to mean a node that is labelled with the control `X`. The control of a node also defines the number of *ports* of the node, referred to as the *arity* of the control and provided by a function $ar : \Sigma \to \omega$ given as part of the signature. The link graph can be viewed as a directed hypergraph with edges mapping a subset of the complete set of all ports of nodes in the place graph and the *inner* names of the bigraph to either a (single) outer name or a single edge. By convention, outer names are drawn above the bigraph, while inner names are drawn below the bigraph.

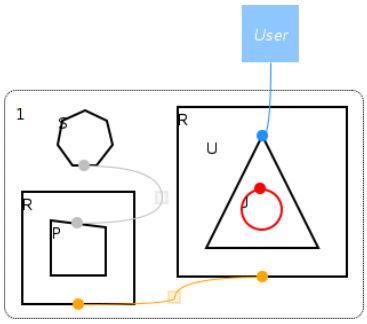


Fig. 1. An example bigraph expressed in the visual syntax of the Big Red tool.

Fig. 1 is an example bigraph constructed within Big Red for a context-aware printing system, inspired by that given in [1]. The example scenario involves a building in which users can submit print jobs to a print spool, and then move into a room with any printer connected to that print spool, at which point the printer will complete the job. The rooms of the buidling are represented by place graph nodes labelled with the control `R`. Similarly, the users, printers, central print spool and print jobs are represented by place graph nodes labelled with the controls `U`, `P`, `S`, and `J` respectively. All controls in this example have arity 1, meaning that every (non-root) node of the place graph has a single port. The port of a room (`R`) node is linked to ports of other `R` nodes that are connected to the room by a door. The port of a user (`U`) node is linked to an outer name representing the identity of the user. The port of a printer (`P`) node is linked to the port of the spool (`S`) node representing the printer spool the printer is connected to. Finally, the port of a job (`J`) node will be linked to the port of the user node to which the job belongs.

The bigraph in Fig. 1 thus represents a single print spool and two rooms connected by a door (represented by the yellow link between the ports of the room nodes). The left room contains a printer which is linked to the print spool, and the right room contains a user with the identity User given as an outer name. The user has a job which has not yet been linked to the user (as long as it has not left the user, its ownership is given implicitly by its location).

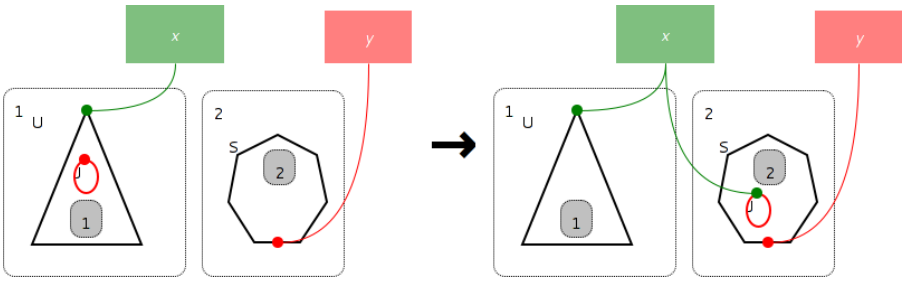
Big Red permits users to specify custom shapes for nodes associated with each type of control. In this example, rooms are represented as boxes, users as triangles, jobs as circles, etc. The shaded box `User` is an outer name that is linked to a port of the `U` node. We reiterate Milner's bigraph definition [5] here:

$$(V, E, ctrl, prnt, link) : \langle n, X \rangle \to \langle m, Y \rangle$$

where V is the set of nodes, E is a set of edges, $ctrl : V \to \Sigma$ assigns controls to nodes drawn from a signature Σ, $prnt : m \uplus V \to V \uplus m$ is the *parent map* that gives the nested place graph structure, and $link : X \uplus P \to E \uplus Y$ is the link map, where $P = \{(v, i) : v \in V \wedge i \in 0 \ldots ar(ctrl(v))\}$ is the set of all ports.

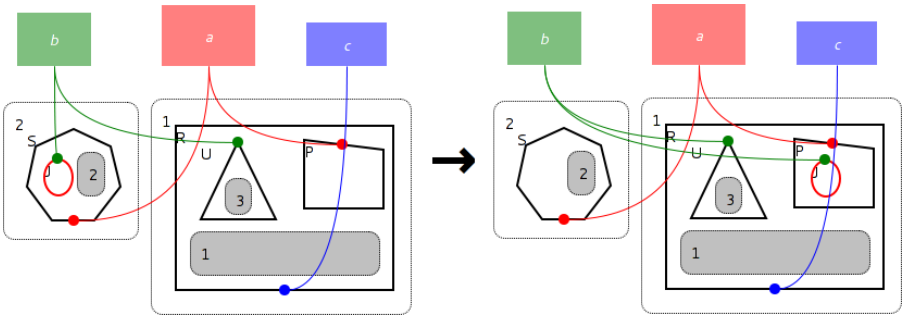
Dynamic behaviour is added to a bigraph model by adding a set of *reaction rules*. The *JobToSpool* rule shown in Fig. 2, in both the visual syntax of Big Red and the textual syntax of BigMC, allows a print job to be transferred from a user to a spool, adding an identifying link to connect users to their submitted print jobs. The *JobToPrinter* rule in Fig. 3 will transfer a job from the spool to a printer that is co-located with the user associated with that job. The full example contains also rules for removing completed jobs and allowing users to move between rooms and is available from `http://bigraph.org/papers/gcm2012`.

Note that the rules are *parametric* in the sense that both the user and the spool node in Fig. 2 contains a *hole*, which Big Red represents as shaded place graph leaf nodes, indexed by integers, and in the BigMC syntax as `$1` and `$2`. This means that both the user and spool node may contain other nodes (e.g. other job nodes) which in this example are left un touched by the rule. Interested readers are referred to [5] for detailed description of bigraphs and bigraphical reactive systems.




```
User[x].(Job[-] | $1) || Spool[y].$2 -> User[x].$1 || Spool[y].($2 |
Job[x]);
```

Fig. 2. The *JobToSpool* rule




```
Room[c].(Printer[a] | User[b].$3 | $1) || Spool[a].(Job[b] | $2) ->
Room[c].(Printer[a].Job[b] | User[b].$3 | $1) || Spool[a].$2;
```

Fig. 3. The *JobToPrinter* rule

4 Implementation

Big Red is implemented as an Eclipse *plugin*, which extends the Eclipse platform with additional file formats representing the objects of a bigraphical reactive system, wizards to create model files, and editors to modify them. In turn, Big Red defines several Eclipse *extension points*: these allow other plugins to contribute extensions to Big Red, adding support for new external tools and export formats.

Big Red's implementation of the bigraphical model strikes a balance between theoretical purity and practicality. No extensions to the bigraphical model are implemented, but the classes have been designed with extensibility in mind; as a result, adding new concepts to the model is easy. Indeed, Big Red uses these mechanisms to implement some of its own functionality — the information used to draw and position objects, for example, has no specific support in the underlying model.

The Eclipse platform includes comprehensive libraries to support the building of modelling tools. At the heart of these is the Graphical Editor Framework, a

toolkit for implementing model-view-controller-based editors. Big Red's bigraph and reaction rule editors are both built on this powerful and flexible component.

Quite apart from its specific support for modelling tools, Eclipse is a very portable and widely-used platform with a vibrant community and many pre-built features, which makes it an ideal choice for the rapid development of an editor.

4.1 Interacting with External Tools

Big Red defines a special extension point for plugins that want to operate on a complete bigraphical reactive system. When an extension registered with this point is activated, Big Red's user interface is suspended, and the extension takes over — essentially, this gives developers the ability to write subprograms that work with Big Red's model objects without having to delve too deeply into the workings of Eclipse.

The integration between Big Red and BigMC [6] is implemented in this way — as an external plugin which converts a Big Red model into its BigMC term language representation, executes BigMC as a subprocess, and parses the results back into Big Red model objects so that they can be visualised.

5 Conclusion

We have presented a brief introduction to the Big Red tool, and described the motivation for developing such a tool. Big Red and accompanying user documentation are available under the Eclipse Public License from `http://bigraph.org`.

Big Red is still under active development. We intend to integrate support for other bigraph tools, and — together with the University of Udine — we are working on developing Big Red into a generally-useful platform for building and hosting new tools for bigraphs.

References

1. L. Birkedal, S. Debois, E. Elsborg, T.T. Hildebrandt, and H. Niss. Bigraphical models of context-aware systems. In *Foundations of Software Science and Computation Structures*, pages 187–201. Springer, 2006.
2. A.J. Glenstrup, T.C. Damgaard, L. Birkedal, and E. Højsgaard. An implementation of bigraph matching. 2007.
3. C. Greenhalgh. bigraphspace, 2009.
4. T.T. Hildebrandt, H. Niss, and M. Olsen. Formalising Business Process Execution with Bigraphs and Reactive XML. In *COORDINATION'06*, volume 4038 of *Lecture Notes in Computer Science*, pages 113–129. Springer-Verlag, January 2006.
5. R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
6. G. Perrone, S. Debois, and T.T. Hildebrandt. A Model Checker for Bigraphs. In *ACM Symposium on Applied Computing 2012 – Software Verification and Testing Track*. ACM, 2012.

XL4C4D – Adding the Graph Transformation Language XL to CINEMA 4D

Ole Kniemeyer[1] and Winfried Kurth[2]

[1] MAXON Computer GmbH, Max-Planck-Str. 20, 61381 Friedrichsdorf, Germany
[2] Georg-August-University Göttingen, Department of Ecoinformatics, Biometrics and Forest Growth, Büsgenweg 4, 37077 Göttingen, Germany

Abstract. A plug-in for the 3D modeling application CINEMA 4D is presented which allows to use the graph transformation language XL to transform the 3D scene graph of CINEMA 4D. XL extends Java by graph query and rewrite facilities via a data model interface, the default rewrite mechanism is that of relational growth grammars which are based on parallel single-pushout derivations. We illustrate the plug-in at several examples, some of which make use of advanced 3D features.

1 Introduction

Most 3D modeling systems represent their 3D content as a scene graph. In general, this is a directed acyclic graph or even just a tree, where nodes contain geometry data and further properties, while edges define spatial and logical relations between nodes. E.g., the coordinate system of a node is typically inherited to its children, and often this also holds for properties like the color.

To create and modify the scene graph, 3D modeling applications typically do not only provide direct user interaction, but also built-in (textual or visual) programming languages. But none of these languages makes use of graph transformation techniques, although they suggest themselves for such a language, given the underlying scene graph.

What has been used for 3D scene graph creation are L(indenmayer)-systems [1], starting with the successful specification of 3D plant models, and nowadays this parallel string-rewriting formalism is directly supported by many 3D modeling applications. But as it is based on strings, it necessitates an additional interpretation step from strings to graphs. In previous research, we developed relational growth grammars as a rewriting mechanism for graphs which incorporates both the possibilities and ease of use of L-systems (but applied to graphs) and of true graph transformations based on single-pushout derivations [2, 3].

We also developed a textual programming language XL which extends Java by graph transformation syntax and semantics. This can be used to implement relational growth grammars, and there is a "reference implementation" within the Java-based open-source 3D platform GroIMP [4]. This software has been developed with the needs of relational growth grammars and plant modeling in mind, so it provides strong support for this field of application, while it has fewer general 3D features than traditional all-purpose 3D modeling systems.

Through the data model interface of XL, it is possible to let XL operate on any kind of graph. We will present an implementation for the scene graph of CINEMA 4D [5] as a plug-in, and we will show some examples of its application.

Graph transformations can not only be used at the topological level of scene graphs, but also to model the geometry itself which is usually described as a polygon mesh. Typical operations on a mesh like extrusion [6] or subdivision [7, 6] can be modelled as graph transformations. We haven't implemented such facilities for the plug-in, but it could be done based on the vv approach [8, 3].

2 Relational Growth Grammars and XL

In this section, we give a short overview of *relational growth grammars* (RGG) and the *XL programming language*, for more details see [2, 3]. RGG graphs are *typed attributed graphs with inheritance* [9], but without attributes for edges. This reflects the typing system of object-oriented programming, and it allows to store any kind of graphical and non-graphical information at nodes.

For the rewriting mechanism of RGG, the single-pushout (SPO) approach with parallel derivations turned out to be most suitable. In their pure shape, SPO productions cannot easily describe L-system-like rules: In L-systems, the left-hand side is completely replaced by the right-hand side, but SPO productions need nodes common to both sides. As a solution, we added the *operator approach* [10] to RGG which establishes *connection transformations* from nodes of the old host graph to nodes of the derived graph. These connection transformations are then used to create additional edges in the derived graph.

XL is a proper extension of Java. The main new features are *rule blocks* and *graph queries*. A rule block can be used everywhere where Java allows a normal code block, it is distinguished by square brackets. This example shows a block with a single rule of L-system kind, namely the rule for the snowflake curve [1]:

```
[ F(x) ==> F(x/3) RU(60) F(x/3) RU(-120) F(x/3) RU(60) F(x/3); ]
```

It has the following semantics: For each node of class `F`, remove it from the graph and insert a chain of seven new nodes. The new nodes are of the classes `F` and `RU`, and the chain is established by edges of the default type (indicated by whitespace). Due to the L-system kind, connection transformations are implicitly added so that the leftmost (rightmost) new `F` node inherits the parents (children) from the deleted `F` node or, if those parents (children) have also been deleted, the rule application successors of the parents (children).

Furthermore, the rule uses the variable `x`, which is set to the default attribute of the matched `F` node, to initialize the `F` nodes with `x/3`. In terms of Java, this is just a constructor invocation, so the new nodes are created by **new** `F(x/3)`. Also the `RU` nodes are initialized with numerical values. Now if `F(x)` nodes stand for lines of length `x` and `RU(a)` nodes for rotations about an angle `a`, all within a scene graph, this is the exact translation of the rule of the snowflake curve.

Tree-like patterns on the left-hand or right-hand side can be specified by surrounding the children of a node with square brackets (a traditional notion of L-systems). For example, the following rule creates a simple binary tree:

```
Tip ==> F(100) [RU(30) RH(90) Tip] [RU(-30) RH(90) Tip];
```

A `Tip` is replaced by a line segment of length 100 followed by two child branches, each of which starts with some rotations and ends with a new `Tip`.

Pure SPO productions are indicated by the rule arrow `==>>`. The rule

```
a:Monomer, b:Monomer ==>> if (condition) (a -bond-> b) else (a, b);
```

finds two `Monomer` nodes which need not have any relationship (the comma indicates a disconnected pattern), binds the matches to the identifiers `a`, `b`, and if a condition is fulfilled, creates a `bond`-typed edge, otherwise nothing happens.

Graph queries are like normal Java expressions, but they yield multiple values – one per match. They are enclosed by `(* *)` and otherwise have the same syntax as left-hand sides. E.g., with `(* a -bond-> b *)` we can check if there is a `bond`-typed edge between known nodes `a`, `b`. That could be used in the condition of the previous example to create `bond` edges only if they don't exist yet.

All graph operations of XL are defined on top of an abstract data model interface. By implementation of this interface, XL can operate on any kind of graph. The most sophisticated implementation exists for GroIMP, but there are also implementations for XML documents or minimalistic Sierpinski graphs [3].

3 The Plug-In XL4C4D and Its Application

The 3D modeling system CINEMA 4D provides a C++ SDK which allows to write custom plug-ins. With the help of the Java Native Interface as a bridge between Java and C++, we implemented XL's data model interface for CINEMA 4D's scene graph. The plug-in (available at [4]) provides a scene graph object `XLObject` which stores XL source code and a console where output from XL is shown and which also allows to interactively execute XL statements.

Fig. 1(a) shows a screenshot of CINEMA 4D with the snowflake example. The code properties of the single `XLObject` are opened. The complete code is

```
public class Main extends XLObject {
  public void init() [ ==>> ^ F(1000) RU(-120) F(1000) RU(-120) F(1000); ]
  public void grow()
    [ F(x) ==> F(x/3) RU(60) F(x/3) RU(-120) F(x/3) RU(60) F(x/3); ]
}
```

`grow` has been discussed in the previous section. `init` uses the special symbol `^` which in general stands for the root of the graph. For XL4C4D this is the `XLObject` node, so `init` adds the nodes of the initial snowflake triangle as descendants of that node. `init` is invoked interactively by selecting init in a drop-down menu of available methods and clicking on the button Invoke. Afterwards, one selects grow and clicks on Invoke as many times as desired.

The plug-in provides several standard node classes for L-system models (`F`, `RU`, `RL`, `RH`, `M` for pure movement), but also extended ones like `Sphere`, `Cube`, `Instance`, `Null` or even `Spring` for physics simulations. Each node of such a class corresponds to a CINEMA 4D object of suitable type, e.g., pure rotation and movement nodes are represented as Null objects in the scene graph.

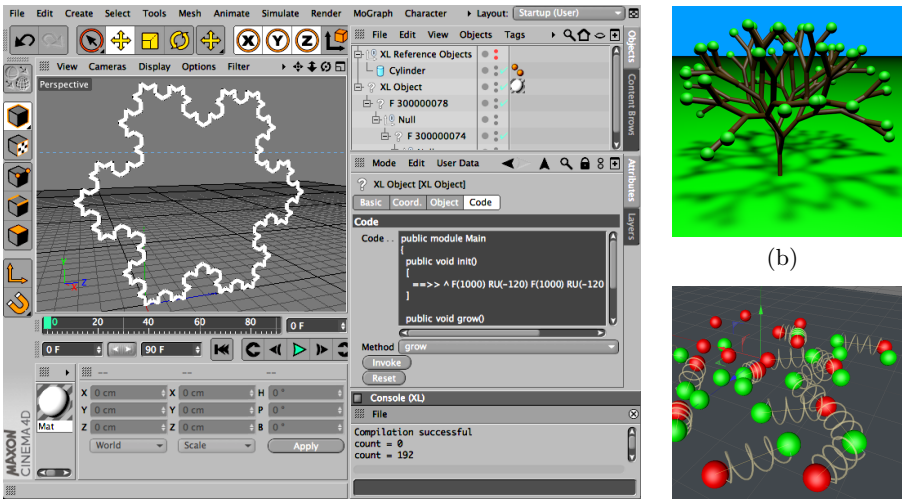


Fig. 1. (a) Screenshot showing the plug-in; **(b)** binary tree; **(c)** polymerization model

The binary tree model from Fig. 1(b) is based on the rule from the previous section. It uses the `Sphere` class to show tips as spheres. This is done by a module declaration, which is basically a simplified form of a complete class declaration:

```
public class Main extends XLObject {
  module Tip extends Sphere(20);
  public void init() [ ==>> ^ Tip; ]
  public void grow() [ Tip ==> F(100) [RU(30) RH(90) Tip]
                                      [RU(-30) RH(90) Tip]; ]
}
```

So far the examples don't use any of CINEMA 4D's advanced features. One such feature is the Dynamics module which provides a rigid body simulation. With its help we can create a toy model of polymerization based on the `Monomer` rule from above: A lot of `Monomer` spheres move around according to the laws of physics. At each frame (i.e., each simulation step), it is checked if two monomers come close to each other. If so, the rule triggers and creates a bond. It will also create a new spring object for the rigid body simulation so that the bond actually manifests itself in the simulation. The complete polymerization rule is

```
a:MonomerA, b:MonomerB ==>>
  if (empty((* a -bond-> b *)) && (distance(a, b) < 50)) (
    a -bond-> b, ^ Spring(a, b)
  ) else (a, b);
```

In addition to the original rule, we use two monomer types, and bonds are only created between monomers of different type. The setup of the simulation environment is done manually. The result after some simulation steps is shown in Fig. 1(c).

4 Discussion

The structure of CINEMA 4D's scene graph is well-suited for the application of XL. The shown examples could be adapted from their original GroIMP implementations [3] without major changes, and they could easily be combined with advanced features like Dynamics. In fact, the GroIMP implementation of the polymerization has to include some simple simulation rules for monomer movement which are superfluous for XL4C4D. There are a lot more features from animation to sophisticated rendering from which one can benefit.

On the other hand, the Java Native Interface introduces a performance bottleneck, and CINEMA 4D isn't able to handle very large amounts of objects or deep structures efficiently. So the system is not suitable for graphs of million nodes, which may appear in detailed plant models and which can be handled by GroIMP. E.g., the snowflake example allows only up to three iterations, reaching a graph depth of 384.

Therefore, in its current state, XL4C4D is a useful addition for CINEMA 4D if one wants to create and modify the graph of scenes within the typical domain of CINEMA 4D such as motion graphics, while GroIMP much better fits the needs of academic research.

References

1. Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer, New York (1990)
2. Kniemeyer, O., Barczik, G., Hemmerling, R., Kurth, W.: Relational growth grammars – a parallel graph transformation approach with applications in biology and architecture. In Schürr, A., Nagl, M., Zündorf, A., eds.: AGTIVE 2007. Volume 5088 of Lecture Notes in Computer Science., Springer (2008) 152–167
3. Kniemeyer, O.: Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling. PhD thesis, BTU Cottbus (2009)
4. Kniemeyer, O., Hemmerling, R., Kurth, W.: GroIMP http://www.grogra.de.
5. MAXON Computer GmbH: CINEMA 4D http://www.maxon.net.
6. Bellet, T., Poudret, M., Arnould, A., Fuchs, L., Gall, P.L.: Designing a topological modeler kernel: A rule-based approach. In: Shape Modeling International, IEEE Computer Society (2010) 100–112
7. Spicher, A., Michel, O., Giavitto, J.L.: Declarative mesh subdivision using topological rewriting in MGS. In Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A., eds.: ICGT. Volume 6372 of Lecture Notes in Computer Science., Springer (2010) 298–313
8. Smith, C., Prusinkiewicz, P., Samavati, F.F.: Local specification of surface subdivision algorithms. In Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: AGTIVE 2003. Volume 3062 of Lecture Notes in Computer Science., Springer (2003) 313–327
9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Secaucus, NJ, USA (2006)
10. Nagl, M.: Graph-Grammatiken: Theorie, Anwendungen, Implementierungen. Vieweg, Braunschweig (1979)

On Derivation Languages of DPO Graph Transformation Systems. Part 1: Introducing Derivation Languages

Nils Erik Flick

Fachbereich Informatik, MIN-Fakultät, Universität Hamburg, DE
email: flick AT informatik.uni-hamburg.de

Abstract. We investigate sequential derivation languages associated with injective double-pushout rewriting systems over finite graphs as a loose generalisation of the corresponding notions of free-labeled Petri net languages, in two variants: without termination rules or with a special rule that terminates a derivation as soon as it applies.
We obtain a first set of results comparing these languages with other well-known classes. In particular, we exhibit simulation relationships with automaton models to show that a class of languages of systems with rule-based termination properly contains the context-free languages as well as the derivation languages of type 0 grammars. All of these languages are decidable in nondeterministic space O(n log n), since that is sufficient to simulate the graph transformation system.

1 Introduction

It is somewhat unusual to consider graph transformation systems as generators of string languages; however, to motivate this paper we argue that these questions are interesting, in their own right and because steps towards better analysis, verification and also synthesis from behavioural specifications, of graph transformation systems are gaining importance. Recasting other known formalisms as special cases of graph transformation systems after forging some generic tools for understanding their languages is also a main motivation. The subject of languages also came up in the context of a more general discussion on the synthesis of graph transformation systems from infinite transition systems specifying the possible state transitions in the system, and can be of use in that context. One usually considers derivation languages of string grammars, often with the goal at providing a notion of controlled application of rules.

Graph transformation systems are related to non-deterministic automata in that events are state transitions, the occurrence of events being governed by the state of the system. If a certain subobject exists in it which happens to be the *precondition* of a rule a, that rule can be applied, which generates an event of type a. The transition rules also have postconditions, which after application replace the preconditions in the global state. The pre- and postcondition may have common parts, that must be present for the rule to apply, but are preserved. It is important to remember that in classical Petri nets, which are very

special transformation systems of the kind studied here [KKHK06], the *effect* of a transition is always the same and can be simply be added to the state. In general graphs, the preconditions for a certain event to occur and the description of the possible successor states are more complicated.

The paper is constructed as follows. Section 2 lists related work. Section 3 reviews the definitions of the underlying formalisms, which can also be found in the literature. Section 4 contains the results of this paper, which are inclusion relations with other language classes, and the undecidability of some problems which follows directly. Section 5 finishes the presentation with an outlook.

2 Related Work

In this section we have gathered some references, both old and new, about related concepts. Szilárd languages, as derivation languages of Chomsky grammars, and Petri net languages appeared in the seventies. There is a connection between both kinds of languages, as pointed out in [CRM77] – at least for context free languages, one can turn non-terminals into places, productions into transitions and obtain a Petri net that has the derivation language of the grammar as its set of firing sequences ending in a deadlock or empty marking.

We stress the connection to grammars because graph transformation systems with termination criterion can be used as graph grammars for generating sets of graphs, and graph grammars are also a generalisation of string grammars, where strings are encoded as linear labeled graphs [KKHK06]. Concepts of regularity and context-freeness have been formulated [Kre79] [CE95] [Ev97] in this context, see also [Hab92] for the related notion of hyperedge replacement grammars. A study of the derivation languages, however, seems to be missing.

A lot more is known about Petri net languages: [Bac11] answers a question raised in [Dar04], where net synthesis from infinite structures is handled. Their questions include finding a Petri net which generates a given language or possibly a superset of it, if an exact realisation is not possible. That question was found by Badouel and Darondeau to be decidable for deterministically context free languages, though undecidable for context free languages in general. Bachmann exactly characterises the languages for which Petri net synthesis is decidable, by semilinearity and prefix-closedness.

Hack [Hac76] has first studied Petri net languages. Jantzen [Jan79] has also studied languages of labeled, λ-free labeled and unlabeled Petri nets; the P-type languages of [Jan87] are languages of firing sequences, $P(N) = \{w \in T^* \mid m_0 \stackrel{w}{\Rightarrow}\}$ (the notation will be introduced in Section 3); the T-type languages are those ending in deadlocks: L-type languages are those ending in one of a set of designated final states and G-type languages are those that end in a state *covering*, i.e. including, a designated state; we will use a termination criterion generalising this. Free-labeled means that the labeling function does not identify any pair of distinct rules and is λ-free (i.e. does not assign the empty word λ); the classes of arbitrarily labeled and arbitrarily but λ-free labeled languages differ from their free-labeled counterparts and from each other in all four cases. Note

that Petri nets, which are injective DPO systems over multisets or labeled graphs without edges, have a monotony property which the arbitrary labeled graph transformation systems considered here lack, such that much more complicated behaviour is seen here; indeed, the formalisms under consideration in this paper are capable of performing arbitrary computations.

Going back to string grammars, derivation languages have also been studied by Monien [Mon77] for automata; all Szilárd languages of type 0 grammars (N, Σ, P, S), $Sz(G) = \{\sigma \in P^* | S \stackrel{\sigma}{\Rightarrow} w, w \in \Sigma^*\}$ are context sensitive languages. Höpner (better known as Jantzen) has studied the family $\mathcal{H}_1(Sz)$ of homomorphic images of Szilárd languages [Hö75] under length-preserving homomorphisms and found it to be endowed with more useful properties than the free-labeled kind. See [Mäk98] for a bibliography of older work. Mäkinen has published a number of articles on Szilárd languages of several kinds of grammars, and complexity of such languages.

3 Basic Definitions

In the following, though we generally assume familiarity with the vocabulary of graph theory, the basic definitions for double-pushout rewriting of graphs are recalled. We would like to refer the reader to the following sources: [HMP01] for several variants of DPO graph rewriting, one of which, called $DPO^{i/i}$ because both matches and right-hand sides of rules must be injective, is precisely the one investigated in this paper; [EPPH06] for the more general theory based on the notion of an adhesive high-level replacement category but also for the special cases graphs and labeled graphs, for the closely related notion of an adhesive category, which the categories of finite graphs and finite labeled graphs we recall here fit, [LS04]. Since the focus of this paper is on the concrete capabilities of injective DPO rewriting over finite graphs and finite labeled graphs, we present only definitions for finite (labeled) graphs and forego a more abstract foundation.

Definition 1. *A finite directed graph with self-loops and multiple edges, called* graph *in the sequel, is a tuple (V, E, s, t) of two finite sets V, E and two total functions called source, target $s, t : E \to V$.*

Definition 2. *A* finite labeled graph *is a tuple $G = (V, E, L, s, t, l_v, l_e)$, adding to the data defining a graph a finite set of labels L and two functions $l_v : V \to L$ and $l_e : E \to L$.*

Definition 3. *A* morphism *of (labeled) graphs is a pair of total functions $f = (f_V, f_E)$ that map nodes / edges, preserving the operations s, t, l, that is $s' \circ f_E = f_V \circ s$, $t' \circ f_E = f_V \circ t$, whilst labels are left unchanged: $l'_e \circ f_E = l_e$, $l'_v \circ f_V = l_v$.*

Categories $FinGraph$, $LFinGraph$ are obtained under component-wise composition of the morphisms. Graphs and labeled graphs can also be seen as two- and three-sorted algebras [EPPH06], which is useful when generalising the theory to other graph-like objects such as hypergraphs. Monomorphisms $m : X \hookrightarrow Y$,

$m \circ i = m \circ j \Rightarrow i = j$ are of special importance in the theory of DPO graph rewriting. In $FinGraph$ and $LFinGraph$, these are exactly the morphisms which are injective mappings component-wise.

Definition 4. *An initialised $DPO^{i/i}$* system $\mathcal{S}$ *over $(L)FinGraph$ is a pair $(\{U_t\}_{t\in T}, s_0)$ of a set of rules U_t indexed on a finite alphabet T of rule names. s_0 is the start graph and the rules U_t are pairs of monomorphisms $l_t : K_t \hookrightarrow L_t$, $r_t : K_t \hookrightarrow R_t$ from a common* interface *graph K_t to a* left-hand-side *graph L_t and a* right-hand-side *graph R_t.*

We are going to need a name for the set of possible states of a system. In principle, the set of isomorphism classes of finite graphs, respectively of finite graphs labeled over some alphabet including all labels to be used, is the right notion. We will let $\mathcal{G}$ denote one of these according to the context. Whenever a state is involved in a construction (notably Definition 6), a representant of the isomorphism class is meant.

The definition of a direct derivation is important, because it defines the dynamics of the systems. In the nomenclature of [HMP01], we are here in the process of defining $DPO^{i/i}$, double pushout rewriting with injective matches and injective (right-hand side morphisms of) rules.

The interface graph K_t of a rule is a subgraph of both the left hand side and the right hand side. The mechanism of replacement of the pattern L_t with the pattern R_t when applying a rule is as follows. A pushout, abstractly, is defined by a universal construction; we will just take a concrete construction describing a pushout in $(L)FinGraph$ as a definition.

Definition 5. *Given morphisms $C \xleftarrow{c} A \xrightarrow{b} B$, let D be the graph having as nodes $V_D = (c(V_A) \uplus b(V_A))/\{(c(v), b(v)) \mid v \in V_A\}$: the quotient of the disjoint union of the images of A via b resp. c by the least equivalence relation identifying those nodes that are the images of a common node of A. Edges likewise. The* pushout *of $C \xleftarrow{c} A \xrightarrow{b} B$ is $C \xrightarrow{c'} D \xleftarrow{b'} B$: D together with the compositions of the injections with the quotient morphism from the construction of D.*

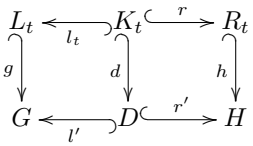
Clearly, the pushout always exists in $(L)FinGraph$. Moreover,monomorphisms push out to monomorphisms in any adhesive category. In the pushout, B and C are then simply glued together along the common subgraph A.

Definition 6. *If there is a monomorphism, called the match, $L_t \overset{g}{\hookrightarrow} G$ such that $K_t \overset{l_t}{\hookrightarrow} L_t \overset{g}{\hookrightarrow} G$ can be complemented by a graph D and morphisms $K_t \overset{d}{\hookrightarrow} D \overset{l'}{\hookrightarrow} G$ in a way that $D \overset{l'}{\hookrightarrow} G \overset{g}{\hookleftarrow} L_t$ is the pushout over $D \overset{d}{\hookleftarrow} K_t \overset{l_t}{\hookrightarrow} L_t$,*

let $D \overset{r'}{\hookrightarrow} H \overset{g'}{\hookleftarrow} R_t$ be the pushout over $D \overset{d}{\hookleftarrow} K_t \overset{r_t}{\hookrightarrow} R_t$.

We write $G \overset{t,g}{\Rightarrow} H$ or $G \overset{t}{\Rightarrow} H$.

As a double pushout *diagram:*



For graphs, concretely this means that nodes and edges in G are of three kinds each: a node/edge x is either
(a) The image of a node/edge in D, but not the image of a node/edge in L_t
(b) The image of a node/edge in L_t and the image of a node/edge in D
(c) The image of a node/edge in L_t, but not the image of a node/edge in D

The nodes and edges of (a) are not touched by the rewriting step, the nodes and edges of (b) are preserved and the nodes and edges of (c) are deleted.

The applicability of a rule given a match then comes down to the satisfaction of a *dangling edge condition*, which is checked on the (c) nodes and the incident edges: a node which is attached to an edge that is not deleted may not be deleted. This implies that the in- and outdegrees of any deleted node are always exactly those indicated in the rule. For labeled graphs, the labels must also match.

We define derivation sequences and a derivation relation between objects.

Definition 7. *Notation. We write*

$G \overset{w}{\Rightarrow}_* G'$ *whenever* w *is the empty string* λ *and* $G = G'$
or $w = tv$, $\exists G'' \in \mathcal{G} : G \overset{t,g}{\Rightarrow} G'' \wedge G'' \overset{v}{\Rightarrow}_* G'$.
$G \Rightarrow_* G'$ *whenever* $\exists w \in T^* : G \overset{w}{\Rightarrow}_* G'$
$G \overset{w}{\Rightarrow}_*$ *whenever* $\exists G \in \mathcal{G} : G \overset{w}{\Rightarrow}_* G'$
$G \Rightarrow_*$ *whenever* $\exists w \in T^* \; \exists G \in \mathcal{G} : G \overset{w}{\Rightarrow}_* G'$
$G \not\Rightarrow$ *whenever* $\neg \exists t \in T : \; G \overset{t}{\Rightarrow}$

We will occasionally subscript the name of the system or omit the asterisk.

Before we go on, we fix some conventions for Graphical Representation: The rule drawings in the examples are to be read as follows: each node or edge in the middle graph (the rule interface) is mapped to the one in the corresponding place on the left hand resp. right hand graph. Labels are encoded as node shadings or strings near the labeled nodes or edges. Sometimes edge directions are omitted in the graphical representation.

3.1 Derivation Languages

Graph transformation systems can be used as generators of languages. The languages considered here are not sets of graphs generated by a graph grammar, but instead the sets of possible sequences of rule applications, which are a revealing aspect of the dynamics of a system. A notion of termination will nevertheless be introduced, and it is one which is internal to the graph transformation system itself, in that termination is recognised by an additional transition rule which disables itself and all other rules.

We start out with the prefix-closed derivation language without termination. If T is the alphabet and s_0 is the start graph of the system $\mathcal{S}$, see Definition 4.

Definition 8. *Let $\mathcal{S} = (\{U_t\}_{t\in T}, s_0)$ be an initialised $DPO^{i/i}$ system over $FinGraph$. Then $\mathcal{L}_D(\mathcal{S}) := \{w \in T^* \mid s_0 \overset{w}{\Rightarrow}_{\mathcal{S}}\}$ is called the prefix-closed* derivation language without termination *of $\mathcal{S}$. We call $\mathcal{L}_D$ the class of all languages L such that $L = \mathcal{L}_D(U, s_0)$ for some $DPO^{i/i}$ system (U, s_0) over $FinGraph$.*

The \$-terminating derivation languages are defined as follows, $\$ \in T$ being a special rule label that is barred from occurring in words of that language.

Definition 9. *Let $\mathcal{S} = (\{U_t\}_{t\in T}, s_0)$ be an initialised $DPO^{i/i}$ system over $FinGraph$. Then $\mathcal{L}_\$(\mathcal{S}) := \{w \in (T \setminus \{\$\})^* \mid s_0 \overset{w\$}{\Rightarrow}_{\mathcal{S}}\}$ is called the* \$-terminating derivation language *of $\mathcal{S}$. $\mathcal{L}_\$$ is the class of all languages L such that $L = \mathcal{L}_\$(U, s_0)$ for some $DPO^{i/i}$ system (U, s_0) over $FinGraph$.*

We could also define $\mathcal{L}'_\$$, with the condition that \$ must disable all other rules and itself. But any $L \in \mathcal{L}'_\$$ can be seen to be in $\mathcal{L}_\$$ by modifying the system such that $L_\$ = L'_\$ \uplus \{v_\$\}$ and $s_0 = s'_0 \uplus \{v_\$\}$ and that special node $v_\$$ is present in the interface K_t for all other transitions, the node $v_\$$ always bearing the unique label \$. Even concurrent executions are preserved by this construction, since side conditions do not influence the concurrent applicability of rules, from the theory of parallel independent rules.

For every initialised $DPO^{i/i}$ transformation system over finite labeled graphs a behaviorally *equivalent* one exists over unlabeled graphs, even preserving concurrency. One possible translation represents the differently labeled nodes and edges (node and edge labels disjoint) as nodes to which m loops and n auxiliary nodes are attached, with numbers $n = 1 + |L| - m$ depending only on the label so as to create incomparable subobjects for distinctly labeled nodes and edges. The principle is illustrated in Figure 1, where m is 2 for a, 3 for b and 1 for label c, and there are 3 labels in total.

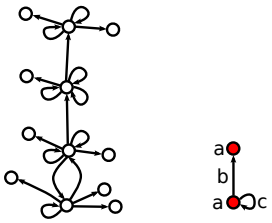


Fig. 1: An unlabeled graph representing a labeled graph.

We think it is safe now to let the statement of equivalence of labeled and unlabeled systems stand without a proof – an initialised system only uses a fixed set of labels, and states are only considered up to isomorphism. The reverse direction is obvious, as unlabeled graphs can trivially be seen as labeled ones.

4 Results

Unlike free-labeled Petri net languages and Szilárd languages, the derivation languages of DPO systems over finite graphs do contain all regular languages and possess basic closure properties already without having to relabel any rules. Only the very simple case of closure under intersection is in this article, which is needed to show an undecidability result at the end of Subsection 4.2. For several other closure properties, see [Fli12].

4.1 Languages in $\mathcal{L}_D$ and $\mathcal{L}_\$$

From the definitions, it is trivial that for any system S, the prefixes of terminating derivations are in $\mathcal{L}_D(S)$.

Proposition 1. $pref(\mathcal{L}_\$(S)) \subseteq \mathcal{L}_D(S)$.

Proof. Immediate. □

The relationship between the classes is also easy to see. It is worth noting (see above) that systems over the ordinary finite graphs have the same expressivity as systems over the labeled graphs; therefore we will work with less tedious constructions for labeled graphs.

Proposition 2. $\mathcal{L}_D \subsetneq \mathcal{L}_\$$.

Proof. Just add a termination rule which is enabled in the beginning and consumes a special precondition, which cannot be consumed by the other rules. One the other hand, there are non-prefix closed languages in $\mathcal{L}_\$$. □

We give some examples of such languages. The system of Figure 2 generates a^{2^n}. It is quite easy to see that a^1 and a^2 are executed deterministically and activate \$; when the marker has reached the loop, it can travel around the loop. The portion of the loop still ahead of the marker is not modified by the rule, while each edge left behind is doubled. Therefore, after reaching an end state again in n hops, the loop is now twice as long, a has already been executed $2n$ times in total and must be executed another $2n$ times to reach the next end state. The system in Figure 3 deterministically generates $a^n b^n c^n$ by letting a build a chain of edges as long as the blue and red markers are together. Then repeated applications of b eventually bring the blue marker back home and double every one of the n edges in the chain. Then the yellow marker can be moved by c, and when that is done, the red and the yellow marker can be removed for termination. That is also possible after zero steps, disabling further applications immediately.

Results like the undecidability of termination of DPO graph rewriting systems (without a fixed start graph, [Plu98]) are already well known. Indeed, DPO systems over graphs are very powerful, enough so to simulate the workings of a Turing machine.

After simulating a successful calculation with silent rules, can read the word off the input tape using the non-silent rules, so the images of the derivation

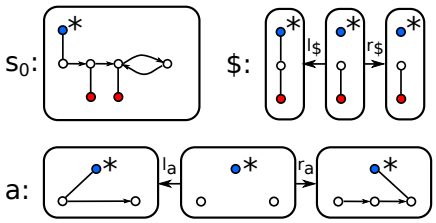


Fig. 2: A system generating $\mathcal{L}_{\$}(\mathcal{S}) = \{a^{2^n} \mid n \in \mathbb{N}\}$

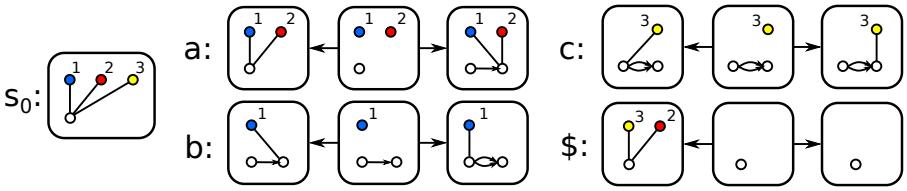


Fig. 3: A system generating $\mathcal{L}_{\$}(\mathcal{S}) = \{a^n b^n c^n \mid n \in \mathbb{N}\}$

languages $\mathcal{L}_{\$}$ under arbitrary, also deleting homomorphisms are indeed all recursively enumerable languages. If $\mathcal{H}$ denotes closure under arbitrary homomorphisms and $\mathcal{RE}$ the class of recursively enumerable languages, Sz the Szilárd languages and $NSPACE(f(n))$ the class of all languages acceptable by a nondeterministic Turing machine in space $O(f(n))$,

Proposition 3. $\mathcal{H}(\mathcal{L}_{\$}) = \mathcal{RE}$, $\mathcal{H}(\mathcal{L}_D) = pref(\mathcal{RE})$.

Proof. By constructing a DPO graph transformation system from a Turing machine with one input tape and one work tape, with one λ-labeled rule for each element of the transition relation, one λ-labeled rule that extends the work tape as needed, and finally a set of rules labeled over T, which read the input once (use a marker initially attached to the first input cell, and the termination rule to read the end-of-tape symbol) if an accepting state has been reached. □

Proposition 4. $Sz \subsetneq \mathcal{L}_{\$}$.

Proof. The inclusion of the Szilárd languages of arbitrary Chomsky grammars in $\mathcal{L}_{\$}$, which are the sequences of free-labeled rule applications can be seen by encoding each word composed of terminals and nonterminals as a linearly shaped graph with two dummy nodes at the ends, and each production as a rewriting rule. Termination, that is derivation of a terminal word, can be recognised internally by always attaching all non-terminals to a single special node labeled \$ which is preserved by all production rules, and putting an isolated \$ labeled node in $L_{\$}$ but not in $K_{\$}$. Szilárd languages never contain the empty word since derivation always starts at the non-terminal string S. □

Proposition 5. $\mathcal{L}_{\$} \subseteq NSPACE(n \log n)$.

Proof. Every derivation of length n can clearly be simulated in $O(n \log n)$ space by a nondeterministic Turing machine, because every graph can be encoded by a string of length $O((v + e) \log v)$ as an adjacency list; each configuration of the system reachable in n steps can only have $c_1 \cdot n$ nodes and $c_2 \cdot n$ edges. The subgraph matching can be done by (nondeterministically) marking off edges and nodes as images of the elements in L_t for each transition; doing this and checking the dangling edge condition can be performed with linear space requirement. □

We show that all context-free languages $\mathcal{CF}$ are $\mathcal{L}_\$$ languages by demonstrating that the runs of a real-time push-down automaton, that is without any λ transitions, can be simulated by derivations in a graph transformation system in such a way that each direct derivation can simulate all transitions of the PDA that read the same input symbol, also preserving determinism. Because only injective matches are allowed, some extra care is needed for encoding self-loops of the automaton. They must be realised with symmetries.

We also provide a visual example of the PDA simulation construction. Figure 4 shows a push-down automaton that accepts the language $\{a^n b^n \mid n \in \mathbb{N}\}$ with end states q_0 and q_3. Figure 5 shows an encoding of the stack as a graph. Figures 6, 7, 8, 9, 10 sketch the corresponding DPO system.

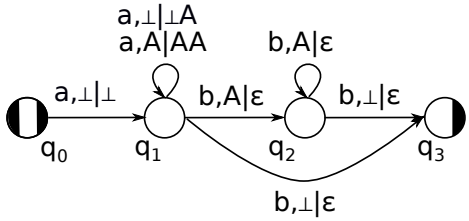


Fig. 4: An λ-free push-down automaton with end states q_0, q_3 and start state q_0 for the language $\{a^n b^n \mid n \in \mathbb{N}\}$

For the following theorem we present the construction of the equivalent graph transformation system and argue its validity verbally and graphically, but without formal rigour.

Theorem 1. *$\mathcal{CF} \subseteq \mathcal{L}_\$$, and from a push-down automaton one obtains a $DPO^{i/i}$ graph transformation system S having the same language as derivation language. If the automaton is deterministic, so is S.*

Proof. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Q_f)$ be a PDA without λ transitions accepting the language L. Q is the finite set of states of the finite control, Σ and Γ are finite alphabets, $\delta \subseteq Q \times \Sigma \times \Gamma \times \Gamma^* \times Q$ is the transition relation, q_0 $q_0 \in Q$ is the start state and $Q_f \subseteq Q$ is the set of end states. A configuration of P is a triple $(q, \gamma, w) \in Q \times \Gamma^* \times \Sigma^*$ representing the finite state, the stack and the remaining input. $(q, \gamma A, aw) \vdash_P (q', \gamma g, w)$ iff $(q, a, A, g, q') \in \delta$. g is always a string of either zero, or one, or two symbols.

Let s_0 be the initial state of the graph transformation system, as shown in the example (Figures 6 and 7). It is composed of a representation of the initial stack, the finite control and assorted other nodes and edges needed to provide the desired behaviour.

- The symbol nodes. One needs one specially labeled node for each symbol. We cannot match nodes labeled A, A' with $A \neq A'$ generically when working with labeled graphs. So for each node labeled with a symbol $A \in \Gamma$ there is a unique node pointing to it. We will call that the symbol node of A.
- The stack. Our representation of the stack will have a backbone, whose nodes alternate between being connected to a *cell* node (which itself points to the symbol node) and potentially to a top-of-stack marker.

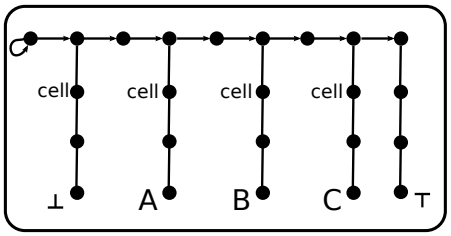


Fig. 5: Representation of the stack $\perp ABC$

- The finite control:
  - The state nodes. One node $(v_q, 0)$ is needed for each state $q \in Q$, and another node $(v_q, 1)$ if q has a self-loop.
  - The transition nodes. One node $(v_d, 0)$ or two nodes $(v_d, 0)$, $(v_d, 1)$ for each element $(q, a, g, g', q') = d \in \delta$, attached to v_q and $v_{q'}$ in the manner indicated in the picture: in case one of q and q' has a self-loop, two nodes are needed; in case $q = q'$, only one is created but it is attached with arrows going both ways.
  - A node v_{marker} labeled *marker* attached to v_{q_0} initially and to the current state during the simulation, and one node each, labeled *end*, attached to v_q where $q \in Q_f$.
- For each transition node, the following supplementary elements encode the instructions:
  - one edge labeled *read*, one edge labeled *top of stack*, and one edge labeled *write* indicate the effect of the rule, as in Figure 8.
  - The portion to be written on the stack is attached to a chain of virtual backbone nodes. There are always five of them, since the mechanism has to be the same whether zero, one or two symbols are written.
  - The top of stack node is always attached to either the first, the third or the fifth cell node on the virtual backbone. The *top of stack* edge of the transition node indicates, which one. This alternating layout is also shared by the representation of the actual stack.

- Every other backbone node (virtual of not) points to a *cell* node serving as a pointer to a symbol node.

A transition of the PDA is effected by searching a match that would map the node marked with a number 1 in Figure 8, to the transition's node (v_d, i), and performing the DPO derivation.

Define the relation $A \subseteq (Q \times \Gamma^*) \times \mathcal{G}$ such that $((q_i, \gamma), G) \in A$ iff the graph G_{marker} consisting of a node v_1 labeled $*$ and a node v_2 labeled q is source of subobject i of G in such a way that $i(v_2)$ is the node (q_i, n) for some $n \in \mathbb{N}$, and a basic bare-bones representation of the stack γ, as and exemplified in 5, is subobject of G in essentially one way, and no other possible stack representation is a subobject of G.

It remains to be proven that the construction provides that $(c, s) \in A \wedge (aw, c) \overset{a}{\vdash} (w, c') \Rightarrow \exists s', g : \ s \overset{a,g}{\Rightarrow} s' \wedge (c', s') \in A$: for the induction over the length of w (there are no λ steps), one mainly needs to show that this simulation property is preserved over one step.

As induction assumption, we posit that the state q a stack γA_i is represented in the DPO system's state s. The node pointing to a cell with the symbol $A_i \in \Gamma$ above the *read* edge is part of the one legitimate stack representation of γA_i, ending in $\top$, since the old connection to $\top$ is broken when applying the rule. The induction basis stands: by construction, the assumption is true in s_0, where the stack $\bot$ is represented.

The induction step goes like this: after extending the stack in the manner of rule a (Figure 8), there are three cases to be considered, depending on the number of symbols to be put back on the stack after removing the old top cell.

But they are all very similar. If nothing is written back, that is encoded in a part of the state graph which is never changed, and where the *top of stack* edge points to the node linked to the one right of the upmost virtual backbone node (number 2 in Figure 8). This gets linked to the predecessor, in the real backbone, of the node pointing to the cell that was read from the stack. If one symbol is written back, the *top of stack* edge of the transition points to the node linked to the one right of the third virtual backbone node, also leading to one valid stack representation being subobject of the new state graph, and if two symbols are written back, then all of the new backbone nodes actually have a use because the fifth will bear the top-of-stack marker.

Please note there are *state invariants* making the analysis easier: there is always only one connection to the single $\top$ node in the graph. The $\top$ is always attached to a node which has only one path to the bottom of the actual stack. Also note that whenever only one transition is possible for the automaton, only one match can be found. □

This is in contrast to the λ-free labeled languages of Petri nets, by an argument reported in [Hac76] and also used there to show that not even λ-free labeled inhibitor nets generate all context-free languages.

Proposition 6. $\mathcal{CF} \subsetneq \mathcal{L}_\$$.

Proof. To see that the inclusion is proper, take the example $\{a^{2^n} | n \in \mathbb{N}\} \in \mathcal{L}_\$$. □

The following figures show part of the initial state s_0 of the graph transformation system corresponding to Figure 4. The part represented in Figure 6 is the finite control of the automaton. All state transitions have three arcs, labeled *read*, *write* and *top of stack*. We have drawn them for only one of the transitions.

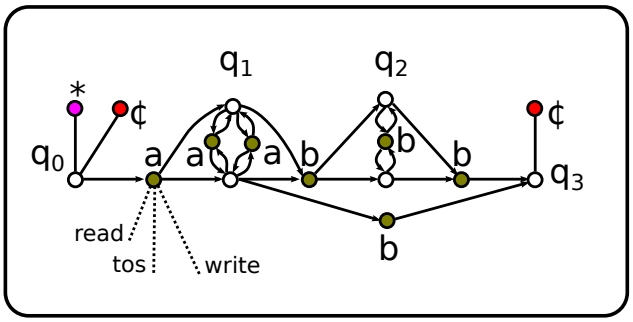


Fig. 6: The part of s_0 encoding the finite control. The q_i are not node labels, but serve to explain the encoding.

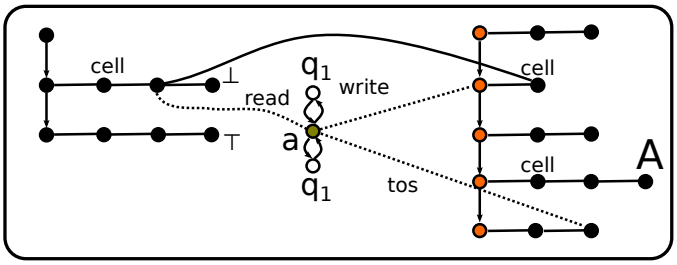


Fig. 7: The part of s_0 encoding the stack, and the instructions for one of the state transitions. The part to the left of the figure, which encodes the stack with only one $\perp$ symbol in it, exists only once in the whole start state s_0. The part to the right, with the "instructions", exists once for every transition in δ.

There are of course as many rules as there are input symbols, plus the end rule. The end rule removes the current state marker, but only if the simulation is currently in an end state. Termination with empty stack, without end state can also be checked.

$DPO^{i/i}$ systems over finite graphs are really much more general than push down automata, and generalisations immediately suggest themselves. For example, it is possible to build up a branching stack, as in the "concurrent finite stack

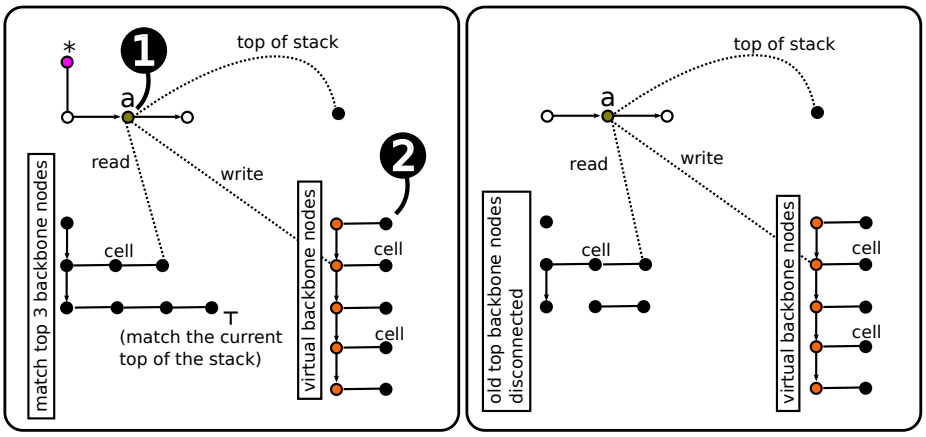


Fig. 8: The rule for a labeled transitions. Left hand side L_a, interface K_a

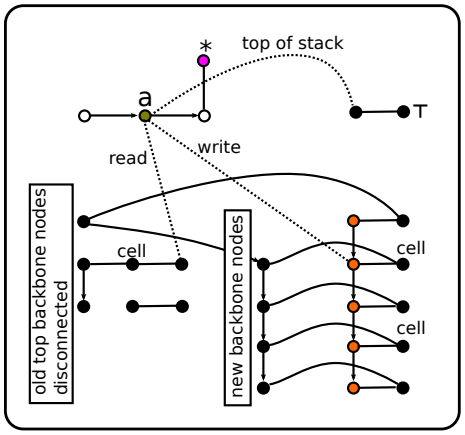


Fig. 9: The rule for a labeled transitions. Right hand side R_a

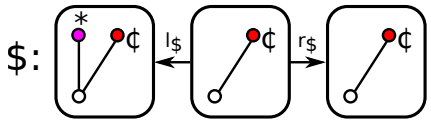


Fig. 10: The end rule

automata" (CFSA) of [BBH11], but it is not clear whether all CFSA languages can be obtained here. That would be the case if λ transitions were avoidable in CFSA. It is also possible to encode various schemes of multiple stacks, but again no λ transitions can be simulated directly.

4.2 Consequences

We immediately obtain decidability results. While the word problem is obviously decidable, many other questions are not decidable for $\mathcal{L}_D$ and $\mathcal{L}_\$$, because the corresponding questions for context free languages, given as grammars or automata, can be reduced to questions for derivation languages of DPO systems over finite graphs. Now,

Lemma 1. *$\mathcal{L}_D$ is closed under intersection, so is $\mathcal{L}_\$$*

Proof. Given two systems S_1 and S_2 over unlabeled graphs with rules indexed over the same alphabet, we map S_1 and S_2 to labeled graph transformation systems for graphs labeled over disjoint alphabets. Then the rules of both systems are composed via disjoint unions for graphs and morphisms. The resulting system is initialised with the disjoint union of the start graphs. One verifies that this system generates the intersection of the languages of S_1 and S_2. □

Proposition 7. *It is not decidable in general whether $\mathcal{L}_\$(\mathcal{S}) = T^*$. The equivalence and inclusion problems are also undecidable. Regularity is also undecidable. The emptiness problem $\mathcal{L}_\$(\mathcal{S}) = \emptyset$ is not decidable.*

Proof. Because of Proposition 6, questions undecidable [Kud04] for context free languages (given as PDA or equivalently as context free grammars) are not decidable for $\mathcal{L}_\$$ languages. Because of Lemma 1 furthermore there is an effective way of constructing a system generating the intersection, and it is undecidable whether the intersection of two context free languages is empty. □

5 Conclusion and Outlook

We have examined the classes defined here for some closure properties. The findings will appear in a separate paper.

The question whether $\mathcal{L}_\$$ is equal to the context sensitive languages $\mathcal{CS}$ had to remain open. We have not found anything that would preclude it, but also not yet been able to relate the two.

Only sequential languages were considered, concurrency issues were entirely ignored in the making of this paper, since they are less relevant when comparing to other classes of sequential languages. That said, double pushout graph rewriting is geared towards describing concurrent rule applications too, so step languages should also be studied.

We hope the investigation of derivation languages will continue to prove a fruitful subject, as one can study hierarchies of such languages by restricting the systems appropriately.

Also, we are currently investigating reachability graphs of the same systems, which can also be defined in analogy to the Petri net case, and generalising some aims of net synthesis – though the questions can not necessarily be addressed by the same means as net synthesis.

References

[Bac11] Jörg P. Bachmann, *Characterization of Petri net languages*, Proceedings of the international workshop CS&P 2011 September 28–30, Pułtusk, Poland (2011), 17–28.

[BBH11] Martin Berglund, Henrik Björklund, and Johanna Högberg, *Recognizing shuffle languages*, LNCS (2011), 142–154.

[CE95] Bruno Courcelle and Joost Engelfriet, *A logical characterization of the sets of hypergraphs defined by hyperedge replacement grammars*, Mathematical Systems Theory **28** (1995), no. 6, 515–552.

[CRM77] Stefano Crespi-Reghizzi and Dino Mandrioli, *Petri nets and Szilard languages*, Information and Control **33** (1977), no. 2, 177–192.

[Dar04] Philippe Darondeau, *Unbounded Petri net synthesis*, Lectures on Concurrency and Petri Nets (2004), 1–20.

[EPPH06] Hartmut Ehrig, Julia Padberg, Ulrike Prange, and Annegret Habel, *Adhesive high-level replacement systems: A new categorical framework for graph transformation*, Fundamenta Informaticae **74** (2006), no. 1, 1–29.

[Ev97] Joost Engelfriet and Vincent van Oostrom, *Logical description of context-free graph languages*, Journal of Computer and System Sciences **55** (1997), no. 3, 489–503.

[Fli12] Nils Erik Flick, *On derivation languages of DPO graph transformation systems. part 2: Closure properties*, 2012, Accepted for GCM 2012, Bremen.

[Hab92] Annegret Habel, *Hyperedge replacement: grammars and languages*, vol. 643, Springer, 1992.

[Hac76] Michel Hack, *Petri net languages*, Tech. Report 159, Massachusetts Institute of Technology, 1976.

[HMP01] Annegret Habel, Jürgen Müller, and Detlef Plump, *Double-pushout graph transformation revisited*, Mathematical Structures in Computer Science **11** (2001), no. 05, 637–688.

[Hö75] Matthias Höpner, *Eine Charakterisierung der Szilardsprachen und ihre Verwendung als Steuersprachen*, Gl-4.Jahrestagung (D. Siefkes, ed.), LNCS, vol. 26, Springer Berlin / Heidelberg, 1975, pp. 113–121.

[Jan79] Matthias Jantzen, *On the hierarchy of Petri net languages*, RAIRO: Theoretical informatics **13** (1979), 19.

[Jan87] ———, *Language theory of Petri nets*, Petri Nets: Central Models and Their Properties (1987), 397–412.

[KKHK06] Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske, *Some essentials of graph transformation*, Recent Advances in Formal Languages and Applications **25** (2006), 229–254.

[Kre79] Hans-Jörg Kreowski, *A pumping lemma for context-free graph languages*, Graph-Grammars and Their Application to Computer Science and Biology, Springer, 1979, pp. 270–283.

[Kud04] Manfred Kudlek, *Context-free languages*, Studies in Fuzziness and Soft Computing **148** (2004), 97–116.

[LS04] Stephen Lack and Paweł Sobociński, *Adhesive categories*, Foundations of software science and computation structures, Springer, 2004, pp. 273–288.

[Mäk98] Erkki Mäkinen, *A bibliography on Szilard languages*.

[Mon77] Burkhard Monien, *About the derivation languages of grammars and machines*, Automata, Languages and Programming (1977), 337–351.

[Plu98] Detlef Plump, *Termination of graph rewriting is undecidable*, Fundamenta Informaticae **33** (1998), no. 2, 201–209.

On Derivation Languages of DPO Graph Transformation Systems. Part 2: Closure Properties

Nils Erik Flick

Fachbereich Informatik, MIN-Fakultät, Universität Hamburg, DE
email: flick AT informatik.uni-hamburg.de

Abstract. We investigate sequential derivation languages associated with injective double-pushout rewriting systems over finite graphs, as a loose generalisation of the corresponding notions of free-labeled Petri net languages. Both the class of the prefix-closed languages of rule application sequences without termination rules and a class of languages with rule-based termination are found to possess several interesting closure properties, whose proofs are sketched in this article.

1 Introduction

Double pushout graph transformation systems, introduced [EPS73] as an algebraic approach to graph rewriting has been extended beyond graphs. In the present paper we will concentrate exclusively on rule application sequences that occur in the rewriting of finite graphs. This means we study string languages that describe the possible rule applications and not sets of graphs generated by a graph grammar. These families of languages possess some closure properties which are shown in this paper. They differ from the languages of Petri nets in that unlabeled variants are equivalent to λ-free labeled variants, and the images under arbitrary homomorphisms are all recursively enumerable languages.

A motivation for their study has been given in [Fli12]: since graph transformation systems can express diverse other formalisms, they can be used as a unifying tool. Also, knowledge about the capabilities of the formalism is useful when using it in designing systems. The paper is structured as follows:

Section 3 contains the basic definitions for double pushout transformation of graphs, on the concrete level needed for the exposition. Section 2 gives literature references to existing publications dealing with language theory of special graph transformation systems, namely Petri nets, and related fields. Section 4 contains the results of this paper, selected closure properties of derivation languages obtained by applying operations to graph transformation systems, providing proof sketches. Section 5 concludes the presentation with an outlook.

2 Related Work

In this section, we list some work of note. There is not much to be found on languages of more general systems, most is on Petri nets. A Petri net, more

precisely a P/T net, is an initialised rewriting system over multisets. It can be viewed as a comparatively simple injective DPO system [KKHK06] [Cor95], with very special properties. Rule activation is monotonous with respect to the partial order on states, states and transition effects are described as integer vectors and the effect of a rule or transition is calculated simply by vector addition, and the special properties allow the synthesis of nets from languages [Dar98,Dar04]. Petri nets have been related by their language theory to several other formalisms such as string grammars, and extensions of regular expressions [CRM77,Gis81].

Several authors have studied the sets of firing sequences or labeled firing sequences generated by Petri nets, with termination recognised by the reaching of a final marking. One of the first such studies was by Hack in 1976 [Hac76]. Hack's report already contains, and compiles, for example the work of Peterson [Pet76], many interesting theorems, including closure and non-closure properties.

Jantzen [Jan79] has also studied languages of labeled, λ-free labeled and unlabeled Petri nets; the P-type languages of [Jan87] are languages of firing sequences, i.e. $P(N) = \{w \in T^* \mid m_0 \stackrel{w}{\Rightarrow}\}$(the notation will be introduced in Section 3); the T-type languages are those ending in deadlocks. L-type languages are those ending in one of a finite set of designated final states and G-type languages are those that end in a state *covering*, i.e. including, a designated state; we will use a termination criterion generalising this. Free-labeled means that the labeling function does not identify any pair of distinct rules by assigning them the same label, and no label is λ (empty); the classes of arbitrarily labeled and arbitrarily, but λ-free labeled languages differ from their free-labeled counterparts and from each other in all four cases. The range of possible termination conditions was extended by Gaubert and Giua [GG99], who considered semilinear and not just finite sets of end markings. Starke [Sta78] investigated free terminal Petri net languages with a finite set of terminal markings and even found a characterisation by closure properties. Petri nets also offer the possibility of definition concurrent activation of rules: step languages of Petri nets under various firing policies were studied in [JZ08]. See also our article [Fli12] for more sources, also for references to the related notion of Szilárd languages.

3 Basic Definitions

In this section, we recall the basic definitions for double pushout rewriting of graphs. First of all, we must define graphs, which are the objects being rewritten.

Definition 1. *A finite directed graph with self-loops and multiple edges, called* graph *in the sequel, is a tuple (V, E, s, t) of two finite sets V, E and two total functions called source, target $s, t : E \to V$.*

Definition 2. *A finite* labeled graph *is a tuple $G = (V, E, L, s, t, l_v, l_e)$, adding to the data defining a finite graph a set of labels L, which can always be assumed finite, and two functions $l_v : V \to L$ and $l_e : E \to L$.*

If in the sequel, we leave nodes and edges not explicitly labeled, these all have the same label, which is none of those explicitly given.

The relationships between graphs, which play an important role in rewriting since it is necessary to express in what precise way one graph is a subgraph of another, are expressed in terms of structure preserving mappings.

Definition 3. *A* morphism *of (labeled) graphs is a pair of total functions* $f = (f_V, f_E)$ *that map nodes and edges respectively, whilst labels are left unchanged, preserving the operations* s, t, l*, that is* $s' \circ f_E = f_V \circ s$*,* $t' \circ f_E = f_V \circ t$*,* $l'_e \circ f_E = l_e$*,* $l'_v \circ f_V = l_v$*.*

Finite graphs and labeled finite graphs together with these morphisms form categories *FinGraph* and *LFinGraph*. The whole theory of graph rewriting according to the approach presented below was generalised to other kinds of objects and morphisms, but we deal only with graphs here. Monomorphisms $m : X \hookrightarrow Y$, $m \circ i = m \circ j \Rightarrow i = j$ are especially important. In our categories, these are exactly the morphisms which are injective mappings component-wise, i.e. do not identify the images of two nodes or edges.

We are going to need a name for the set of possible states of a system. The set of isomorphism classes of finite graphs or, following the context, finite graphs labeled over some alphabet including all labels to be used will be denoted by $\mathcal{G}$. Whenever we speak of graphs, it is understood that these graphs are labeled; every labeled DPO system can be translated to an unlabeled one which is fully equivalent even if concurrent behaviour is considered [Fli12], allowing us to use the results in the unlabeled case.

Double pushout graph rewriting comes in several variants [HMP01]. In the nomenclature of [HMP01], we are interested in $DPO^{i/i}$ systems. That is, matches and left-hand sides of rules are injective, in fact every transformation rule t is specified using injective morphisms l_t, r_t from a common interface graph K_t to the left-hand side L_t and right-hand side R_t graphs.

Definition 4. *A* rule $L_t \overset{l_t}{\hookleftarrow} K_t \overset{r_t}{\hookrightarrow} R_t$ *is a pair of monomorphisms with common source,* $l_t : K_t \hookrightarrow L_t$*,* $r_t : K_t \hookrightarrow R_t$*.*

The interface graph K_t is a subgraph of both the left hand side and the right hand side. The mechanism of replacement of the pattern L_t with the pattern R_t when applying a rule is explained below, in the concrete case of labeled or unlabeled graphs. First, we define the systems that are the object of our study.

Definition 5. *An initialised* $DPO^{i/i}$ *system* $\mathcal{S}$*, or* system *for short, is a pair* $(\{U_t\}_{t \in T}, s_0)$ *of a set of rules* U_t *(Definition 4) indexed on a finite alphabet* T *of rule labels.* $s_0 \in \mathcal{G}$ *is the start graph.*

Next, we move on to the dynamics of the systems. The definition of the actual transformation steps, customarily called direct derivations, is recalled.

Definition 6. *Given morphisms* $C \overset{c}{\hookleftarrow} A \overset{b}{\hookrightarrow} B$*, let* D *be the graph having as nodes* $V_D = (c(V_A) \uplus b(V_A))/\{(c(v), b(v)) \mid v \in V_A\}$*: the quotient of the disjoint union of the images of* A *via* b *resp.* c *by the least equivalence relation identifying*

those nodes that are the images of a common node of A. Edges likewise. The pushout *of* $C \overset{c}{\leftarrow} A \overset{b}{\rightarrow} B$ *is* $C \overset{c'}{\rightarrow} D \overset{b'}{\leftarrow} B$*: D together with the compositions of the injections with the quotient morphism from the construction of D.*

Clearly, the pushout always exists in $(L)FinGraph$. Moreover, monomorphisms push out to monomorphisms in any adhesive category. In the pushout, B and C are then simply glued together along the common subgraph A.

Definition 7. *If there is a monomorphism, called the match,* $L_t \overset{g}{\hookrightarrow} G$ *such that* $K_t \overset{l_t}{\hookrightarrow} L_t \overset{g}{\hookrightarrow} G$ *can be complemented by a graph D and morphisms* $K_t \overset{d}{\hookrightarrow} D \overset{l'}{\hookrightarrow} G$ *in a way that* $D \overset{l'}{\hookrightarrow} G \overset{g}{\hookleftarrow} L_t$ *is the pushout over* $D \overset{d}{\hookleftarrow} K_t \overset{l_t}{\hookrightarrow} L_t$*,*
let $D \overset{r'}{\hookrightarrow} H \overset{g'}{\hookleftarrow} R_t$ *be the pushout over* $D \overset{d}{\hookleftarrow} K_t \overset{r_t}{\hookrightarrow} R_t$*.*
We write $G \overset{t,g}{\Rightarrow} H$ *or* $G \overset{t}{\Rightarrow} H$*.*

Remark 1. For graphs, concretely this means that nodes and edges in G are of three kinds each: a node/edge x is either (a) The image of a node/edge in D, but not the image of a node/edge in L_t
(b) The image of a node/edge in L_t and the image of a node/edge in D
(c) The image of a node/edge in L_t, but not the image of a node/edge in D

The nodes and edges of (a) are left unchanged by the rewriting step, the nodes and edges of (b) are preserved and the nodes and edges of (c) are deleted. The applicability of a rule given a match then comes down to the satisfaction of a dangling edge condition, which is checked on the (c) nodes and the incident edges: a node which is attached to an edge that is not deleted may not be deleted. This implies that the in- and out-degrees of any deleted node are always exactly those indicated in the rule. For labeled graphs, the labels must also match.

Definition 8. *Notation. We write*

$G \overset{w}{\Rightarrow}_* G'$ *whenever* w *is the empty string* λ *and* $G = G'$
or $w = tv$, $\exists G'' \in \mathcal{G} : G \overset{t,g}{\Rightarrow} G'' \wedge G'' \overset{v}{\Rightarrow}_* G'$.
$G \Rightarrow_* G'$ *whenever* $\exists w \in T^* : G \overset{w}{\Rightarrow}_* G'$
$G \overset{w}{\Rightarrow}_*$ *whenever* $\exists G \in \mathcal{G} : G \overset{w}{\Rightarrow}_* G'$
$G \Rightarrow_*$ *whenever* $\exists w \in T^* \; \exists G \in \mathcal{G} : G \overset{w}{\Rightarrow}_* G'$
$G \not\Rightarrow$ *whenever* $\neg \exists t \in T : \; G \overset{t}{\Rightarrow}$

To each system we associate the set $lab(\mathcal{S})$ of labels of nodes or edges occurring in its start state and rules. In describing the derivation languages it would always be sufficient to consider only graphs labeled over $lab(\mathcal{S})$, as no other labels stand to be introduced. Even when varying the start graphs, all labels that do not occur in the rules look the same as far as the possible rule applications are concerned. When describing the reachable states, the assignment of the other labels can change the number of non-isomorphic reachable states, though.

We fix some conventions for the graphical representation of graphs and diagrams: nodes are drawn as little disks whose shading and inscriptions signify different labels. Continuous, dotted or dashed lines represent edges; the different kinds of lines, and inscriptions, stand for different labels. An arrow tip indicates the direction of and edge. Undirected edges can be read as pairs of opposite edges bearing the same label – sometimes we omitted the direction.

Graphs are surrounded with rounded boxes. In diagrams with graphs and morphisms, morphisms are usually drawn as fat arrows linking the rounded boxes. Every element (node, edge) is understood to be mapped to the element in the corresponding place. That convention is also valid for the named subgraphs which appear in little boxes.

3.1 Languages

In this subsection, we define derivation languages, a certain kind of terminal derivation languages and deadlock languages of graph transformation systems. The derivation languages (without termination criterion), which are analogous to Petri net firing sequence sets, are the sets of all possible sequences of rule applications. It is immediate from the definition that they are prefix-closed. If $\mathcal{S} = (\{U_t\}_{t\in T}, s_0)$ is an initialised system in the sense of Definition 5,

Definition 9. $\mathcal{L}_D(\mathcal{S}) = \{w \in T^* \mid s_0 \overset{w}{\Rightarrow}_{\mathcal{S}}\}$

All sequences ending in a state where a special rule labeled \$, that disables all further rule applications, is activated, constitute what we call the terminal language or \$-language.

Definition 10. $\mathcal{L}_\$(\mathcal{S}) = \{w \in (T \setminus \{\$\})^* \mid \exists s' \in \mathcal{G} :\ s_0 \overset{w}{\Rightarrow}_{\mathcal{S}} s' \wedge s' \overset{\$}{\Rightarrow}_{\mathcal{S}}\}$ *or more succinctly* $\{w \in (T \setminus \{\$\})^* \mid s_0 \overset{w\$}{\Rightarrow}_{\mathcal{S}}\}$

$\mathcal{L}_D$ and $\mathcal{L}_\$$ are the classes of all such languages. These are the two classes of languages we will be most concerned with in the present text. One can also define the deadlock language of the system $\mathcal{S}$,

Definition 11. $\mathcal{L}_\delta(\mathcal{S}) = \{w \in T^* \mid \exists s' \in \mathcal{G} :\ s_0 \overset{w}{\Rightarrow}_{\mathcal{S}} s' \wedge s' \not\Rightarrow_{\mathcal{S}}\}$

as the set of all sequences ending in a state where no rule applies anymore. Note that deadlock languages are a little awkward by definition in the handling of the empty word since if $\lambda \neq w \in \mathcal{L}_\delta(\mathcal{S})$, $\lambda \notin \mathcal{L}_\delta(\mathcal{S})$. Similar issues were remarked for Petri net languages and other definitions of language classes [Hac76].

Step languages can be defined as well by stipulating that a multiset of rules $\alpha \in T^\oplus$ is applicable iff the rules have parallel independent [HMP01,KKHK06] applications, and updating the definition of $\Rightarrow_{\mathcal{S}}$ accordingly. Several distinct step firing policies make sense [JZ08], and this paper mainly deals with sequential languages. We will nevertheless sometimes discuss conditions for the constructions to be valid for step languages as well. The simplest case of a parallel rule application is the application of the composition of the rules by disjoint union, without

identifying any common parts in the interface graphs, and one can keep this in mind when reasoning about the constructions, but one will need the actual definition of a parallel independent set of rules when checking whether some of the rules built in the constructions can be applied concurrently.

4 Closure Properties

Having defined several types of languages, we can look for closure properties. Even in case $\mathcal{L}_\$$ does turn out to be some well-known class of languages (with all of the properties shown here already being known), the techniques are specific to DPO systems and might still be adaptable for another variant, or possibly a restricted application of the formalism.

The closure of $\mathcal{L}_\$$ under the prefixes operation $pref(L) = \{w \in \Sigma_L^* \mid \exists u \in \Sigma_L^*, wu \in L\}$ follows from Propositions 1 and 2 in [Fli12].

In all of the systems constructed next, a lot of garbage is created during the execution, due to the fact that the number of nodes or edges added or removed by a rule cannot vary. In fact the amount of garbage created is always linear in the length of the derivation. That does not invalidate the constructions, which deal only with controlling the applicable rules so as to obtain a certain language.

4.1 Operations on Graph Transformation Systems

It is expedient to define some basic operations on graphs, morphisms, rules, which can be used to build up more complicated systems from given ones. A constructive approach will allow us to see many closure properties.

We will introduce a number of simple constructions that will prove to be versatile tools for crafting the derived systems, especially systems which accept a language that can be obtained by any of a number of operations that preserve the individual sequences of rule applications, such as union, concatenation, shuffling. In building new systems from old, an elementary tool for managing the controlled simulation of one system by another, which we will need for many of the proofs, is the suspension construction. It assigns to each graph (V, E, L, s, t, l) the graph $(V \uplus \{\hat{v}\}, E \cup \{(v, \hat{v}) \mid v \in V\}, L \uplus \{\hat{*}\}, s \cup s', t \cup t', l \cup l')$ with $s'((v, \hat{v})) = v$, $t'((v, \hat{v})) = \hat{v}$, $l'(\hat{v}) = \hat{*}$, and to each morphism of graphs one that preserves $\hat{v}$ and adds to the edge mapping the right mapping of the suspension edges. It is a functor and preserves the application of double pushout rules and creates no spurious rule applications. In other words, it defines an augmentation relation on graphs which even guarantees that every rule application in the adapted system corresponds to a unique rule application in the original system. In the graphical representation, it will be drawn like this 

, with a little box around the name of the graph whose nodes are linked to $\hat{v}$. (the hyperedge-like appearance [DKH97] is accidental). We will often call a state of the derived system which corresponds to a state of the input system, and which allows the corresponding translated rule applications, an augmented state. Note that Lemmata 1 and 2 are actually an easy application of the first embedding theorem in [Ehr77].

Lemma 1. *If there is a derivation $G \overset{t,g}{\Rightarrow} H$ in $\mathcal{S}$ and $\hat{*}$ is a label not in $lab(\mathcal{S})$, then there is exactly one derivation with all objects L_t, K_t, R_t, G, D, H (see Def. 7) suspended on a single node $\hat{v}$ labeled $\hat{*}$ and the restriction of every graph and morphism in the diagram to the nodes and edges is the original derivation.*

Proof. Every new edge's image is determined because there is exactly one edge from the node $\hat{v}$ to any other node; one ascertains that the resulting diagram still commutes and is a double-pushout situation.

Uniqueness is also given when considering sequences of derivations, because when the modified direct derivation applies, the new H will just be the suspension of the old H on a uniquely labeled node again. □

Concurrent applications of rules, not strictly under consideration here, are also preserved because any of the new edges which are deleted or created in the derivation are attached to nodes which are already deleted resp. created.

Lemma 2. *Any derivation remains valid if new nodes in K_t, and their images in L_t and R_t and G, D, H, are introduced; any derivation remains valid if any new edge is attached in L and G to nodes that are not deleted by the rule t*

Proof. The morphisms of the rule application now also map the additional nodes and edges. The dangling edge condition is fulfilled, because no new edges involving deleted nodes are introduced. □

The formalism of $DPO^{i/i}$ graph rewriting is fully time symmetric, but since backward derivations play no role in our current investigation, we also state

Lemma 3. *It is always possible to augment the effect of a rule by adding nodes and edges to R_t, and changing the rule by composing the right-hand side with $a : R_t \hookrightarrow R'_t$, without rendering any application of t impossible.*

Proof. The pushout of the right-hand side always exists, regardless of any conditions on the left-hand side. □

However, the addition of edges that involve nodes in R_t can make subsequent derivations invalid by interfering with deletions. This does not happen if all nodes involved are never deleted by any rule.

Also, it is obvious that the image of every match of a connected pattern is connected. For a more rigorous discussion and formal proofs of the following theorems, one would concentrate more on such lemmata and use them to show that certain relations can be found between the original and augmented graphs.

Theorem 1. *$\mathcal{L}_D$ is closed under intersection, and so is $\mathcal{L}_\$$.*

Proof. Given two systems $S_1 = (\{U_{t1}\}_{t \in T_1}, s_1)$ and $S_2 = (\{U_{t2}\}_{t \in T_2}, s_2)$ over unlabeled graphs, we map S_1 and S_2 to labeled graph transformation systems S'_1, S'_2 labeled over disjoint alphabets without changing their languages. Rules are composed via disjoint unions for graphs and morphisms. Let $G \cdot|\cdot G'$, $f \cdot|\cdot g$, $\mathcal{S}_1 \cdot|\cdot \mathcal{S}_2$ denote such an operation on graphs, morphisms and entire systems, discarding rules whose rule label is used in one system only. □

The same would be true for parallel applications of rules. The construction is useless for deadlock languages, as $\mathcal{L}_\delta(\mathcal{S}_1 \cdot|\cdot \mathcal{S}_2) \supseteq \mathcal{L}_\delta(\mathcal{S}_1) \cap \mathcal{L}_\delta(\mathcal{S}_2)$, without equality in general. While a deadlock on one side is a deadlock of the compound system, if w is not a deadlock for either but $s_1 \overset{w}{\Rightarrow}_{\mathcal{S}_1} s_1'$ and $s_2 \overset{w}{\Rightarrow}_{\mathcal{S}_2} s_2'$ and the sets of rules activated in s_1' and s_2' respectively have disjoint sets of labels (which is possible) then w is a new deadlock word of the compound system.

It is possible to create an alternative such that if $G \overset{t,g}{\Rightarrow}_{\mathcal{S}} H$, the graph $\hat{G}$ has the graph G as a subgraph and $\hat{G} \overset{t,\hat{g}}{\Rightarrow}_{\hat{\mathcal{S}}} \hat{H}$ and $\hat{G} \overset{t,g'}{\Rightarrow}_{\hat{\mathcal{S}}} H'$, where H' differs from $\hat{H}$ as to the possible rule applications.

To do this, suspend the graph G on a node $\hat{v}$, link $\hat{v}$ via an edge labeled $l \notin lab(\mathcal{S})$ to a fork-shaped graph $(\{v_0, v_1, v_2\}, \{e_0, e_1\}, \{l_f\}, s, e, l)$, $s(e_0) = s(e_1) = v_0$, $t(e_0) = v_1$, $t(e_1) = v_2$ and l assigns to v_0, v_1, v_2, e_0, v_1 the new label $l_f \notin lab(\mathcal{S})$. The second version of s' in Figure 1 shows this. Place v_1 and v_2 in different contexts, so that no automorphism of $\hat{G}$ swaps them. The rule U_t' of the new system $\mathcal{S}'$ is a suspension of the rule U_t, as in the upper left hand diagram of Figure 1. If v_1 and v_0 share some feature that can be matched together with the attached suspended graph, and v_2 does not, then an alternative can be used to stop one simulation and begin another.

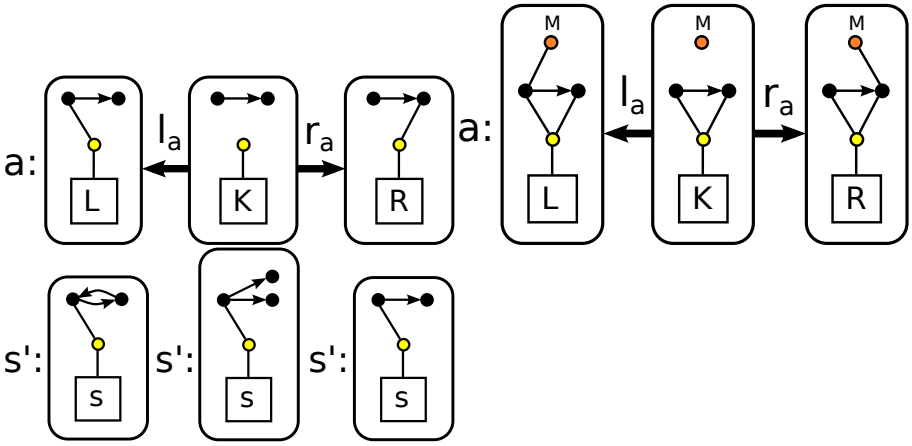


Fig. 1. A number of basic control patterns. Left: a rule featuring a suspension of the rule $L \hookleftarrow K \hookrightarrow R$ whose applicability is now controlled by supplementary nodes and edges, is subject to the presence of an edge between black nodes. Right: the same but with a mobile marker instead of reattachment of the suspended part. Left, bottom: three possible augmented versions of state s, to be used with the rule directly above them. From left to right, these allow: repeated application; alternative; one step only.

4.2 Closure under Homomorphisms

Unlike what one experiences with Petri net languages, where the free-labeled languages form families with weak properties [Jan87], not even containing the regular languages and having few closure properties, the derivation languages of $DPO^{i/i}$ systems over finite graphs are even closed under length-preserving (string) homomorphisms. Relabelings of rules can thus be simulated.

The constructions we devised for proving the closure properties have a commonality in that the states of the systems generating the original languages are in some way augmented, preserving applicability and translating effects of rules as long as the right conditions are met. This is most often achieved by suspending the state of each input system on a node, which is then attached to a scaffolding that governs the applicability of rules, whose nodes and edges bear labels that do not occur in the states of the original systems.

We come to our second theorem, $\mathcal{H}_1$ denoting closure under length-preserving (letter-to-letter) homomorphisms.

Theorem 2. *$\mathcal{L}_D = \mathcal{H}_1(\mathcal{L}_D)$ and $\mathcal{L}_\$ = \mathcal{H}_1(\mathcal{L}_\$)$.*

Proof. The construction takes a system $(\{U_t\}_{t\in T}, s_0\})$ with $e, o \in T$ and a morphism $h : T^* \to ((T \setminus \{e, o\}) \cup \{a\})^*$ with $h(x) := x$ for all $x \in T \setminus \{e, o\}$, and $h(e) = h(o) := a \notin T$, augments the start state by adding dummies, and slightly adapts all rules U_t $t \in T \setminus \{e, o\}$, and combines the rules U_e and U_o to a new rule U_a that has the combined effects of U_e and U_o, one of which is applied to a dummy, a "sandbox" portion of the state. U_a also renews the sandbox precondition that has just been consumed and cleans up the result of the old.

Figure 2 shows the construction. In the middle part of the scaffolding (nodes drawn as empty circles), an edge that is part of any valid match comprising the boxed state is transported. This has the effect of disconnecting the dummy parts which are used only once, whereas for the part containing the real state it has no effect because of the special setup of the system state. The rules whose labels' images under the homomorphism are unequivocal (called "b" in the drawing) are changed so that they can apply only to the part of the augmented state where they should apply. Termination rules for $\mathcal{L}_\$$ are handled the same as other rules.

The construction works because one part of the augmented rule always applies to the portion of the augmented state which is really responsible for the simulation, leading to another state which contains a faithful copy of the successor state of the system being simulated, while the applicability of the remainder of the augmented rule is always ensured. □

Still, less concurrency will in general be possible: if e has unbounded autoconcurrency because it does not delete any nodes, and o always has limited autoconcurrency, the number of times a can be applied in parallel will be limited by the necessary applications of o to the sandbox. One can provide for guaranteed bounded concurrency by attaching not one, but several suspended L_e and L_o.

Incidentally, the following corollary follows (because one rule can be used to simulate all end rules), even though the construction from Theorem 2 is more powerful than necessary in that case.

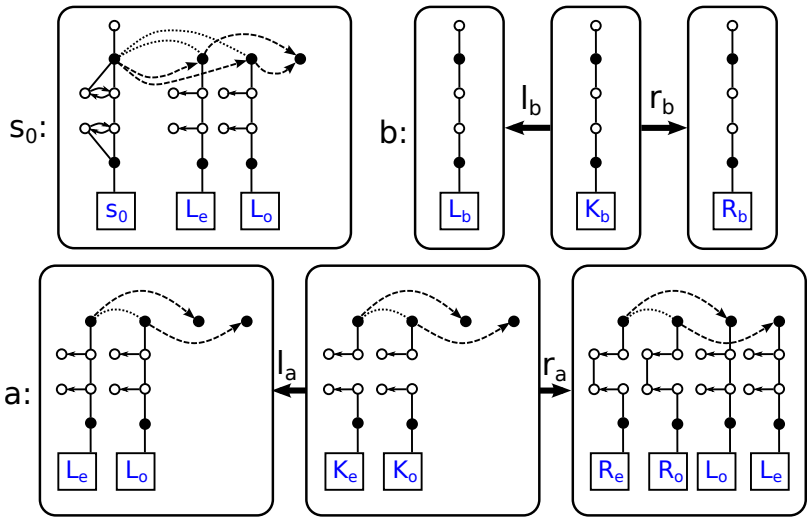


Fig. 2. Closure under length-preserving homomorphisms. The box stands for the graph whose name it contains, see explanation in Section 3

Corollary 1. $\mathcal{L}_{\{\$_1,\ldots,\$_n\}}(\mathcal{S}) := \{w \in (T \setminus \{\$_1,\ldots,S_n\})^* \mid s_0 \overset{w\$}{\Rightarrow}_{\mathcal{S}}\}$ *with* $\$ \in \{\$_1 \ldots \$_n\}$ *is in* $\mathcal{L}_\$$ *for finite* $\{\$_1,\ldots,\$_n\}$*: having multiple termination rules does not give any new languages.*

Note that $DPO^{i/i}$ systems over graphs can simulate a Turing machine; one can introduce one rule for extending the work tape, one rule per transition of the machine, and after a successful calculation read the word off the input tape using a special set of rules finishing with an application of \$ when hitting the end of the input tape. If one labels with λ all rules except those that read the word off the tape, one sees that the images of the derivation languages $\mathcal{L}_\$$ under deleting homomorphisms are indeed all recursively enumerable languages (it is clear that they are r.e.), which gives us non-closure under arbitrary homomorphisms easily for all classes under consideration.

4.3 Closure under Other Operations

The shuffle operation, defined on words as $\lambda \sqcup w = w \sqcup \lambda = \{w\}$ and $au \sqcup bv = \{a\}(u \sqcup bv) \cup \{b\}(v \sqcup au)$, returns all possibilities of interleaving the two words. It is commutative, associative when extended to languages: $L \sqcup M = \bigcup_{l \in L, m \in M} l \sqcup m$.

Theorem 3. $\mathcal{L}_D$ *is closed under shuffling, so is* $\mathcal{L}_\$$ *and so is* $\mathcal{L}_\delta$*. We provide a construction that to any two systems* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *returns a system* $\mathcal{S}_3$ *such that* $\mathcal{L}_x(\mathcal{S}_3) = \mathcal{L}_x(\mathcal{S}_1) \sqcup \mathcal{L}_x(\mathcal{S}_2)$ *for* $x \in \{\delta, D, \$\}$.

Proof. The start state s_0 of $\mathcal{S}_3$ in Figure 3 contains two independent parts. The rules are completely symmetric. When applied to the left hand part of the state,

the compound rule simulates one direct derivation of system $\mathcal{S}_2$ while letting run idle the part which simulates the rule from $\mathcal{S}_1$. When applied to the part shown on the right hand side of s_0 in Figure 3, it works the other way round. This idle running is made possible by providing the unused part of the rule with all preconditions it may need: the disjoint union (without relabeling) of all the left-hand sides of rules in $\mathcal{S}_1$, L_1. Since any amount of initially provided copies L_1 eventually runs out if some of the rules consume more nodes or edges than they produce, every rule of the compound system must provide another copy of L_1 to be used whenever a $\mathcal{S}_2$ derivation is simulated, as well as a copy of L_2 to be used in the converse case.

What do we do if one system is empty, or if a rule is not present in one of the systems? Unlike for intersection (Theorem 1), then the idle transition and empty state come into play, which means the parts belonging to the non-existent rule can simply be left out, leaving the degree of the yellow node at 1.

For $\mathcal{L}_\$$, the end rule must be extended differently from the other rules. It must make sure that both simulations have reached an end state. The construction works for $\mathcal{L}_\delta$ because when deadlock occurs for one system 1 or 2, the other can go on. It is only after both reach a deadlock that the termination condition is met, which is exactly the desired behaviour. □

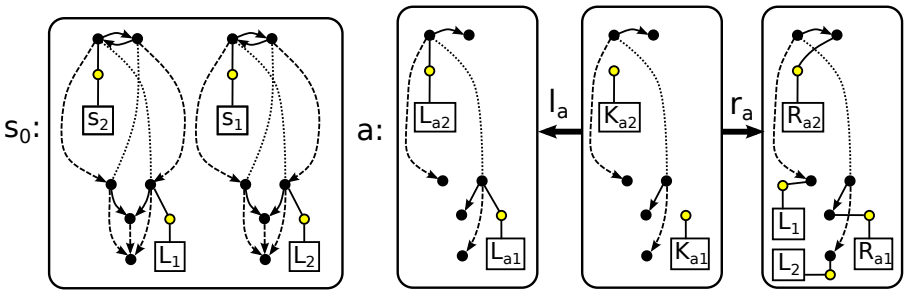


Fig. 3. Construction for closure under shuffling.

Theorem 4. *$\mathcal{L}_D$ is closed under union, so is $\mathcal{L}_\$$ and so is $\mathcal{L}_\delta$. Given two systems $\mathcal{S}_1$ and $\mathcal{S}_2$, one can produce a system $\mathcal{S}_3$ such that $\mathcal{L}_x(\mathcal{S}_3) = \mathcal{L}_x(\mathcal{S}_1) \cup \mathcal{L}_x(\mathcal{S}_2)$ for $x \in \{D, \$, \delta\}$.*

Proof. In the construction, one first assumes or makes sure that systems $\mathcal{S}_1$ and $\mathcal{S}_2$ have disjoint label sets. Then one attaches s_1 and s_2, the respective start states, to a node. A derivation step in the combined system moves the current state, which is suspended to a node of the scaffolding (the black nodes in Figure 4.3) to a new place. But every rule U_t of $\mathcal{S}_3$ must simulate the effects of both $U_{t,1}$ and $U_{t,2}$, its counterparts in the individual systems, except if one does not have a rule with that name, in which case it needs only simulate one.

At the beginning, the compound system will make a decision, choosing nondeterministically whether to simulate $\mathcal{S}_1$ or $\mathcal{S}_2$, since the state is crafted in such a way that it allows both kinds of match for the symmetrically constructed rules: if $\mathcal{S}_2$ is chosen, the part pertaining to $\mathcal{S}_1$ must not interfere with it, but it must be activated no matter what. To guarantee this, one constructs graphs L_1 and L_2 which are the disjoint unions, without relabeling, of all left-hand side graphs of system $\mathcal{S}_1$ resp. $\mathcal{S}_2$. These are produced as post-condition in every R_t to make sure the unused part of rule a which simulates the other system finds a match. The connected component to the right of the start state is used only once.

Of course, the constant activation must only be available under certain circumstances and without effect for the system whose simulation has been chosen in the beginning. Mobile markers (labeled with M) and fixed markers (labeled $*$) are introduced that regulate the availability of the individual derivations. □

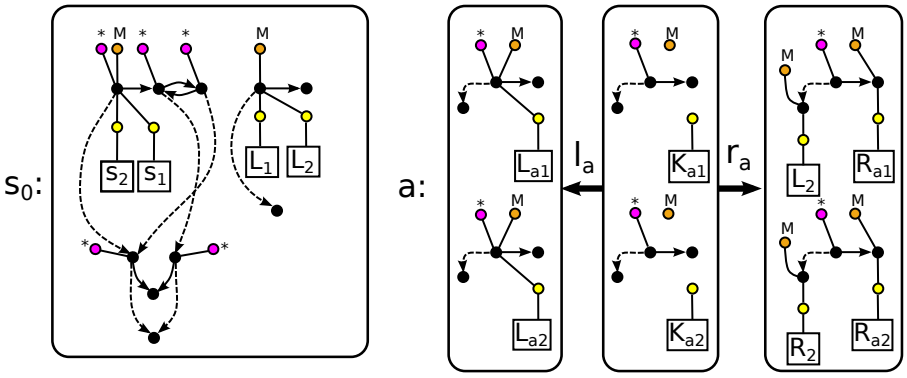


Fig. 4. Construction for union. The dashed lines are also edges. The connected component of the start state with L_1 and L_2 is used only once.

Next, let us try concatenation of languages $L \odot M = \{lm \mid l \in L, m \in M\}$. The idea is to simulate system $\mathcal{S}_1$ whose language is L, letting the other system run idle, always with the possibility of taking an alternative turn (the large "1" beneath the diagram in Figure 5 points out the edges that, if matched by the horizontal edge visible in the rule diagram, lead to this alternative) to a state from where the simulation of $\mathcal{S}_1$ is dead and $\mathcal{S}_2$ takes over.

Theorem 5. *$\mathcal{L}_D$ is closed under concatenation.*

Proof. The construction from Figure 5 is not sufficient by itself for the aim at hand, since it only generates $\{uv \in T^* \mid u \in \mathcal{L}_D(\mathcal{S}_1),\ v \in \mathcal{L}_D(\mathcal{S}_2),\ |u| > 0\}$. The union of this language with $\mathcal{L}_D(\mathcal{S}_2)$ is the desired language $\mathcal{L}_D(\mathcal{S}_1) \odot \mathcal{L}_D(\mathcal{S}_2)$. □

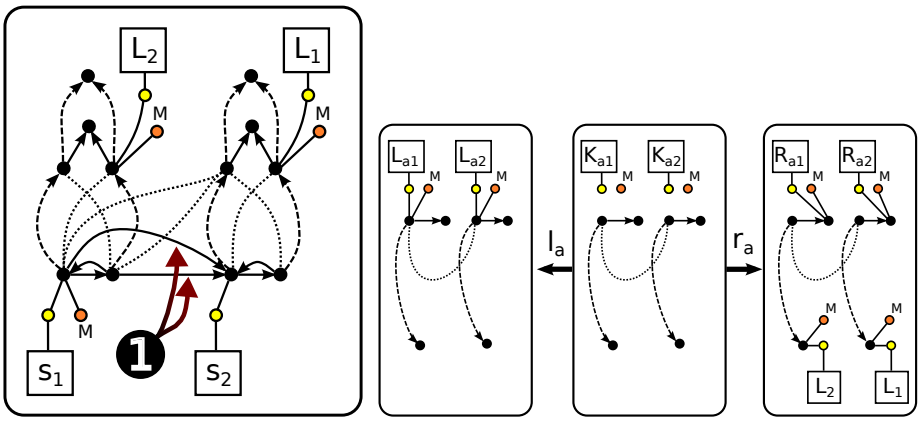


Fig. 5. Construction for concatenation of $\mathcal{L}_D$. The first diagram is the starting state, where L_i is the disjoint union of all left hand sides of $\mathcal{S}_i$ and s_i is the start state.

4.4 Iterated Shuffle and Concatenation

Languages of shuffle expressions [Gis81] are the closure of the finite languages under $(\cup, \odot, ⧢, ^{\odot}, ^{⧢})$ applied in any order. The iterated shuffle $L^{⧢}$ of a language L is defined in analogy with the Kleene star: $L^{(0⧢)} := \{\lambda\}$, for $i \in \mathbb{N} \setminus \{0\}$, $L^{(i⧢)} := L ⧢ L^{((i-1)⧢)}$, $L^{⧢} := \bigcup_{i\in\mathbb{N}} L^{(i⧢)}$. $\mathcal{L}_D$ is also closed under $^{\odot}$ and $^{⧢}$.

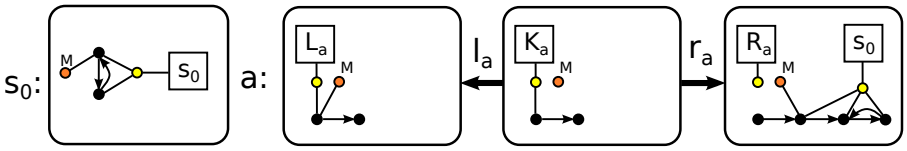


Fig. 6. Construction for iterated catenation (Kleene star) of $\mathcal{L}_D$.

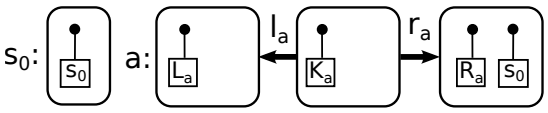


Fig. 7. Construction for iterated shuffling of $\mathcal{L}_D$.

Theorem 6. *$\mathcal{L}_D$ is closed under Kleene star and iterated shuffling.*

Proof. Iterated shuffle is very easily done for $\mathcal{L}_D$ (Figure 7). It is sufficient to add a new suspended version of the start state of the original system after every derivation step of the system that will generate the iterated shuffle of the original language. The iterated concatenation of a $\mathcal{L}_D$ language is also a $\mathcal{L}_D$ language, as shown in Figure 6. After each step, a new subgraph with the suspended start state s_0 is created and the scaffolding is extended in such a way that in the next step, that subgraph can serve for matching L_a. If that happens, the old evolved state is no longer available, because the marker then moves to the new part of the scaffolding which is only connected to the new start state at first. □

Unfortunately this does not mean that all prefix-closed languages of shuffle expressions are also derivation languages, since $pref(A \odot B)$ does in general not equal $pref(A) \odot pref(B)$.

4.5 Synopsis

We summarize the results obtained so far in a table.

	$\cap$	h	h_1	$\cup$	$pref$	$\odot$	$\sqcup$	$\odot$	$\sqcup$	$\Sigma^* \setminus$
$\mathcal{L}_\$$	Y	N	Y	Y	Y	?	Y	?	?	?
$\mathcal{L}_D$	Y	N	Y	Y	Y	Y	Y	Y	Y	N
$\mathcal{L}_\delta$	?	N	Y	Y	N	?	Y	?	?	?

Table 1. Synopsis of closure properties. Question marks indicate open points.

Some of the properties are obviously false for $\mathcal{L}_D$, since it contains only prefix closed languages. If the deadlock languages are not closed under $pref$, this is because the only $\mathcal{L}_D$ language to contain the empty word is $\{\lambda\}$. Otherwise, it would be enough to add disjointly, with 0 and $*$ being new labels, a fragment $(\{v_0, v_1, v_2, v_{marker}\}, \{e_0 : v_0 \to v_1, e_1 : v_1 \to v_0, e_2 : v_0 \to v_2, e_3 : v_1 \to v_2, e_4 : v_{marker} \to v_0\}, l_v = \{(v_0, 0), (v_1, 0), (v_2, 0), (v_{marker}, *)\})$ to the start graph, and to each rule a part moving an $*$ labeled marker node down an edge. A deadlock can be voluntarily entered after $n > 0$ steps.

5 Outlook

Using constructions on graphs, morphisms and systems, we have been able to find a number of closure properties. Obviously we want the remaining ones filled in. Also, we would like to know if Theorem 2 can be extended to cover arbitrary λ-free homomorphisms. Also, we still do not know the relationship between $\mathcal{L}_\$$ and the context sensitive languages $\mathcal{CS}$. We suspect $L_p = \{a^p \mid p \in \mathbb{N}\ \textit{is a prime number}\}$ might serve to distinguish the two, as it is difficult to see how a transformation system over finite graphs might generate it.

Overall, we think the subject deserves attention. Putting the informally described constructions and proof ideas on a firm formal basis is therefore an immediate goal.

References

[Cor95] Andrea Corradini, *Concurrent computing: from Petri nets to graph grammars*, Electronic Notes in Theoretical Computer Science **2** (1995), 56–70.

[CRM77] Stefano Crespi-Reghizzi and Dino Mandrioli, *Petri nets and Szilard languages*, Information and Control **33** (1977), no. 2, 177–192.

[Dar98] Philippe Darondeau, *Deriving unbounded Petri nets from formal languages*, CONCUR'98 Concurrency Theory (1998), 533–548.

[Dar04] ———, *Unbounded Petri net synthesis*, Lectures on Concurrency and Petri Nets (2004), 1–20.

[DKH97] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel, *Hyperedge replacement graph grammars*, Handbook of graph grammars and computing by graph transformation, World Scientific Publishing Co., Inc., 1997, pp. 95–162.

[Ehr77] Hartmut Ehrig, *Embedding theorems in the algebraic theory of graph grammars*, Fundamentals of Computation Theory, Springer, 1977, pp. 245–255.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans-Jürgen Schneider, *Graph-grammars: An algebraic approach*, Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on, IEEE, 1973, pp. 167–180.

[Fli12] Nils Erik Flick, *On derivation languages of DPO graph transformation systems. part 1: Introducing derivation languages*, 2012, Accepted for GCM 2012, Bremen.

[GG99] Stéphane Gaubert and Alessandro Giua, *Petri net languages and infinite subsets of* $\mathbb{N}^m$, Journal of Computer and System Sciences **59** (1999), no. 3, 373–391.

[Gis81] Jay Gischer, *Shuffle languages, Petri nets, and context-sensitive grammars*, Communications of the ACM **24** (1981), no. 9, 597–605.

[Hac76] Michel Hack, *Petri net languages*, Tech. Report 159, Massachusetts Institute of Technology, 1976.

[HMP01] Annegret Habel, Jürgen Müller, and Detlef Plump, *Double-pushout graph transformation revisited*, Mathematical Structures in Computer Science **11** (2001), no. 05, 637–688.

[Jan79] Matthias Jantzen, *On the hierarchy of Petri net languages*, RAIRO: Theoretical informatics **13** (1979), 19.

[Jan87] ———, *Language theory of Petri nets*, Petri Nets: Central Models and Their Properties (1987), 397–412.

[JZ08] Matthias Jantzen and Georg Zetzsche, *Labeled step sequences in Petri nets*, Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings (Kees van Hee, 2008, pp. 270–287.

[KKHK06] Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske, *Some essentials of graph transformation*, Recent Advances in Formal Languages and Applications **25** (2006), 229–254.

[Pet76] James L. Peterson, *Computation sequence sets*, Journal of Computer and System Sciences **13** (1976), no. 1, 1–24.

[Sta78] Peter H. Starke, *Free Petri net languages*, Mathematical Foundations of Computer Science 1978 (1978), 506–515.

Graph Rewriting with Contextual Refinement

Berthold Hoffmann

Fachbereich Mathematik und Informatik, Universität Bremen, Germany

Abstract. Plain graph rewrite rules modify subgraphs of constant size and fixed shape. In this paper, we propose rule schemata that are refined by deriving them with meta-productions based on contextual graph rewrite rules. The application of a refined schema may modify subgraphs of variable size and shape. We show that every rule based on single pushouts, on neighborhood-controlled embedding, or on variable substitution can be modeled by a schema with appropriate meta-productions. It turns out that the question whether schemata may have refinements with critical overlaps is not decidable in general.

1 Introduction

With graph rewriting, modifications of graphs can be specified in a rule-based, "declarative" way. Plain graph rewrite rules as in [5] are rather weak: they only allow to modify subgraphs of fixed shape and constant size. Even this is powerful enough from a theoretical point of view [21], many practical rewriting task need to modify subgraphs of variable shape and size. Now, if a set of plain rules is used to specify such tasks, they often have to be coordinated. Control can be specified in an imperative way (as in Progres [20]), or with separate formalisms like control conditions [17]. Both are only loosely related to the rules, so that it is harder to analyze properties of such a heterogeneous specification.

In this paper, we propose to make rules more expressive so that they can specify rewriting tasks that modify subgraphs of variable size and shape. We propose schemata – rules with variables – that are refined with contextual meta-productions before they are applied. This mechanism is based on the classical definition of rules. Properties of refined rules can be studied, using induction over the meta-productions. Moreover, rules with refinements can be executed, very efficiently, in the rewriting tool GrGen [1].

The paper is organized as follows. The next section defines graphs, plain rules for defining graph rewriting, and context-free and contextual productions for deriving graph languages. Then we recall related work on rules with variables, and define schemata, meta-productions, and the refinement process in Sect. 3. In Sect. 4 we relate rule schemata to other kinds of graph rewrite rules. It turns out that rules defined by single pushouts, by neighborhood-controlled embedding, and generic rules with graph variables can all be modeled by a single schema with appropriate meta-productions. Expressiveness of schema refinement has a price: it can, in general, not be decided whether two schemata can have refinements that are parallelly dependent. We conclude by indicating future work, in Section 5.

2 Graphs, Rewriting, and Derivation

We define graphs with edges that may attach not just to two nodes – a source and a target – but to any number of nodes. Nodes and edges of a graph are labeled. Such graphs are known as labeled hypergraphs in the literature, see [10].

Definition 1 (Graph). A pair $\mathcal{C} = (\dot{\mathcal{C}}, \vec{\mathcal{C}})$ of finite sets of *colors* is used to label nodes and edges, with a finite set $\mathcal{X} \subseteq \vec{\mathcal{C}}$ with a *type function type*: $\mathcal{X} \to \dot{\mathcal{C}}^*$ containing *variable names*.

A *graph* $G = (\dot{G}, \vec{G}, att, \ell)$ consists of disjoint finite sets $\dot{G}$ of *nodes* and $\vec{G}$ of *edges*, a function $att\colon \vec{G} \to \dot{G}^*$ that *attaches* sequences of nodes to edges; and of a pair $\ell = (\dot{\ell}, \vec{\ell})$ of *labeling functions* $\dot{\ell}\colon \dot{G} \to \dot{\mathcal{C}}$ for nodes and $\vec{\ell}\colon \vec{G} \to \vec{\mathcal{C}}$ for edges so that, for every edge $x \in \vec{G}$ with $\ell_G(x) \in \mathcal{X}$, the attached nodes $att_G(x)$ are distinct to each other and labeled so that $\ell^*_G(att_G(x)) = type(\vec{\ell}(x))$.[1] The component functions of a graph G will often be denoted as att_G and ℓ_G.

A (*graph*) *morphism* $m\colon G \to H$ is a pair $m = (\dot{m}, \vec{m})$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\vec{m}\colon \vec{G} \to \vec{H}$ that preserves attachments and labels: $att_H \circ \vec{m} = \dot{m}^* \circ att_G$, $\dot{\ell}_H = \dot{\ell}_G \circ \dot{m}$, and $\vec{\ell}_H = \vec{\ell}_G \circ \vec{m}$. 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 is injective, and maps nodes and edges onto themselves, we call G a *subgraph* of H, and write $G \hookrightarrow H$.

Let G be a graph. For an edge $e \in \vec{G}$ and a set $E \subseteq \vec{G}$ of edges, $G - e$ and $G - E$ shall denote the subgraph without e and E, respectively. We denote an edge $x \in X_G$ with $\vec{\ell}_G(x) = \xi \in \mathcal{X}$ as $x : \xi$, and call it a *variable*. $X_G = \{x \in \vec{G} \mid \vec{\ell}_G(x) \in \mathcal{X}\}$ is the set of *variables* in G. G is *terminal* if $X_G = \emptyset$. The *kernel* of G is its terminal subgraph $\underline{G} = G - X_G$. The subgraph of G consisting of a variable $x : \xi$ and its attached nodes is called the *star* of e, and denoted by $\langle x \rangle$. $\langle X_G \rangle = \bigcup_{x \in X_G} \langle x \rangle$ is the *star set* of G.

We base the rewriting of graphs on a well-known known definition in [5], but represent rules in a particular way.

Definition 2 (Rewriting). A (*graph rewrite*) *rule* $\rho = (P \hookrightarrow B \hookleftarrow R)$ consists of a *body* graph B with subgraphs P and R that are called *pattern* and a *replacement*, respectively. The graph $I = P \cap R$ is called *interface*; the nodes and edges in $O = P \setminus R$ are called *obsolete*, and those in $N = R \setminus P$ are called *new*. Sometimes we refer to components of ρ by P_ρ, R_ρ etc.

An injective morphism $m\colon P \to G$ is a *match* of ρ in G if it satisfies the following *gluing condition*: Every edge of G that is attached to the match of an obsolete node in $m(O)$ is in $m(P)$. Then the rule ρ *rewrites* the graph G under

[1] A^* denotes *finite sequences* over a set A; the empty sequence is denoted by ε, and $|w|$ denotes the *length* of a sequence $w \in A^*$. For a function $f\colon A \to B$, its extension $f^*\colon A^* \to B^*$ to sequences is defined by $f^*(a_1, \ldots, a_n) = f(a_1) \ldots f(a_n)$, for all $a_i \in A$, $1 \leqslant i \leqslant n$, $n \geqslant 0$. For functions or morphisms f and g, $f \circ g$ denotes their composition.

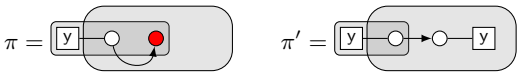


Fig. 1. Two productions

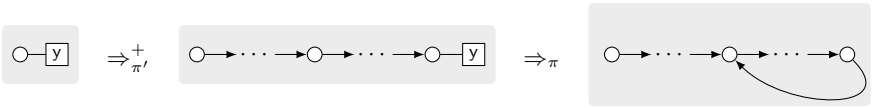


Fig. 2. Deriving lassos with productions π and π'

m into a graph H that can be constructed uniquely (up to isomorphism) by (1) removing the match $m(O)$ of the obsolete nodes and edges, giving the *kept graph* K, and (2) adding a fresh copy of N disjointly to K and substitute, in the attachments of the edges $\vec{N}$, every node from $v \in \dot{I}$ by its match $m(v) \in \dot{K}$. Then we write $G \Rightarrow_{\rho,m} H$.

The requirement that matches should be injective is no restriction [11].

Graph rewriting can be used for computing, by applying rules to some input graph as long as possible to yield an output graph. Rules can also be used to derive a set of graphs (a "language") from some start graph. A special form of rules have turned out to be useful for that purpose: productions replace a variable (edge) by gluing a graph to its attached nodes, and removing the variable. In the literature, this is known as context-free (hyperedge) replacement [10]. Here we extend productions so that some nodes in the context of the variable may be involved as well. This increases their generative power so that practically relevant languages can be derived, such as graphs representing object-oriented programs [4].

Definition 3 (Derivation). A rule $\pi = (P \hookrightarrow B \hookleftarrow R)$ is a *production* for a variable name $\xi \in \mathcal{X}$ if ξ is the label of the only edge x in P and if R equals $B - x$. For a finite set Π of productions, we write $G \Rightarrow_\Pi H$, and say that Π *directly derives* G to H if there is a rewrite $G \Rightarrow_{\pi,m} H$ for some match m of some $\pi \in \Pi$. As usual, $\Rightarrow_\Pi^+$ and $\Rightarrow_\Pi^*$ denote the transitive and reflexive-transitive closure of this relation, respectively. We call π *context-free* if $P = \langle x \rangle$, and *contextual* otherwise; then P is the disjoint union of $\langle x \rangle$ with a discrete *context graph* C_π, and its *underlying context-free production* $\tilde{\pi}$ is obtained by removing C_π from P, but not from R. The production π is *nonterminal* if $X_R \neq \emptyset$, and *monotone* if $|P| \leqslant |R|$. Π is *loop-free* if $P \not\Rightarrow_\Pi^+ R$ for the pattern P of every production in Π.

Example 1 (Contextual Derivation). Figure 1 shows two productions and illustrates our conventions for drawing graphs, rules, and productions. In figures of graphs, nodes are drawn as circles and have their labels inscribed. In our examples, all terminal edges are attached to exactly two nodes, a *source* and a *target*, and are drawn as arrows from their source to their target and have their labels

ascribed. Variables are drawn as boxes and have their labels inscribed, with lines connecting them to their attached nodes. By default these nodes are assumed to be ordered counter-clockwise around the variable, starting in the north; otherwise small numbers ascribed to the lines indicate the order, as in rule $\delta^d_{\pi'}$ of Fig. 6 further below.

The pattern and the replacement graphs of rule bodies are enclosed in boxes, where the boxes extending farther to the left indicate the patterns; their intersection defines the interface. Context nodes are set off by drawing them in red (appearing as dark grey in B/W). We use the convention that an edge belongs only to the subgraph whose box contains it entirely; so the edge is new in π, i.e., it belongs to R, but not to P.

If a star with a variable named y is used as a start graph, the productions in Fig. 1 derive a language of "lassos" as in Fig. 2: n applications of the context-free production π' derive a chain of length n; the terminal contextual production π inserts an edge from the last node in the chain to a node in the context that has been derived previously. The noose of a lasso contains at least one node, since in π, the attached node of y and the context node must match different nodes in a graph.

3 Deriving Rules with Meta-Productions

The idea to derive graph rewrite rules with meta-rules has arisen early, inspired by two-level string grammars [13,9]. D. Plump and A. Habel have introduced substitutive rules whose patterns and replacements contain variables that may be substituted by arbitrary graphs [18]. Operationally, this makes rule matching highly non-deterministic, so that substitutions were restricted to graphs derived with context-free productions in [14]. Here we study the more general case where not just a part of the pattern or of the replacement can be substituted, but the rule can be derived as a whole, and allow the productions to be contextual, not just context-free. This notion of rules is supported by the rewriting tool GrGen [1,15].

We add variables to the body B of a rule $(P \hookrightarrow B \hookleftarrow R)$ in order to indicate the places where it shall be refined with productions Π, and call this a schema. In derivations $B \Rightarrow^*_\Pi B'$ of the body, we must indicate which subgraph of B' shall be the pattern and the replacement of the refined schema $(P' \hookrightarrow B' \hookleftarrow R')$, respectively. This leads to the notion of meta-productions, and to a classification of variable attachments. We assume that the variable names $\mathcal{X}$ come with functions $pat, repl\colon \mathcal{X} \to \mathbb{N}$ so that $pat(\xi) \leqslant |type(\xi)|$ and $1 \leqslant repl(\xi) \leqslant pat(\xi)$ for every variable name $\xi \in \mathcal{X}$. For every variable $x : \xi$ in a graph G with $att_G(x) = v_1 \dots v_k$, these functions designate discrete subgraphs of G: the *pattern attachment* $\langle x \rangle^{\mathrm{p}}$ consists of the leading attached nodes $v_1 \dots v_{pat(x)}$, and the *replacement attachment* $\langle x \rangle^{\mathrm{r}}$ consists of the trailing attached nodes $v_{repl(x)} \dots v_k$.

The meta-productions for refining schemata consist of contextual spine productions, with a body rule indicating which subgraphs of the spine body are meant to refine the pattern and the replacement of the schemata.

Definition 4 (Schema, Meta-Production). A *schema* is a rule $\sigma = (P \hookrightarrow B \hookleftarrow R)$ so that $P \cap R = \underline{B}$, and $\langle x \rangle^{\mathrm{p}} \hookrightarrow P$ and $\langle x \rangle^{\mathrm{r}} \hookrightarrow R$ for every variable $x \in X_B$.

A *meta-production* $\delta = (\pi, \rho)$ consists of a contextual *spine production* $\pi = (P_\pi \hookrightarrow B_\pi \hookleftarrow R_\pi)$, and of a *body rule* $\rho = (P_\rho \hookrightarrow B_\rho \hookleftarrow R_\rho)$ so that $(P_\rho \hookrightarrow B_\pi \hookleftarrow R_\rho)$ is a schema, i.e., $\underline{B_\pi} = B_\rho$.

The following facts are direct consequences of Def. 4.

Fact 1. For meta-production $\delta = (\pi, \rho)$, the following holds:

1. Its spine production π satisfies
   (a) $\underline{P_\pi} = I_\pi$, $\underline{R_\pi} = \underline{B_\pi}$ and $\underline{P_\pi} \hookrightarrow \underline{R_\pi}$, and
   (b) the intersections $P_{\rho\pi} = P_\rho \cap P_\pi$ and $R_{\rho\pi} = P_\rho \cap R_\pi$ are discrete so that $P_{\rho\pi} = \langle e \rangle^{\mathrm{p}} \cup C_\pi$ and $R_{\rho\pi} = \langle e \rangle^{\mathrm{r}} \cup C_\pi$, where C_π is the context graph of π.
2. Consider a schema $\sigma = (P \hookrightarrow B \hookleftarrow R)$, and a derivation $B \Rightarrow_{\pi,m} B'$ so that $m(C_\pi) \hookrightarrow P \cap R$, where g shall denote the morphism $R_\pi \to B'$. Then
   (a) $m(P_{\rho\pi}) \hookrightarrow P$ and $m(R_{\rho\pi}) \hookrightarrow R$,
   (b) $P' = P \cup g(P_\rho)$ and $R' = R \cup g(R_\rho)$ are terminal graphs so that $\sigma' = (P' \hookrightarrow B' \hookleftarrow R')$ is a schema, and $\underline{B} \hookrightarrow \underline{B'}$.

These facts allow to define the refinement of schemata by meta-productions. A schema is refined until a rule is obtained, which can be used to rewrite a graph.

Definition 5 (Schema Refinement). Consider a schema $\sigma = (P \hookrightarrow B \hookleftarrow R)$ and a meta-production $\delta = (\pi, \rho)$ as above.

A match $m\colon P_\pi \to B$ of the spine production π to the body B of σ is a *meta-match* if $m(C_\pi) \hookrightarrow P \cap R$. For the derivation $B \Rightarrow_{\pi,m} B'$, Fact 1 allows to extend B' to a schema $\sigma' = (P' \hookrightarrow B' \hookleftarrow R')$; Then we write $\sigma \Downarrow_{\delta,m} \sigma'$, and say that δ *refines* σ to σ' (at m).

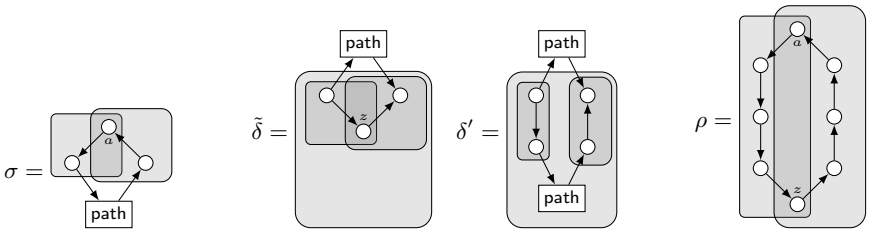
For a contextual meta-production $\delta = (\pi, \rho)$, the *underlying context-free meta-production* is $\tilde{\delta} = (\tilde{\pi}, \rho)$. Let Δ be a finite set of meta-productions so that $\delta \in \Delta$ implies $\tilde{\delta} \in \Delta$. Then $\Downarrow_\Delta$ denotes refinement steps with one of its meta-productions, and $\Downarrow_\Delta^*$ denotes repeated refinement, its reflexive-transitive closure. The *derivates* of a schema $\sigma = (P \hookrightarrow B \hookleftarrow R)$, contain its refinements without variables: $\Delta(\sigma) = \{\rho \mid \sigma \Downarrow_\Delta^* \rho, \underline{\rho} = \rho\}$.

We write $G \Rightarrow_{\Delta,\sigma} H$ and say that σ, refined with Δ, *rewrites* G to H if $G \Rightarrow_\rho H$ for some derivate $\rho \in \Delta(\sigma)$.

A meta-production $\delta = (\pi, \rho)$ is *nonterminal*, *monotone*, and *loop-free* if its spine production π has the respective property. Moreover, it is *pattern-monotone* if it is monotone, and if $|P_\pi| < |R_\pi|$ implies $|P_{\rho\pi}| < |P_\rho|$.

We require that the sets Δ of meta-productions used in this paper are productive and loop-free, and that their nonterminal meta-productions are pattern-monotone.

The structure of context-free meta-rules implies that they really refine the kernels of schemata.



(a) A schema (b) Meta-productions for path (c) A derivate of σ

Fig. 3. Schema refinement with meta-productions

Theorem 1. *For derivates* $(P' \hookrightarrow B' \hookleftarrow R') \in \Delta(P \hookrightarrow B \hookleftarrow R)$, $\underline{B} \hookrightarrow \underline{B'}$, *and* $(P' \hookrightarrow B' \hookleftarrow R')$ *is a rule.*

Proof Sketch. Applying Fig. 1.2b to refinement sequences $(P \hookrightarrow B \hookleftarrow R) \Downarrow^*_\Delta (P' \hookrightarrow B' \hookleftarrow R')$ allows to conclude that $\underline{B} \hookrightarrow \underline{B'}$, $P \hookrightarrow P'$ and $R \hookrightarrow R'$. Since Δ is monotonic and productive, the derivate is free of variables so that $\underline{B'} \cong B'$, confirming the claim. □

Example 2 (Refining a Schema). Fig. 3 illustrates schema refinement and rewriting with derivates, and shows how we draw schemata and meta-productions. In diagrams, the line connecting a variable $x : \xi$ in a body B with some attached node v_i in $att_B(x) = v_1 \dots v_k$ has an arrow tip at x if $i \leqslant repl(x)$, a tip at v_i if $i > pat(x)$, and no tip otherwise.

(a) shows a simple schema σ with a variable name path. As with rules, the pattern and replacement part are enclosed in boxes. The variable lies outside of these boxes since it does belong, neither to the pattern, nor to the replacement. The pattern and replacement of the schema have the node a in common. The variable path attaches to a node v_1 that shall be deleted, and to a node v_2 that shall be inserted, with their incident edges.

(b) shows two context-free meta-productions δ and δ' for the variable name path occurring in σ. Meta-productions are drawn "upright", with their spine variable on top, and their spine replacement in the outer box. Within this box, the pattern and replacement of the body schema are drawn as usual, where the box extending farthest to the left designates the body pattern. The variable path is attached to two nodes v_1 and v_2. The non-recursive meta-production δ replaces an edge from v_1 to some node z by an edge from z to v_2. The recursive meta production δ' extends paths to v_1 and from v_2 by one edge.

(c) shows a derivate ρ of σ. Applying δ' two times and δ once, this generates a rule wherein pattern and replacement have a begin node a (on the top) and an end node z (at the bottom) in common. Pattern and replacement specify paths of the same length between a and z, in opposite direction. In applications of ρ, only the matches of a and of z may be incident to other nodes, due to the dangling condition.

A single rewriting step with some derivate of some schema may match, delete, and insert subgraphs of arbitrary size: In the example, a path of arbitrary length is matched, deleted (up to its start and end node), and a reverse path of the same length is inserted. This goes beyond the expressiveness of plain rewrite rules, which may only match, delete, and insert subgraphs of constant size.

Note that the application of a derivate $\rho \in \Delta(\sigma)$, although it is the result of a compound meta-derivation, is a single rewriting step $G \Rightarrow_\rho H$, like a transaction in a data base. Note also that the refinement process is completely rule-based.

Operationally, we cannot construct the derivates of a schema first, and apply one of them later, because $\Delta(\sigma)$ is infinite in general. Rather, we interleave matching and refinement in a goal-oriented way.

Algorithm 1 (Applying a Schema to a Graph).

Input: A terminal graph G, a schema $\sigma_0 = (P_0 \hookrightarrow B_0 \hookleftarrow R_0)$, and a set Δ of productive and loop-free meta-productions wherein all nonterminal productions are monotone.

Output: Either a refinement $\sigma_i \in \Delta(\sigma_0)$ of σ_0 with a match m_i in G, or **no** if no derivate in $\Delta(\sigma_0)$ applies to G.

1. Search for an injective morphism $m_0 \colon P_0 \to G$ of the original schema σ_0, and set i to 1.
2. Search for an underlying context-free meta-production $\tilde{\delta}_i \in \tilde{\Delta}$ with a refinement $(P_{i-1} \hookrightarrow B_{i-1} \hookleftarrow R_{i-1}) \Downarrow^*_{\tilde{\delta}_i} (P_i \hookrightarrow B_i \hookleftarrow R_i)$ such that the morphism m_{i-1} can be extended to a morphism $m_i \colon P_i \to G$ that identifies, at most, context nodes of the spine production of δ_i with nodes in $P_{i-1} \cap R_{i-1}$.
3. If no such meta-production can be found, undo refinements to the closest step $j < i$ where some meta-production exists that has not been inspected for σ_j previously; if already all meta-productions have been inspected, for all previous steps, and for all initial matches m_0, answer **no**.
4. If σ_i does contain further variables, increase i by one and goto Step 2.
5. Otherwise, σ_i is a rule with a morphism $m_i \colon P_i \to G$. If m_i violates the gluing condition, goto Step 3. Otherwise m_i is a match of the derivate σ_i in G. □

Actually, *all* matches for *all* applicable derivates can be enumerated, so that the potential nondeterminism can be handled by backtracking.

GrGen rules are executed this way; only the match of a particular pattern P_i is determined according to an efficient search plan that is pre-computed, taking into account the shape of the type graph, as well as that of the particular graph G, and the whole rule is compiled into sequences of low level C# instructions that do graph matching and construction [1].

Theorem 2. *Algorithm 1 terminates, and is correct.*

Proof Sketch. (*Termination.*) The search tree spanned by Algorithm 1 is finite: Step 1 chooses among finitely many initial morphisms m_0 in G, and Step 2 chooses among a finite number of morphisms for a finite number of meta-productions. So the search tree has finite breadth. It has finite depth as well, because every δ_i with nonterminal spine production in Step 2 satisfies $|P_{i-1}| \leqslant$

$|P_i|$ due to pattern-monotony of Δ, and the number of consecutive steps with $|P_{i-1}| = |P_i|$ is bound by $|\Delta|$ since Δ is loop-free. Then the search for a match terminates, after applying finitely many terminal meta-productions.

(*Correctness.*) The identification condition in Step 2 makes sure that the morphism m_i either is an injective morphism for the underlying context-free metaproduction $\tilde{\delta}_i$, or an injective morphism for the original meta-production δ_i. Thus Step 5 determines an injective morphism of a derivate of σ_0, and checks whether it is a match. The backtracking in Step 3 and Step 5 makes sure that every possible candidate for a derivate with a match is considered.

Corollary 1. *For a schema σ and meta-productions Δ as above, it is decidable whether some derivate $\rho \in \Delta(\sigma)$ applies to a graph G, or not.*

4 Properties of Contextual Schema Refinement

In this section, we study properties of schema refinement in order to motivate its usefulness. We compare schemata to other ways of graph rewriting: graph rewriting by single pushout, by neighborhood-controlled embedding, and by generic rules. In all three cases, a rule can be modeled by a single schema with appropriate meta-productions, whereas they cannot be modeled by a single plain rule. Finally, we study whether the existence of critical overlaps, a decidable property for plain rewriting, is decidable for schemata as well.

4.1 Graph Rewriting with Node [Dis-]Connections

We investigate ways of graph rewriting where a single rule may delete and redirect a variable number of edges, respectively. First we define generic meta-productions that allow to models such rules.

Definition 6 ([Dis-]Connecting Meta-Productions). Fig. 4 shows *disconnecting meta-productions* that extend a schema by B-nodes in its interface that are connected by obsolete edges to an obsolete A-node. A disconnecting configuration $d \in \dot{\mathcal{C}} \times \vec{\mathcal{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathcal{C}}$ specifies labels and directions of the refined

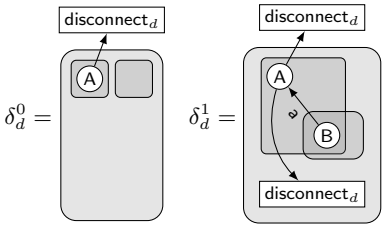


Fig. 4. Meta-productions for a disconnecting configuration $d = (\mathsf{A}, \mathsf{a}, \text{in}, \mathsf{B})$

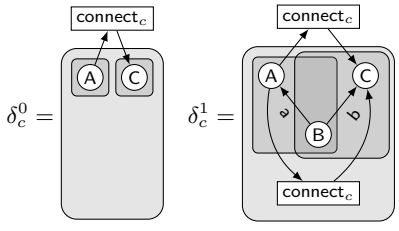


Fig. 5. Meta-productions for a connecting configuration $c = (\mathsf{A}, \mathsf{a}, \text{in}, \mathsf{B}, \mathsf{b}, \text{in}, \mathsf{C})$

nodes and edges. A so refined schema will remove edges that connect nodes in the interface to the obsolete node attached to the variable.

Fig. 5 shows *connecting meta-productions* that extend a schema by B-nodes in its interface that are connected to an obsolete A- node and to a new C-node. Again, a connecting configuration $d \in \dot{\mathcal{C}} \times \vec{\mathcal{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathcal{C}} \times \vec{\mathcal{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathcal{C}}$ specifies the labels and directions of the refined nodes and edges. A so refined schema will remove, for the obsolete node v and the new node v' attached to the variable, edges connecting v to interface nodes, and insert edges connecting these to v' instead.

Δ^{dc} shall denote the set of all disconnecting and connecting meta-productions, for all disconnecting and connecting configurations.

Graph rewriting defined with single pushouts [7] (SPO, for short) is a variation of graph rewriting by double pushout [5] (DPO for short), which in turn is the basis for Def. 2. Apart from techical details of their definition, SPO rules have the same form as DPO rules. However, rewriting is defined differently: no dangling condition needs to be checked when matching a rule; the match $m(v)$ of every obsolete node v of τ is deleted in any case, together with all edges outside the match $m(P)$ that are incident with $m(v)$ (which would violate the dangling condition otherwise).

Definition 7 (SPO Rewriting with Schema Refinement). A plain rule $\rho = (P \hookrightarrow B \hookleftarrow R)$ can be turned into a *SPO schema* $\sigma_\rho = (P \hookrightarrow B \hookleftarrow R)$ that performs SPO rewriting by attaching, to every obsolete node v in $P_\rho \setminus R_\rho$, variables $x : \mathsf{disconnect}_d$, for every disconnecting configuration $d = (\dot{\ell}_P(v), a, io, A)$ where $a \in \vec{\mathcal{C}}$, $io \in \{\text{in}, \text{out}\}$, and $A \in \dot{\mathcal{C}}$.

Fact 2. Consider a rule ρ with an injective morphism $m\colon P_\rho \to G$.

Then the SPO schema σ_ρ has a derivate $\rho' \in \Delta^{\text{dc}}(\sigma_\rho)$ with a rewriting step $G \Rightarrow_{\rho'} H$ so that all obsolete nodes of ρ are deleted, together with their incident ("dangling") edges in $\vec{G} \setminus \vec{m}(\vec{P})$.

Proof Sketch. Consider $m\colon P_\rho \to G$. Then $P_\rho \hookrightarrow P_{\rho'}$ by Fact 1. Now, at every obsolete node v of ρ, the schema can be refined with disconnecting meta-productions so that it covers every "dangling" edge e incident in $\dot{m}(v)$ and its "neighborhood node" – say v' – that is outside $m(P)$. As the pre-image of v' in the schema is in the interface of the refined schema, and the edge is in the obsolete part of the pattern, the dangling edges are removed while the nodes in the neighborhood are preserved. The refined rule satisfies the gluing condition, and does exactly what SPO rewriting does. [2] □

Graph grammars with neighborhood-controlled embedding are "the other notion" of graph rewriting. Their definition is set-theoretic, and their most widely

[2] We have ignored the fact that matches of SPO rules need not be injective. However, we can construct the (finite) set the *quotients* $Q(\rho)$ by identifying arbitrary isomorphic subgraphs in P_ρ. Then for every non-injective match of ρ, $Q(\rho)$ contains a quotient with an equivalent injective match.

investigated subclass has rules where the pattern is a single node [8]. Here, we consider the more general case where the pattern is a graph.

More precisely, a *neighborhood-controlled embedding rewrite rule* (NCE rewriting, for short) $\eta = (M, D, I)$ consists of terminal graphs M and D, called *mother graph* and *daughter graph* respectively, and of a set of *connecting instructions* $I \subseteq \dot{M} \times \vec{\mathcal{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathcal{C}} \times \vec{\mathcal{C}} \times \{\text{in}, \text{out}\} \times \dot{D}$. Let $\tilde{I}$ denote the tuples $(v, \mathsf{a}, io, \mathsf{A}) \in \dot{M} \times \vec{\mathcal{C}} \times \{\text{in}, \text{out}\} \times \dot{\mathcal{C}}$ without a connecting instruction in $(v, \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', v') \in I$.

A rule η as above *applies* to a graph G if there is an injective morphism $m\colon M \to G$ so that the subgraph $m(M)$ is induced by the nodes $\dot{m}(\dot{M})$. Then the application of η at m deletes the subgraph $m(M)$ from G, adds a fresh copy of the daughter graph D, and obtains the graph H by executing the connecting instructions as follows:

- If $c = (v, \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', v') \in I$, for all A-nodes $z \in \dot{G} \setminus \dot{m}(\dot{M})$ connected by an a-edge e with $\dot{m}(v)$ in direction io, a b-edge connecting z with the node $v' \in \dot{D}$ in direction io' is inserted in H.
- For every $d = (v, \mathsf{a}, io, \mathsf{A}) \in \tilde{I}$, a-edges between v and A-nodes in direction io are just deleted from G.

NCE rewriting can be modeled with schema refinement as follows.

Definition 8 (NCE Rewriting with Schema Refinement). An NCE rule $\eta = (M, D, I)$ as above can be transformed into a schema $\sigma_\eta = (M \to B \hookleftarrow D)$ where the underlying rule $\underline{\sigma_\eta}$ has an empty interface (i.e., $\underline{B} = M \uplus D$), and B contains, for every connecting instruction $c = (v, \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', v') \in I$, a variable $x : \mathsf{connect}_{c'}$ with $att_B(x) = vv'$ and $c' = (\dot{\ell}_M(v), \mathsf{a}, io, \mathsf{A}, \mathsf{b}, io', \dot{\ell}_D(v'))$, and for every $d = (v, \mathsf{a}, io, \mathsf{A}) \in \tilde{I}$, a variable $x : \mathsf{disconnect}_{d'}$ with $att_B(x) = v$, and $d' = (\dot{\ell}_M(v), \mathsf{a}, io, \mathsf{A})$.

Fact 3. For an NCE rule η as above, there exists an NCE graph rewriting step $G \Rightarrow_\eta H$ if and only if there is a rewriting step $G \Rightarrow_\rho H$ using some derivate $\rho \in \Delta^{\text{dc}}(\sigma_\eta)$ of its schema σ_η.

Proof Sketch. As in the case of modeling SPO rewriting, a match $m\colon M_\eta \to G$ can be taken as a basis to refine σ_η so that all connection instructions are "performed", and all other "dangling" edges are deleted.

There are other derivates that do not satisfy the dangling condition although the NCE rule applies. As with SPO rewriting, this is the case when the recursive meta-productions are not applied exhaustively. Even then, a refinement of the schema may violate the dangling condition when the match of the refined pattern contains nodes that are connected by edges outside the match. This, however, violates the application condition for NCE rules as well, where the match must be the induced subgraph of the matched nodes. □

4.2 Generic Graph Rewriting

Let us recall the definition of generic rules [14], slightly re-phrasing it according to our definition of plain rules, and extending it to contextual substitution.

More precisely, a *generic rule* $\gamma = (P \hookrightarrow B \hookleftarrow R)$ over a finite set Π of (monotone, productive) contextual productions consists of two graphs P, R with variables so that (1) all variables in B belong to P or to R and (2) every variable name occurring in R occurs in P as well.

An *instance* of γ is obtained by replacing, in its body B, every variable $x : \xi$ occurring in B by the same terminal graph G_ξ with a derivation $\langle x \rangle \Rightarrow^*_\Pi G_\xi$ so that variables carrying the same name have identical instances, and contextual nodes occurring in the derivation belong to the intersection of the pattern and the replacement, and are identical for all variables names ξ. A generic rewrite step $G \Rightarrow_\rho H$ applies an arbitrary instance ρ of γ.

Generic rules can be modeled by schema refinement as follows.

Definition 9 (Meta-Productions and Schemata for Generic Rules). Let γ be a generic rule for productions Π.

Then $r = (k, m, n)$ is a *reproduction instruction* of a variable name $\xi \in \mathcal{X}$ if

(a) ξ labels k variables in the body B of $\gamma = (P \hookrightarrow B \hookleftarrow R)$, whereof m variables occur in P and n variables occur in R, or
(b) Π contains a production $(P' \hookrightarrow B' \hookleftarrow R')$ where the variable name ξ occurs in R', and the variable name ξ' in P has a reproduction instruction r.

We transform γ and Π into a schema σ_γ and meta-productions $\Delta_{\gamma,\Pi}$ as follows:

1. Whenever a variable ξ with $\mathit{type}(\xi) = l_1 \dots l_a$ has a reproduction instruction $r = (k, m, n)$, $\mathcal{X}$ shall contain a fresh variable name ξ^d with $\mathit{type}(\xi^d) = l_1^k \dots l_a^k$ and $\mathit{pat}(\xi^d) = k \cdot \mathit{pat}(\xi)$ and $\mathit{repl}(\xi^d) = k \cdot \mathit{repl}(\xi)$.
2. We transform every generic rule $\gamma = (P \hookrightarrow B \hookleftarrow R)$ into a schema $\sigma_\gamma = (\underline{P} \hookrightarrow B' \hookleftarrow \underline{R})$ by replacing all variables $x_1 : \xi, \dots, x_k : \xi$ in B (where we assume, for $1 \leqslant i \leqslant k$, that if x_i is in P, so are $x_1 \dots x_{i-1}$, and if x_i is in R, so are $x_{i+1} \dots x_k$) by a reproduction variable $y : \xi^d$ with $\mathit{att}_{B'}(y) = \mathsf{pcat}\,(\mathit{att}_{B_1}(x_1), \dots, \mathit{att}_{B_k}(x_k))$.[3]
3. For every production $\pi = (P \hookrightarrow B \hookleftarrow R) \in \Pi$ with $R = B - x$ and $x : \xi \in P$, and for every reproduction instruction $r = (k, m, n)$ of ξ, $\Delta_{\gamma,\Pi}$ shall contain a context-free *reproducing production* $\delta^r_\pi = (\pi_r, \rho)$ so that
   (a) the kernel $\underline{B_{\pi_r}}$ of its spine production consists of n disjoint copies $B_1, \dots, B_n$ of $\underline{B}$ so that $B_1, \dots, B_m$ are in its body pattern P_ρ and $B_{k-n+1}, \dots, B_k$ are in its body replacement R_ρ;
   (b) for every variable $x : y$ in B, where we assume that w_i is the copy of the attached node sequence $\mathit{att}_B(x)$ in B_i (for $1 \leqslant i \leqslant k$), B_{π_r} contains a variable $y : \xi^r$ with $\mathit{att}_{B_{\pi_r}}(y) = \mathsf{pcat}\,(w_1, \dots, w_k)$.[3]

   Note that $\Delta_{\gamma,\Pi}$ does contain also the underlying context-free rule $\tilde{\delta}^r_\pi$ of δ^r_π if π is contextual. Thus $\langle y \rangle \hookrightarrow P_{\pi_r}$ if x is the variable in P, and all other variables belong to R_{π_r}. In B_{π_r}, a copy B_i belongs to P_ρ if $1 \leqslant i \leqslant m$, and to R_ρ if $k - n + 1 < i \leqslant k$. If δ is contextual, its contextual nodes belong both to P_ρ and to R_ρ, are shared by all copies, and are contextual nodes in π_r.

[3] If, for $1 \leqslant i \leqslant k$, the words $w_i = a_{i,1} \dots a_{i,n}$ have equal length $n \geqslant 0$, their *pointwise concatenation* is $\mathsf{pcat}\,(w_1, \dots, w_k) = a_{1,1} \dots a_{k,1} \dots a_{1,n} \dots a_{k,n}$.

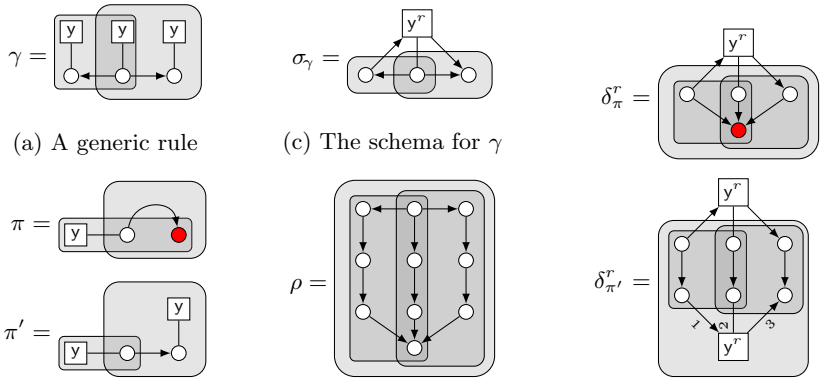


Fig. 6. Modeling a generic rule with a schema

Example 3 (Modeling a Generic Rule by a Schema). Fig. 6 shows (a) a generic rule γ, and (b) the productions $\Pi = \{\pi, \pi'\}$ for the variable y which are already known from Fig. 1.

This is modeled as follows: (c) the schema σ_γ for the generic rule uses the variable y^r for $r = (3, 2, 2)$, as y occurs trice in γ, and twice in its pattern and replacement. (d) The meta-productions $\Delta_{\mathcal{R},\Pi} = \{\delta^r_\pi, \delta^r_{\pi'}\}$ reproduce the productions. Note that the contextual node in production π is *not* reproduced in meta-productions δ^r_π. (e) A derivate ρ of the schema (using $\delta^r_{\pi'}$ twice and δ^r_π once) equals an instance of γ using the productions π' twice and π once, for each occurrence of y. By using the contextual meta-production $\tilde{\delta}^r_\pi$, the contextual node in the derivate ρ would be identified with another node its body, which has to be in the intersection of the pattern and replacement to satisfy the gluing condition.

Fact 4. A rule ρ is the instance of a generic rule γ if and only if it is a derivate of the schema ρ of γ.

4.3 Existence of Critical Overlaps

Two rules $\rho = (P \hookrightarrow B \hookleftarrow R)$ and $\rho' = (P' \hookrightarrow B' \hookleftarrow R')$ *overlap critically* if there exists a graph G with matches $m\colon P \to G$ and $m'\colon P' \to G$ that intersect in deleted nodes or edges, i.e., if $m(P) \cup m'(P') \not\leftrightarrow m(P \cap R) \cup m'(P' \cap R')$. It suffices to check, in the finite number of *minimal overlaps* (where every node and edge of G is in the image either of P or of P'), whether the matches are parallel independent in the sense of [5, Def. 5.9] or not. In the case of schema refinement, the more interesting case is whether two schemata may have refinements that overlap critically, or not. Since schemata have infinitely many derivates in general, this question is harder to answer. Unfortunately, we obtain:

Theorem 3. *For rule schemata σ_1 and σ_2 with meta-productions Δ, it is, in general, undecidable whether or not σ_1 and σ_2 have derivates $\rho_1 \in \Delta(\sigma_1)$ and $\rho_2 \in \Delta(\sigma_2)$ with critical overlaps.*

Proof. (By contradiction.) We reduce this problem to deciding whether context-free languages are disjoint, which is known to be undecidable [16, Thm. 14.3].

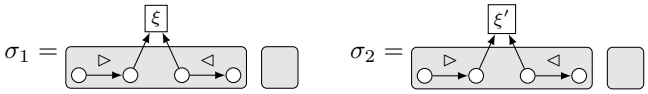
We define a representation of words as string graphs, and context-free rules as context-free productions over string graphs.

(1) Let $\dot{\mathcal{C}} = \{\circ\}$ and let all variable names $\xi \in \mathcal{X}$ have arity $\mathit{arity}(\xi) = \circ\circ$.

(2) A graph G with nodes $\dot{G} = \{v_o, v_1, \ldots, v_n\}$, edges $\vec{G} = \{e_o, e_1, \ldots, e_n\}$ where $\mathit{att}_G(e_i) = v_{i-1}v_i$ for $1 \leqslant i \leqslant n$ is a *string graph* representing the word $w = \vec{\ell}^*_G(e_1 \ldots e_n) \in \vec{\mathcal{C}}^*$. The string graph representing a word w is unique up to isomorphism, and denoted by $w^\bullet$.

(3) Now a proper context-free rule $r = (\xi, w) \in \mathcal{X} \times \vec{\mathcal{C}}^+$ can be represented as a meta-production δ_r with a spine production $\pi = (P_\pi \hookrightarrow B_\pi \hookleftarrow R_\pi)$, where $P = \xi^\bullet$, $R = w^\bullet$, and, in B, P and R are disjoint up to their start and end nodes. It is shown in [10] that these productions perform context-free derivations on string graphs. In the body rule $\rho = (P_\rho \hookrightarrow B_\rho \hookleftarrow R_\rho)$ of δ, $P_\rho = B_\pi$ and R_ρ is empty.

Now consider schemata where the string graphs for ξ and ξ' are enclosed in edges labeled with triangles (as start and end markers):



Then derivates $\rho_1 = (P'_1 \hookrightarrow B'_1 \hookleftarrow R'_1) \in \Delta(\sigma_1)$ and $\rho_2 = (P'_2 \hookrightarrow B'_2 \hookleftarrow R'_2) \in \Delta(\sigma_2)$ are parallelly dependent if $P'_1 \cong P'_2$. By the definition of derivates, and meta-productions, this implies that $P'_1 \cong P'_2 \cong w^\bullet$ for some word $w \in (\vec{\mathcal{C}} \setminus \mathcal{X})^*$, i.e., if w is in the context-free string languages derived from ξ and from ξ', respectively. In other words, σ_1 and σ_2 are parallelly independent if the languages derived from ξ and ξ' have an empty intersection.[4] □

Note that this result holds even if the meta-productions are context-free.

5 Conclusions

In this paper we have defined how schemata of plain graph rewrite rules can be refined with contextual meta-productions. This kind of rules has been implemented in the GrGen rewriting tool [1]. Graph rewriting with schema refinement allows

[4] There is no loss of generality if we assume that the context-free rules are proper, i.e., have non-empty right-hand sides, since every context-free grammar can be transformed into one where only the start symbol has a rule with an empty right-hand side. Then the question is still undecidable for the sublanguages without the empty word ε.

to model graph rewriting with SPO, NCE, and generic rules. Although schema refinement is based on DPO, some properties are lost since schemata may have infinitely many derivates. So it is not decidable whether schemata have derivates with critical overlaps, or not.

Until now, rules, (meta-) productions, and schemata are unconditional. Recently, the theory of plain graph rewriting has been extended to rules with nested application conditions [6], and with "HR-conditions" proposed in [12] where the graphs in nested conditions can be refined by context-free productions as well. The constructions of this paper shall be extended by such application conditions as well. We also work on a transformation of schema refinement into sets of plain rules that are applied according to a certain strategy (which applies transformed meta-productions as long as possible). This will allow us – hopefully – to determine advanced properties of schema refinement, like confluence and termination, by analyzing the transformed rules. Because, this is a major motivation of for future work: to provide assistance for users of GrGen in analyzing the behavior of their specifications.

Acknowledgments. I wish to thank several people: Katharina Saemann and Ruth Schönbacher found an obvious error in Def. 5, and Giorgio Busatto, Annegret Habel, and Hendrik Radke have lent their ears (and brains) to simplify and consolidate early drafts of this paper. Frank Drewes carefully reviewed a later version of the paper. Edgar Jakumeit conceived and implemented recursive refinements for rules in the GrGen rewriting tool, under the supervision of Rubino Geiß.

References

1. J. Blomer, R. Geiß, and E. Jakumeit. GrGen.net: A generative system for graph-rewriting, user manual. www.grgen.net, 2006-2012. Version V3.5RC.
2. F. Drewes, B. Hoffmann, D. Janssens, and M. Minas. Adaptive star grammars and their languages. *Theoretical Computer Science*, 411:3090–3109, 2010.
3. 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.
4. 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. Springer, 2012. To appear.
5. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs. Springer, 2006.
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, R. Heckel, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation part II: Single pushout approach and comparison to double pushout approach. In Rozenberg [19], chapter 4, pages 247–312.
8. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [19], chapter 1, pages 1–94.
9. H. Göttler. Semantical descriptions by two-level gaph-grammars for quasi-hierarchical graphs. In M. Nagl and H.-J. Schneider, editors, *Graphs, Data Structures, Algorithms (WG'79)*, number 13 in Applied Computer Science, pages 207–225, München-Wien, 1979. Carl-Hanser Verlag.
10. A. Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.
11. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
12. 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).
13. W. Hesse. Two-level graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 255–269. Springer, 1979.
14. B. Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.
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. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.
17. S. Kuske. More about control conditions for transformation units. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764 in Lecture Notes in Computer Science, pages 323–337. Springer, 2000.
18. D. Plump and A. Habel. Graph unification and matching. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
19. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
20. A. Schürr. Programmed graph replacement systems. In Rozenberg [19], chapter 7, pages 479–546.
21. T. Uesu. A system of graph grammars which generates all recursively enumerable sets of labelled graphs. *Tsukuba J. Math.*, 2:11–26, 1978.

Parallel Graph Grammars with Instantiation Rules Allow Efficient Structural Factorization of Virtual Vegetation

Katarína Smoleňová[1], Winfried Kurth[1], and Paul-Henry Cournède[2]

[1] Department Ecoinformatics, Biometrics and Forest Growth, Georg-August University of Göttingen, 37077 Göttingen, Germany
ksmolen@gwdg.de,wk@informatik.uni-goettingen.de
[2] Ecole Centrale Paris, France
paul-henry.cournede@ecp.fr

Abstract. Parallel rewriting of typed attributed graphs, based on the single-pushout approach extended by connection transformations, serves as the backbone of the multi-paradigm language XL, which is widely used in functional-structural plant modelling. XL allows to define instantiation rules, which enable an instancing of graphs at runtime for frequently occurring substructures, e.g., in 3-d models of botanical trees. This helps to save computer memory during complex simulations of vegetation structure. Instantiation rules can be called recursively and with references to graph nodes, thus providing a unifying formal framework for various concepts from the literature: object instancing, structural factorization, Xfrog multiplyer nodes, L-systems with interpretation. We give simple examples and measure the computation time for an idealized growing virtual plant, taken from the GreenLab model, in its implementation with instantiation rules in XL, compared to a version without instantiation rules.

Keywords: instantiation rules, structural factorization, XL, growth grammar

1 Introduction

Realistic simulation of large vegetation scenes or growing plant structure may become computationally very intensive if no methods to reduce the geometrical complexity are used. To save computer memory, different techniques have already been proposed in the field of computer graphics. When employing one of them, object instancing [21], the geometric representation is specified just for one or more representative plants (master). Other identical plants (instances) are placed in the scene, having a reference to the master object. Instances may be exact copies of the referenced object [8] or approximate ones [4, 2].

Thanks to the repetitive nature of plants, instancing may be applied on several levels. To efficiently render trees that look realistic from both distant and close views Max [15] modelled larger structures from smaller ones: a leaf is

used to form a twig with several leaves, several twigs form a big twig, big twigs form a subbranch, and so on. Cournède proposed a mathematical formalism [3] for decomposition of a plant into substructures [23], used in the GreenLab model [3, 14], that allows fast simulation (deterministic and stochastic) and relies on botanical principles. It assumes that all organs of the same kind, created at the same time, behave identically. This botanically-based formalism of "structural factorization" of plants is of special interest in the area of functional-structural plant modelling.

In higher plants, the elementary botanical units named metamers or phytomers are set in place rhythmically or continuously depending on the type of plants [1]. In the rhythmic case, the plant grows by successive shoots of several metamers produced by buds. The appearance of these shoots defines the architectural growth cycle. A growth unit is the set of metamers built by a bud during a growth cycle. Plant growth is said continuous when meristems keep on functioning and generate metamers one by one. The number of metamers on a given axis (that is to say generated by the same meristem) is generally proportional to the sum of daily temperatures (above a reference temperature depending on the species) received by the plant. In such case, the growth cycle is defined as the thermal time unit necessary for a meristem to build a new metamer. In both continuous and rhythmic cases, the chronological age of a plant (or of an organ) is defined as the number of growth cycles it has existed for, and the organogenesis is used as the time step to model the plant as a discrete dynamical system.

Quite naturally, owing to the simultaneous functioning of the meristems, parallel string rewriting grammars, i.e. L-systems [17], are most widely used to model plant organogenesis. If only organogenesis is modelled, the formalism of structural factorization can be expressed by L-systems [3]. The possibility to factorize L-system productions was pointed out already by Smith [19]. Improvement of the efficiency of an L-system-based algorithm for creation of virtual plants using factorization was also discussed in [6]. The use of instancing for improved representation of L-system-based models was explored, e.g., in [8, 2].

Widely used in the area of functional-structural plant modelling is the language XL (eXtended L-system language) [11]. XL is a superset of the language Java and supports the specification of graph grammars. It also allows to define instantiation rules, which enable an instancing of graphs for frequently occuring substructures, e.g., in 3-d models of botanical trees.

We first give a theoretical background on the language XL, including the formal definition of instantiation rules. Afterwards we show how to use these rules to produce inflorescence patterns, fractals, and to efficiently simulate plant development based on structural factorization.

2 Theoretical Background

The graph model which describes the basic data structure of the language XL is that of *typed attributed graphs with inheritance over a type graph* [11] (p. 98). "Typed" means that nodes of the graph belong to a finite number of classes,

called "modules" in XL, and in an application of a graph grammar rule, the types of matching nodes have to coincide. Likewise, the edges belong to a finite number of edge types, which have also to be taken into account during graph matching. For a given edge type, at most one edge of this type is allowed between two given nodes. The edge types can thus be interpreted as *relations* on the set of nodes. A "type graph" can be specified to restrict the allowed types of edges between nodes of given types. "Inheritance" is a concept taken from object-oriented programming and allows a hierarchy among the node types. "Attributed" refers to the nodes which can have a finite number of attributes of arbitrary data type.

The parallel application of a *graph grammar* on a graph conforming to this model is in [11] (p. 101ff.) defined as a special case of the *single-pushout approach* from algebraic graph grammar theory [7] extended by connection transformations and operators. The latter provide an embedding mechanism for the right-hand sides of rules into the host graph which allows to consider classical parallel string-rewriting rules (L-system rules) as a special case of graph rewriting.

In the following, we simplify the definitions from [11] in order to avoid too much technical overhead. Particularly, we omit the attributes, the inheritance relation and the restriction imposed by a type graph. "Instantiation rules" as a special sort of graph transformation rules have not been formally defined in [11], although they have been described as part of the language XL there and have been used in various examples.

Definition 1. A Relational Growth Grammar graph (*RGG graph*) is a quintuple $G = (T_N, T_E, N, E, t)$, where $T_N \neq \emptyset$ is a finite set of node types, $T_E \neq \emptyset$ a finite set of edge types, N a set of nodes, $E \subseteq N \times T_E \times N$ a set of typed edges, and $t : N \to T_N$ a node typing function. An *RGG graph isomorphism* is a graph isomorphism which preserves node and edge types.

The language XL makes use of a string notation of RGG graphs which is demonstrated in Fig. 1.

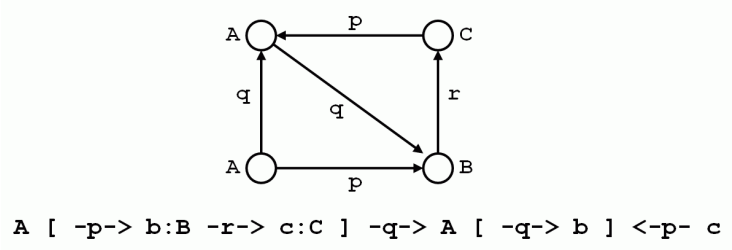


Fig. 1. Example of an RGG graph G with node types A, B, C and edge types p, q, r. Given is also a string notation $s(G)$ of the graph in XL (not unique). b and c are labels referring to particular nodes.

Definition 2. A *bi-rooted RGG graph* (G, n_0, n_1) is an RGG graph with two distinguished nodes $n_0, n_1 \in N$.

Definition 3. An (unconditional) *RGG rule* with L-system style embedding is a pair $((G_1, m_0, m_1), (G_2, n_0, n_1))$ of two bi-rooted RGG graphs (G_1, m_0, m_1) (the *left-hand side* or *search pattern*) and (G_2, n_0, n_1) (the *right-hand side* or *production pattern*). The *application* of an RGG rule to a host RGG graph H consists of the simultaneous replacement of all subgraphs of H which are isomorphic (via an RGG graph isomorphism) to G_1 by G_2, with a subsequent redefinition of some of the incoming and outgoing edges: Each edge going from the rest of the host graph to the node m_0 in G_1 is redirected to n_0, and each edge going from m_1 in G_1 to the rest of the host graph gets n_1 as its new start node. All other edges going into or out of G_1 are deleted.

In the case that an edge goes from m_1 in one isomorphic copy of G_1 to m_0 in another copy, an edge of the same type is inserted from n_1 of the first copy of G_2 to n_0 of the second copy of G_2 (see [11] for a more rigorous definition). In the case of overlapping copies of G_1 in H, the result of rule application is undefined.

For shortness, we write `G1 ==> G2` for an RGG rule, omitting the nodes m_0, m_1, n_0 and n_1. In the language XL, the distinguished nodes of an RGG rule are not given explicitly, but are taken from the string representation of the graphs: m_0 is the leftmost and m_1 the rightmost node in $s(G_1)$, and analogously for G_2. For instance, the application of the RGG rule `A ==> D -r-> E` to the graph in Fig. 1 will result in the graph depicted in Fig. 2.

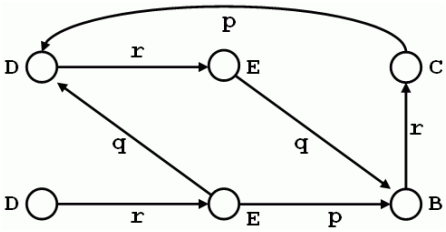


Fig. 2. Result of a parallel rule application to the RGG graph of Fig. 1 (see text).

The advantage of this form of embedding the right-hand side into the host graph is its consistency with parallel string rewriting by L-systems. A string which is transformed by an L-system can simply be represented by a linear chain of nodes, connected by edges of just one special type, which we call "successor edges". The symbols of the L-system correspond to the node types.

Definition 4. An *RGG* (Relational Growth Grammar) (T_N, T_E, G_0, R) consists of a finite set of node types $T_N \neq \emptyset$, a finite set of edge types $T_E \neq \emptyset$, a start RGG graph (axiom) G_0 and a finite set of RGG rules R, with G_0 and R having all node and edge types from T_N, resp., T_E.

The parallel application of rules from an RGG to a host graph is performed in the same way as in the case of a single rule. In XL, all node types in T_N have to be declared first, using the keyword "module" (also possible: "class"). XL allows to split the set of rules into several subsets which are accessible under user-defined names. Control structures like conditional branches or loops can be used. Likewise, these control structures can also occur on the r.h.s. of rules to control the production of subgraphs. We omit these extensions in our definitions because they are not central for the notion of instantiation rules.

Definition 5. An *instantiation rule* is an RGG rule $N ==> G$, where the l.h.s. consists only of a single node of type N. An *RGG with instantiation rules* is a quintuple (T_N, T_E, G_0, R, I) where the first 4 components form an RGG and I is a finite set of instantiation rules with node and edge types from T_N, resp., T_E.

The application of an RGG with instantiation rules is performed as shown in Fig. 3. $G_1, G_2, ...$ are the RGG graphs which are successively obtained from the start graph G_0 by parallel application of (potentially) all rules from R in each transformation step. They usually correspond to rough structural descriptions of the simulated objects at discrete timesteps. $S_0, S_1, ...$ are the RGG graphs which are obtained from $G_0, G_1, ...$ by parallel application of (potentially) all rules from I. These rules can also be applied recursively. The graphs S_i usually stand for detailed geometrical descriptions (scene graphs) of the simulated objects. $S_0, S_1, ...$ can be regarded as a simulated developmental sequence of the objects. Note that this way of applying two sets of rules is not equivalent to that used in "table L-systems", since the graphs S_i do not undergo a new application of rules in the next step. Nondeterminism is possible for both rule sets, R and I, when it makes sense in an application, and can be resolved in XL by stochastic choice of rules with given probabilities. When XL is used, the graphs S_i are not stored as data structures in the computer memory, but are derived "on the fly" by applying rules from I to G_i at runtime in the moment when they are needed (e.g., for rendering an image of the simulated structures). In the case of large graphs, this can lead to a gain in efficiency.

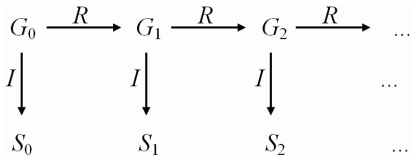


Fig. 3. Way of operation of an RGG with normal RGG rules R and instantiation rules I, starting from RGG graph G_0.

The same scheme of rule application is also valid for L-systems or graph grammars with *interpretive rules* [11] (also called *interpretative* [12] (p. 25), *interpretation* [18], or *homomorphism rules* [16]). Interpretive rules differ from

instantiation rules in the feature that multiple instances of a right-hand side of an interpretive rule which are obtained from G_i are stored as different copies (being parts of S_i) and can thus later be accessed and manipulated individually. This is not possible for the results of instantiation rule application. The latter are, however, less demanding in terms of computation space and time.

A further extension of the concept of instantiation rule allows to construct graphs which encode in a concise manner the way how objects or groups of objects are multiplied and transformed in space, including recursive constructions leading to fractals. As special cases, the multiplicator nodes used in the interactive plant modelling software Xfrog [5] can be derived this way [9].

Definition 6. An *RGG with instantiation rules and references* is an RGG where the right-hand sides of the instantiation rules can contain specific nodes of the types $Ref(e)$, where e is an edge type. When applied to a node a in the host graph H, each node of type $Ref(e)$ is replaced by a copy of the subgraph of H which is directly connected with a via an edge of type e emerging from a, if such an edge exists (otherwise the $Ref(e)$ node is deleted). In case of more than one edge of type e starting in a, the choice of the adjacent subgraph is nondeterministic.

For instance, the instantiation rule `R ==> Ref(p) -q-> Ref(p)`, applied to the host graph `R -p-> A`, will result in the graph `A -q-> A -p-> A`. Since in most applications, the edges of type `p` will be ignored during further processing (e.g., rendering - only the subgraph spanned by a special subset of edge types is used for rendering in XL), we will effectively have a copy of the referenced `A` node: `A -q-> A`. This can be extended to more complex copy operations, as we will see in the examples sections. In the language XL, the `Ref` nodes are effectively realized by calls of the method `getFirst(e)`.

3 Examples

3.1 Inflorescence Patterns and Fractals

We will first show two simple examples, providing the complete code in the language XL, which demonstrate that the concepts introduced above can be expressed in XL in a straightforward way. The first example (code listed in Table 1) defines a node "`Infl`" (standing for "inflorescence") by an instantiation rule with references. Similar to the effect of the "Hydra" node from Xfrog [5], the execution of this rule creates 250 scaled copies of the structure attached via an edge of type m and arranges them in a pattern following spiral phyllotaxis (Fig. 4, right-hand part; see [9] for the emulation of the other Xfrog arrangement nodes in XL).

In XL, edges of the standard type "successor" are automatically generated when nodes are separated by blanks (or linefeeds), as in the right-hand side of the rule in line 9. An edge type other than the standard ones requires a declaration of a corresponding identifier (here "`m`", done in line 1) and can then

Table 1. Listing of the XL code for the inflorescence example (see text).

```
 1 const int m = EDGE_0;
 2 module Infl ==> for (int i: 1:250) ([
 3    { float h = i * 0.02;
 4      float s = 0.2 * Math.sqrt(i); }
 5    M(-h) RH(i*137.5) Translate(s,0,0) RU(i*80/250)
 6    Scale(0.2,0.2,0.3*h+0.1) getFirst(m)
 7 ]);
 8 public void run() [
 9    Axiom ==> P(10) Cylinder(20, 1) Cone(5.2, 2.4) P(14)
10              Infl -m-> Sphere;
11 ]
```

be used in the form "`-m->`" (line 10), here to connect the node of type `Infl` with a node corresponding to a geometrical object, here a `Sphere` with default radius 1. `Sphere` belongs, together with `Cylinder`, `Cone` and the geometrical transformation node types `M` (move), `Translate` (shift by a vector), `RH` (rotate around head direction), `RU` (rotate around up direction), `P` (set colour index) and `Scale` (rescale coordinates for the subsequent scene subgraph) to the default node types of XL, which do not need to be declared in the code, whereas `Infl` is a user-defined node type declared with an instantiation rule in lines 2-7. The "`for`" control structure in line 2 (with its body ending in line 7) iterates the production of the nodes which are specified in lines 5-6, together with the execution of the imperative statements in lines 3-4, with a counting index `i` running from 1 to 250. The calculations of `h` and `s` and the rotation angles and scaling factors in lines 5-6 were fitted to the botanical object to be modelled. The user can edit these lines to adapt the structure of the created inflorescence to empirical findings and has thus much more flexibility than with a given standard portfolio of multiplyer nodes as provided by the Xfrog software [5].

From the default start node called "Axiom", the execution of the "`run`" method (line 8) generates in one step of rule application a chain of four nodes with geometrical/visible meaning (line 9), followed by a node of type `Infl` and – attached by an `m` edge – a single `Sphere` node (line 10). This graph is saved internally (Fig. 4 left).

During rendering, the `Infl` node is further transformed by the instantiation rule: In each iteration of the loop (line 2), a separate branch (enclosed by brackets [ ], line 2 and 7) is created, consisting of geometry nodes (line 5) and a scaled copy of the sphere from line 10, which is found by following the `getFirst` reference (line 6) along the edge `-m->` from the processed `Infl` node in line 10. `getFirst(m)` corresponds to a reference node $Ref(m)$. This results in the visual arrangement of 250 ellipsoids shown in Fig. 4 (right part).

In an analogous way, the code from Table 2 produces after n steps of application of the `run` rules (lines 7-8) a chain of $n-1$ nodes of type `Rec` and

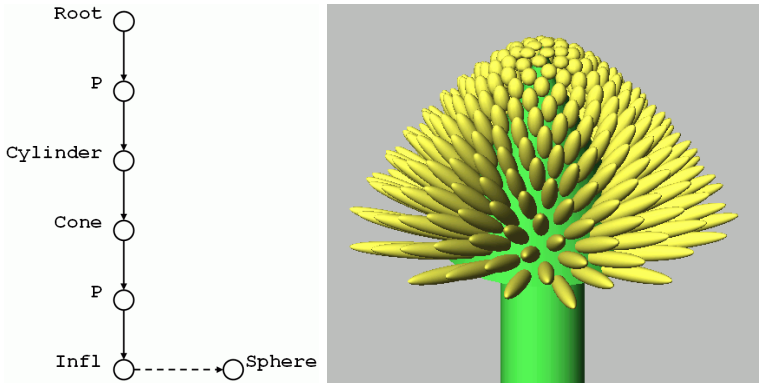


Fig. 4. Structure resulting from the inflorescence example (Table 1). Left: Generated graph after the application of the single non-instantiation rule (lines 9-10 in Tab. 1). Unbroken edges stand for the successor relation, broken edges for a type m connection. Right: Rendered view of the corresponding 3-d structure, taking the application of the instantiation rule (lines 2-7) into account.

one node of type `A` (Fig. 5, left part). The instantiation of the `Rec` nodes by the translated copies which are referenced in the instantiation rule in lines 3-5 renders this graph into a Sierpinski triangle of recursion depth n (Fig. 5, right part). Note that the stored graph is growing only linearly in n. The initiator `A` is here getting its shape from a sphere (line 2), but the shape of the initiator has no influence on the resulting fractal pattern for large n. The method can easily be generalized to other iterated function systems (IFS) or recurrent IFS (RIFS), cf. [8].

Table 2. XL code for Sierpinski triangle, version with instantiation rules with references.

```
1 const int m = EDGE_0;
2 module A extends Sphere(0.4);
3 module Rec ==> Scale(0.5) [ Translate(0, 0.5, 0)        getFirst(m) ]
4                           [ Translate(-0.433, -0.25, 0) getFirst(m) ]
5                           [ Translate(0.433, -0.25, 0)  getFirst(m) ];
6 public void run() [
7    Axiom ==> A;
8    A ==> Rec -m-> A;
9 ]
```

The Sierpinski triangle can also be generated by an instantiation rule in a purely recursive way, without references (Table 3). The generated graph is then even simpler: it consists of only two nodes, `Root` → `Rec`, and its structure does not

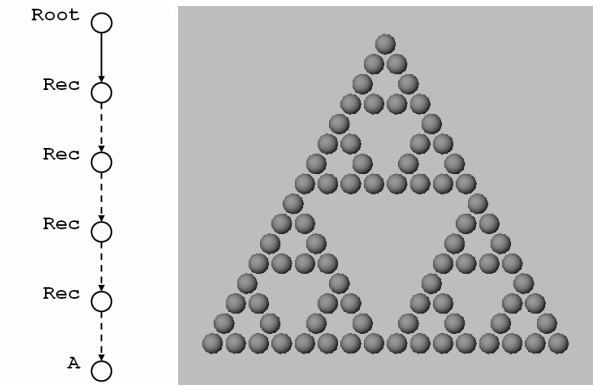


Fig. 5. Structure resulting from the Siepinski triangle example (Table 2). Left: Generated graph, right: rendered view of the corresponding 3-d structure.

change during rule application. Only the parameter n of the `Rec` node, denoting recursion depth, is incremented. A disadvantage is that the graph carries less visible information for the user.

Table 3. XL code for the Sierpinski triangle, recursive version without references (output as in Fig. 5, right-hand side).

```
1 module Rec(int n) ==> if (n == 0) ( Sphere(0.4) )
2    else ( Scale(0.5) [ Translate(0, 0.5, 0)        Rec(n-1) ]
3                      [ Translate(-0.433, -0.25, 0) Rec(n-1) ]
4                      [ Translate(0.433, -0.25, 0)  Rec(n-1) ] );
5 public void run()  [
6    Axiom ==> Rec(0);
7    Rec(n) ==> Rec(n+1);
8 ]
```

Both recursive versions of the XL code for the Sierpinski fractal are not optimal in terms of efficiency, since in an implementation on a real computer the size of the recursion stack will soon be limiting when n is increased. However, XL provides also the possibility to implement the rewriting rule of the Sierpinski triangle in a direct way (without instantiation), avoiding heavy stack loading, as was shown in [22] (p. 528). This solution turned out to be quite efficient and allowed to reach a recursion depth of 20, resulting in a graph with more than 5 million nodes.

3.2 Structural Factorization of Plant Architecture

In an individual plant, cohorts of similar organs are created at each growth cycle. Simulation models handle each of them individually, which may lead to cumbersome computation in the case of tree growth simulations, as the number of organs may exceed several millions. However, it is often not necessary to consider local environmental conditions at the organ level. Thus, we suppose that all organs of the same kind, created at the same growth cycle, behave identically. From a modelling point of view, this leads to a powerful structural factorization of the plant, based on botanical instantiations: organs are grouped by categories, for example based on their physiological ages, characterizing their morphogenetic differentiation, see [1]. Compact inductive equations of organogenesis can thus be deduced, as detailed in [3].

Let P be the maximum number of metamer categories that we consider in the plant. It is generally very small, inferior to 5. At growth cycle t, a metamer is characterized by its physiological age p, the physiological age of its axillary branches q (with $q \geq p$) and its chronological age n. It will be denoted by $m_{pq}^t(n)$. These three indices p, q, n are sufficient to describe all the metamers and their numbers grow linearly with t. A bud is only characterized by its physiological age p and will be denoted by s_p.

The terminal bud of a plant axis produces different kinds of metamers bearing axillary buds of various physiological ages. These buds themselves give birth to axillary branches and so on. A substructure is the complete plant structure that is generated after one or several cycles by a bud. In the deterministic case, all the substructures with the same physiological and chronological ages are identical if they were set in place at the same moment in the tree architecture. At cycle t, a substructure is thus characterized by its physiological age p and its chronological age n. It will be denoted by $S_p^t(n)$. Since the physiological age of the main trunk is 1, at growth cycle t, the substructure of physiological age 1 and of chronological age t, $S_1^t(t)$, represents the whole plant. Figure 6 illustrates how substructures are organized. The total number of different substructures in a plant of chronological age t is small, usually less than 30, even if the total number of organs is high. Substructures and metamers are repeated a lot of times in the tree architecture, but they need to be computed only once for each kind.

We use the concatenation operator (represented by the product symbol) to describe the organization of plant metamers and substructures and deduce their construction at growth cycle t by induction, as follows:

- Substructures of chronological age 0 are buds:
$$S_p^t(0) = s_p$$
- If we suppose that they built all substructures of chronological age $n-1$, we deduce the substructures of chronological age n

$$S_p^t(n) = \left[\prod_{p \leq q \leq P} \left(m_{pq}^t(n)\right)^{u_{pq}(t+1-n)} \left(S_q^t(n-1)\right)^{b_{pq}(t+1-n)} \right] S_p^t(n-1) \quad (1)$$

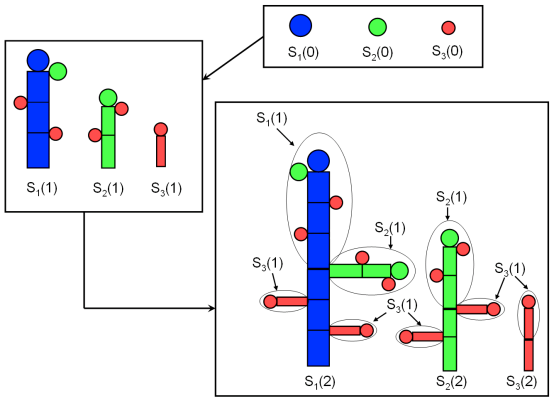


Fig. 6. Construction of substructures for a plant with deterministic development. The substructure of physiological age 1 and chronological age 2, $S_1(2)$, is built of the base growth unit, consisting of two metamers of type m_{13} and one metamer of type m_{12}, and of substructures of chronological age 1 (created in the previous step): two lateral substructures of physiological ages 2, and 3, and the terminal substructure of physiological age 1.

For all (p, q) such that $1 \leq p \leq P$, $p \leq q \leq P$, $\left(u_{pq}(t)\right)_t$ and $\left(b_{pq}(t)\right)_t$ are sequences of integers that are characteristic of the plant organogenesis: $u_{pq}(t)$ corresponds to the number of metamers m_{pq} in growth units of physiological age p appearing at growth cycle t; $b_{pq}(t)$ is the number of axillary substructures of physiological age q in growth units of physiological age p that appeared at growth cycle t. These sequences can be deterministic (fixed or determined by the functional part of the plant as detailed in [14]) or stochastic (the induction equation (1) would be generalized in such case with the generating functions of the number of elements in plant architecture, see [13]).

In equation 1, substructure $S_p^t(n)$ is decomposed into:

– its oldest growth unit, called base growth unit:

$$\prod_{p \leq q \leq P} \left(m_{pq}^t(n)\right)^{u_{pq}(t+1-n)}$$

– the lateral substructures borne by the base growth unit (they are one cycle younger):

$$\prod_{p \leq q \leq P} \left(S_q^t(n-1)\right)^{b_{pq}(t+1-n)}$$

– the substructure grown from the apical bud of the base growth unit (also one cycle younger):

$$S_p^t(n-1)$$

This decomposition is illustrated on $S_1(2)$ in Fig. 6. Note that for deterministic growth, the substructures are independent of growth cycle t. If we append geometrical rules (e.g., internode lengths, branching angles, phyllotaxy) to the structural equations, we will obtain the 3-d architecture of a geometrical tree.

The (deterministic) construction of substructures can be directly translated to XL code, using instantiation, as shown in Table 4. The phyllotactic and branching angles are part of the instantiation rule, with parameters `ang_ph` and `ang_br`, respectively. The arrays `u`, `b` stand for u_{pq}, resp. b_{pq}, with values set as in [10]. The resulting structure is shown in Fig. 7 (middle, right). The graph generated after the rule applications consists, similarly to the purely recursive Sierpinski triangle example, of two nodes, `Root -> Substructure`.

Table 4. XL code for plant development based on structural factorization, recursive version with an instantiation rule.

```
 1 module Bud(int p) extends Sphere
 2 { ... /* set radius, shader */ }
 3 module Metamer(int p, int q) extends Cylinder
 4 { ... /* set radius, length, shader */ }
 5 module Substructure(int p, int n) ==>
 6    if (n > 0) (
 7       for (int j = 5; j >= p-1; j--) (
 8          for (int i = 0; i < u[p-1][j]; i++) (
 9             RH(ang_ph[p-1]) Metamer(p, j+1)
10             for (int k = 0; k < b[p-1][j]; k++) ( [
11                if (k > 0) ( RH(360.0 / b[p-1][j] * k) )
12                RU(ang_br[p-1][j]) Substructure(j+1, n-1)
13             ] )
14          )
15       )
16       Substructure(p, n-1)
17    ) else ( Bud(p) );
18 public void run() [
19    Axiom ==> Substructure(1, 0);
20    Substructure(p, n) ==> Substructure(p, n+1);
21 ]
```

We have modified the recursive algorithm from Table 4 to store the substructures of all physiological and chronological ages (as shown in Fig. 6) in a list, so that they could be reused as instances later on. An example of the underlying graph is shown in Fig. 7 (left).

Furthermore, our basic algorithm for plant development, not taking advantage of structural factorization and instantiation rules, was presented in [20].

To compare the computation cost of structure generation in the three abovementioned implementations of plant development, simulation time for the development of the same virtual plant (Fig. 7, cf. [10, 20]) was measured on a

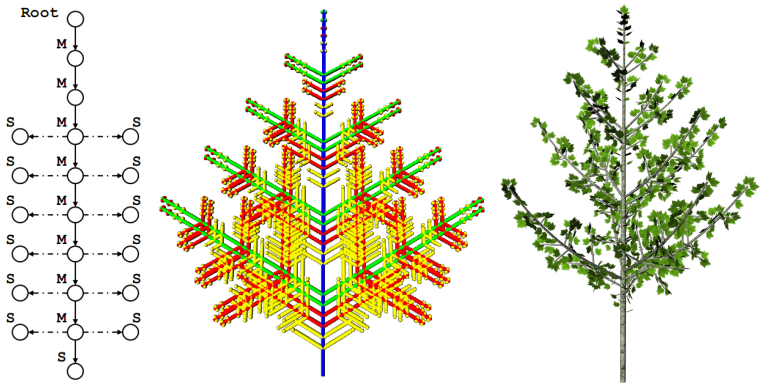


Fig. 7. Model example. Left: The essential structure of the generated graph for the substructure of physiological age 1 and chronological age higher than 0. It consists of metamers (`M`) of the base unit and corresponding substructures (`S`). Lateral substructures are connected to metamers by a branching edge (dash-dot style). Middle: Simulated topology at growth cycle 5 for $S_1(5)$. Right: The same topological structure with improved geometrical representation.

computer with Intel Core i7 CPU 950, 3.07 GHz, and 12 GB RAM. We show the results in Table 5 and Fig. 8.

Table 5. Simulation time (in ms) for the growing plant from Fig. 7 (without rendering). w/o SF (A) refers to the basic algorithm without structural factorization from [20], w/ SF (B) refers to the recursive algorithm from Table 4, and w/ SF (C) shows the performance of the algorithm that stores the substructures in a list.

Plant age	Metamers	w/o SF (A)	w/ SF (B)	w/ SF (C)	Ratio A/C	Ratio C/B
5	2,440	30.96	0.29	6.55	4.7	22.6
10	44,480	618.61	0.59	14.39	43.0	24.4
15	238,120	3,915.22	0.83	23.17	169.0	27.9
20	775,360	15,421.00	1.06	31.38	491.4	29.6
25	1,928,200	40,540.11	1.39	41.39	979.5	29.8
30	4,048,640	103,633.07	1.71	50.63	2,046.9	29.6

4 Conclusions

Object instancing is an established technique improving the performance when modelling and rendering vegetation. Giving the theoretical background first, we presented the features of the XL language, supporting instancing of graphs, i.e., instantiation rules. We demonstrated their use for modelling inflorescence patterns and fractals, and finally for the structural factorization of deterministically growing plants, decreasing the computational cost greatly. We would

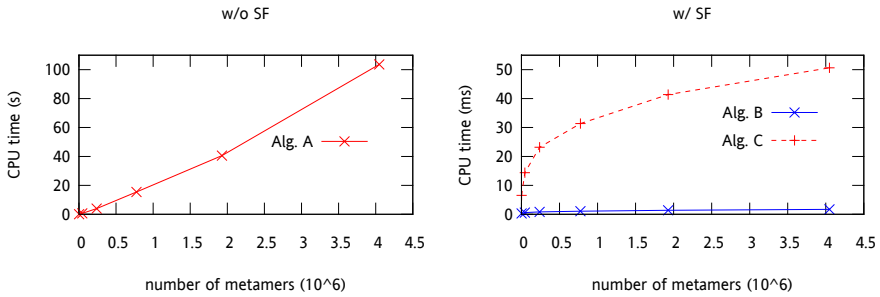


Fig. 8. Simulation time for the growing plant from Fig. 7 (for algorithms A, B, C according to Table 5).

further like to explore which algorithm, implementing structural factorization, is most suitable for modelling stochastic structures and physiological processes within plants.

Acknowledgments We thank Reinhard Hemmerling and Ole Kniemeyer for helpful comments. This research was funded by the German Research Foundation (DFG) under project identifier KU 847/8-1.

References

1. Barthélémy, D., Caraglio, Y.: Plant Architecture: A Dynamic, Multilevel and Comprehensive Approach of Plant Form, Structure and Ontogeny. Annals of Botany 99, 375–407 (2007)
2. Brownbill, A.: Reducing the Storage Required to Render L-system Based Models. Masters Thesis, University of Calgary (1996)
3. Cournède, P.-H., Kang, M.-Z., Mathieu, A., Barczi, J.-F., Yan, H.-P., Hu, B.-G., de Reffye, P.: Structural Factorization of Plants to Compute Their Functional and Architectural Growth. Simulation 82, 427–438 (2006)
4. Deussen, O., Hanrahan, P., Lintermann, B., Měch, R., Pharr, M., Prusinkiewicz, P.: Realistic modeling and rendering of plant ecosystems. In: 25th Annual Conference on Computer Graphics and Interactive Techniques, pp. 275–286. ACM, New York (1998)
5. Deussen, O., Lintermann, B.: Digital Design of Nature: Computer Generated Plants and Organics. Springer-Verlag, Berlin Heidelberg (2005)
6. Ding, W.-L., Zhang, W.-T., Zhou, X.: An Improved Algorithm Based on Sub-Structures for Creating Virtual Plant. In: 16th IEEE International Conference on Artificial Reality and Telexistence–Workshops, pp. 200–204. IEEE Computer Society Press, Los Alamitos (2006)
7. 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: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, pp. 247–312. World Scientific, River Edge (1997)

8. Hart, J. C.: The Object Instancing Paradigm for Linear Fractal Modeling. In: Conference on Graphics Interface, pp. 224–231. Morgan Kaufmann Publishers Inc., San Francisco (1992)
9. Henke, M.: Entwurf und Implementation eines Baukastens zur 3D-Pflanzenvisualisierung in GroIMP mittels Instanzierungsregeln. Masters Thesis, University of Technology at Cottbus (2006)
10. Kang, M.-Z., de Reffye, P., Barczi, J.-F., Hu, B.-G., Houllier, F.: Stochastic 3D tree simulation using substructure instancing. In: International Symposium on Plant growth Modeling, simulation, visualization and their Applications, pp. 154–168. Springer / Tsinghua University Press, Beijing (2003)
11. Kniemeyer, O.: Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling. Ph.D. dissertation, BTU Cottbus (2008)
12. Kurth, W.: Growth Grammar Interpreter GROGRA 2.4: A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modelling. Introduction and reference manual. Ber. FZW Göttingen, Ser. B, vol. 38 (1994)
13. Loi, C., Cournède, P.-H.: Generating functions of stochastic L-systems and application to models of plant development. In: Fifth Colloquium on Mathematics and Computer Science, pp. 325–338 (2008)
14. Mathieu, A., Cournède, P.-H., Letort, V., Barthélémy, D., de Reffye, P.: A dynamic model of plant growth with interactions between development and functional mechanisms to study plant structural plasticity related to trophic competition. Annals of Botany 103, 1173–1186 (2009)
15. Max, N. L.: Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In: Eurographics Workshop on Rendering Techniques, pp. 165–174. Springer-Verlag, Wien (1996)
16. Měch, R.: CPFG Version 4.0 User's Manual, `http://algorithmicbotany.org/lstudio/CPFGman.pdf`.
17. Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer-Verlag, New York (1990)
18. Prusinkiewicz, P., Hanan, J., Měch, R.: An L-system-based plant modeling language. In: Nagl, M., Schürr, A. Münch, M. (eds.) AGTIVE'99. LNCS, vol. 1779, pp. 395–410. Springer-Verlag, Berlin, Heidelberg (2000)
19. Smith, A. R.: Plants, fractals, and formal languages. Computer Graphics 18, 1–10 (1984)
20. Smoleňová, K., Henke, M., Kurth, W.: Rule-based integration of GreenLab into GroIMP with GUI aided parameter input. In: 4th IEEE International Symposium on Plant Growth Modeling, Simulation, Visualization and Application (accepted) (2012)
21. Sutherland, I. E.: Sketchpad: A man-machine graphical communication system. In: Spring Joint Computer Conference, pp. 329–346. ACM, New York (1963)
22. Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiger, L., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T.: Generation of Sierpinski triangles: A case study for graph transformation tools. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE'07. LNCS, vol. 5088, pp. 514–539. Springer-Verlag, Berlin, Heidelberg (2008)
23. Yan, H.-P., de Reffye, P., Pan, C.-H., Hu, B.-G.: Fast construction of plant architectural models based on substructure decomposition. Journal of Computer Science and Technology 18, 780–787 (2003)

Generalized Cayley Graphs and Cellular Automata over them

Pablo Arrighi[1,2] and Simon Martiel[3]

[1] Université de Grenoble, LIG, 220 rue de la chimie, 38400 SMH, France
[2] Université de Lyon, LIP, 46 allée d'Italie, 69008 Lyon, France
parrighi@imag.fr
[3] Université Nice-Sophia Antipolis, I3S, 2000 routes des Lucioles, 06900 SA, France
simon.martiel@gmail.com

Abstract. Cayley graphs have a number of useful features: the ability to graphically represent finitely generated group elements and their equality; to name all vertices relative to a point; the fact that they have a well-defined notion of translation, and that they can be endowed with a compact metric. We propose a notion of graph associated to a language, which conserves or generalizes these features. Whereas Cayley graphs are regular; associated graphs are arbitrary, although of a bounded degree. Moreover, it is well-known that cellular automata can be characterized as the set of translation-invariant continuous functions for a certain compact metric, which makes it easy to extend their definition from grids to Cayley graphs. Similarly, we extend their definition to these arbitrary, bounded degree, time-varying graphs.
Keywords. *Causal Graph Dynamics, Curtis-Hedlund-Lynden, Dynamical networks, Boolean networks, Generative networks automata, Graph Automata, Graph rewriting automata, Parallel graph transformations, Amalgamated graph transformations, Time-varying graphs, Regge calculus, Local, No-signalling.*

Introduction

Cayley graphs. Cayley graphs are graphs associated to a finite set of generators of a group; together with their inverses. For instance let this set be $\pi = \{a, a^{-1}, b, b^{-1}, \ldots\}$. Then the vertices of the graph can be designated by the terms $\pi^* = \{a, a^2, \ldots, a^{-1}, \ldots, a.b, \ldots\}$, but more precisely they are the equivalence classes of these terms with respect to the group equivalence $\equiv$, e.g. $b^{-1}.b.a \equiv a$. The edges on the other hand are those pairs $(u\!:\!a, v\!:\!a^{-1})$ such that there exists a in π so that $u.a \equiv v$. Cayley graphs have been used intensively because they have a number of useful features:

- Once a point representing the identity has been chosen, all other vertices can be named relative to that point.
- The resulting graph represents the group, i.e. the set of terms and their equality.

– There is a well-defined notion of translation of the graph, which corresponds to changing the point representing the identity, or equivalently applying an element of the group to all vertices.
– The graph can be endowed with a compact Hausdorff metric.

In this paper, we propose a notion of graph associated to an adjacency language L and its equivalence relation $\equiv_L$, which conserves or generalizes these features. Whereas Cayley graphs are very regular; associated graphs are arbitrary, albeit of a bounded degree.

Cellular Automata. Cellular Automata (CA) consist of a grid of identical square cells, each of which may take one of a finite number of possible states. The entire array evolves in discrete time steps. The time evolution is required to be translation-invariant (it commutes with translations of the grid) and causal (information cannot be transmitted faster than a fixed number of cells per time step). Whilst Cellular Automata are usually defined as exactly the functions having those physics-like symmetries, it turns out that they can also be characterized in purely mathematical terms as the set of translation-invariant continuous functions [9] for a certain compact Hausdorff metric. As a consequence CA definitions are quite naturally extended from grids to Cayley graphs, where most of the theory carries though [16, 3]. Moving on, there have been several approaches to generalize Cellular Automata not just to Cayley graphs, but to arbitrary graphs of bounded degree:

– With a fixed topology [15, 4, 8], in order to describe certain distributed algorithms.
– Through the simulation environments of [7, 22, 13] which offer the possibility of applying a local rewriting rule simultaneously in different non-conflicting places.
– Through concrete instances advocating the concept of CA extended to time-varying graphs as in [21, 12, 11], some of which are advanced algorithmic constructions [20, 19].
– Through Amalgamated Graph Transformations [2, 14] and Parallel Graph Transformations [5, 17, 18], which work out rigorous ways to apply a local rewriting rule synchronously throughout a graph.

The approach of this paper is different in the sense that it first generalizes Cayley graphs, and then applies the mathematical characterization of Cellular Automata as the set of translation-invariant continuous functions in order to generalize CA. Compared with the above mentioned CA papers, the contribution is to extend the fundamental structure theorems about Cellular Automata to arbitrary, bounded degree, time-varying graphs. Compared with the above mentioned Graph Rewriting papers, the contribution is to deduce aspects of Amalgamated/Parallel Graph Transformations from the axiomatic and topological properties of the global function.

Causal Graph Dynamics. The work by [1] by Dowek and one of the authors already achieves an extension of Cellular Automata to arbitrary, bounded degree,

time-varying graphs, also through a notion of continuity, with the same motivations. However, graphs in [1] lack a compact Hausdorff metric over graphs, which is left as an open question. As a consequence all the necessary facts about the topology of Cayley graphs get reproven. It also leaves open whether the notion of invertible causal graph dynamics is the most general one, and whether causal graph dynamics are computable. *Most of these issues vanish in the new formalism; which suggests that the new formalism itself is the main contribution of this paper.*
This paper. Section 1 provides a generalization of Cayley graphs. This takes the form of an isomorphism between graphs and languages endowed with an equivalence. Section 2 provides basic operations upon generalized Cayley graphs. Section 3 provides facts about the topology of generalized Cayley graphs. It follows that continuous functions are uniformly continuous, composable, and that their inverses are also continuous. Section 4 establishes a notion of Cellular Automata over generalized Cayley graphs. A theorem of equivalence between a mathematical and a constructive approach is given. Section 5 provides some examples. Section 6 shows that the instances of the model thereby obtained are recursively enumerable, and that their effect over finite graphs is computable; which grants our model the status of a model of computation.

1 Generalized Cayley graphs: definitions

Notations. All graphs are assumed to be connected. The *vertices* of the graphs we consider in this paper are uniquely identified by a name u in V. Vertices may also be labelled with a *state* $\sigma(u)$ in Σ a finite set. Each vertex has *ports* i in π a finite set. A vertex and its port are written $u:a$. An *edge* is a two element set $\{u:a, v:b\}$. The port of a node can only appear in one edge, so that the degree of the graphs is always bounded by $|\pi|$. Edges may also be labelled with a *state* $\delta(\{u:a, v:b\})$ in Δ a finite set. All languages defined are on the finite alphabet $\Pi = \pi^2$ with a suffix in $S = \varepsilon \cup \{1 \dots s\}$, i.e. they are subsets of $\Pi^* . S$. Here '.' represents the concatenation of words and ε the empty word, as usual. The word operation $w[u/v]$ substitutes v for u in w, but only if u is a prefix of w, otherwise it leaves it unchanged.

1.1 Graphs as paths

Definitions 1 to 4 are as in [1].

Definition 1 (Graph). *A* graph G *is given by*

- *An at most countable subset* $V(G)$ *of* V*, whose elements are called* vertices.
- *A finite set* π*, whose elements are called* ports.
- *A set* $E(G)$ *of non-intersecting two element subsets of* $V(G):\pi$*, whose elements are called edges.*

such that for all u*,* w *in* $V(G)$*, there exists* $n, v_0, a_0, b_0, \dots v_n, a_n, b_n$ *such that for all* $i \in \{0 \dots n-1\}$*,* $(v_i : a_i, v_{i+1} : b_{i+1}) \in E(G)$ *with* $v_0 = u$ *and* $v_n = w$*,* i.e. *the graph is assumed to be connected.*

Definition 2 (Labelled graph). *Let G be a graph. A labelling with states Σ, Δ is given by:*

- *A partial function σ from $V(G)$ to Σ giving the label of the vertices.*
- *A partial function δ from $E(G)$ to Δ giving the label of the edges.*

A labelled graph is a graph together with a labelling. The set of all graphs *with ports π is written $\mathcal{G}_\pi$. The* set of labelled graphs *with states Σ, Δ and ports π is written $\mathcal{G}_{\Sigma,\Delta,\pi}$. To ease notations, we sometimes write $v \in G$ for $v \in V(G)$.*

Definition 3 (Pointed graph). *A* pointed (labelled) graph *is a pair (G, p) with $p \in G$. The* set of pointed graphs *with ports π is written $\mathcal{P}_\pi$. The* set of pointed labelled graphs *with states Σ, Δ and ports π is written $\mathcal{P}_{\Sigma,\Delta,\pi}$.*

Definition 4 (Isomorphism). *An* isomorphism *R is a function from $\mathcal{G}_\pi$ to $\mathcal{G}_\pi$ which is specified by a bijection $R(.)$ from V to V. The image of a graph G under the isomorphism R is a graph RG whose set of vertices is $R(V(G))$, and whose set of edges is $\{\{R(u) : a, R(v) : b\} \mid \{u : a, v : b\} \in E(G)\}$. Similarly, the image of a pointed graphs $P = (G, p)$ is the pointed graph $RP = (RG, R(p))$. When P and Q are isomorphic we write $P \approx Q$, defining an equivalence relation on the set of Graphs. The definition extends to pointed labelled graphs.*

In the particular graphs we are considering, the vertices can be uniquely distinguished by the paths that lead to them starting from the pointer vertex. Hence, we might just as well forget about vertex names.

Definition 5 (Pointed graph modulo). *Let P be a pointed (labelled) graph (G, p). The* pointed (labelled) graph modulo *$\tilde{P}$ is the equivalence class of P with respect to the equivalence relation $\approx$. The* set of pointed graphs modulo *with ports π is written $\tilde{\mathcal{P}}_\pi$. The* set of pointed labelled graphs modulo *with states Σ, Δ and ports π is written $\tilde{\mathcal{P}}_{\Sigma,\Delta,\pi}$.*

Given such a pointed graph modulo, its set of paths forms a language, endowed with a notion of equivalence whenever two paths designate the same vertex. The language, together with its equivalence, is referred to as a the paths structure.

Definition 6 (Language of paths). *Given a pointed graph modulo $\tilde{P}$, we say that u is a path of $\tilde{P}$ if and only if there is a sequence u of ports $a_i b_i$ such that, starting from the pointer, it is possible to travel in the graph according to this sequence. We define the* language of paths *$L(\tilde{P})$ of $\tilde{P}$ as the set of these paths. More formally, $u \in L(\tilde{P})$ if and only if there exists $(G, p) \in \tilde{P}$ and $v_1, \ldots, v_{|u|} \in V(G)$ such that for all $i \in \{0 \ldots |u| - 1\}$, one has $\{v_i : a_i, v_{i+1} : b_i\} \in E(G)$, with $v_0 = p$ and $u_i = a_i b_i$.*

Definition 7 (Equivalence of paths). *Given a pointed graph modulo $\tilde{P}$, we define the* equivalence of paths *relation $\equiv_{\tilde{P}}$ on $L(\tilde{P})$ such that for all paths $u, u' \in L(\tilde{P})$, $u \equiv_{\tilde{P}} u'$ if and only if, starting from the pointer, u and u' lead to the same vertex of $\tilde{P}$. More formally, $u \equiv_{\tilde{P}} u'$ if and only if there exists $(G, p) \in \tilde{P}$ and $v_1, \ldots, v_{|u|}, v'_1, \ldots, v'_{|u'|} \in V(G)$ such that for all $i \in \{0 \ldots |u|-1\}$, $i' \in \{0 \ldots |u'| - 1\}$, one has $\{v_i : a_i, v_{i+1} : b_i\} \in E(G)$, $\{v'_{i'} : a'_{i'}, v'_{i'+1} : b'_{i'}\} \in E(G)$, with $v_0 = p$, $v'_0 = p$, $u_i = a_i b_i$, $u'_{i'} = a'_{i'} b'_{i'}$ and $v_{|u|} = v_{|u'|}$.*

Definition 8 (Paths structure). *Given a pointed graph modulo $\tilde{P}$, we define the* structure of paths $X(\tilde{P})$ *as the structure* $\langle L(\tilde{P}), \equiv_{\tilde{P}} \rangle$. *The* set of all paths structures *is the set* $\{X(\tilde{P}) \mid \tilde{P} \in \mathcal{P}_\pi\}$. *It is written* $X(\tilde{\mathcal{P}}_\pi)$.

Given two pointed graphs modulo, any difference between them shows up in their paths structure.

Proposition 1 (Pointed graphs modulo and paths structures isomorphism). *The function* $\tilde{P} \mapsto X(\tilde{P})$ *is a bijection between* $\tilde{\mathcal{P}}_\pi$ *and* $X(\tilde{\mathcal{P}}_\pi)$.

1.2 Paths as languages

Inversely, we could have started by defining a certain class of languages endowed with an equivalence, namely adjacency structures, and then see whether the paths structures of graph modulo fall into this class. This is the purpose of the following definitions and lemma.

Definition 9 (Completeness). *Let* $L \subseteq \Pi^*.S$ *be a language and* $\equiv_L$ *an equivalence on this language. The tuple* $(L, \equiv_L)$ *is said to be* complete *if and only if*

- $\forall u, u' \in L, v \in \Pi^*.S, u \equiv_L u' \wedge u.v \in L \Rightarrow u'.v \in L \wedge u'.v \equiv_L u.v$
- $\forall u, \in L, a, b \in \pi, u.ab \in L \Rightarrow u.ab.ba \in L \wedge u.ab.ba \equiv_L u$

The completion $(L', \equiv_{L'})$ *of* $(L, \equiv_L)$ *is the smallest complete extension of* $(L, \equiv_L)$.

Definition 10 (Prefix-stable language). $L \subseteq \Pi^*.S$ *is a* prefix-stable *language if and only if*

$$\forall u, v \in \Pi^*.S, u.v \in L \Rightarrow u \in L.$$

Definition 11 (Adjacency structure). *Let* $L \subseteq \Pi^*$ *be a prefix-stable language and* $\equiv_L$ *an equivalence on this language. The tuple* $(L, \equiv_L)$ *defines an* adjacency structure *if and only if it is complete and*

$$\forall u, u' \in L, a, b, c \in \pi, u \equiv_L u' \wedge u.ab \in L \wedge u'.ac \in L \Rightarrow b = c.$$

When this is the case, L *is referred to as an* adjacency language *and* $\equiv_L$ *as an* adjacency equivalence. *The adjacency structure* X *is denoted* $\langle L, \equiv \rangle$. *The* set of all adjacency structures X *is written* $\mathfrak{X}_\pi$.

Definition 12 (Associated pointed graph modulo). *Let* X *be some adjacency structure* $\langle L, \equiv_L \rangle$. *Let* $V(X)$ *be the set of equivalence classes of* X. *Let* $P(X)$ *be the pointed graph* $(G(X), \tilde{\varepsilon})$, *with* $G(X)$ *such that:*

- *The set of vertices* $V(G(X))$ *is* $V(X)$;
- *The edge* $\{\tilde{u} : a, \tilde{v} : b\}$ *is in* $E(G(X))$ *if and only if* $u.ab \equiv_L v$, *for all* $u \in \tilde{u}$ *and* $v \in \tilde{v}$.

We define the associated pointed graph modulo *to be $\tilde{P}(X)$, i.e. the equivalence class of $P(X)$.*

Soundness: Two properties of adjacency structures ensure that the ports of the vertices are not used several times and thus that the graph associated to an adjacency structure is always a well-defined object.

Definition 13 (Labelled adjacency structure). *Let $X = \langle L, \equiv_L \rangle$ be a an adjacency structure. A labelling with states Σ, Δ is given by a labelling for $\tilde{P}(X)$. The* set of labelled adjacency structures *with states Σ, Δ and ports π is written $\mathfrak{X}_{\Sigma,\Delta,\pi}$.*

Not only do we have that paths structures are adjacency structures, but it also turns out that any adjacency structure can be generated this way, i.e. it is the paths structure of some pointed graph modulo.

Proposition 2 (Adjacency structures are paths structures). *Let X be some adjacency structure. The equality $X = X(\tilde{P}(X))$ holds. Hence $\mathfrak{X}_\pi = X(\tilde{\mathcal{P}}_\pi)$.*

1.3 Graphs as languages

Generalized Cayley graphs. The following result comes out as a corollary of Propositions 1 and 2:

Theorem 1 (Pointed graphs modulo and adjacency structures isomorphism). *The function $\tilde{P} \mapsto X(\tilde{P})$ is a bijection between $\tilde{\mathcal{P}}_\pi$ and $\mathfrak{X}_\pi$, whose inverse is the function $X \mapsto \tilde{P}(X)$. It can be extended into a bijection between $\tilde{\mathcal{P}}_{\Sigma,\Delta,\pi}$ and $\mathfrak{X}_{\Sigma,\Delta,\pi}$.*

Conventions. This theorem justifies the fact that

- a (labelled) pointed graph modulo $\tilde{P}(X)$ (resp. $\tilde{P}$),
- and a (labelled) adjacency structure X (resp. $X(\tilde{P})$).

can be viewed as being the same mathematical object. Together with Definitions 8 and 12, it also justifies the fact that the vertices of this mathematical object can be designated by

- $\tilde{u}$ an equivalence class of X (resp. $X(\tilde{P})$), i.e. the set of all paths leading to this vertex starting from $\tilde{\varepsilon}$,
- or more directly by u an element of an equivalence class $\tilde{u}$ of X (resp. $X(\tilde{P})$), i.e. a particular path leading to this vertex starting from ε.

These two remarks lead to the following mathematical conventions, which we adopt for convenience. From now on:

- $\tilde{\mathcal{P}}_{\Sigma,\Delta,\pi}$ and $\mathfrak{X}_{\Sigma,\Delta,\pi}$ will no longer be distinguished. The latter notation will be preferred but most often given meaning of the former, i.e. we shall speak of a "generalized Cayley graph" X in $\mathfrak{X}_{\Sigma,\Delta,\pi}$,

– $\tilde{u}$ and u will no longer be distinguished, and the latter notation will be preferred but most often given the meaning of the former, i.e. we shall speak of a "vertex" u in $V(X)$, or simply $x \in X$.
– it follows that '$\equiv$' and '$=$' will no longer be distinguished, and the latter notation will be preferred but most often given the meaning of the former, i.e. we shall write $u = v$ when strictly speaking we just have $\tilde{u} = \tilde{v}$.

Such conventions may seem shocking at first but are very common in algebraic structures, for instance it is common to write $2 + 2 = 4$ even though they are syntactically distinct. In any case, we will make sure that a rigorous meaning can always be recovered by placing tildes back.
Discussion. Clearly this mathematical object, namely adjacency structures or pointed graphs modulo, extends the notion of Cayley graph: those are recovered for adjacency structures $\langle L, \equiv_L \rangle$ having $L = M^*$, with M the subset $\{aa^{-1}, a^{-1}a \mid a \in \pi\}$ of π^2. The extension is a strict one, because now arbitrary bounded degree graphs are allowed. The extension is an advantageous one, since all of the key features of Cayley graphs remain: we are able to name vertices relative to a point, though the word describing the path from that point, and in fact the topology of the graph describes the equivalence structure upon words. Another important feature of Cayley graphs is that they admit a well-defined notion of translation, or shift. This again is maintained. This will be one of the basic operations upon these generalized Cayley graphs which we present in Section 2. Yet another important feature of Cayley graphs is that they admit a compact, Hausdorff topology. This will be done in in Section 3.

2 Generalized Cayley graphs: basic operations

In terms of pointed graphs, the neighbours of radius r are just those nodes which can be reached in r steps starting from the pointer. For the sake of mathematical rigour we define them for generalized Cayley graphs as follows:

Definition 14 (Neighbours, neighbourhood structure). *Let $X \in \mathcal{X}_\pi$ be a generalized Cayley graph given by $\langle L, \equiv_L \rangle$. Let R be the subset of words of length less than or equal to r in L, and $\equiv_R$ the restriction of $\equiv_R$ to this subset. The* neighbours of radius r in X *is the set of vertices of X than can be designated by an element of R. The* neighbourhood structure of radius r *is the completion of* $(R, \equiv_R)$.

Soundness: The completion of $(R, \equiv_R)$ is an adjacency structure because it is a substructure of $\langle L, \equiv_L \rangle$.

In terms of pointed labelled graphs, the disk of radius r is the subgraph induced by those nodes which can be reached in $r + 1$ steps starting from the pointer, but with labellings restricted to those that can be reached in r steps, see [1]. For the sake of mathematical rigour we define them for generalized Cayley graphs as follows:

Definition 15 (Disk). *Let $X \in \mathcal{X}_{\Sigma,\Delta,\pi}$ be a generalized Cayley graph given by $(\langle L, \equiv_L \rangle, \sigma, \delta)$. The* disk of radius *$r$, namely X^r, is given by the neighbourhood structure of radius $r+1$, together with σ^r and δ^r the restrictions of σ and δ to the neighbours of radius r. The* set of disks of radius *r of the set of labelled adjacency structures $\mathcal{X}_{\Sigma,\Delta,\pi}$ is the set of disks $\{X^r \mid X \in \mathcal{X}_{\Sigma,\Delta,\pi}\}$. It is denoted $\mathcal{D}^r_{\Sigma,\Delta,\pi}$.*

Given a path from ε to a vertex u, it helps to have a notation for the inverse path.

Definition 16 (Inverse). *Given $u \in \Pi^*$ we define $\overline{u}$ so that for all $a, b \in \pi$, $v, w \in \Pi^*$ we have $\overline{ab} = ba$ and $\overline{v.w} = \overline{w}.\overline{v}$. The word $\overline{u}$ is in Π^*, it is referred to as* the inverse of *u.*

In a generalized Cayley graph, vertices are named after those paths that lead to them, starting from the vertex ε. Given such a graph, it helps to have a notation for the graph obtained by naming vertices relative to some other vertex u.

Definition 17 (Shift). *Let $X \in \mathcal{X}_\pi$ be a generalized Cayley graph given by $\langle L, \equiv_L \rangle$. Given $u \in X$ we define X_u as $\langle L', \equiv_{L'} \rangle$, with L' equal to $(L \cap u.\Pi^*)[u/\varepsilon]$ and $\equiv_{L'}$ equal to $(\equiv_L \cap (u.\Pi^* \times u.\Pi^*))[u/\varepsilon]$. The resulting generalized Cayley graph X_u is referred to as X* shifted by *u, and sometimes also denoted $\overline{u}.X$. Indeed, notice that if $u \in X$ and $v \in X^r$ for some r, then $v_u = \overline{u}.v$ is in X_u. The vertex v_u is referred to as v* shifted by *u. The definition extends to labelled generalized Cayley graphs.*

A shift is just a graph isomorphism aiming at changing the position of the pointer. Thus it makes sense to define a similar notion over graphs non-modulo. Because words are usually endowed with an equality in our setting, which graphs non-modulo do not provide, the definition is given relative to an adjacency structure X.

Definition 18 (Shift isomorphism). *Let X be an adjacency structure and $V(X)$ its equivalence classes. Let $G \in \mathcal{G}_\pi$ be a graph taken to have vertices that are disjoint subsets of $V(X^r).S$. Given $u \in V(X)$, consider R the isomorphism obtained as the pointwise extension of a bijection mapping $v.z \mapsto v_u.z$, for any $v \in V(X^r), z \in S$. We define G_u to be RG; it has vertices that are disjoint subsets of $V(X_u).S$. The resulting graph G_u is referred to as G* shifted by *u, and sometimes also denoted $\overline{u}.G$. Indeed, notice that if $v.z \in G^r$ for some r, then $v_u.z = \overline{u}.z.b$ is in G_u. The definition extends to labelled graphs.*

These two definitions are consistent in the following sense: if $X = X(P) \in \mathcal{X}_\pi$ and $u \in X$, then $X_u = X(P_u)$. As the functions P and X are inverses of each other, we have the symmetric impliation: if $P = P(X)$ and $u \in P$, then $P_u = P(X_u)$.
The next two definitions are standard, as they give a well-defined meaning to the notion of union of graphs. See for instance [1], although here again the definition is given relative to an adjacency structure X.

Definition 19 (Consistency). *Let X be an adjacency structure and $V(X)$ its equivalence classes. Let G be a labelled graph (G, σ, δ), and G' be a labelled graph (G', σ', δ'), both taken to have vertices that are disjoint subsets of $V(X).S$. The graphs are said to be* consistent *if and only if:*

- $\forall u \in G, u' \in G',\ u \cap u' \neq \emptyset \Rightarrow u = u'$,
- $\forall u, v \in G, u', v' \in G', a, a', b, b' \in \pi,\ \{u : a, v : b\} \in E(G) \wedge \{u' : a', v' : b'\} \in E(G') \wedge u = u' \wedge a = a' \Rightarrow b = b' \wedge v = v'$,
- $\forall u, v \in G, u', v' \in G', a, b \in \pi,\ u = u' \Rightarrow \delta(\{u : a, v : b\}) = \delta'(\{u' : a, v' : b\})$ *when both are defined,*
- $\forall u \in G, u' \in G',\ u = u' \Rightarrow \sigma(u) = \sigma'(u')$ *when both are defined.*

They are said to be trivially consistent *if and only if there is no $u \in G$, $u' \in G'$ such that $u = u'$.*

Definition 20 (Union). *Let X be an adjacency structure and $V(X)$ its equivalence classes. Let G be a labelled graph (G, σ, δ), and G' be a labelled graph (G', σ', δ'), both taken to have vertices that are disjoint subsets of $V(X).S$. Whenever they are consistent, their* union *is defined. The resulting graph $G \cup G'$ is the labelled graph with vertices $V(G) \cup V(G')$, edges $E(G) \cup E(G')$, labels that are the union of the labels of G and G'.*

3 Generalized Cayley graphs: topological properties

Having a well-defined notion of disks allows us to define a topology upon $\mathcal{X}_{\Sigma,\Delta,\pi}$, which is the natural generalization of the well-studied Kantor metric upon CA configurations [9]. It is interesting to compare its properties with the analogous metric upon $\mathcal{P}_{\Sigma,\Delta,\pi}$.

Definition 21 (Gromov-Hausdorff-Kantor metrics). *Consider the function*

$$\begin{aligned} d_{\mathcal{X}} : \mathcal{X}_{\Sigma,\Delta,\pi} \times \mathcal{X}_{\Sigma,\Delta,\pi} &\longrightarrow \mathbb{R}^+ \\ (X, X') &\mapsto d_{\mathcal{X}}(X, X') = 0 \quad \textit{if } X = X' \\ (X, X') &\mapsto d_{\mathcal{X}}(X, X') = 1/2^r \quad \textit{otherwise} \end{aligned}$$

where r is the minimal radius such that $X^r \neq X'^r$.
Also consider the function:

$$\begin{aligned} d_{\mathcal{P}} : \mathcal{P}_{\Sigma,\Delta,\pi} \times \mathcal{P}_{\Sigma,\Delta,\pi} &\longrightarrow \mathbb{R}^+ \\ (P, P') &\mapsto d_{\mathcal{P}}(P, P') = 0 \quad \textit{if } P = P' \\ (P, P') &\mapsto d_{\mathcal{P}}(P, P') = 1/2^r \quad \textit{otherwise} \end{aligned}$$

where r is the minimal radius such that P and P' differ, starting from their pointers p and p'.
Both functions are such that for $\epsilon > 0$ we have (with $r = \lfloor \log_2(\epsilon) \rfloor$):

$$d(A, A') < \epsilon \Leftrightarrow A^r = A'^r.$$

Both are metric, actually they are Hausdorff, ultrametric, i.e. for all A, B, C, $d(A, C) = \max\{d(A, B), d(B, C)\}$.

The fact that generalized Cayley graphs are pointed graphs modulo, i.e. the fact that they have no "vertex name degree of freedom" is key to proving the following property. Indeed, compactness crucially relies on the set being "finite-branching", meaning that the set of possible graphs, as one progressively enlarges the radius of a disk, remains finite. This does not hold for usual graphs.

Lemma 1 (Compactness). *The metric* $d_{\mathcal{X}}$ *makes* $\mathcal{X}_{\Sigma,\Delta,\pi}$ *into a compact space.*

Continuity is a crucial notion in mathematics, which was used to characterize CA [9].

Definition 22 (Continuous function). *A function* F *from* $\mathcal{X}_{\Sigma,\Delta,\pi}$ *to* $\mathcal{Y}_{\Sigma,\Delta,\pi}$ *is said to be continuous if and only if for all* X*, for all* η*, there exists* ϵ *such that for all* X'*,* $d_{\mathcal{X}}(X, X') < \epsilon$ *implies* $d_{\mathcal{Y}}(F(X), F(X')) < \eta$.

Uniform continuity, on the other hand, is crucial notion in physics, as it captures the fact that information does not propagate too fast.

Definition 23 (Uniformly continuous function). *A function* F *from* $\mathcal{X}_{\Sigma,\Delta,\pi}$ *to* $\mathcal{Y}_{\Sigma,\Delta,\pi}$ *is said to be uniformly continuous if and only if for all* η*, there exists* ϵ *such that for all* X, X'*,* $d_{\mathcal{X}}(X, X') < \epsilon$ *implies* $d_{\mathcal{Y}}(F(X), F(X')) < \eta$.

Both are known to be equivalent, provided we have compactness.

Theorem 2 (Topology recap.).
Consider continuous functions $F : \mathcal{X} \longrightarrow \mathcal{Y}$*, with* $\mathcal{X}$ *a Hausdorff compact space and* $\mathcal{Y}$ *a Hausdorff space. Then*

- F *is uniformly continuous.*
- *If* F *has an inverse* F^{-1}*, then this inverse is also continuous.*

Theorem 2 summarizes well-known facts of general topology [6]. Their implications for Cellular Automata were first studied in [9], with self-contained elementary proofs available in [10]. For Cellular Automata over Cayley graphs a complete reference is [3]. For Causal Graph Dynamics [1], these had to be re-proven by hand, due to the lack of a clear topology in the set of graphs that was considered. Here we are able rely on the topology of generalized Cayley graphs and reuse Theorem 2 out-of-the-box, which makes generalized Cayley graph a very attractive setting in order to generalize CA.

4 Causality and Localizability

Causality. The notion of causality extends the mathematical definition of Cellular Automata over Cayley graphs. The extension is a strict one: not only the graphs have become arbitrary, but their topology can also vary in time.

Definition 24 (Dynamics). *A function F from $\mathcal{X}_{\Sigma,\Delta,\pi}$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$ is said to be a* dynamics *if and only if there exists suffixes S such that for all X, the vertices of $F(X)$ are disjoint subsets of $V(X).S$.*

Definition 25 (Shift-invariant dynamics). *A dynamics $F : \mathcal{X}_{\Sigma,\Delta,\pi} \to \mathcal{G}_{\Sigma,\Delta,\pi}$ with suffixes $S = \varepsilon \cup \{1 \dots s\}$ is said to be* shift-invariant *if and only if for all X, $u \in X$, $F(X)_u = F(X_u)$.*

Definition 26 (Causal dynamics). *A function $F : \mathcal{X}_{\Sigma,\Delta,\pi} \to \mathcal{G}_{\Sigma,\Delta,\pi}$ is said to be a* causal dynamics *if and only if it is a shift-invariant dynamics suffixes S and*

- *$X \mapsto (F(X), \varepsilon)$ is continuous.*
- *For all X, $V(F(X))$ is a partition of $V(X).S$.*

Lemma 2 (Bounded inflation). *Consider a causal dynamics F from $\mathcal{X}_{\Sigma,\Delta,\pi}$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$. There exists a bound b such that for all X, $v \in F(X)$, we have that $v \in F(X)^{|v|.b}$.*

Localizability. The notion of localizability captures the exact same idea of the constructive definition of Cellular Automata, namely a single local rule f applied in a translation-invariant manner on the input graph.

Definition 27 (Local rule). *A function f from $\mathcal{D}^r_{\Sigma,\Delta,\pi}$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$ is a* local rule *if and only if it is a dynamics with suffixes S and*

- *For any disk X^r, we have that $z \in S$ implies $z \in f(X^r)$.*
- *For any disk X^{r+1} and any $u \in X^0$ we have that $f(X^r)$ and $u.f(X^r_u)$ are non-trivially consistent.*
- *For any disk X^{3r+2} and any $u \in X^{2r+1}$ we have that $f(X^r)$ and $u.f(X^r_u)$ are consistent.*

The conventions taken for the local rules are so that integer z stands for the 'successor of rank z'. Hence the words in $\{1 \dots s\}$ designate the successors of the vertex ε, whereas those in $\Pi^{\leq r}.\{1 \dots s\}$ are the successors of its neighbours of radius r. For instance a vertex named $\{1, ab.2\}$ is understood to be both the first successor of vertex ε, and the second successor of vertex ab.

Definition 28 (Localizable function). *A function F from $\mathcal{X}_{\Sigma,\Delta,\pi}$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$ is said to be* localizable *if and only if there exists a radius r and a local rule f from $\mathcal{D}^r$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$ such that for all X, $F(X)$ is given by:*

$$\bigcup_{u \in X} u.f(X^r_u).$$

Equivalence. The following theorem shows that this constructive definition is in fact equivalent to the topological definition of causal functions.

Theorem 3 (Causal is localizable). *A function F from $\mathcal{X}_{\Sigma,\Delta,\pi}$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$ is causal if and only if it is localizable.*

5 Example

The following is an example of causal, hence localizable dynamics. It is adapted from [1], for comparison.

Inflating grid

In this example, each vertex gives birth to four distinct vertices such that the structure of the initial graph is preserved. The degree of the graph is bounded by $|\pi| = 4$, as the ports take their names in $\pi = \{n, s, e, w\}$ (for north, south, east and west directions). Here the labels of the vertices and edges are ignored. The general case of the local rule, defined on disks of radius one, is described in Figure 1. This local rule generates a subgraph of 12 vertices (named 0 through

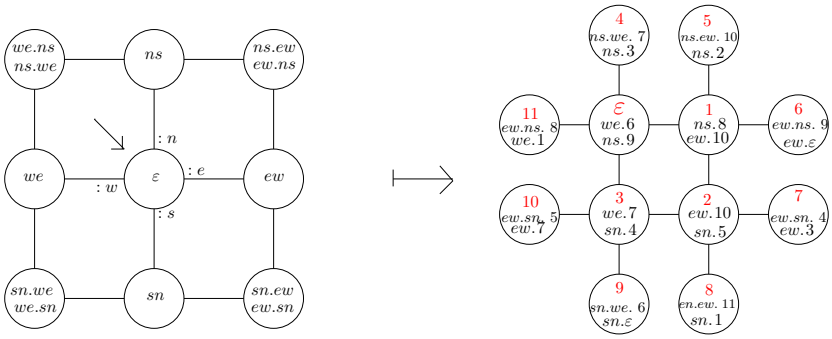


Fig. 1. General local rule for the inflating grid (8 neighbours)

11), some with further names serving as identification information, so that the generated graphs glue back together. For a complete definition we would also have to include the border cases, where the pointed vertex is surrounded by less than 8 neighbours (see Figure 2, for instance). The gluing rule of the diffent

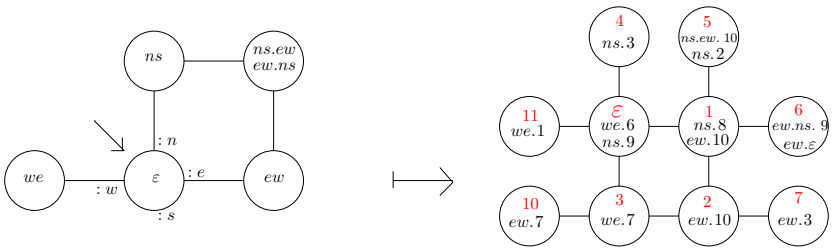


Fig. 2. General local rule for the inflating grid (4 neighbours)

$u.f(X_u^r)$ is done by identifying the vertices having the same set of names, as illustrated in Figure 3.

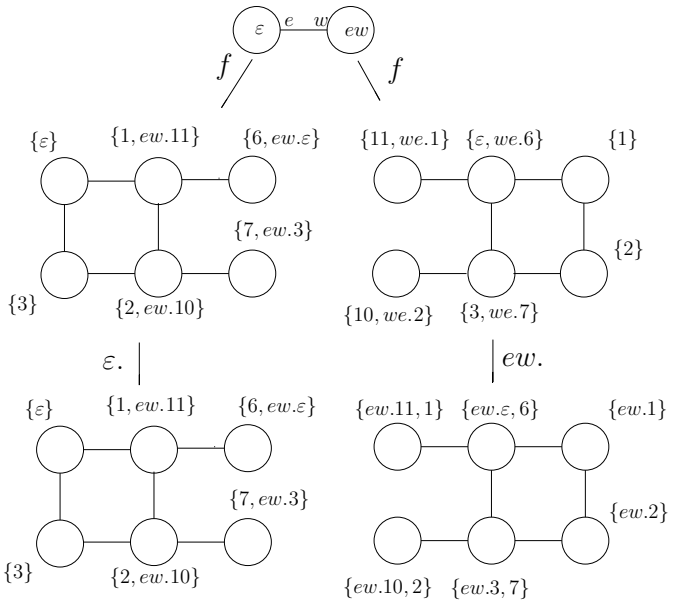


Fig. 3. Inflation of a pair of vertices.

6 Computability

Our causal dynamics over generalized Cayley graphs is a candidate model of computation accounting for space, but without this space being fixed. As a candidate model of computation, we must check that it is computable. The following shows that we can decide whether a syntactic object is a valid instance of the model.

Proposition 3 (Decidability of consistency). *Given a dynamics f from $\mathcal{D}^r_{\Sigma,\Delta,\pi}$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$ with suffixes S, it is decidable whether f is a local rule.*

Sketch of the proof:
We can enumerate the possible X^r and check that $\varepsilon, 1, \ldots, s \in f(X^r)$. We can enumerate the possible X^{r+1} and check that for all $u \in X^0$, $f(X^r)$ and $u.f(X_u^r)$ are non-trivially consistent. We can enumerate the possible X^{3r+2} and check that for all $u \in X^{2r+1}$, $f(X^r)$ and $u.f(X_u^r)$ are consistent.
Next, we prove that we can enumerate all instances of the model.

Proposition 4 (Recursive enumeration of local rules). *The set of all local rules is recursively enumerable.*

Sketch of the proof:
If we fix a radius r and suffixes $S = \varepsilon \cup \{1, \ldots, s\}$, there is a finite number of dynamics f from $\mathcal{D}^r$ to $\mathcal{G}_{\Sigma,\Delta,\pi}$ with bound s. For each of these, from the above lemma, it can be decided whether it is a local rule.
Finally, we prove that if the initial state is finite, its evolution can be computed.

Proposition 5 (Computability of causal functions). *Given a local rule f and a finite generalized Cayley graph X, then $F(X)$ is computable, with F the causal dynamics induced by f.*

Sketch of the proof:
Since f is a local rule, the images of disks of radius r included in X are all finite, and consistent with one another. The finite union of finite, consistent graphs, is computable.

7 Further works

The mathematical relation between the causal dynamics of [1] and ours remains to be clarified. The propositions of Section 6 remain to be proven for the causal graph dynamics of [1]. Still, they are important features of models of computation. The fact that they are relatively straightforward to prove in this paper is a good indicator that the formalism presented is appropriate. Another important issue is to characterize the localizable dynamics from $\mathcal{X}$ to $\mathcal{X}$, just like we did for those from $\mathcal{X}$ to $\mathcal{G}$.

Acknowledgements

This work has been funded by the ANR-10-JCJC-0208 CausaQ grant. Christophe Crespelle, Gilles Dowek, Viv Kendon, Jean Mairesse, Vincent Nesme, Eric Thierry.

References

1. P. Arrighi and G. Dowek. Causal graph dynamics. In *Proceedings of ICALP 2012, Warwick, July 2012, to appear in LNCS. Pre-print arXiv:1202.1098*, 2012.
2. P. Boehm, H.R. Fonio, and A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *Journal of Computer and System Sciences*, 34(2-3):377–408, 1987.
3. T. Ceccherini-Silberstein and M. Coornaert. *Cellular automata and groups*. Springer Verlag, 2010.
4. B. Derbel, M. Mosbah, and S. Gruner. Mobile agents implementing local computations in graphs. *Graph Transformations*, pages 99–114, 2008.
5. H. Ehrig and M. Lowe. Parallel and distributed derivations in the single-pushout approach. *Theoretical Computer Science*, 109(1-2):123–143, 1993.
6. V.V. Fedorchuk, A.V. Arkhangelskiui, and L.S. Pontriagin. *General topology I*, volume 1. Springer, 1990.

7. J.L. Giavitto and A. Spicher. Topological rewriting and the geometrization of programming. *Physica D: Nonlinear Phenomena*, 237(9):1302–1314, 2008.
8. Stefan Gruner. Mobile agent systems and cellular automata. *Autonomous Agents and Multi-Agent Systems*, 20:198–233, 2010. 10.1007/s10458-009-9090-0.
9. G. A. Hedlund. Endomorphisms and automorphisms of the shift dynamical system. *Math. Systems Theory*, 3:320–375, 1969.
10. K. Kari. Cellular Automata, Lecture notes. http:users.utu.fijkarica, 2011.
11. A. Klales, D. Cianci, Z. Needell, D. A. Meyer, and P. J. Love. Lattice gas simulations of dynamical geometry in two dimensions. *Phys. Rev. E.*, 82(4):046705, Oct 2010.
12. H.J. Kreowski and S. Kuske. Autonomous units and their semantics-the parallel case. *Recent Trends in Algebraic Development Techniques*, pages 56–73, 2007.
13. W. Kurth, O. Kniemeyer, and G. Buck-Sorlin. Relational growth grammars–a graph rewriting approach to dynamical systems with a dynamical structure. *Unconventional Programming Paradigms*, pages 97–97, 2005.
14. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1-2):181–224, 1993.
15. C. Papazian and E. Remila. Hyperbolic recognition by graph automata. In *Automata, languages and programming: 29th international colloquium, ICALP 2002, Málaga, Spain, July 8-13, 2002: proceedings*, volume 2380, page 330. Springer Verlag, 2002.
16. Z. Róka. Simulations between cellular automata on Cayley graphs. *Theoretical Computer Science*, 225(1-2):81–111, 1999.
17. G. Taentzer. *Parallel and distributed graph transformation: Formal description and application to communication-based systems.* PhD thesis, Technische Universitat Berlin, 1996.
18. G. Taentzer. Parallel high-level replacement systems. *Theoretical computer science*, 186(1-2):43–81, 1997.
19. K. Tomita, S. Murata, A. Kamimura, and H. Kurokawa. Self-description for construction and execution in graph rewriting automata. *Advances in Artificial Life*, pages 705–715, 2005.
20. Kohji Tomita, Haruhisa Kurokawa, and Satoshi Murata. Graph automata: natural expression of self-reproduction. *Physica D: Nonlinear Phenomena*, 171(4):197 – 210, 2002.
21. Kohji Tomita, Haruhisa Kurokawa, and Satoshi Murata. Graph-rewriting automata as a natural extension of cellular automata. In Thilo Gross and Hiroki Sayama, editors, *Adaptive Networks*, volume 51 of *Understanding Complex Systems*, pages 291–309. Springer Berlin / Heidelberg, 2009.
22. S. Von Mammen, D. Phillips, T. Davison, and C. Jacob. A graph-based developmental swarm representation and algorithm. *Swarm Intelligence*, pages 1–12, 2011.