

GCM 2019

THE TENTH INTERNATIONAL WORKSHOP ON GRAPH
COMPUTATION MODELS

Proceedings

Eindhoven, The Netherlands, July 2019

Editors:
Rachid Echahed and Detlef Plump

Contents

Preface	iv
AARON LYE	
Transformation of Turing Machines into Context-Dependent Fusion Grammars	1
MARTIN BERGLUND	
Analyzing and Pumping Hyperedge Replacement Formalisms in a Common Framework	17
MARK MINAS	
Speeding up Generalized PSR Parsers by Memoization Techniques	33
CHRISTIAN SANDMANN AND ANNEGRET HABEL	
Rule-Based Graph Repair	49
NICOLAS BEHR	
Sesqui-Pushout Rewriting: Concurrency, Associativity and Rule Algebra Framework	65
HOLGER GIESE, MARIA MAXIMOVA, LUCAS SAKIZLOGLOU AND SVEN SCHNEIDER	
Metric Temporal Graph Logic over Typed Attributed Graphs (Short Version)	101
SVEN SCHNEIDER, LEEN LAMBERS AND FERNANDO OREJAS	
Automated Reasoning for Attributed Graph Properties (Short Version)	103
REIKO HECKEL, LEEN LAMBERS AND MARYAM GHAFARI SAADAT	
Analysis of Graph Transformation Systems : Native vs Translation-based Approaches	105

Preface

This volume contains the proceedings of the Tenth International Workshop on *Graph Computation Models (GCM 2019¹)*. The workshop was held in Eindhoven, The Netherlands, on July 17th, 2019, as part of STAF 2019 (Software Technologies: Applications and Foundations).

Graphs are common mathematical structures that are visual and intuitive. They constitute a natural and seamless way for system modelling in science, engineering and beyond, including computer science, biology, business process modelling, etc. Graph computation models constitute a class of very high-level models where graphs are first-class citizens. The aim of the International GCM Workshop series is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation. It promotes the cross-fertilizing exchange of ideas and experiences among senior and young researchers from the different communities interested in the foundations, applications, and implementations of graph computation models and related areas.

Previous editions of GCM series were held in Natal, Brazil (GCM 2006), in Leicester, UK (GCM 2008), in Enschede, The Netherlands (GCM 2010), in Bremen, Germany (GCM 2012), in York, UK (GCM 2014), in L'Aquila, Italy (GCM 2015), in Wien, Austria (GCM 2016), in Marburg, Germany (GCM 2017) and in Toulouse, France (GCM 2018).

These proceedings contain seven accepted papers and an introduction to the panel discussion dedicated to the *Analysis of Graph Transformation Systems*. All submissions were subject to careful refereeing. The topics of accepted papers range over a wide spectrum, including theoretical aspects of graph transformation, verification and parsing techniques as well as application issues of graph computation models. Selected papers from these proceedings will be published online by Electronic Proceedings in Theoretical Computer Science (EPTCS, <http://www.eptcs.org/>).

We would like to thank all the people who contributed to the success of GCM 2019, especially the Program Committee and the additional reviewers for their valuable contributions to the selection process as well as the contributing authors without whom this volume would not exist. We would like also to express our gratitude to all members of the STAF 2019 Organizing Committee.

July, 2019

Rachid Echahed and Detlef Plump

Program chairs of GCM 2019

¹GCM2019 web site: <http://gcm2019.imag.fr>

Program committee of GCM 2019

Anthony Anjorin	University of Paderborn, Germany
Andrea Corradini	University of Pisa, Italy
Juan de Lara	Universidad Autónoma de Madrid, Spain
Rachid Echahed	CNRS and Univ. Grenoble Alpes, France (co-chair)
Maribel Fernández	King's College London, UK
Annegret Habel	University of Oldenburg, Germany
Hans-Joerg Kreowski	University of Bremen, Germany
Leen Lambers	University of Potsdam, Germany
Detlef Plump	University of York, UK (co-chair)
Christopher M. Poskitt	Singapore University of Technology and Design
Leila Ribeiro	Universidade Federal do Rio Grande do Sul, Brazil

Additional Reviewers

Okan Özkan
Christian Sandmann
Nils Weidmann

Transformation of Turing Machines into Context-Dependent Fusion Grammars

Aaron Lye

University of Bremen, Department of Computer Science and Mathematics
P.O.Box 33 04 40, 28334 Bremen, Germany

`lye@math.uni-bremen.de`

Abstract. Context-dependent fusion grammars were recently introduced as devices for the generation of hypergraph languages. In this paper, we show that this new type of hypergraph grammars, where the application of fusion rules is restricted by positive and negative context conditions, is a universal computation model. Our main result is that Turing machines can be transformed into these grammars such that the recognized language of the Turing machine and the generated language of the corresponding context-dependent fusion grammar coincide up to representation of strings as graphs. As a corollary we get that context-dependent fusion grammars can generate all recursive enumerable languages.

1 Introduction

In 2017 we introduced fusion grammars as generative devices on hypergraphs [1]. They are motivated by the observation, that one encounters various fusion processes in various scientific fields like DNA computing, chemistry, tiling, fractal geometry, visual modeling and others. The common principle is, that a few small entities may be copied and fused to produce more complicated entities. For example, the fusion of DNA double strands according to the Watson-Crick complementarity is a key operation of DNA computing (see, e.g. [2,3]). Similar effects can be seen in the iteration of some fractals (see, e.g., [4]), can be found in mosaics and tilings (see, e.g., [5]), or in the area of visual modeling where a spectrum of diagrams is composed of some basic forms. However, it seems that the generative power of fusion grammars is limited (cf. [1,6]) and that they are insufficient whenever context, regulation or regularity is required. There are numerous examples of fusion processes restricted to certain conditions, e.g. the presence enzymes accelerating chemical reactions. In [7] we introduced context-dependent fusion grammars as a generalization of fusion grammars. The basic entities are provided by the start hypergraph. The fusion is done by means of context-dependent fusion rules. A context-dependent fusion rule consumes two complementary hyperedges and identifies the attachment vertices provided that certain positive and negative context conditions are satisfied. It turns out, that context-dependent fusion grammars are powerful enough to simulate Turing machines. We construct a transformation of Turing machines into context-dependent fusion grammars in such a way that the recognized language of the

Turing machine and the language generated by the corresponding grammar coincide up to representation of strings as graphs.¹ The key of the transformation is to simulate a step of the Turing machine by a sequence of applications of fusion rules. As a corollary we get that context-dependent fusion grammars can generate all recursive enumerable languages.

Relating computational models to Turing machines is an old and established approach which can be found in most foundations textbooks in theoretical computer science. Moreover, it is well known that graph transformation in general is Turing-complete. In 1978 Uesu presented a system of graph grammars that generates all recursively enumerable sets of labeled graphs (cf. [8]). Furthermore, asking “what programming constructs are needed on top of graph transformation rules to obtain a computationally complete language” [9] is not a new question. Habel and Plump also presented a graph program that simulates a Turing machine (cf. [9]). With respect to variants of fusion grammars many questions are open due to the novelty of the approach. However, in [10] it is shown that fusion grammars with additional rules for the inverse operation to fusion can simulate Chomsky grammars and connective hypergraph grammars.

The paper is organized as follows. In Section 2, basic notions and notations of hypergraphs are recalled. Section 3 and 4 recall the notions of Turing machines and context-dependent fusion grammars, respectively. Section 5 presents the reduction of Turing machines to context-dependent fusion grammars. Section 6 concludes the paper pointing out some open problems. All the proofs are omitted due to the lack of space.

2 Preliminaries

We consider hypergraphs the hyperedges of which have multiple sources and multiple targets. A *hypergraph* over a given label alphabet Σ is a system $H = (V, E, s, t, lab)$ where V is a finite set of *vertices*, E is a finite set of *hyperedges*, $s, t: E \rightarrow V^*$ are two functions assigning to each hyperedge a sequence of *sources* and *targets*, respectively, and $lab: E \rightarrow \Sigma$ is a function, called *labeling*. The components of $H = (V, E, s, t, lab)$ may also be denoted by V_H , E_H , s_H , t_H , and lab_H respectively. The class of all hypergraphs over Σ is denoted by \mathcal{H}_Σ .

Let $pr: V^* \times \mathbb{N} \rightarrow V$ be defined as $pr(v_1 v_2 \dots v_n, i) = v_i$ if $1 \leq i \leq n$, where n is the length of the sequence. It is undefined otherwise.

Let $H \in \mathcal{H}_\Sigma$, and let \equiv be an equivalence relation on V_H . Then the *fusion of the vertices in H with respect to \equiv* yields the hypergraph $H/\equiv = (V_H/\equiv, E_H, s_{H/\equiv}, t_{H/\equiv}, lab_H)$ with the set of equivalence classes $V_H/\equiv = \{[v] \mid v \in V_H\}$ and $s_{H/\equiv}(e) = [v_1] \dots [v_{k_1}]$, $t_{H/\equiv}(e) = [w_1] \dots [w_{k_2}]$ for each $e \in E_H$ with $s_H(e) = v_1 \dots v_{k_1}$, $t_H(e) = w_1 \dots w_{k_2}$.

¹ Instead of Turing machines some other equivalent computational formalism could be chosen, e.g. Petri nets with inhibitor arcs which would be an extension of the transformation presented in [7]. In the considered approaches the transformations were technical and of similar complexity. Turing machines have the advantage of being an established and well known computational model.

Given $H, H' \in \mathcal{H}_\Sigma$, a *hypergraph morphism* $g: H \rightarrow H'$ consists of two mappings $g_V: V_H \rightarrow V_{H'}$ and $g_E: E_H \rightarrow E_{H'}$ such that $s_{H'}(g_E(e)) = g_V^*(s_H(e))$, $t_{H'}(g_E(e)) = g_V^*(t_H(e))$ and $lab_{H'}(g_E(e)) = lab_H(e)$ for all $e \in E_H$, where $g_V^*: V_H^* \rightarrow V_{H'}^*$ is the canonical extension of g_V , given by $g_V^*(v_1 \cdots v_n) = g_V(v_1) \cdots g_V(v_n)$ for all $v_1 \cdots v_n \in V_H^*$.

Given $H, H' \in \mathcal{H}_\Sigma$, H is a *subhypergraph* of H' , denoted by $H \subseteq H'$, if $V_H \subseteq V_{H'}$, $E_H \subseteq E_{H'}$, $s_H(e) = s_{H'}(e)$, $t_H(e) = t_{H'}(e)$, and $lab_H(e) = lab_{H'}(e)$ for all $e \in E_H$. $H \subseteq H'$ implies that the two inclusions $V_H \subseteq V_{H'}$ and $E_H \subseteq E_{H'}$ form a hypergraph morphism from $H \rightarrow H'$.

Let $H' \in \mathcal{H}_\Sigma$ as well as $V \subseteq V_{H'}$ and $E \subseteq E_{H'}$. Then the *removal* of (V, E) from H' given by $H = H' - (V, E) = (V_{H'} - V, E_{H'} - E, s_H, t_H, lab_H)$ with $s_H(e) = s_{H'}(e)$, $t_H(e) = t_{H'}(e)$ and $lab_H(e) = lab_{H'}(e)$ for all $e \in E_{H'} - E$ defines a subgraph $H \subseteq H'$ if $s_{H'}(e), t_{H'}(e) \in (V_{H'} - V)^*$ for all $e \in E_{H'} - E$. Let $H \in \mathcal{H}_\Sigma$, $H' \subseteq H$. Then $H - H' = H - (V_{H'}, E_{H'})$.

Let $H \in \mathcal{H}_\Sigma$ and $H' = (V', E', s', t', lab': E' \rightarrow \Sigma)$ be some quintuple with two sets V' , E' and three mappings s' , t' and lab' where $+$ denotes the disjoint union of sets. Then the *extension* of H by H' given by $H'' = (V_H + V', E_H + E', s, t, lab)$ with $s(e) = s_H(e)$, $t(e) = t_H(e)$ and $lab(e) = lab_H(e)$ for all $e \in E_H$ as well as $s(e) = s'(e)$, $t(e) = t'(e)$ and $lab(e) = lab'(e)$ for all $e \in E'$ is a hypergraph with $H \subseteq H''$.

Let $H \in \mathcal{H}_\Sigma$ and let $att(e)$ be the set of source and target vertices for $e \in E_H$. H is *connected* if for each $v, v' \in V_H$, there exists a sequence of triples $(v_1, e_1, w_1) \cdots (v_n, e_n, w_n) \in (V_H \times E_H \times V_H)^*$ such that $v = v_1, v' = w_n$ and $v_i, w_i \in att(e_i)$ for $i = 1, \dots, n$ and $w_i = v_{i+1}$ for $i = 1, \dots, n-1$. A subgraph C of H , denoted by $C \subseteq H$, is a *connected component* of H if it is connected and there is no larger connected subgraph, i.e., $C \subseteq C' \subseteq H$ and C' connected implies $C = C'$. The set of connected components of H is denoted by $\mathcal{C}(H)$.

Given $H, H' \in \mathcal{H}_\Sigma$, the *disjoint union* of H and H' is denoted by $H + H'$. Further, $k \cdot H$ denotes the disjoint union of H with itself k times. We use the *multiplication* of H defined by means of $\mathcal{C}(H)$ as follows. Let $m: \mathcal{C}(H) \rightarrow \mathbb{N}$ be a mapping, called *multiplicity*, then $m \cdot H = \sum_{C \in \mathcal{C}(H)} m(C) \cdot C$.

A string can be represented by a simple path where the sequence of labels along the path equals the given string. Let $w = x_1 \dots x_n \in \Sigma^*$ for $n \geq 1$ and $x_i \in \Sigma$ for $i = 1, \dots, n$. Let $[n] = \{1, \dots, n\}$. Then the *string graph* of w is defined by $sg(w) = (\{0\} \cup [n], [n], s_w, t_w, lab_w)$ with $s_w(i) = (i-1)$, $t_w(i) = i$ and $lab(i) = x_i$ for $i = 1, \dots, n$. The string graph of the empty string ε , denoted by $sg(\varepsilon)$, is the discrete graph with a single node 0. Obviously, there is a one-to-one correspondence between Σ^* and $sg(\Sigma^*) = \{sg(w) \mid w \in \Sigma^*\}$.

3 Turing Machines

In this section, we shortly recall the notion of Turing machines (see, e.g., [11, 12, 13]) and their recognized languages. We consider Turing machines with a designated start and accept state and one two-sided infinite tape.

Definition 1. 1. A *Turing machine* is a system $TM = (Q, \Omega, \Gamma, \Delta)$, where Q is a finite set of states with two designated different states q_{start} and q_{accept} , Ω is the input alphabet, Γ is the tape alphabet with $\Omega \subseteq \Gamma$ and $\square \in \Gamma \setminus \Omega$, where \square is the blank symbol, and $\Delta \subseteq (Q \setminus \{q_{accept}\}) \times \Gamma \times \Gamma \times \{l, n, r\} \times Q$ is the transition relation.

2. $conf(TM) = Q \times (\square^\infty \cdot \Gamma^*) \times \Gamma \times (\Gamma^* \cdot \square^\infty)$ is the set of *configurations*, where \square^∞ denotes an infinite sequence of \square -symbols.

3. A *step* of TM is defined by the relation $\vdash_{TM} \subseteq conf(TM) \times conf(TM)$:

$$\begin{aligned} (p, \alpha u, x, \beta) \vdash_{TM} (q, \alpha, u, y\beta) & \quad \text{if } (p, x, y, l, q) \in \Delta \\ (p, \alpha, x, \beta) \vdash_{TM} (q, \alpha, y, \beta) & \quad \text{if } (p, x, y, n, q) \in \Delta \\ (p, \alpha, x, u\beta) \vdash_{TM} (q, \alpha y, u, \beta) & \quad \text{if } (p, x, y, r, q) \in \Delta \end{aligned}$$

4. A *computation* of TM is a potentially infinite sequence of configurations c_0, c_1, \dots where $c_0 = (q_{start} \times \square^\infty \times x \times \beta \square^\infty)$ for some $x \in \Gamma, \beta \in \Gamma^*$ is the start configuration, and $c_i \vdash_{TM} c_{i+1}$ for all $i \in \mathbb{N}$.

5. The recognized language of TM is defined as

$$L(TM) = \{w \in \Omega^* \mid (q_{start}, \square^\infty, x, w') \vdash_{TM}^* (q_{accept}, \square^\infty \alpha, y, \beta \square^\infty)\},$$

where either $x = \square$ and $w' = \square^\infty$ if $w = \varepsilon$, i.e., the empty string, or $x = w_1$ and $w' = w_2 \dots w_n \square^\infty$ if $w = w_1 w_2 \dots w_n \neq \varepsilon$; $\alpha, \beta \in \Gamma^*, y \in \Gamma$ are arbitrary.

Remark 1. 1. $(p, x, y, dir, q) \in \Delta$ means if the Turing machine is in state p and reads the symbol x than it can replace x by y and move the (read/write) head to the left if $dir = l$, to the right if $dir = r$ or leave the head stationary if $dir = n$. Afterwards the machine is in state q .

2. A configuration is of the form (q, α, x, β) which means the machine is in state q , reads currently the symbol x and the contents of the tape to the left and right of the head are α and β , respectively.

3. A computation is finite if a halting configuration is reached, i.e., if there is no possibility of continuing the computation. If the machine enters the state q_{accept} , then it accepts the input.

4. The recognized language consists of all strings for which a computation exists such that the machine enters the accepting state q_{accept} .

4 Context-Dependent Fusion Grammars

In this section, we recall context-dependent fusion grammars introduced in [7]. Context-dependent fusion grammars generate hypergraph languages from start hypergraphs via successive applications of context-dependent fusion rules, multiplications of connected components, and a filtering mechanism. A fusion rule is defined by two complementary-labeled hyperedges and positive and negative context-conditions. Such a rule is applicable if both the positive and negative context-conditions of the rule are satisfied. Its application consumes the two hyperedges and fuses the sources of the one hyperedge with the sources of the other as well as the targets of the one with the targets of the other.

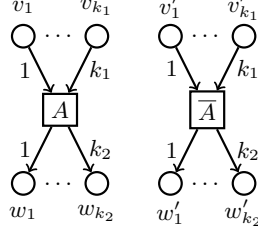


Fig. 1: The fusion rule $fr(A)$ with $type(A) = (k_1, k_2)$

- Definition 2.*
1. $F \subseteq \Sigma$ is a *fusion alphabet* if it is accompanied by a *complementary fusion alphabet* $\bar{F} = \{\bar{A} \mid A \in F\} \subseteq \Sigma$, where $F \cap \bar{F} = \emptyset$ and $\bar{A} \neq \bar{B}$ for $A, B \in F$ with $A \neq B$ and a *type function* $type: F \cup \bar{F} \rightarrow (\mathbb{N} \times \mathbb{N})$ with $type(A) = type(\bar{A})$ for each $A \in F$.
 2. For each $A \in F$ with $type(A) = (k_1, k_2)$, the *fusion rule* $fr(A)$ is the hypergraph, depicted in Figure 1, with $V_{fr(A)} = \{v_i, v'_i \mid i = 1, \dots, k_1\} \cup \{w_j, w'_j \mid j = 1, \dots, k_2\}$, $E_{fr(A)} = \{e, \bar{e}\}$, $s_{fr(A)}(e) = v_1 \cdots v_{k_1}$, $s_{fr(A)}(\bar{e}) = v'_1 \cdots v'_{k_1}$, $t_{fr(A)}(e) = w_1 \cdots w_{k_2}$, $t_{fr(A)}(\bar{e}) = w'_1 \cdots w'_{k_2}$, and $lab_{fr(A)}(e) = A$ and $lab_{fr(A)}(\bar{e}) = \bar{A}$.
 3. The application of $fr(A)$ to a hypergraph $H \in \mathcal{H}_\Sigma$ proceeds according to the following steps: (1) Choose a *matching morphism* $g: fr(A) \rightarrow H$. (2) Remove the images of the two hyperedges of $fr(A)$ yielding $X = H - (\emptyset, \{g(e), g(\bar{e})\})$. (3) Fuse the corresponding source and target vertices of the removed hyperedges yielding the hypergraph $H' = X/\equiv$ where \equiv is generated by the relation $\{(g(v_i), g(v'_i)) \mid i = 1, \dots, k_1\} \cup \{(g(w_j), g(w'_j)) \mid j = 1, \dots, k_2\}$. The application of $fr(A)$ to H is denoted by $H \xRightarrow[fr(A)]{} H'$ and called a *direct derivation*.
 4. A *context-dependent fusion rule* is a triple $cdfr = (fr(A), PC, NC)$ for some $A \in F$ where PC and NC are two finite sets of hypergraph morphisms with domain $fr(A)$ mapping into finite contexts defining *positive* and *negative context conditions* respectively.
 5. The rule $cdfr$ is applicable to some hypergraph H via a matching morphism $g: fr(A) \rightarrow H$ if for each $(c: fr(A) \rightarrow C) \in PC$ there exists a hypergraph morphism $h: C \rightarrow H$ such that h is injective on the set of hyperedges and $h \circ c = g$, and for all $(c: fr(A) \rightarrow C) \in NC$ there does not exist a hypergraph morphism $h: C \rightarrow H$ such that $h \circ c = g$.
 6. If $cdfr$ is applicable to H via g , then the direct derivation $H \xRightarrow[cdfr]{} H'$ is the direct derivation $H \xRightarrow[fr(A)]{} H'$.

Remark 2. $fr(A)$ and $(fr(A), \emptyset, \emptyset)$ are equivalent. We use the first as an abbreviation for the latter.

Given a finite hypergraph, the set of all possible successive fusions is finite as fusion rules never create anything. To overcome this limitation, arbitrary multi-

plications of disjoint components within derivations are allowed. The generated language consists of the terminal part of all resulting connected components that contain no fusion symbols and at least one marker symbol, where marker symbols are removed in the end. These marker symbols allow us to distinguish between wanted and unwanted terminal components.

- Definition 3.* 1. A *context-dependent fusion grammar* is a system $CDFG = (Z, F, M, T, P)$ where $Z \in \mathcal{H}_{F \cup \bar{F} \cup T \cup M}$ is a *start hypergraph* consisting of a finite number of connected components, $F \subseteq \Sigma$ is a finite fusion alphabet, $M \subseteq \Sigma$ with $M \cap (F \cup \bar{F}) = \emptyset$ is a finite set of *markers*, $T \subseteq \Sigma$ with $T \cap (F \cup \bar{F}) = \emptyset = T \cap M$ is a finite set of *terminal labels*, and P is a finite set of context-dependent fusion rules.
2. A *direct derivation* $H \Rightarrow H'$ is either a context-dependent fusion rule application $H \xRightarrow{cdf} H'$ for some $cdf \in P$ or a multiplication $H \xRightarrow{m} m \cdot H$ for some multiplicity $m: \mathcal{C}(H) \rightarrow \mathbb{N}$. A *derivation* $H \xRightarrow{n} H'$ of length $n \geq 0$ is a sequence of direct derivations $H_0 \Rightarrow H_1 \Rightarrow \dots \Rightarrow H_n$ with $H = H_0$ and $H' = H_n$. If the length does not matter, we may write $H \xRightarrow{*} H'$.
3. $L(CDFG) = \{rem_M(Y) \mid Z \xRightarrow{*} H, Y \in \mathcal{C}(H) \cap (\mathcal{H}_{T \cup M} \setminus \mathcal{H}_T)\}$ is the *generated language* where $rem_M(Y)$ is the terminal hypergraph obtained by removing all hyperedges with labels in M from Y .

5 Transformation of Turing Machines into Context-Dependent Fusion Grammars

In this section, we transform Turing machines into context-dependent fusion grammars such that the recognized language of the Turing machine and the generated language of the corresponding context-dependent fusion grammar coincide up to representation of strings as graphs. The construction uses a graphical representation of Turing machines and simulates a step of the machine by a sequence of context-dependent fusion rules. In our construction positive and negative context conditions are needed to restrict the applicability of fusion rules in order to obtain a correct and sound transformation. Some of the context conditions derive directly from the semantics of a Turing machine. For example, the step $(p, \alpha u, x, \beta) \vdash_{TM} (q, \alpha, u, y\beta)$ can only be applied if $(p, x, y, l, q) \in \Delta$, the Turing machine is in state p and reads the symbol x . Other context conditions are needed because (context-dependent) fusion rules can only consume two hyperedges at a time.

Because the transformation is quite complicated we introduce the ideas step by step. First, we introduce the tape graph representing the working tape as well as the input to the Turing machine. Afterwards, we give a hypergraph representation of Turing machines and configurations and demonstrate how a step can be simulated by a sequence of context-dependent fusion rules. Finally, the two constructions are combined and our main theorem is presented.

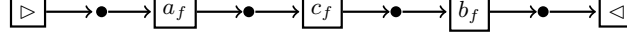


Fig. 2: The extendable string graph $\triangleright sg(f(acb)) \triangleleft$

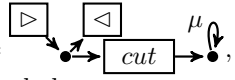
5.1 Tape graph

In our construction the tape is represented by an infinitely extendable tape graph. We cut off the infinite \square -strings to the left and to the right and add \square by rules if they are needed. Furthermore, due to technical reasons, the tape graph contains two connected corresponding string graphs, where one is labeled over the terminal alphabet Ω and the other is labeled over the fusion alphabet $\Gamma_f = (\Gamma \setminus \Omega) + \Omega_f$, where $\Omega_f = \{x_f \mid x \in \Omega\}$. This is because in fusion grammars fusion alphabets and terminal alphabets are disjoint but $\Omega \subsetneq \Gamma$ by definition of the Turing machine. The construction can be seen as having two tapes initially with the same content, where the first tape is left invariant. If the machine halts in the accepting state, then the content of the first tape is used as a contribution to the generated language. Due to the construction of Γ_f we need a mapping $f: \Gamma^* \rightarrow \Gamma_f^*$ defined by $f(x) = x_f$, if $x \in \Omega$, and $f(x) = x$, otherwise. The mapping is defined for infinite strings because $f(\square^\infty \alpha \square^\infty) = \square^\infty f(\alpha) \square^\infty$, where $\alpha = \alpha_1 \cdots \alpha_n$ with $\alpha_1 \neq \square \neq \alpha_n$. For example, the infinite string $\square^\infty acb \square^\infty$ is represented by the graph $\triangleright sg(f(acb)) \triangleleft$, depicted in Figure 2, where \triangleright and \triangleleft are fusion labels used for extending the string graphs $sg(f(acb))$. The fusion labels \triangleright and \triangleleft allow us to generate for each $i, j \in \mathbb{N}$ the graph $\triangleright sg(\square^i f(acb) \square^j) \triangleleft$.

Definition 4. 1. Let $\alpha, \beta \in \Gamma^*, x \in \Gamma, w \in \Omega^*$. Let cut, \triangleright and \triangleleft be fusion symbols with $type(\triangleright) = (0, 1)$, $type(\triangleleft) = (1, 0)$, and $type(cut) = (1, 1)$, and let $sg(cut \cdot w)_\mu$ be the string graph with an additional μ -labeled loop attached to the first vertex of $sg(w)$. Then

$$tg(\alpha, x, \beta, w) = (\triangleright sg(f(\alpha x \beta)) \triangleleft + sg(cut \cdot w)_\mu) /_{s(f(x)) \equiv s(cut)}$$

where $f(x)$ and cut are the respective labeled hyperedge in the two string graphs. We call these graphs *tape graphs*.

2. As a special case we define the hypergraph $tg(\varepsilon, \perp, \varepsilon, \varepsilon) =$

, where $\perp \notin \Gamma$ is used for representing the absence of a symbol.
3. For each $tg(\alpha, x, \beta, w)$ define $tg(\alpha, x, \beta, w)_{tape}$ which is $tg(\alpha, x, \beta, w)$ with an additional *tape*-hyperedge attached to the source vertex of the *cut*-hyperedge.
4. Let cut, \triangleright and \triangleleft be as before. Let $F_{tg} = \{tape, gen, cut, \triangleright, \triangleleft\} + \Gamma_f$ be a fusion alphabet with $type(tape) = (0, 1)$, and $type(gen) = type(x) = (1, 1)$ for each $x \in \Gamma_f$. Define

$$CDFG_{tg}(\Omega, \Gamma) = (Z_{tg}, F_{tg}, \{\mu\}, \Omega, \{fr(gen), fr(\triangleright), fr(\triangleleft)\}),$$

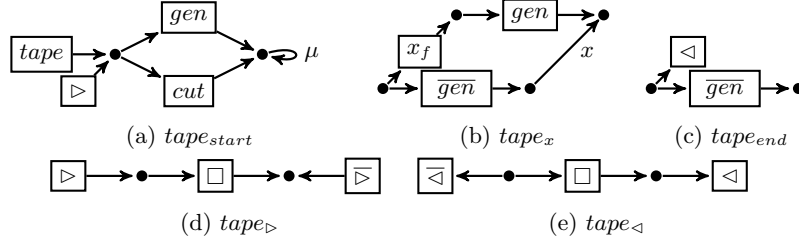


Fig. 3: The connected components of the start hypergraph Z_{tg} .

where the start hypergraph $Z_{tg} = tape_{start} + tape_{end} + \sum_{x \in \Omega} tape_x + tape_{\triangleright} + tape_{\triangleleft}$ consists of the connected components depicted² in Figure 3.

Remark 3. The connected components $tape_{start}, tape_x$ for $x \in \Omega$ and $tape_{end}$ and the fusion rule $fr(gen)$ are used to generate the two connected corresponding string graphs. The terminal-labeled string graph carries a marker hyperedge. The two corresponding string graphs are connected via a hyperedge labeled cut . $tape_{\triangleright}$ and $tape_{\triangleleft}$ as well as the fusion rules $fr(\triangleright)$ and $fr(\triangleleft)$ are used to extend the latter string graphs with \square -labeled hyperedges an unbounded number of times. The $tape$ -hyperedge is later used for attaching the tape graph to a hypergraph representation of a Turing machine and the cut -hyperedge is used to disconnect the terminal- and marker-labeled string graph.

Example 1. Let $\Omega = \{a, b, c\}$ and $\Gamma = \{a, b, c, \square\}$. Then by definition $CDFG_{example}(\Omega, \Gamma) = (Z_{example}, \{tape, gen, cut, \triangleright, \triangleleft, a_f, b_f, c_f, \square\}, \{\mu\}, \Omega, P_{tg})$ with $Z_{example} = tape_{start} + tape_{end} + tape_a + tape_b + tape_c + tape_{\triangleright} + tape_{\triangleleft}$ and P_{tg} denotes the set of context-dependent fusion rules specified in $CDFG_{tg}(\Omega, \Gamma)$. A derivation may be

$$\begin{aligned}
Z_{example} &\xRightarrow{fr(gen)} tape_{starta} + tape_{end} + tape_b + tape_c + tape_{\triangleright} + tape_{\triangleleft} \\
&\xRightarrow{fr(gen)} tape_{startac} + tape_{end} + tape_b + tape_{\triangleright} + tape_{\triangleleft} \\
&\xRightarrow{fr(gen)} tape_{startac} + tape_{bend} + tape_{\triangleright} + tape_{\triangleleft} \\
&\xRightarrow{fr(gen)} tg(\varepsilon, a, cb, acb)_{tape} + tape_{\triangleright} + tape_{\triangleleft},
\end{aligned}$$

where first $tape_{start}$ and $tape_a$ are fused wrt the two complementary gen - and \overline{gen} -hyperedges yielding $tape_{starta}$, then $tape_{starta}$ and $tape_c$ are fused yielding $tape_{startac}$, then $tape_{end}$ and $tape_b$ are fused yielding $tape_{bend}$, and finally $tape_{startac}$ and $tape_{bend}$ are fused yielding $tg(\varepsilon, a, cb, acb)_{tape}$ which is depicted in Figure 4a.

² We depict hyperedges with one source and one target with labels $x \in \Sigma \setminus F$ by $\bullet \xrightarrow{x} \bullet$.

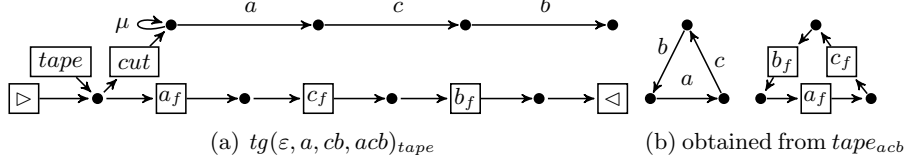


Fig. 4: Some hypergraphs derivable in $CDFG_{example}(\Omega, \Gamma)$

Another example is the hypergraph in Figure 4b. This hypergraph is derivable because fusions within some connected component – in this case $tape_{acb}$ – is possible. Note that the left connected component is terminal labeled. However, it does not contribute to the generated language because it lacks a marker hyperedge.

Proposition 1. *For each $tg(\square^i, x, w \cdot \square^j, w')$ where $i, j \in \mathbb{N}$ and either $x = \perp$ and $w = w' = \varepsilon$ or $x \in \Omega$, $w \in \Omega^*$ and $w' = xw$ exists a derivation $Z \xRightarrow{*} tg(\square^i, x, w \cdot \square^j, w')_{tape}$ in $CDFG_{tg}(\Omega, \Gamma)$.*

Proposition 2. $L(CDFG_{tg}(\Omega, \Gamma)) = \emptyset$.

If the *cut*-hyperedge in some tape graph is fused with the complementary hyperedge in $z_{\overline{cut}} = \bullet \rightarrow \boxed{\overline{cut}} \rightarrow \bullet$ (which is not present in Z_{tg}) then this yields two connected components one of which is $sg(w)_\mu$ for some $w \in \Omega^*$.

Proposition 3. *Let $CDFG_{tg+\overline{cut}}(\Omega, \Gamma) = (Z_{tg} + z_{\overline{cut}}, F_{tg}, \{\mu\}, \Omega, P'_{tg})$, where $P'_{tg} = P_{tg} \cup \{fr(\overline{cut})\}$. Then $L(CDFG_{tg+\overline{cut}}(\Omega, \Gamma)) = \{sg(w) \mid w \in \Omega^*\}$.*

5.2 A hypergraph representation of Turing machines and their configurations

In the hypergraphical representation of a Turing machine, denoted by $hg(TM)$, vertices represent states and hyperedges between these vertices represent the elements of the transition relation, i.e., $(p, x, y, dir, q) \in \Delta$ implies $e \in E_{hg(TM)}$ with $s(e) = p, t(e) = q, lab(e) = x/y/dir$. The tape graph is connected to $hg(TM)$ by a hyperedge with $|Q|$ sources and one target, called *head*, where the sources are the states of the Turing machine and the target is a vertex in the tape graph. The order in which the states of the Turing machine are connected to the sources of the *head*-hyperedge implements a permutation σ . The *head*-hyperedge has three purposes: (1) It is used to connect $hg(TM)$ to a tape graph, (2) it signifies the current state (specifically, the current state is the first source), and (3) it points to the current symbol to be read. This connected component consisting of the tape graph, the hypergraph representation of the Turing machine and the *head*-hyperedge is used for representing configurations. Therefore, it is called hypergraph representation of the configuration with adjunct w and permutation σ , where w is the terminal labeled string graph in the tape graph.

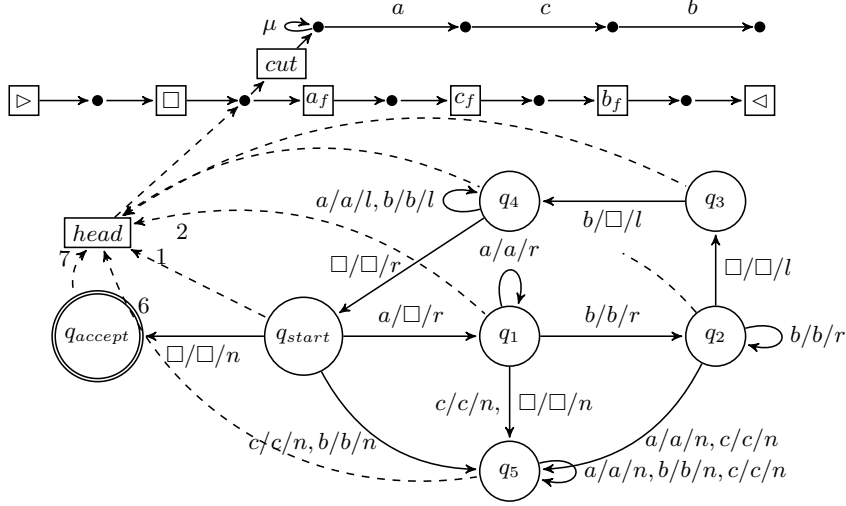


Fig. 5: $hg(q_{start}q_2q_3q_4q_5q_{accept}, \square, a, cb, acb)$ where the *head*-hyperedge is dashed. Parallel edges are indicated by a comma-separated list in the labels in order to clarify the drawing.

Definition 5. Let $TM = (Q, \Omega, \Gamma, \Delta)$ be a Turing machine.

1. Let $q_1 \dots q_{|Q|}$ be a sequence of states of Q , where each state occurs exactly once. Define $hg(TM, q_1 \dots q_{|Q|}) = (Q + \{v_{head}\}, \{head\} + \Delta, s, t, lab)$, where
 - $s(head) = q_1 \dots q_{|Q|}$, $t(head) = v_{head}$, and $lab(head) = head$,
 - $s(\delta) = p$, $t(\delta) = q$, and $lab(\delta) = x/y/dir$, where $\delta = (p, x, y, dir, q) \in \Delta$.
2. Let $c = (q, \square^\infty \cdot \alpha', x, \beta' \cdot \square^\infty) \in conf(TM)$, $\alpha' = \alpha'_1 \dots \alpha'_n$, $\alpha'_1 \neq \square$, $\beta' = \beta'_1 \dots \beta'_n$, $\beta'_n \neq \square$, let $w \in \Omega^*$, let again *cut* be a fusion symbol with type $(1, 1)$, and let $sg(cut \cdot w)_\mu$ be the string graph with an additional μ -labeled loop attached to the first vertex of $sg(w) \subset sg(cut \cdot w)_\mu$. Let $q_1 \dots q_{|Q|}$ be a sequence of states of Q where $q_1 = q$. Define $hg(q_1 \dots q_{|Q|}, \alpha', x, \beta', w) = (hg(TM, q_1 \dots q_{|Q|}) + tg(\alpha', x, \beta', w)) /_{v_{head} \equiv s(cut)}$ where *cut* is the *cut*-labeled hyperedge. We call $hg(q_1 \dots q_{|Q|}, \alpha', x, \beta', w)$ the *hypergraph representation of the configuration c with adjunct w and permutation $q_1 \dots q_{|Q|}$* .

Example 2. Easily one can construct a Turing machine which recognizes the language $\{a^n b^n \mid n \geq 0\}$. A hypergraph representation of the configuration $(q_{start}, \square^\infty \square, a, cb \square^\infty)$ with adjunct *acb* and permutation $q_{start}q_2q_3q_4q_5q_{accept}$ is shown in Figure 5.

5.3 Simulating steps of a Turing machine by context-dependent fusion rules

In order to simulate a step further connected components are needed. These connected components encode the substitution of the symbol on the tape, the

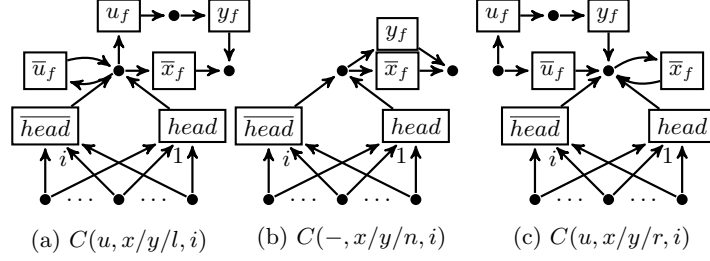


Fig. 6: Schematic drawings of connected components used for simulating a step of a Turing machine

movement of the head and the transition to the next state. In order to move the head to the left or to the right our construction takes both the current symbol and the symbol to the left of the head into account. The relations of the Turing machine can be seen as replacing or deleting the current symbol and (maybe) inserting a new symbol left or right of the head. In the graph representation this corresponds to deleting and inserting edges. These deletions and insertions are done with respective fusions of complementary labeled hyperedges. The hypergraphs in Figure 6a–6c are schematic drawings of the connected components used for simulating a step of a Turing machine. The dots indicate that there are $|Q|$ vertices as sources. Two complementary $head$ and \overline{head} -hyperedges attached to the same vertices are part of this connected component, where the ordering of the source attachments of the $head$ -hyperedge implements a permutation such that the first and i th source are swapped. Formally, these components are defined as follows.

Definition 6. Let $\{head\} + \Gamma_f$ be a fusion alphabet with $type(head) = (|Q|, 1)$ and $type(x) = (1, 1)$ for each $x \in \Gamma_f$. Let $\Lambda = \{x/y/dir \mid x, y \in \Gamma, dir \in \{l, n, r\}\}$.

1. For each $u \in \Gamma_f, x/y/l \in \Lambda, i \in \mathbb{N}, 0 < i \leq |Q|$ define $C(u, x/y/l, i) = (\{v_1, \dots, v_{|Q|+3}\}, \{e_1, \dots, e_6\}, s, t, lab)$ where
 - $s(e_1) = v_1 v_2 \dots v_i \dots v_{|Q|}, t(e_1) = v_{|Q|+1}, lab(e_1) = head$
 - $s(e_2) = v_i v_2 \dots v_1 \dots v_{|Q|}, t(e_2) = v_{|Q|+1}, lab(e_2) = \overline{head}$
 - $s(e_3) = t(e_3) = v_{|Q|+1}, lab(e_3) = \overline{u}_f$
 - $s(e_4) = v_{|Q|+1}, t(e_4) = v_{|Q|+3}, lab(e_4) = u_f$
 - $s(e_5) = v_{|Q|+1}, t(e_5) = v_{|Q|+2}, lab(e_5) = \overline{x}_f$
 - $s(e_6) = v_{|Q|+2}, t(e_6) = v_{|Q|+3}, lab(e_6) = y_f$.
2. For each $x/y/n \in \Lambda, i \in \mathbb{N}, 0 < i \leq |Q|$ define $C(-, x/y/n, i) = (\{v_1, \dots, v_{|Q|+2}\}, \{e_1, \dots, e_4\}, s, t, lab)$ where
 - $s(e_1) = v_1 v_2 \dots v_i \dots v_{|Q|}, t(e_1) = v_{|Q|+1}, lab(e_1) = head$
 - $s(e_2) = v_i v_2 \dots v_1 \dots v_{|Q|}, t(e_2) = v_{|Q|+1}, lab(e_2) = \overline{head}$
 - $s(e_3) = s(e_4) v_{|Q|+1}, t(e_3) = t(e_4) = v_{|Q|+2}, lab(e_3) = y_f, lab(e_4) = \overline{x}_f$.
3. For each $u \in \Gamma_f, x/y/r \in \Lambda, i \in \mathbb{N}, 0 < i \leq |Q|$ define $C(u, x/y/r, i) = (\{v_1, \dots, v_{|Q|+3}\}, \{e_1, \dots, e_6\}, s, t, lab)$ where

- $s(e_1) = v_1 v_2 \cdots v_i \cdots v_{|Q|}, t(e_1) = v_{|Q|+2}, lab(e_1) = head$
- $s(e_2) = v_i v_2 \cdots v_1 \cdots v_{|Q|}, t(e_2) = v_{|Q|+2}, lab(e_2) = \overline{head}$
- $s(e_3) = t(e_3) = v_{|Q|+2}, lab(e_3) = \overline{x}_f$
- $s(e_4) = v_{|Q|+1}, t(e_4) = v_{|Q|+3}, lab(e_4) = u_f$
- $s(e_5) = v_{|Q|+1}, t(e_5) = v_{|Q|+2}, lab(e_5) = \overline{u}_f$
- $s(e_6) = v_{|Q|+2}, t(e_6) = v_{|Q|+3}, lab(e_6) = y_f$.

In order to simulate a step of a Turing machine context-dependent fusion rules are needed. Because (context-dependent) fusion rules can only fuse two complementary hyperedges in one derivation step but a step of a Turing machine is much more complicated (head movement, tape manipulation, state transition) several rules and rule applications are needed to simulate such a step. Furthermore, in our construction positive and negative context conditions are needed to restrict the application to obtain a correct and sound simulation. The in the following defined set of context-dependent fusion rules P_Δ contains rules with respect to the fusion symbol *head* and rules with respect to the fusion symbol x for each $x \in \Gamma$. The first are used to fuse the *head*-hyperedge in the graph representation of a configuration with the correct connected component used for simulating the step of the Turing machine and perform the state transition. The latter are used to modify the tape and move the head correctly. We use $fuse(x)$ as a shorthand for some rule in the latter set in P_Δ if there is no need to distinguish. The $fuse(x)$ rules are constructed in such a way that the two complementary hyperedges must be attached to the same vertex and that never two rules are applicable at the same time with respect to the same connected component. This is achieved by defining $fuse_2in(x)$ and $fuse_loop_in(x)$ as well as $fuse_2out(x)$ and $fuse_loop_out(x)$ in such a way that the positive context condition of the first is reflected in the negative context condition of the other. Further, in order to distinguish the applicability of $fuse_loop_in(x)$ and $fuse_loop_out(x)$ if in-going and out-going edges (wrt some vertex where the loop is attached) are labeled with the same symbol, the number of out-going (in-going) edges, respectively, is of relevance. If the number of out-going (in-going) edges is 4, then the loop must be fused with the in-going (out-going) edge, respectively. $fuse_loop_in(x)$ and $fuse_loop_out(x)$ do not need negative context conditions because the positive context does not occur in the C -components of Definition 6.

Definition 7. Define P_Δ as the following set of context-dependent fusion rules.

$$P_\Delta = \{ \Delta(u, \lambda) \mid u \in \Gamma, \lambda \in \Lambda \} \\ \cup \{ fuse_2in(x), fuse_2out(x), fuse_loop_in(x), fuse_loop_out(x) \mid x \in \Gamma \}$$

$\Delta(u, \lambda) = (fr(head), \{ fr(head) \rightarrow PC(u, \lambda, j) + C(u, \lambda, j) \mid 0 < j \leq |Q| \}, \{ fr(head) \rightarrow twoin(u) + \overline{h}^\bullet \mid u \in \Gamma \}) \cup \{ fr(head) \rightarrow twoout(u) + \overline{h}^\bullet \mid u \in \Gamma \}$, where the morphism in the positive context maps the *head*-hyperedge to the one in $PC(u, \lambda, j)$, depicted in Figure 7a, and the *head*-hyperedge to the one in $C(u, \lambda, j)$. Note that, the connected component $C(u, \lambda, j)$ is induced by the j th source of the *head*-hyperedge and the parameters u, λ . $twoin(u)$, $twoout(u)$ and \overline{h}^\bullet are depicted in Figure 7b, 7c and 7d, respectively.

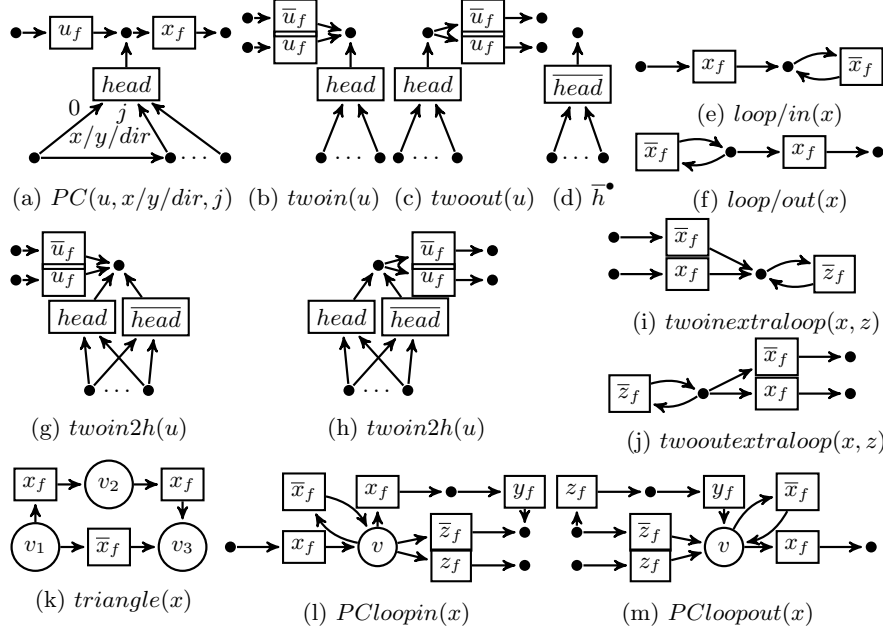


Fig. 7: Some contexts for applying rules simulating a step

$\text{fuse_2in}(x) = (\text{fr}(x), \{\text{fr}(x) \rightarrow \text{twoin}(x)\}, \{\text{fr}(x) \rightarrow \text{twoin2h}(x), \text{fr}(x) \rightarrow \text{loop/in}(x), \text{fr}(x) \rightarrow \text{triangle}(x)\} \cup \{\text{fr}(x) \rightarrow \text{twoinextraloop}(x, z) \mid z \in \Gamma\})$, where $\text{fr}(x) \rightarrow \text{triangle}(x)$ maps the x -hyperedge to $e \in E_{\text{triangle}(x)}$ with $s(e) = v_2$ and $t(e) = v_3$; $\text{fr}(x) \rightarrow \text{twoinextraloop}(x, z)$ maps the \bar{x} -hyperedge to the one above the the x -hyperedge (which is relevant for the case $z = x$). The respective hypergraphs used in the context conditions are depicted in Figure 7.

$\text{fuse_2out}(x) = (\text{fr}(x), \{\text{fr}(x) \rightarrow \text{twoout}(x)\}, \{\text{fr}(x) \rightarrow \text{twoout2h}(x), \text{fr}(x) \rightarrow \text{loop/out}(x), \text{fr}(x) \rightarrow \text{triangle}(x)\} \cup \{\text{fr}(x) \rightarrow \text{twooutextraloop}(x, z) \mid z \in \Gamma\})$ is analog but $\text{fr}(x) \rightarrow \text{triangle}(x)$ maps the x -hyperedge to $e \in E_{\text{triangle}(x)}$ with $s(e) = v_1$ and $t(e) = v_2$.

$\text{fuse_loop_in}(x) = (\text{fr}(x), \{\text{fr}(x) \rightarrow \text{PCloopin}(x)\}, \emptyset)$, where $\text{PCloopin}(x)$ is depicted in Figure 7l and in the positive context condition the x -hyperedge matches the one with target v and the \bar{x} -hyperedge matches the one with source and target v .

$\text{fuse_loop_out}(x) = (\text{fr}(x), \{\text{fr}(x) \rightarrow \text{PCloopout}(x)\}, \emptyset)$, is analog but the x -hyperedge matches the one with source v . $\text{PCloopout}(x)$ is depicted in Figure 7m.

Example 3. Consider the hypergraph $hg(q_{\text{start}}q_2q_3q_4q_5q_{\text{accept}}, \square, a, cb, acb) + C(\square, a/\square/r, 2)$. The context-dependent fusion rule $\Delta(\square, a/\square/r)$ can be applied by matching the *head*-hyperedge in $hg(q_{\text{start}}q_2q_3q_4q_5q_{\text{accept}}, \square, a, cb, acb)$ and the *head*-hyperedge in $C(\square, a/\square/r, 2)$. The two complementary hyperedges are fused

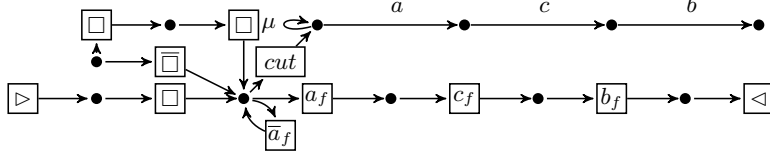


Fig. 8: The tape graph with the additional vertices and hyperedges

and due to the *head*-hyperedge in $C(\square, a/\square/r, 2)$ a new *head*-hyperedge reconstructed; its first source is q_2 . Furthermore, the application attaches additional vertices and hyperedges to the tape graph. The tape graph is depicted in Figure 8. The \bar{a}_f -hyperedge is then fused with the out-going a_f -hyperedge by an application of *fuse_loop_out*(a) with the result that the source and the target vertex of the a_f -hyperedge are glued together. Afterwards, *fuse_2in*(\square) can be applied to the \square - and \square -hyperedge depicted to the left of the center vertex v_{tape} . The resulting connected component is $hg(q_2q_{start}q_3q_4q_5q_{accept}, \square\square, c, b, acb)$.

5.4 Construction of a context-dependent fusion grammar corresponding to a Turing machine

Now we combine the previous constructions in a context-dependent fusion grammar. Using a similar construction as in Definition 5 the initial hypergraph representation of the Turing machine is defined. This connected component is equipped with an *tape*-hyperedge, such that if the *tape*-hyperedge and the *tape*-hyperedge in the generated tape graph (with additional *tape*-hyperedge as in Definition 4) are fused, then this yields a hypergraph representation of a start configuration. Furthermore, additional context-dependent fusion rules are needed (1) for connecting the initial hypergraph representation of the Turing machine with the generated tape graph, (2) for acceptance and (3) for disconnecting³ the terminal and marker labeled string graph $sg(w)_\mu$ for some $w \in \Omega^*$.

Definition 8. Let $TM = (Q, \Omega, \Gamma, \Delta)$ be a Turing machine. Let $CDFG_{tg}(\Omega, \Gamma) = (Z_{tg}, F_{tg}, \{\mu\}, \Omega, P_{tg})$ be the context-dependent fusion grammar generating tape graphs according to Definition 4. Let $C(u, \lambda, i)$ be the connected component and P_Δ be the set of context-dependent fusion rules defined in Definition 6 and 7, respectively. Then

$$CDFG(TM) = (Z_{TM}, \{head\} + F_{tg}, \{\mu\}, \{term\} + \Omega + \Lambda, P_{TM})$$

is the corresponding context-dependent fusion grammar where

$$Z_{TM} = Z_{tg} + hg(TM)_{init} + Acc + \sum_{\substack{u \in \Gamma_f, \lambda \in \Lambda \\ 0 < i \leq |Q|}} C(u, \lambda, i) \quad \text{where} \\ hg(TM)_{init} = hg(TM, q_{start}q_2 \cdots q_{|Q|}) + (\emptyset, \{\overline{tape}\}) \text{ with } s(\overline{tape}) = \varepsilon, t(\overline{tape}) =$$

³ Disconnecting $sg(w)_\mu$ uses similar ideas as discussed in Proposition 3.

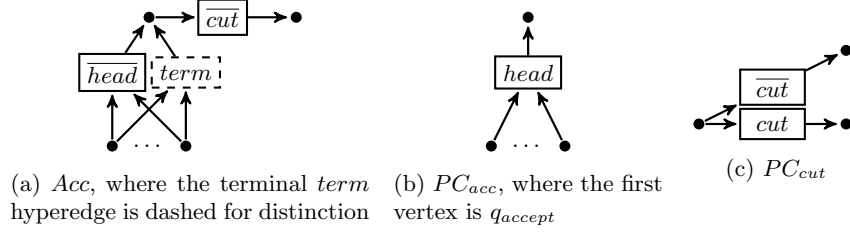


Fig. 9: Schematic drawings of connected components used for acceptance and for disconnecting the terminal and marker labeled string graph.

v_{tape} , and $lab(\overline{tape}) = \overline{tape}$, where ε denotes the empty sequence, and $Acc = (\{v_1, \dots, v_{|Q|+2}\}, \{e_1, \dots, e_3\}, s, t, lab)$ where $s(e_1) = v_1 v_2 \dots v_i \dots v_{|Q|}$, $t(e_1) = v_{|Q|+1}$, $lab(e_1) = term$, $s(e_2) = v_i v_2 \dots v_1 \dots v_{|Q|}$, $t(e_2) = v_{|Q|+1}$, $lab(e_2) = \overline{head}$, $s(e_3) = v_{|Q|+1}$, $t(e_3) = v_{|Q|+2}$, $lab(e_3) = cut$. A schematic drawings of Acc is depicted in Figure 9a.

$P_{TM} = P_{tg} \cup P_{\Delta} \cup \{fr(tape), accept, fuse_cut\}$ where $accept = (fr(head), \{fr(head) \rightarrow PC_{acc} + Acc\}, \{fr(head) \rightarrow c + Acc \mid c \in \mathcal{C}(\sum_{\substack{x \in \Gamma_f, \lambda \in A \\ 0 < i \leq |Q|}} C(x, \lambda, i))\})$, where $PC_{acc} = (Q + \{v_{tape}\}, \{head\}, s, t, lab)$ with $s(head) =$

$q_{accept} q_1 \dots q_{|Q|-1}$, $t(head) = v_{tape}$, and $lab(head) = head$, where $q_1 \dots q_{|Q|-1}$ are the states of $Q \setminus \{q_{accept}\}$ in arbitrary order. A schematic drawings of PC_{acc} is depicted in Figure 9b;

$fuse_cut = (fr(cut), \{fr(cut) \rightarrow (\{v_1, v_2, v_3\}, \{e_1, e_2\}, s, t, lab)\}, \emptyset)$, where $s(e_1) = s(e_2) = v_1$, $t(e_1) = v_2$, $t(e_2) = v_3$ and $lab(e_1) = cut$, $lab(e_2) = \overline{cut}$, i.e., the source of the two complementary hyperedges must be the same and the targets different. The connected component is depicted in Figure 9c.

Our main theorem is that the language recognized by some Turing machine and the generated language of the corresponding context-dependent fusion grammar coincide up to representation of strings as graphs.

Theorem 1. $L(CDFG(TM)) = \{sg(w) \mid w \in L(TM)\}$.

6 Conclusion

In this paper, we have continued the research on context-dependent fusion grammars by transforming Turing machines into this type of hypergraph grammars. This reduction gives us interesting insights into these grammars because the transformation proves that context-dependent fusion grammars are another universal computing model and can generate all recursive enumerable languages. Note that a similar construction also works for computation of partial functions. In this case the connected components $tape_{start}$, $tape_{end}$ and $tape_x$ are replaced by a tape graph representing the Turing machines input $x_1 \dots x_n \in \Sigma^*$, where

the start is attached to some *tape*-hyperedge. However, further research is needed including the following open question. Is it true that only positive or only negative context conditions are not powerful enough to simulate Turing machines?

Acknowledgment. We are grateful to Hans-Jörg Kreowski and Sabine Kuske as well as to the anonymous reviewers for their valuable comments.

References

1. Hans-Jörg Kreowski, Sabine Kuske, and Aaron Lye. Fusion Grammars: A Novel Approach to the Generation of Graph Languages. In Detlef Plump and Juan de Lara, editors, *Proc. 10th International Conference on Graph Transformation (ICGT 2017)*, volume 10373 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2017.
2. Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *DNA Computing — New Computing Paradigms*. Springer, 1998.
3. Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.
4. Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and fractals - new frontiers of science (2. ed.)*. Springer, 2004.
5. Branko Grünbaum and Goffrey C. Shephard. *Tilings and Patterns*. W. H. Freeman and Company, 1987.
6. Aaron Lye. Decidability and complexity of the membership and emptiness problem of fusion grammars. In *Pre-Proc. 9th International Workshop on Graph Computation Models, (GCM 2018)*, 2018.
7. Hans-Jörg Kreowski, Sabine Kuske, and Aaron Lye. Transformation of Petri nets into context-dependent fusion grammars. In Carlos Martín-Vide, Alexander Okhotin, and Dana Shapira, editors, *Proc. 13th International Conference on Language and Automata Theory and Applications (LATA 2019)*, volume 11417 of *LNCS*, pages 246–258. Springer, 2019.
8. Tadahiro Uesu. A system of graph grammars which generates all recursively enumerable sets of labelled graphs. *Tsukuba journal of mathematics*, 2:11–26, 1978.
9. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In Furio Honsell and Marino Miculan, editors, *Proc. Foundations of Software Science and Computation Structures (FOSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.
10. Hans-Jörg Kreowski, Sabine Kuske, and Aaron Lye. Splicing/fusion grammars and their relation to hypergraph grammars. In Leen Lambers and Jens H. Weber, editors, *Proc. 11th International Conference on Graph Transformation (ICGT 2018)*, volume 10887 of *LNCS*, pages 3–19. Springer, 2018.
11. Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. A correction was published in *Proceedings of the London Mathematical Society*. 2 (1937). 43 (6): 544–6.
12. Juraj Hromkovic. *Theoretical computer science*. Texts in theoretical computer science, EATCS series. Springer, Berlin, 2004.
13. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

Analyzing and Pumping Hyperedge Replacement Formalisms in a Common Framework

Martin Berglund

Institute for Software Technology, Universität der Bundeswehr München
`martin.berglund@unibw.de`

Abstract. We present the first steps of an investigation into the relationship between contextual hyperedge replacement grammars and the standard non-contextual kind. The formalisms are restated in a common framework, discarding some of the specialized details (distinguished node labels), and discuss the effects these changes have on the formalisms. The resulting definitions are then applied to give a strengthened pumping lemma for the normal case, which is then transferred into a novel pumping lemma for the contextual case.

1 Introduction

Contextual hyperedge replacement grammars were introduced in [HM10] as an extension of hyperedge replacement grammars [Hab92]. The more expressive rules make it possible to, among other things, capture graph languages with arbitrarily high levels of connectivity, which is not possible in normal hyperedge replacement grammars. Examples of additional languages which can be captured in this formalism include the language of all directed acyclic graphs and broad classes of *chart languages* describing software architectures and code [DH15]. While many of the properties of contextual hyperedge replacement grammars have been investigated, some of the technical choices have made it hard to directly relate them to hyperedge replacement grammars, and thus hard to reuse the many deep results which exist for those. In this paper we give definitions of both contextual and hyperedge replacement grammars within a common framework, illustrate the effects of the small adjustments this necessitates, and use this common framework to state a powerful pumping lemma for contextual hyperedge replacement languages (in the process also offering a Ogden-style [Ogd68] pumping lemma for hyperedge replacement grammars). This is a first step towards not only refining this lemma, but also investigating how further formalisms could be treated within the same framework.

2 Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \dots\}$. A multiset A over (a normal set) S is defined by a multiplicity function $S \rightarrow \text{nat}$, for $a \in S$ let $|A|_a$ be the result of this multiplicity function (intuitively the number of occurrences of a in A). For multisets A, B and

any a we define the multiset operations by letting $|A \cup B|_a = \max(|A|_a, |B|_a)$, $|A \cap B|_a = \min(|A|_a, |B|_a)$, $|A \setminus B|_a = \max(0, |A|_a - |B|_a)$, $|A \uplus B|_a = |A|_a + |B|_a$, and $a \in A$ iff $|A|_a > 0$. Applying an operator to a multiset and a normal set S together we treat S as the multiset S' over S with $|S'|_a = 1$ for $a \in S$. For normal sets S and T let $S \uplus T$ denote the union where we assume, w.l.o.g. that S and T are (or can obviously be made, by e.g. renaming) disjoint.

Let S^* denote the Kleene closure, the set of all finite sequences over the set S , including the empty sequence, denoted ε . A ranked symbol α has $\text{rank}(\alpha) \in \mathbb{N}$ defined, we may write $\alpha^{(k)}$ to denote α with $\text{rank}(\alpha) = k$. A (ranked) alphabet Σ is a finite set of (ranked) symbols. Alphabets and ranks will often be implicit in their use, e.g. the existence of a ternary edge labeled a implies $a^{(3)} \in \Sigma$.

We will often use a function f interchangeably with the characterizing set $\{(x, y) \mid y = f(x) \text{ for some } x\}$. We let a function $f : S \rightarrow T$ generalize to powersets and sequences (e.g. writing $f(s_1, \dots, s_n)$ to mean $f(s_1) \cdots f(s_n)$, for $s_1, \dots, s_n \in S$) when there is no room for confusion.

A *hypergraph* g is a pair (\dot{g}, \bar{g}) where \dot{g} is a finite set of nodes and \bar{g} a multiset of edges, each edge $(\alpha, v_1 \cdots v_n) \in \bar{g}$ consisting of a labeling ranked symbol $\alpha^{(n)}$ and the sequence of attached nodes $v_1, \dots, v_n \in \dot{g}^*$ for $n \in \mathbb{N}$. We will denote such an edge $\alpha(v_1, \dots, v_n)$, further let $\text{lab}(\alpha(v_1, \dots, v_n)) = \alpha$ and $|\alpha(v_1, \dots, v_n)| = n$.

From here we refer to hypergraphs as just *graphs*. For a graph h we, without specifying the tuple, let \dot{h} and \bar{h} denote the nodes and edges. In examples we will usually draw edge labels from the lowercase Latin letters a, b, c, \dots (once introduced we will use uppercase for the *nonterminal* edge labels), using u, v, \dots specifically for *unary* edge labels, and the nodes from the integers $1, 2, 3, \dots$. The edge $a(1, 2)$ is considered a *directed edge* from 1 to 2 (a graph with only directed edges is a *digraph*). We will often specify a graph simply as a list of edges (bracketed if needed) and (when unconnected) nodes, e.g. the directed triangle with two parallel a -labeled edges and two unconnected nodes: $[a(1, 2), a(1, 2), b(2, 3), c(3, 1), 4, 5]$. Where the contextual hyperedge replacement grammars in literature [DH15] label nodes separately from edges we opt to encode this in terms of unary edge labels (see Remark 2).

Let G_Σ denote the set of all graphs labeled by the ranked alphabet Σ . Let $|g| = |\dot{g}| + |\bar{g}|$. For a graph g and $e \in \bar{g}$ we let $g - e = (\dot{g}, \bar{g} \setminus \{e\})$ (i.e. removing *one occurrence* of e), for g and $v \subseteq \dot{g}$ let $g - v$ be g with all nodes in v , and all edges incident with those nodes, removed. For any graphs g and h let $g \uplus h = (\dot{g} \uplus \dot{h}, \bar{g} \uplus \bar{h})$, that is, the graph consisting of the two unconnected subgraphs g and h (renaming some nodes if needed).

Remark 1. This definition of graphs differs from e.g. [Hab92] in three key ways.

Firstly hyperedges do not have outgoing and incoming attachments. This adjustment is now standard, see e.g. the survey [DKH97], as differentiating between attachments adds nothing of structural importance to the languages. That is, the information can be encoded in the labeling symbol instead.

Secondly the common definitions tend to differentiate some nodes as “external” to facilitate graph compositions. These are mostly a technicality and are not usually desirable in the languages generated. As such we here stick to a

more traditional mathematical graph definition, and represent the points which compositions act upon using specially labeled edges.

Thirdly, usually edges are defined as a set of opaque objects, mapped to labels and attached nodes by functions. Here we instead opt to use multisets of edges determined by those properties. This approach is used to avoid edge renaming introducing additional symmetries, harmonizing with some software systems.

These changes are intended to make the representation simpler, but all results can be restated within Habel's notation with a minimum of effort. \diamond

3 Grammars and Graph Operations

In this section we define the core of a common framework for the two formalisms of immediate interest here, contextual and normal hyperedge replacement grammars. This effort pays off in some ways within this paper, enabling us to provide novel pumping lemmas for both while limiting duplicated effort. The intention, however, is to in the future extend this work to treat additional formalisms. A notable example is the regular tree folding formalism of [Bjö18], which captures many interesting graph languages by way of an iterative node merging technique, and is very amenable to being stated in the style here proposed.

Definition 1. For any graph g and $R \subseteq \dot{g} \times \dot{g}$ and some fixed global order on the nodes¹ let $g[R]$ be (a graph isomorphic to) g with the nodes related by R transitively collapsed into the smallest node. That is, for all $i \in \dot{g}$, let S_i be the smallest set such that $i \in S_i$ and if $(j, k) \in R$ then $S_j = S_k$, then taking $\sigma(i) = \min S_i$ we have $g[R] = (\sigma(g), \{(\alpha, \sigma(s)) \mid (\alpha, \sigma) \in \bar{g}\})$.

That is, $g[R]$ is produced by collapsing all nodes which are related by R , transitively, into a single node.

Definition 2. For any symbol x of rank k an x -handle in a graph g is a unique edge $x(v_1, \dots, v_k) \in \bar{g}$. That is, $(\sum_{v_1, \dots, v_k \in \dot{g}} |\bar{g}|_{x(v_1, \dots, v_k)}) = 1$.

For any graphs g and h and symbols $x^{(k)}$ and $y^{(k)}$, such that g has an x -handle $e_x = x(v_1, \dots, v_k)$, and h has a y -handle $e_y = y(v'_1, \dots, v'_k)$ define

$$g \otimes_y h \equiv ((g - e_x) \uplus (h - e_y))[\{(v_1, v'_1), \dots, (v_n, v'_n)\}].$$

Let \otimes_y be undefined for graphs lacking the handles indicated by the subscripts.

Further, let $\mathbf{0}_y$ be the zero graph $[x(1, \dots, k), y(1, \dots, k)]$, then for any graph g which is x - and y -handled, define the \otimes_y -closure as the smallest set such that

$$g^{\otimes_y} = \{\mathbf{0}_y\} \cup \{g \otimes_y g' \mid g' \in g^{\otimes_y}\}.$$

¹ For the purposes of graph languages the specific order is irrelevant, as different orders only produce different isomorphic graphs, and it is thus for those definitions left unspecified (a general order for all nodes can always be fixed by choosing any Turing machine which recursively and uniquely enumerates all graphs, choosing its output order as the ordering, but such a specific order is not necessary for this paper). In a later construction where the exact graphs matter (to keep the same relation defined on a constructed graph) we will state explicit internal orders.

More formally $x \otimes_y$ is the operator in a graph combining monoid, concatenating graphs at sites indicated by the handles. The zero graph is then the identity, and the closure is analogous to the Kleene closure, $g^{x \otimes_y}$ being the set of all graphs $g \otimes_y \dots \otimes_y g \otimes_y \mathbf{0}_y$ for any number of repetitions of g , including zero.

A common case for compositions is that the handles connect to each node at most once (i.e. if both sides have this property no nodes are merged). We introduce additional notation for this more specific case.

Definition 3. We may write $g_1 {}^*_{x \otimes_y} g_2$ (or equivalently $g_2 {}^*_y \otimes_x g_1$) to denote $g_1 \otimes_y g_2$ when the x -handle in g_1 attaches to each node at most once. Let $g_1 {}^*_{x \otimes_y} g_2$ denote when both handles have this property.

This is a shorthand for stating the property on the graphs involved, in that $g {}^*_{x \otimes_y} h = g {}^*_{x \otimes_y} h = g \otimes_y h$ if g and h fulfill the property. The ${}^*_{x \otimes_y}$ case recurs in grammar definitions and pumping lemmas, where the shorthand aids brevity.

Definition 4. A graph replacement rule x over Σ consists of two handles and two graphs such that $\text{lhs}(x) {}_{x \otimes_{\hat{x}}} \text{rhs}(x) \in G_\Sigma$, where

- x and \hat{x} are the rule handle and rule cohandle, identified by globally unique² ranked symbols, and,
- $\text{lhs}(x)$ and $\text{rhs}(x)$ are the left-hand and right-hand side graphs, which are x - and \hat{x} -handled, respectively.

We write the rule as $\text{lhs}(x) {}_{x \otimes_{\hat{x}}} \text{rhs}(x)$. We will often refer to such a rule simply by its unique handle x .

For any graph g the rule $l {}_{x \otimes_{\hat{x}}} r$ can be applied to produce h if there exists a graph g' such that $g = l {}^*_{x \otimes_{\hat{x}}} g'$ and $h = g' {}_{x \otimes_{\hat{x}}} r$. For a (set of) rule(s) e we write $g \rightarrow_e h$ to denote that h can be produced applying (some rule from) e to g .

Definition 5. A graph grammar G is a tuple $G = (N, \Sigma, R, S)$ where: (i) N is a ranked alphabet of nonterminal labels; (ii) Σ is a ranked alphabet of terminal labels; disjoint from N ; (iii) R is a finite set of replacement rules over $N \cup \Sigma$; and; (iv) $S \in N$ is the initial nonterminal.

The language accepted by G is denoted $\mathcal{L}(G) \subseteq G_\Sigma$ and contains g if and only if there exists a graph g' , isomorphic to g , which can be produced by a sequence of rule applications $S() \rightarrow_R \dots \rightarrow_R g'$.

Definition 6. A graph grammar $G = (N, \Sigma, R, S)$ is a hyperedge replacement (HR) grammar if all rules in R are of the form $l {}^*_{x \otimes_{\hat{x}}} r$ where the left hand side $l = A(1, \dots, k), x(1, \dots, k)$ for some $A^{(k)} \in N$.

Definition 7. A graph grammar $G = (N, \Sigma, R, S)$ is a contextual hyperedge replacement (CHR) grammar if there is a set $\text{cxlabs}(G) \subseteq \{\alpha \in \Sigma \mid \text{rank}(\alpha) = 1\}$ (induced by the conditions on the rules below), the contextual labels, such that each rule $l {}^*_{x \otimes_{\hat{x}}} r \in R$ has

² We leave this somewhat informal, the purpose is to conflate a rule and its handle and not have to explicitly rule out symbols in other alphabets making it *not* be a possible handle in some graph. Notably $x, \hat{x} \notin \Sigma$.

1. $l = [x(1, \dots, n+m), \alpha_1(1), \dots, \alpha_n(n), A(n+1, \dots, n+m)]$ for some $n, m \in \mathbb{N}$, $A^{(m)} \in N$, and $\alpha_1^{(1)}, \dots, \alpha_n^{(1)} \in \text{cxlabeled}(G)$, where $\alpha_1(1), \dots, \alpha_n(n)$ are called the contextual edges (the nodes, $1, \dots, n$, are contextual nodes); and;
2. $\hat{x}(1, \dots, n+m), \alpha_1(1), \dots, \alpha_n(n) \in \bar{r}$, with the contextual label restriction: only those n instances of an edge $\beta(i) \in \bar{r}$ with $1 \leq i \leq n+m$ and $\beta \in \text{cxlabeled}(G)$ exist (for other i there is no restriction).

More informally, the lhs l identifies any number of contextual nodes, which are nodes to which a terminal unary edge is attached, and one nonterminal edge. Each node in l is attached to by precisely one edge. The rhs has the same contextual edges, i.e. nodes labeled by contextual labels attached to the cohandle such that applying the rule will always preserve contextual edges. To illustrate the structure of a grammar, consider the example in Figure 1, which generates

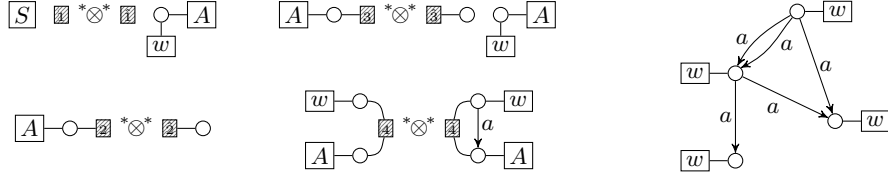


Fig. 1. On the left are the four rules of a CHR grammar generating the set of all not necessarily connected directed acyclic graphs, labeling all nodes w (i.e. each node has a single incident unary edge labeled $w^{(1)}$), and the directed edges $a^{(2)}$. The handles are shown in shaded boxes (and elided from the operator). Rules [1] and [2] start and end the derivation, respectively, while [3] creates a new node (attached to the A nonterminal), and [4] adds an edge from some w -labeled node (i.e. any node) to the node attached to the nonterminal A . The example graph on the right can be produced by this sequence of rule applications: [1], [2], [3], [4], [3], [4], [4], [3], [4], [2]. Intuitively the nonterminal A identifies the unique current “lowest” node, allowing edges to it from any other (due to the injective condition, see Remark 3) to it.

the language of all directed acyclic digraphs. This example is further expanded in Figure 4 on page 13.

The contextual label restriction only prevents adding labels from $\text{cxlabeled}(G)$ to “already existing” nodes, i.e. nodes connected to the cohandle (nodes 1 through $n+m$ as listed above). Its motivation requires some further discussion.

Remark 2. The contextual label restriction of Definition 7 enforces that if a unary label is used to identify a contextual node in some rule then no rule may add that label to an *existing* node. That is, these labels must be created together with the node, simulating *node labels*, mimicking the definition in [DH15].

Note that $\text{cxlabeled}(G) = \emptyset$ is possible if and only if G is a HR grammar (as it is equivalent to no rule having a contextual node), so the CHR languages form a strict superset of the HR languages as defined in the literature (e.g. [Hab92]), with a small caveat in Remark 3. In Theorem 2 we will demonstrate that this

restriction is necessary, that is, its inclusion changes the class of languages which can be accepted by CHR grammars.

To keep things simple we will, where not otherwise explicitly stated, assume that every (non-HR) CHR has $\text{cxlabs}(G) = \{\alpha \in \Sigma \mid \text{rank}(\alpha) = 1\}$. In other words, we monopolize the unary edges for the purpose of denoting node labels, to avoid having the reader confuse what is a node label and what is not. There are some graph languages which cannot be captured in this way, specifically precisely those where there is no bound on the number of unary edges attached to a single node (e.g. $L = \{\varepsilon, [a(1)], [a(1), a(1)], [a(1), a(1), a(1)], \dots\}$) but this difference has no impact on any example or proof made here. Such languages can be encoded in other ways, e.g. adding a “dummy” node to have a binary edge act as a unary one, making the difference an unimportant technical detail. \diamond

Remark 3. Note further that the $g = l \otimes_x^* g'$ lhs anchoring of Definition 4 requires that l is *injectively* embedded in g (by the little asterisk notation). This agrees with CHR as defined in e.g. [DH15], where a replacement must identify distinct nodes, but this is traditionally not required in HR. Requiring it in general harmonizes the definitions without causing any loss of generality however, as the only circumstance where a non-injective embedding of the lhs may be necessary in a HR grammar is when a nonterminal attaching to the same node more than once is generated by some rhs. A simple rewriting of the grammar can be applied to remove such cases, by e.g. creating a new nonterminal to represent that case while only attaching to the node once.

More importantly: all the pumping results of Section 5 are formulated such that they hold whether this injective requirement is present or not. \diamond

4 Formalism Motivations

In this section we motivate some of the choices in the formalisms. Notably the effort to harmonize them causes some small differences from the literature which may require some explanation.

Remark 4. Let us refer to the grammars defined in [DH15] as *DHM-CHR grammars* (using the author initials from the formative [DHM11], though the very original paper is by “HM” [HM10]). Setting the cosmetic difference of node labels aside (i.e. monopolizing unary edges to represent labels here) we can define DHM-CHR by modifying Definition 7 to have $l = [x(1, \dots, n+m), \alpha_1(1), \dots, \alpha_n(n+m), A(n+1, \dots, n+m)]$ in item 1, and require all nodes on both lhs and rhs to have a label. We adopt CHR to harmonize better with HR grammars.

This difference does not cause a difference in the languages which may be represented, as the labels of the attached nodes in DHM-CHR can be encoded in the nonterminal symbol in CHR. Such a construction may cause an exponential blowup in the number of nonterminals, but the number of rules remains the same, as only the labelings which occur as literal left hand sides among the rules are useful. As usual no grammar can have more useful nonterminals than it has rules (i.e. there is no need to create a nonterminal encoding for all sets of node

labels, if those sets do not exist as a left-hand side of some rule) the grammar only grows linearly. However, in the other direction, there are some languages where HR (and CHR) is exponentially more succinct than DHM-CHR. \diamond

Theorem 1. *There exists a family of graph languages L_1, L_2, \dots such that the smallest HR (and CHR) grammar accepting L_k is of size $\mathcal{O}(k)$, while the smallest DHM-CHR grammar accepting L_k is of size $\Omega(2^k)$.*

Proof. We demonstrate this by, for any $k \in \mathbb{N}$, constructing a language L_k fulfilling these conditions. For any n let $g_{k,n}$ be the $n \times k$ grid graph, i.e. there are kn nodes on a grid, identifying the nodes with their coordinates node (i, j) has outgoing edges; $x((i, j), (i+1, j))$ when $i < n$; and; $y((i, j), (i, j+1))$ when $j < k$. Define $f_{\text{lab}} : G_\Sigma \rightarrow 2^{G_\Sigma}$ by letting $f_{\text{lab}}(g)$ contain every graph which can be produced by labeling the nodes in g with either u or v (i.e. $f_{\text{lab}}([x(1, 2), y(1, 3)]) = \{[x(1, 2), y(1, 3), u(1), u(2), u(3)], [x(1, 2), y(1, 3), u(1), u(2), v(3)], \dots\}$).

Then taking $L_k = \bigcup_{n \in \mathbb{N}} f_{\text{lab}}(g_{k,n})$ fulfills the conditions of the theorem, containing graphs of the form shown in Figure 2. This language can, for any

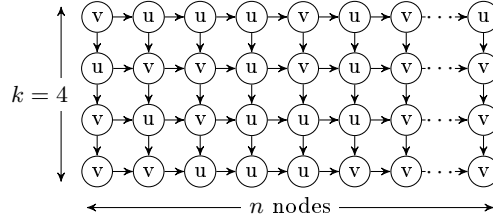


Fig. 2. Example graph from the language L_4 as described in Theorem 1, with the x and y labels on the horizontal and vertical binary edges elided. The u and v labels are in actuality unary edges attached to the nodes, here shown as node labels.

k , be captured by a HR grammar with no more than six rules (the main rule generating one column of the grid in each step, attaching to each new node a nonterminal which nondeterministically generates either a u - or v -labeled unary edge). This graph language is lifted from Chapter 5 of [Hab92] (with the addition of the arbitrary node labels), where it is by a pumping argument demonstrated that a HR grammar accepting this language *must* repeatedly (as n grows) use a rule replacing a nonterminal which is connected to at least k nodes (i.e. the column-wise generation is *necessary* for arbitrary n). A DHM-CHR grammar G capturing L_k cannot usefully use the ability to attach edges to contextual nodes (once three or more nodes are generated there is no way to correctly control where the edge goes), so it too must generate the graph by repeatedly replacing a nonterminal connected to at least k nodes. However, as the nodes are arbitrarily labeled a simple adversarial argument (assigning u and v labels to sentential forms) dictates that G must have $\Omega(2^k)$ rules to account for the possible labels on the left hand sides. \square

Next we demonstrate that the contextual label restriction in CHR is necessary to generate a class of languages comparable to those captured by DHM-CHR.

Theorem 2. Let CHR^+ be grammars defined by modifying Definition 7 to remove the contextual label restriction. There then exists a CHR^+ grammar G such that there exists no CHR grammar G' that has $\mathcal{L}(G') = \mathcal{L}(G)$.

Proof. We proceed by constructing such a G , demonstrating that no G' accepting $\mathcal{L}(G)$ exists due to the necessary structure of the derivation. Define G with the rules: (i) $[S(), x()] \xrightarrow{*}_{\hat{x}} [\hat{x}(), A(1)]$; (ii) $[A(1), y(1)] \xrightarrow{*}_{\hat{y}} [\hat{y}(1), A(1), u(2), a(1, 2)]$; (iii) $[A(1), u(2), u(3), z(1, 2, 3)] \xrightarrow{*}_{\hat{z}} [\hat{z}(1, 2, 3), A(1), u(2), u(3), a(2, 3)]$; and; (iv) $[A(1), \tau(1)] \xrightarrow{*}_{\hat{\tau}} [\hat{\tau}(1), u(1)]$. That is, $\mathcal{L}(G)$ contains the graphs illustrated in Figure 3, consisting of a distinguished (“node 1”) node with a single outgoing edge to *each other node*, and those other nodes form an arbitrary digraph. All nodes are labeled u and all (binary) edges labeled a .

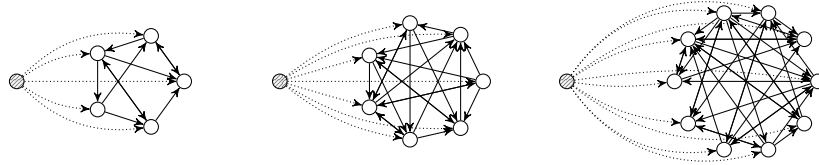


Fig. 3. Three examples of graphs from the language $\mathcal{L}(G)$ from the proof of Theorem 2 (opting not to draw labels, self-loops, or multiple instances of the same edge). The nodes are all labeled u , the edges a . The marked node is “node 1”, the first node created by the application of the rule $[S(), x()] \xrightarrow{*}_{\hat{x}} [\hat{x}(), A(1)]$.

No CHR G' accepting $\mathcal{L}(G)$ exists, since the “arbitrary digraph” part cannot be created without repeatedly applying contextual rules (see e.g. the Connectivity Theorem of [Hab92], which precludes arbitrarily densely connected graphs). However, it follows that “node 1” cannot be created before this arbitrary number of contextual edge additions, as it too is labeled u , so there is no way to avoid the derivation being able to add incoming edges to “node 1”, which brings us outside the language. On the other hand, “node 1” also cannot be created last, as the edges to the unbounded number of other nodes would then need to be added, meaning no lhs is sufficiently large to systematically identify them all for any sufficiently large graph. Adding those edges by contextual rules again risks failing to create some of the needed edges and creating multiples of others. \square

5 Derivation Structures and Pumping Lemmas

One of the main goals of the framework developed in the previous two sections is to be able to discuss structural properties of various formalisms (HR and CHR being considered here, but others to be added in the future). This kind of comparison is important, as the formalisms are powerful enough that it is often not intuitively obvious whether a certain language can be captured within them.

Specifically we will here consider Ogden-style lemmas, for example (adapting the original lemma [Ogd68] for context-free to regular languages): for every

regular string language L there exists a constant k such that for every string $w \in L$ with at least k string positions *marked* there exists strings v_1, v_2, v_3 such that: (i) $w = v_1 v_2 v_3$; (ii) $v_1 (v_2)^i v_3 \in L$ for all $i \in \mathbb{N}$ (i.e. v_2 can be repeated any number of times while remaining in the language); (iii) v_2 contains at least one of the marked positions; and (iv) $v_2 v_3$ contains at most k of the marked positions. However, where marking string positions is clear enough to be talked about quire informally more careful notation is necessary in the graph case (i.e. resolving symmetries due to automorphisms), and to make the results usefully comparable (e.g. yielding some insight on the power added by contextual rules) the notation should be kept as consistent as possible between formalisms.

Pumping and Habel-Kreowski's lemma. First let us define the idea of “pumping” independent of the formalism.

Definition 8. For any $L \subseteq G_\Sigma$ and $g \in \mathcal{L}$ we say that $g = g_1 \otimes_u g_2 \otimes_v g_3$, for any graphs g_1, g_2 , and g_3 , and any $x^{(k)}, y^{(k)} \notin \Sigma$, is a pumpable decomposition of g in L if and only if

$$g_1 \otimes_u^* (g_2 \otimes_v^*) \otimes_y^* g_3 \subseteq L.$$

Let us recall Habel-Kreowski's pumping lemma [HK87, Hab92] phrased using the notation of this paper (making it differ slightly in presentation as discussed in Remark 1, but adapting the proof is a simple matter of changing external nodes into handles, replacing the graph embedding with our operator).

Theorem 3 (Habel-Kreowski's lemma). For every HR language L there exists constants k and n (depending only on L) such that every $g \in L$ with $|g| \geq n$ has a pumpable decomposition $g = g_1 \otimes_u^* g_2 \otimes_v^* g_3$ with

1. $|g_2 \otimes_v^* g_3| \leq n$, and
2. $|g_2| > |{}_u \mathbf{0}_v|$ (i.e. $|g_2| > k + 2$, it creates at least one node or edge).

Marking graphs and derivation trees. While Habel-Kreowski's lemma is very powerful it can, using largely existing literature, be extended into an Ogden-style lemma, and with some care this can be extended into CHR. Let us first define marking graphs and recall an existing lemma.

Definition 9. For an alphabet Σ define the marked alphabet $\text{mark}(\Sigma) = \Sigma \uplus \{\text{mark}(\alpha)^{(k)} \mid \alpha^{(k)} \in \Sigma\} \uplus \{\text{mark}(\perp)^{(1)}\}$. An edge labeled $\text{mark}(\alpha)$ for $\alpha \in \Sigma$, or a node with exactly one incident edge labeled $\text{mark}(\perp)$, is said to be marked. Let $\text{unmark}(g)$ be the graph resulting when removing all marks from g , i.e. changing each label $\text{mark}(\alpha)$ into α , and deleting the $\text{mark}(\perp)^{(1)}$ edge from marked nodes (we leave unmark undefined if some node has more than one incident edge marked $\text{mark}(\perp)^{(1)}$). Let $\text{mark}(L) = \{g \in G_{\text{mark}(\Sigma)} \mid \text{unmark}(g) \in L\}$ for any graph language L over Σ .

Marks not being defined for a node being marked multiple times is a small technicality that simplifies the statement of pumping lemmas going forward, as

it makes it easy to avoid cases where the pumping would otherwise just add marking edges, as opposed to “real” edges. Next we refine the idea of derivations into trees (for HR only).

Let us define a weak form of derivation trees, which will structurally correspond to *tree decompositions* for HR graph languages, as defined in [Lau90]. These will only usefully apply to the HR grammars here, and to simplify them we assume a normal form for the grammars.

Definition 10. A simple HR grammar $G = (N, \Sigma, R, S)$ is such that for all rules $l_{x \otimes_{\hat{x}}} r \in R$

1. each $A \in N$ occurs at most once in r , and,
2. r either contains at most one node not connected to the cohandle, or at most one Σ -labeled edge, i.e. $(|\bar{r}| - \text{rank}(x)) + |\{\alpha(v_1, \dots, v_k) \in \bar{r} \mid \alpha^{(k)} \in \Sigma\}| \leq 1$.
We say that this is the node or edge created by x , denoted $\text{created}(x) \in \bar{r} \cup \bar{r}$ (which is not defined for all x as r is permitted to not create a node or edge).

A CHR grammar G is simple if it fulfills condition 1 alone.

We may assume simple (C)HR grammars w.l.o.g. as any (C)HR grammar can be made simple (with only a constant blowup in the overall size of the grammar) using the usual rewriting techniques: creating additional nonterminals under unique names as needed, and, for HR, splitting rules with multiple nodes and/or edges on the rhs into a chain of rules (enforced by new unique nonterminals).

Definition 11. For a simple HR grammar $G = (N, \Sigma, R, S)$ a derivation tree t , for a graph g , is a tree (a digraph where all but one node, the root, has precisely one incoming edge) over $\{r^{(2)} \mid r \in R\}$ (that is, the directed edges are labeled by rules from R), such that $g = \Psi(t)$, where Ψ is a tree-to-graph mapping defined as follows.

For every internal node i with incident edges $x(p, i), x_1(i, j_1), \dots, x_n(i, j_n)$ let $\Psi_t(i) = r'_{x_1 \otimes_{\hat{x}_1}} \Psi_t(i_1) \cdots_{x_n \otimes_{\hat{x}_n}} \Psi_t(i_n)$ where

1. r' is such that $\text{rhs}(x) = r'_{x_1 \otimes_{\hat{x}_1}} \text{lhs}(x_1) \cdots_{x_n \otimes_{\hat{x}_n}} \text{lhs}(x_n)$, which is unique as each nonterminal occurs at most once in $\text{rhs}(x)$ (as G is simple), and,
2. r' contains no nonterminals. A tree where this does not hold is called a derivation tree fragment, Ψ is defined the same way for such fragments.

The root node p has precisely one incident edge, let it be $x(p, j)$, and let $\Psi(t)$ denote $\Psi_t(j) - \hat{x}$. We call $\Psi(t)$ the graph derived from A by t , where $A \in N$ is the nonterminal in $\text{lhs}(x)$. Let $\text{deriv}(G)$ denote the set of all derivation trees which derive a graph from the initial nonterminal S .

Finally, we extend Ψ_t to marked derivation trees by treating the label $\text{mark}(x)$ as x if $\text{created}(x)$ is undefined, and by replacing $\text{created}(x)$ with $\text{mark}(\text{created}(x))$ in the rhs of x otherwise (for example, if t' is t with all edges marked then $\Psi(t')$ is the graph $\Psi(t)$ with all nodes and edges marked).

Lemma 1. For all HR grammars G we have $\mathcal{L}(G) = \{\Psi(t) \mid t \in \text{deriv}(G)\}$.

Proof. This can largely be taken as a consequence of the definitions with a reference to [Hab92] for derivation trees, and [Lau90] for a deeper look at tree decompositions of HR languages. As we will operate on derivation trees in later proofs as well however let us spell out the details.

Let $G = (N, \Sigma, R, S)$ be a simple HR grammar, and take any derivation sequence $g_1 \rightarrow_{r_1} \dots \rightarrow_{r_n} g_n$ for G , such that $g_1 = S()$ and $g_n \in \mathcal{L}(G)$. Let $\sigma : \{1, \dots, n\} \rightarrow \{0, 1, \dots, n\}$ be a mapping such that the derivation step \rightarrow_{r_i} replaces a nonterminal added by derivation step $\rightarrow_{r_{\sigma(i)}}$ (symmetries in the sentential graphs means that σ is not necessarily unique, but any σ consistent with the derivation can be chosen), with $\sigma(1) = 0$. More strictly stated σ is such that marking some nonterminal on the rhs in the $\rightarrow_{r_{\sigma(i)}}$ and replacing the marked nonterminal in \rightarrow_{r_i} does not otherwise change the derivation.

Then construct the tree t with nodes $0, 1, \dots, n$ and the edges $\{r_i(i, \sigma(i)) \mid 1 \leq i \leq n\}$. Now $t \in \text{deriv}(G)$ and $\Psi(t) = g_n$, which can be seen by induction on the length of the derivation: for each $1 \leq i \leq n$ the derivation tree fragment t' induced by deleting the nodes $\{i+1, \dots, n\}$ and all their incident edges has $\Psi(t') = g_i$. \square

Ogden's lemma for derivation trees and graph grammars. We now have all the pieces necessarily to easily pluck the Ogden-style lemma from [Kuh08] and reapply it to give us a similar lemma for HR grammars.

Theorem 4 (Ogden's for Regular Tree Languages (by Kuhlmann)). *For every HR grammar G there exists a constant p such that every $t \in \text{mark}(\text{deriv}(G))$ (by virtue of being a regular tree language [Lau90]) with at least p marked edges has a pumpable decomposition $t = t_1 \overset{*}{x \otimes_u} t_2 \overset{*}{v \otimes_y} t_3$ such that*

1. x, u, v and y are all rank 1,
2. at least one edge in t_2 is marked,
3. $t_2 \overset{*}{v \otimes_y} t_3$ contains at most p marked edges.

Note that the marked decomposition in $\text{mark}(\text{deriv}(G))$ corresponds to an unmarked pumpable decomposition in $\text{deriv}(G)$.

With this we are ready to state an Ogden-style pumping lemma for HR.

Theorem 5. *For every HR language L there exists a constant $p \in \mathbb{N}$ such that every $g \in \text{mark}(L)$ which has at least p marked nodes and edges has a pumpable decomposition $g = g_1 \overset{*}{x \otimes_u} g_2 \overset{*}{v \otimes_y} g_3$ where*

1. x, u, v , and y have rank at most p ,
2. g_2 has at least one edge or node marked, and,
3. $g_2 \overset{*}{v \otimes_y} g_3$ contains at most p marked nodes and edges.

Proof. Note that if there is an edge labeled $\text{mark}(\perp)$ in g_2 then the node it is attached to is “created” in g_2 , in that it is not attached to both the u - and v -handle. This must be the case as otherwise pumping g_2 would add multiple $\text{mark}(\perp)$ labels to the same node, which Definition 9 disallows. In consequence

pumping g_2 does add additional nodes and/or edges to the resulting graph, even viewed under unmark.

Let $G = (N, \Sigma, R, S)$ be a simple HR grammar accepting L , which exists by definition. Let p_T be the pumping constant for $\text{deriv}(G)$ indicated by Theorem 4, and set $p = \max(p_T, \max_{r \in R} \text{rank}(r))$.

Then there exists some $t \in \text{mark}(\text{deriv}(G))$ such that $\Psi(t) = g$, and t will have at least p marked edges. This is so since there by definition is a tree $t' \in \text{deriv}(G)$ with $\text{unmark}(g) = \Psi(t')$, and then t with $\text{unmark}(t) = t'$ exists. Since G is simple each edge in t is responsible for creating at most one node or edge in the final graph, so each edge also introduces at most one mark, which means generating the p marks in g requires at least p (which is at least as large as p_T) marked edges in t . So, simply apply Theorem 4 to t to get the pumpable decomposition $t_1 \overset{*}{x} \overset{*}{u} t_2 \overset{*}{v} \overset{*}{y} t_3$, and all that remains is to show that the pumpable decomposition $g = g_1 \overset{*}{x} \overset{*}{u} g_2 \overset{*}{v} \overset{*}{y} g_3$ this induces has the properties indicated by the lemma. This is the case, as $\Psi(t_1)$ and $\Psi(t_1 \overset{*}{x} \overset{*}{u} t_2)$ (note that these are derivation tree fragments) will contain precisely one nonterminal for some rank $k \leq p$, and as t_2 contains at least one mark, and $t_2 \overset{*}{v} \overset{*}{y} t_3$ at most p marks, so will g_2 , and $g_2 \overset{*}{v} \overset{*}{y} g_3$, respectively. \square

The statement of this kind of pumping lemma would often use a separate constant bounding the rank of the decomposition (i.e. in part 1 of the statement above), as it is not directly related to the pumping constant p itself. The practical usefulness of this separation is limited however, and we opt to assume a single constant which is sufficiently large (taking the maximum of the two constants).

Extending pumping to CHR. While the pumping lemma for HR is in itself useful, our actual goal is an analogous lemma for CHR. We will achieve this by characterizing all CHR languages in terms of HR grammars under an operator responsible for introducing contextual edges. While we will need to augment this approach further, let us launch straight to it.

Definition 12. For a simple CHR grammar $G = (N, \Sigma, R, S)$ construct the HR grammar $G_\tau = (N, \Sigma \uplus \{\perp_\alpha^{(1)} \mid \alpha \in \text{cxlabs}(G)\}, R_\tau, S)$ by for each rule $r \in R$ having the rule $r_\tau \in R'$, where

- assuming $\text{lhs}(r)$ is of the form (renaming nodes if necessary) $[r(1, \dots, n+m), A(1, \dots, n), \alpha_1(n+1), \dots, \alpha_m(n+m)]$ construct $\text{lhs}(r_\tau)$ to be the graph $[r_\tau(1, \dots, n), A(1, \dots, n)]$, and,
- construct $\text{rhs}(r_\tau)$ by (renaming nodes if necessary) taking $\text{rhs}(r)$ and constructing $\text{rhs}(r_\tau)$ by replacing the edges $\hat{r}(1, \dots, n+m), \alpha_1(n+1), \dots, \alpha_m(n+m)$ with the edges $\hat{r}_\tau(1, \dots, n), \perp_{\alpha_1}(n+1), \dots, \perp_{\alpha_m}(n+m)$.

Clearly G_τ can, when needed, be made simple by applying the same argument as made in connection with Definition 10.

Let $P_G = \{[x_\alpha(1, 2), \alpha(1), \perp_\alpha(2)] \overset{x_\alpha}{x_\alpha} \overset{\hat{x}_\alpha}{\hat{x}_\alpha} [\hat{x}_\alpha(1, 1), \alpha(1)] \mid \alpha \in \text{cxlabs}(G)\}$ (note that these rules are neither HR nor CHR, as the rhs handle connects to the same node twice, causing a merging behavior) and define the operator $\tau_G : G_{\Sigma'} \rightarrow 2^{G_\Sigma}$ by having $g' \in \tau_G(g)$ if and only if $g \rightarrow_{P_G} \dots \rightarrow_{P_G} g'$, with $g' \in G_\Sigma$.

That is, where G would attach some edges to a contextual node $\alpha(i)$ in G_τ a “placeholder” node $\perp_\alpha(j)$ is instead created. τ_G then merges such placeholders into candidate contextual nodes, e.g. $\tau_G([u(1), a(1, 2), a(2, 3), \perp_u(3)]) = \{[u(1), a(1, 2), a(2, 1)]\}$. See Figure 4 for an example of the relationship between the derivation tree, the graph it produces, and a graph that results under τ_G .

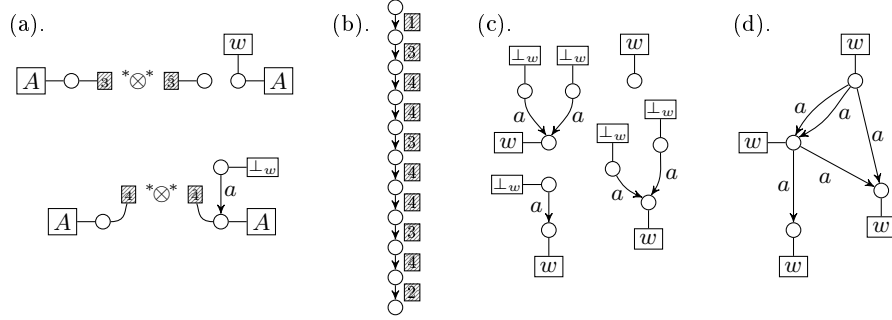


Fig. 4. (a) Shows the two important rules (rule \boxtimes is the only one actually changed) of the HR grammar resulting when applying Definition 12 to the CHR grammar of Figure 1. (b) A derivation tree t , in this case unary, in the indicated HR grammar. (c) The graph $\Psi(t)$ produced by the derivation. (d) One of the graphs in $\tau_G(\Psi(t))$ (i.e. it is produced by merging \perp_w -labeled nodes into w -labeled ones), in fact the same graph that was the example in Figure 1, demonstrating the relationship established. Note that marking one edge in (b) produces a marked node (if a \boxtimes -labeled edge) or edge (if a \boxtimes -labeled edge) in (c), and such a mark in turn turns into a mark in (d) under τ_G .

We will go on to argue that for every CHR grammar G we will have $g \in \mathcal{L}(G)$ only if there exists some $g' \in \mathcal{L}(G_\tau)$ with $g \in \tau_G(g')$, simulating contextual nodes by instead creating \perp -labeled placeholder nodes, which the τ_G operator merges into a node with the correct label. The reverse does not hold however, as not every way of merging the nodes is valid.

Where Figure 4 also indirectly illustrates a case where the combination of G_τ and τ_G produces a loosening of the language (i.e. the graph in (d) is only one of many possible), another fact from [DH15] supports the reasoning behind the construction.

Definition 13 (From [DH15]). A CHR grammar $G = (N, \Sigma, R, S)$ is context-safe if all graphs $g \in G_{\Sigma \sqcup N}$ which can be constructed by a sequence of rule applications $S() \rightarrow_R \dots \rightarrow_R g$ necessarily have a sequence of rule applications $g \rightarrow_R \dots \rightarrow_R g'$ such that $g' \in G_\Sigma$.

That is, G is context-safe if a derivation in G cannot get “stuck” due to missing contextual nodes (i.e. for each rule x in G if the *nonterminal* in $\text{lhs}(x)$ exists in a sentential graph then the *contextual nodes* in the lhs necessarily also exist).

Remark 5. In [DH15] it is shown that *every* CHR grammar G can be put on a normal form with this property, and on such grammars we consequently have

the property that all $g \in \mathcal{L}(G_\tau)$ will have $\tau_G(g) \cap \mathcal{L}(G) \neq \emptyset$. This is largely by definition, context-safety meaning that there is at least one way for τ_G to join up the nodes which is consistent with the CHR derivation. Note that the DAG grammar sketched above is context-safe, so the issue that not *all* the graphs produced by τ_G necessarily correspond to valid CHR derivations remains. This is not necessary for any result in this paper, but means that all graphs $g \in G_\tau$ can be pumped and will always produce at least some graph in G under τ_G . \diamond

Lemma 2. *For all CHR grammars G it holds that $\mathcal{L}(G) \subseteq \cup_{g \in \mathcal{L}(G_\tau)} \tau(g)$.*

Proof. Firstly note that for HR rules R and any graphs g (on which τ_G is defined), g' and g'' we have $\tau_G(\tau_G(g) \rightarrow g') \subseteq \tau_G(g \rightarrow_R g'')$ (in a slight abuse of notation we write $f(g \rightarrow_R g')$ to mean $\{f(g') \mid g \rightarrow g'\}$). This is the case as HR rule applications preserve contextual labels (replacing only nonterminals), so if $\tau_G(g)$ merges $\perp_\alpha(i)$ into $\alpha(j)$ then $\tau_G(g \rightarrow_R g'')$ can do the same (the reverse does not hold in general as $\tau_G(g \rightarrow_R g'')$ can merge a node $\perp_\alpha(i)$ from g into a node $\alpha(j)$ from g'' , which the other case cannot).

Now consider a derivation $g_1 \rightarrow_{r_1} \dots \rightarrow_{r_n} g_{n+1}$ in G , where $g_1 = S()$ and $g_{n+1} \in \mathcal{L}(G)$. For each r_i let r'_i be the corresponding rule in G_τ (as constructed in Definition 12), and then we show that $g_{n+1} \in \tau_G(g)$ for a $g \in \mathcal{L}(G_\tau)$ by inductively showing that $g_i \in \tau_G(\dots \tau_G(\tau_G(g'_1) \rightarrow_{r'_1} g'_2) \dots \rightarrow_{r'_{i-1}} g'_i)$ for sentential graphs g'_1, \dots, g'_i for G_τ , equivalent to those in G . For the base case: $g_1 = \tau_G(g'_1)$, as both are the initial nonterminal $S()$ (recall that G and G_τ have the same set of nonterminals); and; for inductive step i : the lhs r_i of G will apply to some (possibly zero) contextual edges, where in the equivalent step in the G_τ derivation the equivalent \perp -labeled nodes are instead introduced on the rhs, and the immediate τ_G application can merge those into the same nodes which r_i in G took as contextual (again note that τ_G may be able to merge in other ways, but we only prove that one of the ways is equivalent to the derivation in G). Noting that all the τ_G applications can be collapsed up into a single top-level one (as shown in the first part) completes the proof. \square

Theorem 6. *For every CHR language $L \subseteq G_\Sigma$ there exists a constant $p \in \mathbb{N}$ such that every $g \in \text{mark}(L)$ which has at least p marked nodes and edges there are graphs g_1, g_2, g_3 , with (nodes renamed such that) $i < j$ for all $i \in \dot{g}_1$ and $j \in \dot{g}_2$, and two one-to-many (i.e. describing the inverse of a function) relations $R_1 \subseteq \dot{g}_1 \times \dot{g}_2$ and $R_2 \subseteq (\dot{g}_1 \cup \dot{g}_2) \times \dot{g}_3$, such that*

1. x, u, v , and y have rank at most p ,
2. g_2 has at least one edge or node which is marked,
3. $g_2 \otimes_y^* g_3$ contains at most p marked nodes and edges,
4. $g = ((g_1 \otimes_u^* g_2) \llbracket R_1 \rrbracket \otimes_y^* g_3) \llbracket R_2 \rrbracket$, and,
5. $(G \otimes_y^* g_3) \llbracket R_2 \rrbracket \subseteq \text{mark}(L)$, where G is the smallest language such that $g_1 \in G$ and $(G \otimes_u^* g_2) \llbracket R_1 \rrbracket \subseteq G$.

Proof. The statement of this theorem appears somewhat complex, but it is in practice a very direct consequence of pumping G_τ .

Note that if there is an edge labeled $\text{mark}(\perp)^{(1)}$ in g_2 then the node it is attached to is “created” in g_2 , in that it is neither attached to both the u - and v -handle, nor is it in R_1 (which gets merged into a node in g_1). This must be case as otherwise pumping g_2 twice would add two $\text{mark}(\perp)$ labels to the same node, which Definition 9 disallows. From this it follows that pumping g_2 does indeed add new copies of a marked node or edge to the resulting graph.

By definition there exists a CHR grammar G accepting L , apply Lemma 2 to find the $g' \in \text{mark}(G_\tau)$ which has $g \in \tau_G(g')$, apply Theorem 5 (thus choosing p as indicated by that theorem) to get a pumpable decomposition $g' = g'_1 \otimes_u^* g'_2 \otimes_v^* g'_3$, then construct g_1, g_2, g_3, R_1 and R_2 by, following the steps of Lemma 2, merging any pairs of nodes which are in the same component graph, and recording in R_1 the nodes which should be merged from g_2 into g_1 , and in R_2 the nodes which should be merged from g_3 into g_1 and g_2 (deleting the \perp -labeled edges in the process). Then the part of the derivation corresponding to g_2 can be repeated arbitrarily many (including zero) times, each time applying R_1 (note that the way Definition 1 chooses a representative of the merged nodes is abused to retain the node identity from g_1 in G) to merge new copies of the same nodes into the ones in g_1 indicated by the relation. This procedure necessarily produces new graphs in L , amounting to repeating derivation steps choosing the same contextual nodes as in the original derivation. \square

Example 1. Numbering the nodes of the graph from Figure 4(d). top to bottom: $w(1), w(2), w(3), w(4), a(1, 2), a(1, 2), a(1, 3), a(2, 3), a(2, 4)$, then, marking node 3, Theorem 6 is fulfilled by the following decomposition with rank-0 handles $g_1 = [w(1), w(2), a(1, 2), a(1, 2), x()]$, $g_2 = [u(), w(3), a(1', 3), a(2', 3), v()]$, $g_3 = [y(), w(4), a(2', 4)]$ with the relations $R_1 = \{(1, 1'), (2, 2')\}$ and $R_2 = \{(2, 2')\}$. See Figure 5 for an illustration of the graph resulting from repeating g_2 four times. Note that the handles are rank zero as all edges are contextual, in general cases they handle the “Habel-Kreowski-style” HR part of the pumping. \diamond

Example 2. Many conclusions can be drawn by such a pumping result, to illustrate let us note two simple examples.

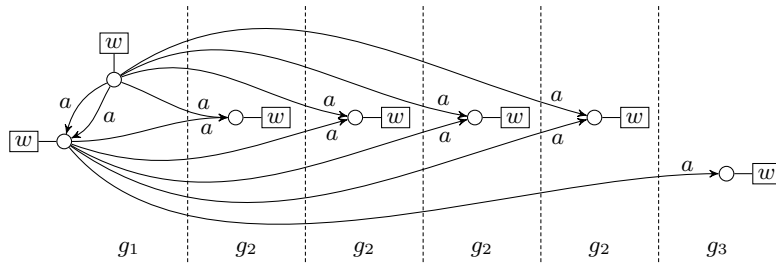


Fig. 5. An illustration of the result of pumping the decomposition of Example 1 four times. To the left of the first dashed boundary is the part corresponding to g_1 (i.e. node 1 and 2 and some edges), the top eight edges crossing that dashed boundary are created by R_1 collapsing $1'$ and $2'$ from each copy of g_2 into node 1 and 2 in g_1 . The bottom edge is created by the application of R_2 .

The language of complete graphs is not in CHR: marking a node in any such graph, and applying Theorem 6, gives a way to pump the graph which grows the number of nodes while growing the number of edges only linearly

For every CHR language L either: (i) every $g \in \text{mark}(L)$ can be pumped with Theorem 5 (i.e. it can be pumped as a HR grammar); or; (ii) the node degrees in L are unbounded, i.e. for every $k \in \mathbb{N}$ there exists a graph $g \in L$ which has at least one node of degree $\geq k$. This can be seen as either R_1 can be made empty, or it add edges to some node on each iteration of the pumping procedure. \diamond

6 Conclusions and Future Work

In summary we have introduced and to some extent motivated definitions and notation suitable for discussing HR and CHR grammars in the same framework, and have applied it to discuss pumping properties of these formalisms in a unified way. This is only an initial look at the work planned however, with several loose threads to be considered. Notably the framework is intended to cover some further classes of formalisms, such as the regular tree folding mechanism from [Bjö18]. The pumping result for CHR is merely an early example, and should be refined, e.g.: making the G_τ and τ_G constructions precise); and considering the frequent case where the R_1 relation should be allowed to attach edges to “newer” copies of g_2 rather than always to g_1 .

References

- [Bjö18] Johanna Björklund. Tree-to-graph transductions with scope. In *DLT'18*, volume 11088 of *LNCS*, pages 133–144. Springer, 2018.
- [DH15] Frank Drewes and Berthold Hoffmann. Contextual hyperedge replacement. *Acta Informatica*, 52(6):497–524, 2015.
- [DHM11] Frank Drewes, Berthold Hoffmann, and Mark Minas. Contextual hyperedge replacement. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, pages 182–197. Springer, 2011.
- [DKH97] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge replacement graph grammars. In *Handbook Of Graph Grammars And Computing By Graph Transformation, Vol. 1*, pages 95–162. World Scientific, 1997.
- [Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *LNCS*. Springer Berlin Heidelberg, 1992.
- [HK87] Annegret Habel and Hans-Jörg Kreowski. Some structural aspects of hypergraph languages generated by hyperedge replacement. In *STACS'87*, volume 247 of *LNCS*, pages 207–219. Springer, 1987.
- [HM10] Berthold Hoffmann and Mark Minas. Defining models-meta models versus graph grammars. *Electronic Communications of the EASST*, 29, 2010.
- [Kuh08] Marco Kuhlmann. Ogden's lemma for regular tree languages. *CoRR*, abs/0810.4249, 2008.
- [Lau90] Clemens Lautemann. Tree automata, tree decomposition and hyperedge replacement. In *International Workshop on Graph Grammars and Their Application to Computer Science*, pages 520–537. Springer, 1990.
- [Ogd68] William Ogden. A helpful result for proving inherent ambiguity. *Mathematical systems theory*, 2(3):191–194, 1968.

Speeding up Generalized PSR Parsers by Memoization Techniques

Mark Minas

Universität der Bundeswehr München, Germany
mark.minas@unibw.de

Abstract. Predictive shift-reduce (PSR) parsing for hyperedge replacement (HR) grammars is very efficient, but restricted to a subclass of unambiguous HR grammars. To overcome this restriction, we have recently extended PSR parsing to generalized PSR (GPSR) parsing along the lines of Tomita’s generalized LR parsing. Unfortunately, GPSR parsers turned out to be too inefficient without manual tuning. This paper proposes to use memoization techniques to speed up GPSR parsers without any need of manual tuning, and which has been realized within the graph parser distiller GRAPPA. We present running time measurements for some example languages; they show a significant speed up by some orders of magnitude when parsing valid graphs. But memoization techniques do not help when parsing invalid graphs or if all parses of an ambiguous input graph shall be determined.

Keywords: hyperedge replacement grammar, graph parsing, parser generator

1 Introduction

In earlier work [4], we have devised *predictive shift-reduce* parsing (PSR), which lifts D.E. Knuth’s LR string parsing [9] to graphs and runs in at most expected linear time in the size of the input graph. However, parsing for graph grammars based on hyperedge replacement (HR) is in general NP-hard, even for a particular grammar [6]. Therefore, PSR parsing is restricted to a subclass of HR grammars, which particularly must be unambiguous. We have recently extended PSR parsing to *generalized PSR* (GPSR) parsing [7], which can be applied to every HR grammar.

GPSR parsing has been motivated by M. Tomita’s *generalized LR* (GLR) parsing for strings [12]. The original GLR parsing algorithm by Tomita runs in $O(n^{k+1})$ time where n is the length of the input string and k the length of the longest rule in the grammar, but J.R. Kipps improved it to $O(n^3)$ [8], which is the same complexity as of a parser using the *Cocke-Younger-Kasami* (CYK) algorithm [13].

GPSR parsing cannot be efficient in general because GPSR parsers can be applied to every HR grammar. But our experiments [7] have shown that GPSR parsers are even slower than simple graph parsers that extend the CYK algorithm

to graphs [10,11]. Manual tuning of GPSR parsers by using language specific strategies (see Sect. 4) helped to improve their efficiency, but even those tailored parsers have not always been faster than the corresponding CYK parsers.

GPSR parsers identify parses of an input graph in a search process that may run into dead ends. They are inefficient because they waste time in this process and because they discard all information collected in these dead ends, even if it could be used later. This paper proposes to use memoization techniques to keep the information and to reuse it later. Reuse allows to skip long sequences of parsing operations that would just recreate information that has already been collected earlier.

GPSR parsing with memoization has been implemented in the graph-parser distiller GRAPPA¹. Experiments with generated parsers for different example languages demonstrate that memoization substantially improves parsing speed.

The remainder of this paper is structured as follows. After recalling HR grammars in Sect. 2, PSR parsing in Sect. 3, and GPSR parsing in Sect. 4, we describe how memoization can speed up GPSR parsing in Sect. 5. We compare its performance with plain GPSR parsing and with CYK parsing using three example graph languages in Sect. 6: Sierpinski graphs, series-parallel graphs, and structured flowcharts, where GPSR parsing with memoization substantially improves plain GPSR parsing. Sect. 7 concludes the paper.

2 Graph Grammars Based on Hyperedge Replacement

Throughout the paper, we assume that X is a global, countably infinite supply of *nodes*, and that Σ is a finite set of *symbols* that comes with an *arity function* $\text{arity}: \Sigma \rightarrow \mathbb{N}$, and is partitioned into disjoint subsets \mathcal{N} of *nonterminals* and \mathcal{T} of *terminals*.

We represent hypergraphs as ordered sequences of edge literals, where each literal represents an edge with its attached nodes. This is convenient as we shall derive (and parse) the edges of a graph in a fixed order.

Definition 1 (Hypergraph). For a symbol $\mathbf{a} \in \Sigma$ and $k = \text{arity}(\mathbf{a})$ pairwise distinct nodes $x_1, \dots, x_k \in X$, $\mathbf{a} = \mathbf{a}^{x_1 \dots x_k}$ represents a *hyperedge* that is labeled with \mathbf{a} and attached to x_1, \dots, x_k . \mathcal{E}_Σ denotes the set of hyperedges (over Σ).

A *graph* $\gamma = \langle V, \varphi \rangle$ over Σ consists of a finite set $V \subseteq X$ of nodes and a sequence $\varphi = \mathbf{e}_1 \dots \mathbf{e}_n \in \mathcal{E}_\Sigma^*$ of hyperedges such that all nodes in these literals are in V . \mathcal{G}_Σ denotes the set of all graphs over Σ .

We say that two graphs $\gamma = \langle V, \varphi \rangle$ and $\gamma' = \langle V', \varphi' \rangle$ are *equivalent*, written $\gamma \bowtie \gamma'$, if $V = V'$ and φ is a permutation of φ' .

In the following, we usually call hypergraphs just *graphs* and hyperedges just *edges*. For a graph $\gamma = \langle V, \varphi \rangle$, we use the notation $V_\gamma = V$.

Note that graphs are sequences rather than multisets of literals, i.e., two graphs $\langle V, \varphi \rangle$ and $\langle V', \varphi' \rangle$ with the same set of nodes, but with different sequences of literals are considered to differ, even if $V = V'$ and φ' is just a

¹ Available under www.unibw.de/inf2/grappa.

permutation of φ . However, such graphs are equivalent, denoted by the equivalence relation \bowtie . In contrast, “ordinary” graphs would rather be represented using multisets of literals instead of sequences. The equivalence classes of graphs, therefore, correspond to conventional graphs. The ordering of literals is technically convenient for the constructions in this paper. However, input graphs to be parsed should of course be considered up to equivalence. Thus, we will make sure that the developed parsers yield identical results on graphs g, g' with $g \bowtie g'$.

An injective function $\varrho: X \rightarrow X$ is called a *renaming*, and γ^ϱ denotes the graph obtained by replacing all nodes in γ according to ϱ . Although renamings are, for technical simplicity, defined as functions on the whole of X , only the finite subset $V_\gamma \subseteq X$ will be relevant. We define the “concatenation” of two graphs $\gamma = \langle V, \varphi \rangle, \gamma' = \langle V', \varphi' \rangle \in \mathcal{G}_\Sigma$ as $\gamma\gamma' = \langle V \cup V', \varphi\varphi' \rangle$. If a graph $\gamma = \langle V, \varphi \rangle$ is completely determined by its sequence φ of literals, i.e., if each node in V also occurs in some literal in φ , we simply use φ as a shorthand for γ . In particular, a literal $\mathbf{a} = \mathbf{a}^{x_1 \dots x_k} \in \mathcal{E}_\Sigma$ is identified with the graph $\langle \{x_1, \dots, x_k\}, \mathbf{a} \rangle$.

A *hyperedge replacement rule* $r = (\mathbf{A} \rightarrow \alpha)$ (*rule* for short) has a nonterminal edge $\mathbf{A} \in \mathcal{E}_\mathcal{N}$ as its *left-hand side*, and a graph $\alpha \in \mathcal{G}_\Sigma$ with $V_\mathbf{A} \subseteq V_\alpha$ as its *right-hand side*.

Consider a graph $\gamma = \beta \bar{\mathbf{A}} \bar{\beta} \in \mathcal{G}_\Sigma$ with a nonterminal edge $\bar{\mathbf{A}}$ and a rule $r = (\mathbf{A} \rightarrow \alpha)$. A renaming $\mu: X \rightarrow X$ is a *match* (of r in γ) if $\mathbf{A}^\mu = \bar{\mathbf{A}}$ and if $V_\gamma \cap V_{\alpha^\mu} \subseteq V_{\mathbf{A}^\mu}$.² A match μ of r *derives* γ to the graph $\gamma' = \beta \alpha^\mu \bar{\beta}$. This is denoted as $\gamma \Rightarrow_{r, \mu} \gamma'$. If \mathcal{R} is a finite set of rules, we write $\gamma \Rightarrow_{\mathcal{R}} \gamma'$ if $\gamma \Rightarrow_{r, \mu} \gamma'$ for some match μ of some rule $r \in \mathcal{R}$.

Definition 2 (HR Grammar). A *hyperedge replacement grammar* $\Gamma = (\Sigma, \mathcal{T}, \mathcal{R}, Z)$ (*HR grammar* for short) consists of *symbols* Σ with *terminals* $\mathcal{T} \subseteq \Sigma$ as assumed above, a finite set \mathcal{R} of rules, and a *start graph* $Z = Z^\varepsilon$ with $Z \in \mathcal{N}$ of arity 0. Γ generates the language $\mathcal{L}(\Gamma) = \{g \in \mathcal{G}_\mathcal{T} \mid Z \Rightarrow_{\mathcal{R}}^* g\}$.

In the following, we simply write \Rightarrow and \Rightarrow^* because the rule set \mathcal{R} in question will always be clear from the context.

Example 1 (A HR Grammar for Sierpinski Triangles). The following rules

$$Z^\varepsilon \xrightarrow{0} \mathbf{D}^{xyz} \quad \mathbf{D}^{xyz} \xrightarrow{1} \mathbf{D}^{xuw} \mathbf{D}^{uyv} \mathbf{D}^{wvz} \quad \mathbf{D}^{xyz} \xrightarrow{2} \mathbf{t}^{xyz}$$

generate Sierpinski triangles as graphs where triangles are represented by ternary edges of type \mathbf{t} . This grammar is in fact a slightly modified version of [6, p. 189] where edges of triangles are represented by binary edges.

Fig. 1 shows a derivation with graphs as diagrams, in particular with \mathbf{t} -edges drawn as triangles. This corresponds to the following derivation. Underlines indicate rewritten nonterminal edges:

$$\begin{aligned} Z^\varepsilon &\xRightarrow{0} \underline{\mathbf{D}^{ahf}} \xRightarrow{1} \mathbf{D}^{abc} \underline{\mathbf{D}^{bhj}} \mathbf{D}^{cjl} \xRightarrow{1} \mathbf{D}^{abc} \mathbf{D}^{bde} \mathbf{D}^{dhi} \mathbf{D}^{eij} \underline{\mathbf{D}^{cjl}} \\ &\xRightarrow{1} \mathbf{D}^{abc} \mathbf{D}^{bde} \mathbf{D}^{dhi} \mathbf{D}^{eij} \mathbf{D}^{cfg} \mathbf{D}^{fje} \mathbf{D}^{gfl} \xRightarrow{7}{2} \mathbf{t}^{abc} \mathbf{t}^{bde} \mathbf{t}^{dhi} \mathbf{t}^{eij} \mathbf{t}^{cfg} \mathbf{t}^{fje} \mathbf{t}^{gfl} \end{aligned}$$

² I.e., a match μ makes sure that the nodes of α^μ that do not occur in $\bar{\mathbf{A}} = \mathbf{A}^\mu$ do not collide with the other nodes in γ .

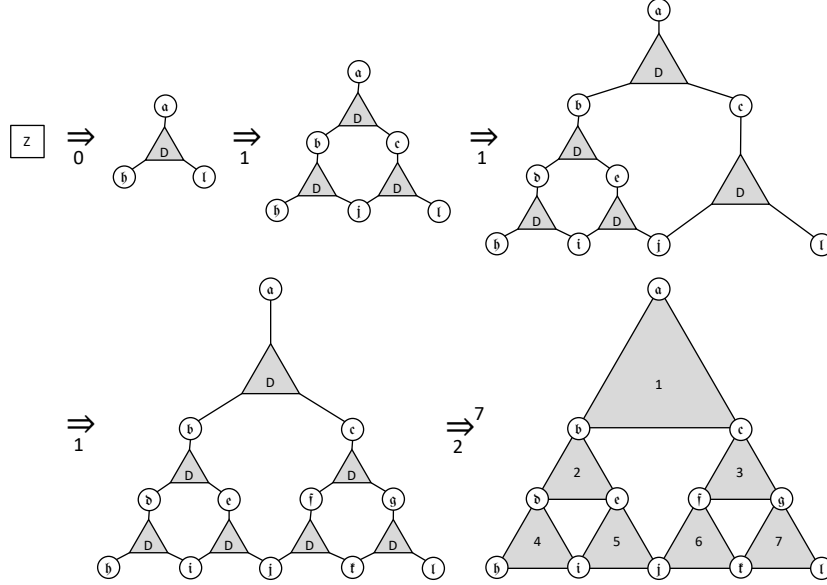


Fig. 1. A derivation of the graph $t^{abc}t^{bdc}t^{dhi}t^{cij}t^{efg}t^{fjt}t^{gfl}$. Nodes are drawn as circles with their names inscribed, nonterminal edges as boxes or triangles around their label, with lines to their attached nodes, and terminal edges as triangles visiting their attached nodes counter-clockwise, starting at the top corner. The numbers inside the terminal triangles are used later to refer to the corresponding terminal edges.

3 Predictive Shift-Reduce Parsing

The article [4] gives detailed definitions and correctness proofs for PSR parsing. Here we recall the concepts only so far that we can describe its generalization in the next section.

A PSR parser attempts to construct a derivation by reading the edges of a given input graph one after the other.³ However, the parser must not assume that the edges of the input graph come in the same order as in a derivation. E.g., when constructing the derivation in Fig. 1, it must also accept an input graph $t^{abc}t^{bdc}t^{efg}t^{dhi}t^{cij}t^{fjt}t^{gfl}$ where the edges are permuted.

Before parsing starts, a procedure described in [3, Sect. 4] analyzes the grammar for the *unique start node* property, by computing the possible incidences of all nodes created by a grammar. The unique start nodes have to be matched by some nodes in the right-hand side of the start rule of the grammar, thus determining where parsing begins. For our example, the procedure detects that every Sierpinski graph has a unique topmost node. That is a node with a single t-edge attached where the node is the first in the edge's attachments. The node

³ We silently assume that input graphs do not have isolated nodes. This is no real restriction as one can add special edges to such nodes.

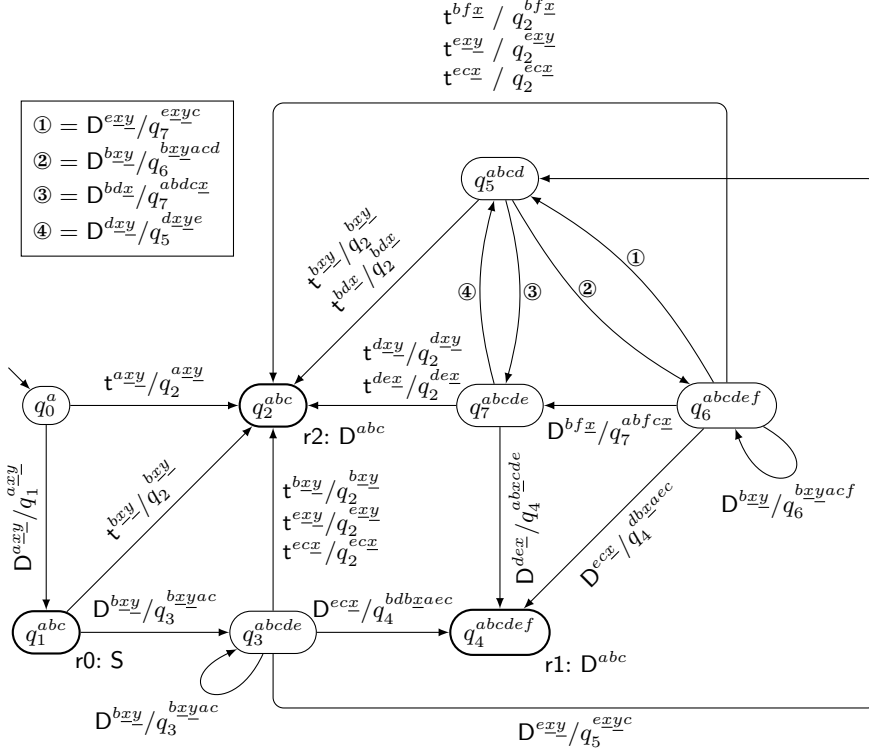


Fig. 2. The characteristic finite automaton for the HR grammar of Sierpinski triangles.

x in the start rule $Z^\varepsilon \rightarrow D^{xyz}$ must be bound to the topmost node of any input graph.⁴ If the input graph has no topmost node, or more than one, it cannot be a Sierpinski graph, so that parsing fails immediately.

A PSR parser is a push-down automaton that is controlled by a *characteristic finite automaton* (CFA). The stack of the PSR parser consists of states of the CFA. The parser makes sure that the sequence of states on its stack always describes a valid walk through its CFA.

Fig. 2 shows the CFA for our example of Sierpinski graphs. It has been generated by the graph parser distiller GRAPPA,¹ using the constructions described in [4], and consists of eight states. Each state has a unique state number and a number of *parameters*, which are written as subscript and superscript, respectively. Parameters are placeholders for nodes of the input graph, which have already been read by the parser. The initial state is q_0^a . Its parameter a is bound

⁴ The other two nodes of the start rule, in fact, can be uniquely identified, too, which could be used as a second and a third start node bound to y and z , respectively. However, the corresponding CFA is too complicated for a presentation in this paper.

to the start node of the input graph, i.e., the topmost node, when parsing starts. Transitions between states are labeled by pairs with a slash as a separator. The first part of a label is the *trigger* of the transition whereas the second part of a label determines the parameters of the target state of the transition. Note that the latter is in fact the target state of the transition with its parameters set to the values used in the label. The trigger is a placeholder for an edge whose attached nodes are either parameters of the source state of the transition, or placeholders \underline{x} or \underline{y} , which stand for nodes of the input graph that have not yet been read by the parser. Note that some transitions have multiple labels. This is in fact a shortcut for different transitions, each with one of these labels. We are going to describe the meaning of labels in the following and shall use $t^{abc}t^{bdc}t^{cfa}t^{dhi}t^{eij}t^{fjt}t^{gkl}$ as an input graph.

A PSR parser starts with a stack that contains the input state with its parameters bound to the start nodes. In our example this is q_0^a with a being bound to node \mathbf{a} , written as q_0^a . We call such a state with all its parameters being bound to input graph nodes a *concrete state*. The next action of the parser is always determined by the topmost state on the stack, which is concrete, and by consulting the corresponding state in the CFA. Three different types of actions are distinguished:

A *shift* action reads a yet unread edge of the input graph. This corresponds to an outgoing transition with a terminal trigger. The trigger fits if the input graph contains an unread edge labeled with the trigger label and being attached to input graph nodes as specified by the node placeholders of the trigger. If the topmost state is q_0^a , there is an outgoing transition to state q_2^{abc} with a trigger $t^{a\underline{x}\underline{y}}$. Parameter a is bound to \mathbf{a} , and its second and third attached nodes must be unread nodes, indicated by \underline{x} and \underline{y} . Edge t^{abc} fits this trigger because t^{abc} and \mathbf{b} as well as \mathbf{c} are yet unread. The shift action marks this edge as well as its attached nodes as read, and pushes the target state of the transition on the stack. The second part of the label determines the binding of this state. In our example, this is $q_2^{a\underline{x}\underline{y}}$ where $a, \underline{x}, \underline{y}$ are bound to $\mathbf{a}, \mathbf{b}, \mathbf{c}$, respectively. As a consequence, the stack will now contain q_0^a and q_2^{abc} with the latter being the new topmost state.

A *reduce* action is performed when the top state of the stack corresponds to the right-hand side of a rule which is then replaced by the corresponding left-hand side. The parser recognizes this situation by inspecting just the topmost state of the stack; states that allow a reduce action are marked accordingly. In Fig. 2, these states are drawn with a thick border and additionally labeled by $r0$, $r1$, and $r2$ together with a placeholder for a nonterminal edge. For instance, q_2^{abc} is labeled by $r2:D^{abc}$ where $r2$ means a reduction using rule 2 of the grammar. The reduce action in fact consists of three consecutive steps. In the first step, the parser creates a nonterminal as indicated by the state label. In our example, it is $r2:D^{abc}$. With a topmost state q_2^{abc} , a, b, c are bound to $\mathbf{a}, \mathbf{b}, \mathbf{c}$, which produces a nonterminal D^{abc} . In the second step of the reduce action, the parser pops as many states off the stack as this rule's right-hand side contains edges, i.e., just one state for rule 2. For instance, when starting with stack contents $q_0^a q_2^{abc}$, q_2^{abc} is popped off, yielding a stack just containing q_0^a . The third step is called a *goto*

step. It inspects the new topmost state, i.e., q_0^a here, and selects an outgoing transition whose trigger fits the nonterminal edge produced in the first step, i.e., D^{abc} and the transition to q_1^{abc} . The parser then pushes the target state with its parameters bound according to the transition label. In our example, the stack is then $q_0^a q_1^{abc}$.

An ***accept*** action is in fact a particular reduce action for the start rule, i.e., rule 0 in our example. The input graph is accepted if the topmost state of the stack is labeled with $r0:S$, i.e., state q_1^{abc} in our example, and if all nodes and edges of the input graph are marked as read. In our example with stack contents $q_0^a q_1^{abc}$, the parser has rather reached the accepting state, but there are some unread edges and nodes, i.e., the input graph cannot be accepted yet.

The parser fails if neither a shift, reduce, nor accept action is possible.

As described in [4], such a CFA can be computed for every HR grammar. But it can control a PSR parser as described above only if its states do not have *conflicts*. A conflict is a situation where the parser must choose between different actions. It is clear that the parser cannot run into a dead end if no state of the CFA has a conflict; the parser can then always predict the correct action which avoids a dead end for valid graphs.⁵ But in the case of conflicts, the parser must choose between several actions; it cannot predict the correct next action. A grammar with such a CFA is not PSR parseable.

Our example grammar for Sierpinski graphs is not PSR parseable because states q_3^{abcde} and q_6^{abcdef} have conflicts. When the parser reaches q_3^{abcde} , for instance, it must read a t -edge in the next shift step, and it must choose between an edge being attached to b (or rather the node that b is bound to) or e , indicated by the transition to q_2^{abc} .

4 Generalized Predictive Shift-Reduce Parsing

In [7] we have proposed *generalized PSR* (GPSR) parsing for grammars that are not PSR parseable. A GPSR parser is primarily a PSR parser that follows all different choices if a state has conflicts. It tries to save time and space in a similar way as M. Tomita's *generalized LR* parser for context-free string grammars [12]. Let us briefly summarize how GPSR parsing works.

Whereas a PSR parser maintains a single stack for parsing, a GPSR parser in fact maintains a set of stacks. This set is stored as a so-called *graph-structured stack* (GSS), which is described in the next paragraph. For each stack, the parser determines all possible actions based on the CFA as described for the PSR parser. The parser has found a successful parse if the action is *accept* and the entire input graph has been read. (It may proceed if further parses shall be found.) If the parser fails for a specific stack, the parser just discards this stack, stops if this has been the last remaining stack, and fails altogether if it has not found a successful

⁵ This does not necessarily mean that PSR parsers are deterministic; different edges may be chosen for the same shift action. This does not occur in our example of Sierpinski graphs. In general, a grammar can only be PSR parseable if it additionally satisfies the *free edge choice* property [4].

parse previously. If the CFA, however, indicates more than one possible action, the parser duplicates the stack for each of them, and performs each action on one of the copies.

In fact, a GPSR parser does not store complete copies of stacks, but shares their common prefixes and suffixes. The resulting structure is a DAG known as a *graph-structured stack* (GSS) as proposed by M. Tomita [12]. Each node of this DAG (called *GSS node* in the following) is a state. An individual stack is represented as a path in the GSS, from some topmost state to the unique initial state. Working on the GSS instead of on a set of complete copies of different stacks does not only save space, but also time: instead of repeating the same operations on different stacks that share the same suffix, the parser has to perform these actions only once. Furthermore, maintaining the GSS simplifies the construction of all parse trees (the so-called parse forest) of an ambiguous input. But we ignore this aspect in this paper.

Remember that we represent graphs as permutations of edges. By trying out every action offered by the CFA in each step, the GPSR parser effectively performs an exhaustive search in the set of all permutations of the input graph edges permitted by the CFA. This has two immediate effects for a GPSR parser:

Consider two different stacks reached by the GPSR parser. Each stack represents a different history of choices the parser has made. In particular, different input graph edges may have been read in these histories. The parser, therefore, cannot globally mark edges as read, but it must store, for each stack separately, which edges of the input graph have been read. In fact, each GSS node keeps track of the set of input graph edges that have been read so far. Note that GSS nodes may be shared only if both their concrete states and their sets of read edges coincide.

Whenever the parser has a GSS that represents at least two different stacks, it must choose the stack that it considers next for its actions. It may employ a breadth-first strategy or a depth-first strategy. In [7], we have shown for two example languages (series-parallel graphs and structured flowcharts; see Sect. 6) that the chosen strategy strongly affects the parser speed. In fact, a standard strategy was always too slow, even slower than a simple CYK parser. Instead, specifically tailored strategies have been used that give certain grammar rules preference over others. This requires extra manual work when building a parser and was the motivation for this paper, in particular because even this does not always help in creating a GPSR parser that is faster than a CYK parser.

As a matter of fact, breadth-first and depth-first produce slow parsers for the language of Sierpinski graphs, too. We shall demonstrate this by describing the steps of the GPSR parser for the input graph $t^{abc}t^{bde}t^{cfe}t^{dhi}t^{eij}t^{fje}t^{gfi}$ (see Fig. 1). To save space, we refer to these edges by numbers $1 = t^{abc}$, $2 = t^{bde}$, $3 = t^{cfe}$, $4 = t^{dhi}$, $5 = t^{eij}$, $6 = t^{fje}$, $7 = t^{gfi}$. These numbers correspond to the numbers within the triangles in Fig. 1. And we write GSS nodes in compact form: e.g., 2_{124}^{dhi} refers to the concrete state q_2^{dhi} and indicates that the edges $1 = t^{abc}$, $2 = t^{bde}$, and $4 = t^{dhi}$ have been read already. Fig. 3 shows the graph-structured stacks after each step of the GPSR parser where a step consists of all actions performed by

0	0_{\emptyset}^a	
1	$0_{\emptyset}^a - 2_1^{abc}$	
2	$0_{\emptyset}^a - 1_1^{abc}$	
3	$0_{\emptyset}^a - 1_1^{abc} - 2_{12}^{bde}$	
4	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac}$	
5	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} \begin{matrix} \nearrow 2_{123}^{cfa} \\ \searrow 2_{124}^{dhi} \end{matrix}$	
6	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} \begin{matrix} \nearrow 5_{123}^{cfge} \\ \searrow 2_{124}^{dhi} \end{matrix}$	
7	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} \begin{matrix} \nearrow 5_{123}^{cfge} - 2_{1236}^{fjtcge} \\ \searrow 2_{124}^{dhi} \end{matrix}$	
8	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} \begin{matrix} \nearrow 5_{123}^{cfge} - 6_{1236}^{fjtcge} \\ \searrow 2_{124}^{dhi} \end{matrix}$	
9	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} \begin{matrix} \nearrow 5_{123}^{cfge} - 6_{1236}^{fjtcge} - 2_{12367}^{gfl} \\ \searrow 2_{124}^{dhi} \end{matrix}$	
10	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} \begin{matrix} \nearrow 5_{123}^{cfge} - 6_{1236}^{fjtcge} - 4_{12367}^{cflfgt} \\ \searrow 2_{124}^{dhi} \end{matrix}$	
11	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} \begin{matrix} \nearrow 5_{12367}^{cjle} \\ \searrow 2_{124}^{dhi} \end{matrix}$	
12	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} - 2_{124}^{dhi}$	
13	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} - 3_{124}^{dhibe}$	
14	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} - 3_{124}^{dhibe} - 2_{1245}^{cij}$	
15	$0_{\emptyset}^a - 1_1^{abc} - 3_{12}^{bdeac} - 3_{124}^{dhibe} - 4_{1245}^{bhjdc}$	
16	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac}$	
17	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac} - 2_{12345}^{cfa}$	
18	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac} - 5_{12345}^{cfaj}$	
19	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac} - 5_{12345}^{cfaj} - 2_{123456}^{fjt}$	
20	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac} - 5_{12345}^{cfaj} - 7_{123456}^{cjfgt}$	
21	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac} - 5_{12345}^{cfaj} - 7_{123456}^{cjfgt} - 2_{1234567}^{gfl}$	
22	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac} - 5_{12345}^{cfaj} - 7_{123456}^{cjfgt} - 4_{1234567}^{cflfgt}$	
23	$0_{\emptyset}^a - 1_1^{abc} - 3_{1245}^{bhjac} - 4_{1234567}^{ahlbci}$	
24	$0_{\emptyset}^a - 1_{1234567}^{ahl}$	

Fig. 3. Graph-structured stacks and steps of the GPSR parser when parsing the Sierpinski graph with the edges $1 = t^{abc}$, $2 = t^{bde}$, $3 = t^{cfa}$, $4 = t^{dhi}$, $5 = t^{cij}$, $6 = t^{fjt}$, $7 = t^{gfl}$.

the parser when working on a specific state. Stacks grow to the right, i.e., the initial state is at the left end whereas topmost states are at the right ends. The steps in fact follow the depth-first strategy which turned out to be a bit faster than the breadth-first strategy.

The parser starts (step 0) with a single stack that contains just 0_{\emptyset}^a , i.e., the initial (concrete) state q_0^a where no edge has been read yet. The first four steps are just PSR steps as described in the previous section: edge $1 = t^{abc}$ is shifted in step 1, a reduce action for rule 2 happens in step 2. Edge $2 = t^{bde}$ is shifted in step 3, and this edge is reduced using rule 2 in step 4, reaching state q_3^{bdeac} . This state allows to shift $3 = t^{cfa}$ as well as $4 = t^{dhi}$ (see Fig. 2), producing two stacks, represented by the GSS after step 5. Note that the topmost states of these stacks are both q_2 -states, but with different parameter bindings and differing sets of read input graph edges. 2_{123}^{cfa} is reduced in step 6, resulting in 5_{123}^{cfge} , which is considered next in step 7 because of the depth-first strategy. In fact, the parser continues working on this stack until it fails in step 12: the stack has the topmost state 5_{12367}^{cjle} when step 12 starts, i.e., only $4 = t^{dhi}$ and $5 = t^{cij}$ are yet unread, but they do not fit to any outgoing transition of q_5^{cjle} . The parser continues with working on the remaining stack, i.e., with topmost state 2_{124}^{dhi} . The remaining steps are again plain PSR steps because the parser need not choose between different actions until it accepts the input graph in step 24 in state $1_{1234567}^{ahl}$, i.e., the accepting state with all edges having been read.

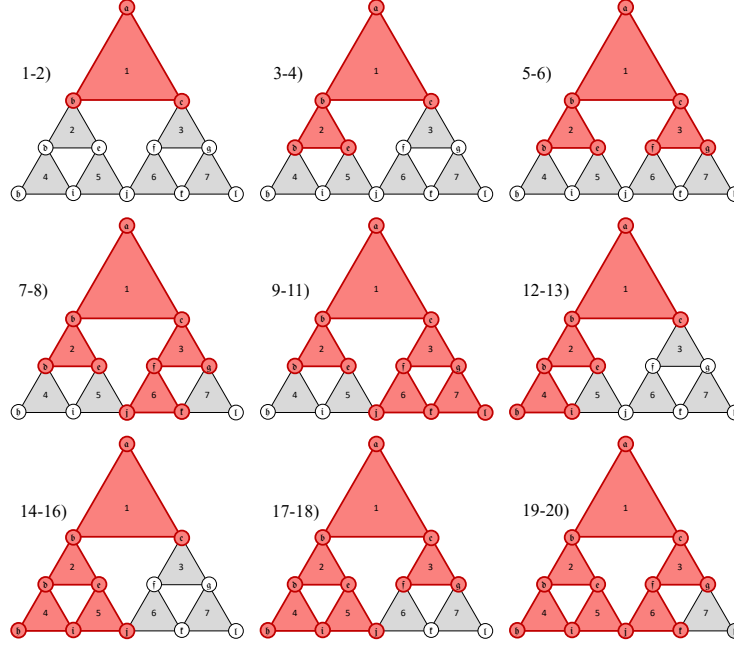


Fig. 4. Visual representation of the edges that are marked as read after the different steps in Fig. 3.

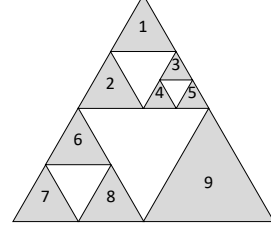
The parser has in fact wasted time by choosing the topmost state 2_{123}^{cfa} for the next stack to work on in step 6. If it had chosen 2_{124}^{dhi} instead, it would have eventually reached the following GSS in step 17:

$$0_{\emptyset}^a \prec \begin{matrix} 1_{12}^{abc} - 3_{12}^{bdeac} - 2_{124}^{dhi} \\ 1_{1234567}^{ahit} \end{matrix}$$

i.e., the parser would have found a successful parse after 17 instead of 24 steps. Of course, the parser could then continue with the remaining stack, which would correspond to the steps 6–12 in Fig. 3., i.e., it would not produce further results. The parser can terminate as soon as it has found a parse because the grammar is unambiguous. But even if the grammar were ambiguous, the parser could terminate after the first parse being found if one is interested in just one parse.

So the question remains whether the parser could be improved by more carefully choosing the stack where the parser continues. For this purpose, consider Fig. 4, which shows the diagram of the input graph after steps 1–20 in Fig. 3. It highlights those edges that are marked as read in the state that has just been pushed to the GSS in the corresponding step. In the steps 1 and 2, for instance, it is the topmost triangle $1 = t^{abc}$, in steps 3 and 4 triangles $1 = t^{abc}$ as well as $2 = t^{bde}$, and so on. Fig. 4 does not show the situation for steps 21–24 where all edges are marked as read. As one can see, the parser erroneously “walks down” to the

lower right triangles 3, 6, and 7 in steps 6–11, discards the corresponding stack in step 12, and again walks down the same path in steps 17–20. So one could assume that a strategy that chooses the left walk first (which corresponds to state 2_{124}^{dhi}) instead of the right walk (which corresponds to state 2_{123}^{fjg}) would improve the parser behavior. However, this is in general not the case. The figure to the right shows a Sierpinski graph where walking down right first finds the parse faster than walking down left first (25 vs. 30 steps): Walking left down first reduces triangles 6–8 twice, once before reducing triangles 3–5 and once after that, which is avoided when walking right down first. Apparently, there is not an easy strategy to always find the parse fast. But memoization solves this problem.



5 Memoization

The GPSR parser finds a parse for a valid input graph faster if it either avoids dead ends like the erroneous walk right down in steps 6–11 (Fig. 4) or if the effort spent in such a dead end is not wasted, but is reused later. To see this, let us consider the parsing steps in Fig. 3 more closely. In step 11, it performs a reduce action for rule 1 on state 4_{12367}^{cjlfgt} producing a nonterminal D^{cjl} (see Fig. 2), popping three states off the stack yielding 3_{12}^{bdca} as a (temporary) topmost state and then pushes state 5_{12367}^{jle} . Moreover, it is known, by comparing the set of read edges of this new topmost state with the set of its predecessor on the stack, that D^{cjl} represents the subgraph consisting of the triangles 3, 6, and 7. But the same nonterminal D^{cjl} representing the same subgraph is again produced in step 23 where the parser performs a reduce action for rule 1 on state $4_{1234567}^{cjlfgt}$, pops three states off the stack yielding state 3_{1245}^{bhjac} , and performs a goto step to $4_{1234567}^{ahlbci}$ triggered by D^{cjl} (see Fig. 2). Note, however, that 3_{1245}^{bhjac} was already the topmost state after step 16. So if the parser remembered that it has produced a D^{cjl} earlier, it could reuse it in step 17 and perform a goto step to $4_{1234567}^{ahlbci}$ right away. As a result, the parser would immediately reach the GSS that Fig. 3 shows after step 23, i.e., the parser would skip six steps and accept the input graph in 18 instead of 24 steps.

In the following, we describe how this observation leads to a systematic approach that allows to skip entire sequences of parsing steps by reusing nonterminal edges that have been produced earlier. This is a *memoization* approach because it depends on memorizing these nonterminal edges.

The main idea is to store a nonterminal edge in a *memo store* whenever it is produced in a reduce action and to look up nonterminals in the memo store whenever the parser reaches a state with an outgoing transition triggered by nonterminal edges. The memo store in fact must store nonterminal edges together with the set of terminal edges that are represented by them. To be more precise, let us assume that the parser analyzes the input graph $h \in \mathcal{G}_{\mathcal{T}}$.

The memo store then contains pairs $\langle \mathbf{A}, g \rangle$ where \mathbf{A} is a nonterminal edge, $g \in \mathcal{G}_{\mathcal{T}}$ is a terminal graph with $\mathbf{A} \Rightarrow^* g$ and $h \bowtie gh'$ for some graph $h' \in \mathcal{G}_{\mathcal{T}}$, i.e., g consists of input graph edges. For instance, the memo store after step 16 in Fig. 3 consists of the following pairs, produced by the reduce actions in one of the previous steps:

$$\begin{aligned} & \{ \langle \mathbf{D}^{bj}, 245 \rangle, \langle \mathbf{D}^{bd}, 2 \rangle, \langle \mathbf{D}^{cl}, 367 \rangle, \langle \mathbf{D}^{fg}, 3 \rangle, \\ & \langle \mathbf{D}^{ac}, 1 \rangle, \langle \mathbf{D}^{bh}, 4 \rangle, \langle \mathbf{D}^{ej}, 5 \rangle, \langle \mathbf{D}^{fi}, 6 \rangle, \langle \mathbf{D}^{gl}, 7 \rangle \}, \end{aligned}$$

Edges in the second components of pairs are again represented by their numbers.

The lookup operation is controlled by the nodes bound to parameters of the current state, by the (nonterminal) label of the transition, and by the set R of edges that are marked as read in the current state. The lookup operation may return *valid* pairs only. These are pairs $\langle \mathbf{A}, g \rangle$ whose graph g does not contain any edge that is also a member of R . Otherwise, edges in g and in R would be read twice.⁶

As an example, let us now consider state 3_{1245}^{bhjac} after step 16. The CFA (Fig. 2) has three outgoing transitions with nonterminal triggers \mathbf{D}^{hxy} , \mathbf{D}^{jx} , and \mathbf{D}^{cxy} when replacing parameters by nodes bound to them. x and y may be bound only to nodes that have not yet been read in state 3_{1245}^{bhjac} . Unread nodes are determined by the set 1245 of read edges, i.e., f, g, i, l are unread in this state. The memo store, therefore, does not contain a pair for \mathbf{D}^{hxy} , but it contains $\langle \mathbf{D}^{cl}, 367 \rangle$ for \mathbf{D}^{jx} and $\langle \mathbf{D}^{fg}, 3 \rangle$ for \mathbf{D}^{cxy} . Note that $\langle \mathbf{D}^{cl}, 367 \rangle$ does not fit \mathbf{D}^{cxy} because node j has been read already. The lookup operation, therefore, has in fact found two valid pairs, and the parser could perform goto actions with both of them. Moreover, it can ignore them both and continue in the regular way, i.e., shift edge $3 = t^{fg}$ (see step 17 in Fig. 3). Because the GPSR parser, by design, does not rule out any choice, it will consider all of them. That way, memoization does not affect the correctness of the parser; if reusing of nonterminals does not lead to acceptance of a valid input graph, regular GPSR will do. But the parser needs a criterion which of the choices to try first. The natural choice is to prioritize the nonterminal edge that represents the largest subgraph; the corresponding goto has the potential to skip the longest sequence of parsing steps. In our example, this is $\langle \mathbf{D}^{cl}, 367 \rangle$, i.e., just the case described at the beginning of this section.

The GRAPPA¹ parser distiller has been extended to generate parsers that maintain a memo store in hash tables and that look up all valid pairs when the parser reaches a state with outgoing nonterminal edges. Looked up pairs are ordered by the size of their represented subgraph and tried in that sequence. And it tries the regular GPSR actions if none of these choices leads to acceptance.

⁶ We assume that there are no parallel edges with the same label. Otherwise, each edge must have a unique name and the lookup operation must make sure that it does not return an edge with a name that is also a member of R .

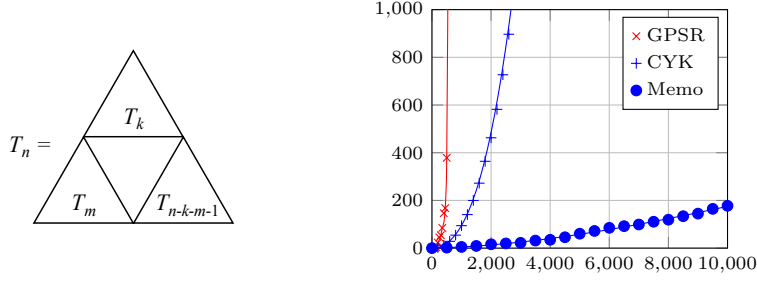


Fig. 5. Recursive definition of Sierpinski graphs T_n for $n > 0$ (left) and running time (in ms) of different parsers analyzing T_n for varying values of n (right).

6 Evaluation

We now report on running time experiments when parsing Sierpinski graphs. We generated three different parsers: a CYK parser using DIAGEN⁷ [11], a GPSR parser using the depth-first strategy described in Sect. 4, and finally a GPSR parser using the depth-first strategy and memoization as described in the previous section. The GPSR parsers have been generated using GRAPPA, and they stop as soon they can accept the input graph. The CYK parser was in fact optimized in two ways: the parser creates nonterminal edges by dynamic programming, and each of these edges can be derived to a certain subgraph of the input graph. The optimized parser makes sure that it does not create two or more indistinguishable nonterminals for the same subgraph, even if the nonterminals represent different derivation trees (which does not occur here.) And it stops as soon as it finds a derivation of the entire input graph.

Running time of the three parsers has been measured for Sierpinski graphs T_n for different values of n . Each T_n consists of $2n + 1$ triangles. T_0 is just a single triangle, and T_n (for $n > 0$) is made of T_k , T_m , and $T_{n-k-m-1}$ as shown in Fig. 5 where $k = \lfloor (n-1)/3 \rfloor$ and $m = \lfloor (n-k-1)/2 \rfloor$, i.e., the $2n + 1$ triangles of T_n are as equally distributed to T_k , T_m , and $T_{n-k-m-1}$ as possible.

Fig. 5 shows the running time of the different parsers applied to T_n with varying value n , measured on an iMac 2017, 4.2 GHz Intel Core i7, OpenJDK 12.0.1 with standard configuration, and is shown in milliseconds on the y -axis while n is shown on the x -axis. Note the substantial speed-up when using memoization (called “Memo” in the legend) compared to the plain GPSR parser (called “GPSR”). In fact, the GPSR parser using memoization allows to parse Sierpinski graphs which cannot be parsed in practice by the other two parsers.

Moreover, we reconsider the examples of series-parallel graphs and structured flowcharts, which we have used in [7]:

The following rules generate series-parallel graphs [6, p. 99]:

$$\begin{array}{cccc} Z^\varepsilon \xrightarrow{0} G^{xy} & G^{xy} \xrightarrow{1} e^{xy} & G^{xy} \xrightarrow{2} G^{xy} G^{xy} & G^{xy} \xrightarrow{3} G^{xz} G^{zy} \end{array}$$

⁷ Homepage: www.unibw.de/inf2/diagen

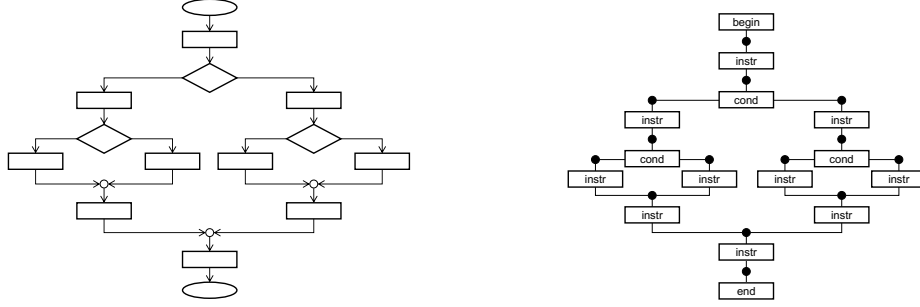


Fig. 6. A structured flowchart and its graph representation.

Structured flowcharts are flowcharts that do not allow arbitrary jumps, but represent structured programs with conditional statements and while loops. They consist of rectangles containing instructions, diamonds that indicate conditions, and ovals indicating begin and end of the program. Arrows indicate control flow; see Fig. 6 for an example (text within the blocks has been omitted). Flowcharts are easily represented by graphs as also shown in Fig. 6. The following rules generate all graphs representing structured flowcharts:

$$\begin{aligned}
 Z^\varepsilon &\rightarrow \text{begin}^x P^{xy} \text{end}^y \\
 P^{xy} &\rightarrow S^{xy} \mid P^{xz} S^{zy} \\
 S^{xy} &\rightarrow \text{instr}^{xy} \mid \text{cond}^{xuv} P^{uy} P^{vy} \mid \text{cond}^{xuy} P^{ux}
 \end{aligned}$$

None of these grammars is PSR because their CFAs have conflicts. We used these examples in [7] to compare GPSR parsers with CYK parsers. We extend these experiments here and additionally compare these parsers with a GPSR parser using memoization.

As in [7], we employ GPSR parsers with two different strategies for series-parallel graphs and for structured flowcharts. GPSR 1 employs a breadth-first strategy whereas GPSR 2 applies a more sophisticated strategy. It requires grammar rules to be annotated with either first or second priority. The GPSR 2 parser for series-parallel graphs gives rule 3 (series) precedence over rule 2 (parallel) whereas the GPSR 2 parser for structured flowcharts gives sequences priority over conditional statements.

Running time of the parsers has been measured for series-parallel graphs S_n as shown in Fig. 7 and for flowcharts F_n defined in Fig. 8. Each F_n consists of n conditions and $3n + 1$ instructions. The flowchart in Fig. 6 is in fact F_3 . F_n has a subgraph D_n , which, for $n > 0$, contains subgraphs D_m and $D_{m'}$ with $n = m + m' + 1$. Note that the conditions in F_n form a binary tree with n nodes when we ignore instructions. We always choose m and m' such that it is a complete binary tree.

Fig. 9 shows the running time of the different parsers applied to S_n and F_n with varying value n on the same platform as for Sierpinski graphs. The experiments again show that the GPSR parser with memoization is substantially faster

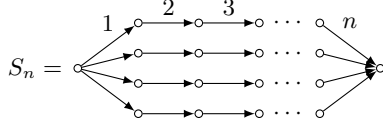


Fig. 7. Definition of series-parallel graphs S_n .

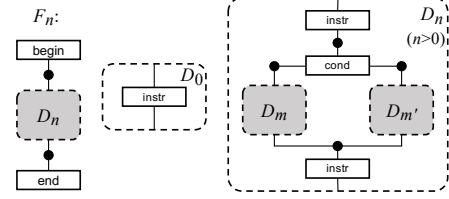


Fig. 8. Definition of flowchart graphs F_n .

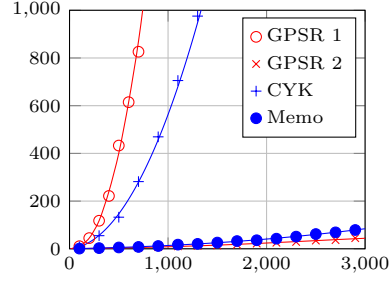
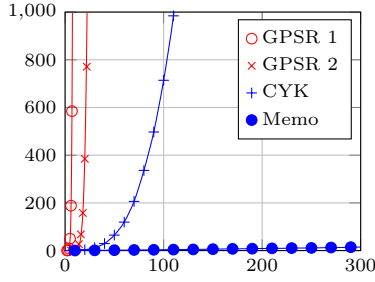


Fig. 9. Running time (in ms) of different parsers analyzing series-parallel graphs S_n (left) and structured flowcharts F_n (right) with varying value n .

than the CYK faster and even more faster than most of the GPSR parsers. Only GPSR 2 for structured flowcharts is a bit faster than the memoization parser because it need not maintain the memo store. But note that realizing the hand-tailored strategy for the GPSR 2 parser required additional programming work, whereas the memoization parser has been generated by the GRAPPA distiller without any further manual work.

7 Conclusions

We have proposed to use memoization to make GPSR parsing faster by memorizing nonterminal edges that have been created in the search process and that are discarded by plain GPSR parsing although this information could be reused later. Our experiments with three example languages (Sierpinski graphs, series-parallel graphs, and structured flowcharts) have shown that GPSR parsing with memoization is in fact substantially faster for these examples. However, memoization is not a silver bullet. It cannot speed up GPSR parsing when analyzing invalid input graphs. In these cases, they must completely traverse the entire search space, essentially falling back to plain GPSR parsing. The same applies if one is not only interested in one successful parse, but in all parses if the input graph is ambiguous.

Memoization techniques have also been used to speed up GLR parsers for strings; Kipps improved the original GLR algorithm from $O(n^{k+1})$ to $O(n^3)$ (see Sect. 1) using memoization [8]. And this speed-up is independent of the input

string being valid or invalid. But memoization for GLR parsing differs entirely from memoization for GPSR parsers proposed here: A GLR parser searches all parsers of the input graph in parallel, and all these “parsing processes” are synchronized by reading one input string token after the other. Memoization helps to speed up reduce steps in the graph-structured stack. A GPSR parser, instead, must try different “reading sequences” of the input graph, and memoization helps to reuse information that has been found earlier in a different reading sequence.

In future work, we will apply GPSR parsing with memoization to examples from natural language processing, in particular for parsing *Abstract Meaning Representations* (AMR) [1]. PSR parsing cannot be applied there because almost all grammars are ambiguous in this field. In particular, we would like to compare our parser with the state of the art in this field, i.e., the Bolinas parser [2] by D. Chiang, K. Knight *et al.* that implements the polynomial algorithm for HR grammars devised in [10] and the s-graph parser [5] by A. Koller *et al.*

References

1. Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., Schneider, N.: Abstract meaning representation for sembanking. In: Proc. 7th Linguistic Annotation Workshop at ACL 2013 Workshop. pp. 178–186 (2013)
2. Chiang, D., Andreas, J., Bauer, D., Hermann, K.M., Jones, B., Knight, K.: Parsing graphs with hyperedge replacement grammars. In: Proc. 51st Ann. Meeting of the Assoc. for Computational Linguistic (Vol. 1: Long Papers). pp. 924–932 (2013)
3. Drewes, F., Hoffmann, B., Minas, M.: Approximating Parikh images for generating deterministic graph parsers. In: Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops. LNCS 9946, pp. 112–128 (2016)
4. Drewes, F., Hoffmann, B., Minas, M.: Formalization and correctness of predictive shift-reduce parsers for graph grammars based on hyperedge replacement. *Journal of Logical and Algebraic Methods in Programming* **104**, 303–341 (Apr 2019).
5. Groschwitz, J., Koller, A., Teichmann, C.: Graph parsing with s-graph grammars. In: Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics, ACL 2015, Vol. 1: Long Papers. pp. 1481–1490 (2015)
6. Habel, A.: Hyperedge Replacement: Grammars and Languages, LNCS 643. Springer (1992)
7. Hoffmann, B., Minas, M.: Generalized predictive shift-reduce parsing for hyperedge replacement graph grammars. In Proc. LATA 2019, LNCS 11417, pp. 233–245.
8. Kipps, J.R.: GLR parsing in time $O(n^3)$. In: Tomita, M. (ed.) Generalized LR Parsing, pp. 43–59. Springer US, Boston, MA (1991).
9. Knuth, D.E.: On the translation of languages from left to right. *Information and Control* **8**(6), 607 – 639 (1965)
10. Lautemann, C.: The complexity of graph languages generated by hyperedge replacement. *Acta Informatica* **27**, 399–421 (1990)
11. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* **44**(2), 157–180 (2002)
12. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proc. of the 9th Int. Joint Conf. on Artificial Intelligence. pp. 756–764 (1985)
13. Younger, D.H.: Recognition and parsing of context-free languages in time n^3 . *Information and Control* **10**(2), 189–208 (1967)

Rule-based graph repair^{*}

Annegret Habel, Christian Sandmann

Universität Oldenburg
`{habel,sandmann}@informatik.uni-oldenburg.de`

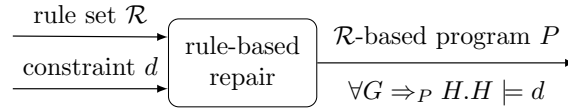
Abstract. Model repair is an essential topic in model-driven engineering. Since models are suitably formalized as graph-like structures, we consider the problem of rule-based graph repair: Given a rule set and a graph constraint, try to construct a graph program based on the given set of rules, such that the application to any graph yields a graph satisfying the graph constraint. We show the existence of rule-based repair programs for specific constraints compatible with the rule set.

1 Introduction

In model-driven software engineering the primary artifacts are models, which have to be consistent wrt. a set of constraints (see e.g. [EEGH15]). These constraints can be specified by the Object Constraint Language (OCL). To increase the productivity of software development, it is necessary to automatically detect and resolve inconsistencies arising during the development process, called model repair (see, e.g. [NEF03,MTC17,NKR17]). Since models can be represented as graph-like structures [BET12] and a subset of OCL constraints can be represented as graph conditions [RAB⁺18], we investigate graph repair and rule-based graph repair.

In [HS18], the problem of *graph repair* is considered: Given a graph constraint d , we derive a rule set $\mathcal{R}(d)$ from the constraint d and try to construct a graph program using this rule set, called *repair program*. The repair program is constructed, such that the application to any graph yields a graph satisfying the graph constraint. In this paper, we consider the problem of *rule-based graph repair*: Given a set of rules \mathcal{R} and a constraint d , try to construct a repair program P based on the rule set \mathcal{R} , i.e. we equip the rules of \mathcal{R} with application conditions [HP09], with interface [Pen09], and a context. The repair program in [HS18], and an $\mathcal{R}(d)$ -based repair program for d are equal.

Rule-based repair problem

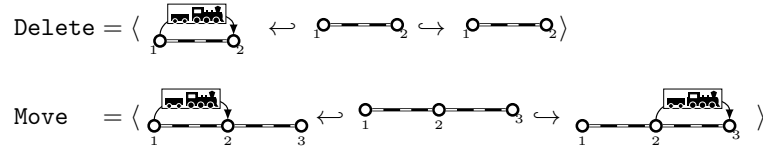


^{*} This work is partly supported by the German Research Foundation (DFG), Grants HA 2936/4-2 and TA 2941/3-2 (Meta-Modeling and Graph Grammars: Generating Development Environments for Modeling Languages).

If a graph G is generated by a grammar with rule set \mathcal{R} , then, after the application of an \mathcal{R} -based program, the result can be generated by the grammar, too.

We show that, given a rule-system compatible with a “proper” constraint without conjunctions and disjunctions, a rule-based repair program for the constraint can be constructed. Unfortunately, compatibility is undecidable. Fortunately, it turns out to be semi-decidable. We illustrate our approach by a small railroad system.

Example 1 (railroad system). The specification of a railroad system is given in terms of graphs, rules (for moving the trains), and constraints. The basic items are waypoints, bi-directional tracks and trains. The static part of the system is given by a directed rail net graph: tracks are modeled by undirected edges (or a pair of directed edges, respectively) and trains are modeled by edges. Source and target nodes of a train edge encode the train’s position on the track and the direction of its movement. The dynamic part of the system is specified by graph transformation rules. The rules model the deletion and movement of trains thereon.



The structure of the paper is as follows. In Section 2, we review the definitions of graphs, graph conditions, and graph programs. In Section 3, we introduce rule-based repair programs, show that there are rule-based repair programs for so-called proper conditions (without conjunctions and disjunctions) compatible with a rule set, and show that compatibility is semi-decidable. In Section 4, we present some related concepts. In Section 5, we give a conclusion and mention some further work.

2 Preliminaries

In the following, we recall the definitions of directed, labelled graphs, graph conditions, rules, and graph programs [EEPT06,HP09].

A directed, labelled graph consists of a set of nodes and a set of edges where each edge is equipped with a source and a target node and where each node and edge is equipped with a label.

Definition 1 (graphs & morphisms). A *(directed, labelled) graph* (over a label alphabet \mathcal{L}) is a system $G = (V_G, E_G, s_G, t_G, l_{V,G}, l_{E,G})$ where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*, $s_G, t_G: E_G \rightarrow V_G$ are total functions assigning *source* and *target* to each edge, and $l_{V,G}: V_G \rightarrow \mathcal{L}$, $l_{E,G}: E_G \rightarrow \mathcal{L}$ are total labeling functions. If $V_G = \emptyset$, then G is the *empty graph*, denoted by \emptyset .

A graph is *unlabelled* if the label alphabet is a singleton. Given graphs G and H , a (graph) *morphism* $g: G \rightarrow H$ consists of total functions $g_V: V_G \rightarrow V_H$ and $g_E: E_G \rightarrow E_H$ that preserve sources, targets, and labels, that is, $g_V \circ s_G = s_H \circ g_E$, $g_V \circ t_G = t_H \circ g_E$, $l_{V,G} = l_{V,H} \circ g_V$, $l_{E,G} = l_{E,H} \circ g_E$. The morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is injective and surjective. In the latter case, G and H are *isomorphic*, which is denoted by $G \cong H$. An injective morphism $g: G \hookrightarrow H$ is an *inclusion morphism* if $g_V(v) = v$ and $g_E(e) = e$ for all $v \in V_G$ and all $e \in E_G$.

Graph conditions are nested constructs, which can be represented as trees of morphisms equipped with quantifiers and Boolean connectives. Graph conditions and first-order graph formulas are expressively equivalent [HP09].

Definition 2 (graph conditions). A (graph) *condition* over a graph A is of the form (a) **true** or (b) $\exists(a, c)$ where $a: A \hookrightarrow C$ is a proper inclusion morphism¹ and c is a condition over C . For conditions c, c_i ($i \in I$ for some finite index set I) over A , $\neg c$ and $\bigwedge_{i \in I} c_i$ are conditions over A . Conditions over the empty graph \emptyset are called *constraints*. In the context of rules, conditions are called *application conditions*.

Graph conditions may be written in a more compact form: $\exists a := \exists(a, \mathbf{true})$, **false** $:= \neg \mathbf{true}$ and $\forall(a, c) := \neg \exists(a, \neg c)$. The expressions $\bigvee_{i \in I} c_i$ and $c \Rightarrow c'$ are defined as usual. For an inclusion morphism $a: A \hookrightarrow C$ in a condition, we just depict the codomain C , if the domain A can be unambiguously inferred.

Definition 3 (semantics). Any injective morphism $p: A \hookrightarrow G$ *satisfies* **true**. An injective morphism p *satisfies* $\exists(a, c)$ with $a: A \hookrightarrow C$ if there exists an injective morphism $q: C \hookrightarrow G$ such that $q \circ a = p$ and q satisfies c .

An injective morphism p *satisfies* $\neg c$ if p does not satisfy c , and p *satisfies* $\bigwedge_{i \in I} c_i$ if p satisfies each c_i ($i \in I$). We write $p \models c$ if p satisfies the condition c (over A). A graph G *satisfies* a constraint c , $G \models c$, if the morphism $p: \emptyset \hookrightarrow G$ satisfies c . $\llbracket c \rrbracket$ denotes the class of all graphs satisfying c . A constraint c is *satisfiable* if there is a graph G that satisfies c . Two conditions c and c' over A are *equivalent*, denoted by $c \equiv c'$, if for all graphs G and all injective morphisms $p: A \hookrightarrow G$, $p \models c$ iff $p \models c'$. A condition c *implies* a condition c' , denoted by $c \Rightarrow c'$, if for all graphs and all injective morphisms $p: A \hookrightarrow G$, $p \models c$ implies $p \models c'$.

Definition 4 (conditions with alternating quantifiers). Conditions of the form $Q(a_1, \overline{Q}(a_2, Q(a_3, \dots)))$ with $Q \in \{\forall, \exists\}$, $\overline{\forall} = \exists$, $\overline{\exists} = \forall$ ending with a condition of the form $\exists b$ or $\neg \exists b$ are conditions with *alternating quantifiers*.

Fact 1 (alternating quantifiers [HS18]). For every condition, an equivalent condition with alternating quantifiers can be constructed.

¹ Without loss of generality, we may assume that for all inclusion morphisms $a: A \hookrightarrow C$ in the condition, A is a proper subgraph of C .

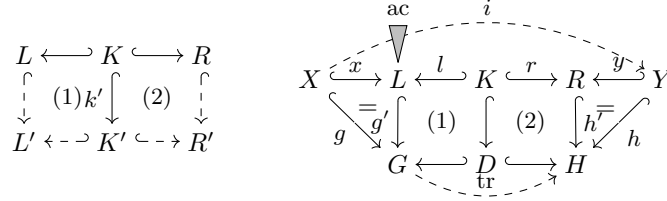
Definition 5 (proper conditions). A condition with alternating quantifiers is *proper* if it ends with an existential condition $\exists b$, or it is a condition of the form $\exists(a, \forall(b, \text{false})) \equiv \exists(a, \nexists b)$.

Fact 2 ([HS18]). Proper conditions are satisfiable.

Plain rules are specified by a pair of injective graph morphisms. For restricting the applicability of rules, the rules are equipped with a left application condition. Moreover, they may be equipped with a left- and a right interface. By the interfaces, it becomes possible to hand over information between the transformation steps.

Definition 6 (rules with context & interface).

1. A *plain rule* $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ consists of two inclusion morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$.
2. The rule p *equipped with context* $k': K \hookrightarrow K'$ is $p[k'] = \langle L' \hookleftarrow K' \hookrightarrow R' \rangle$ where L' and R' are constructed as pushouts of $L \hookleftarrow K \hookrightarrow K'$ and $R \hookleftarrow K \hookrightarrow K'$, respectively.
3. A *rule* $\varrho = \langle x, p, \text{ac}, y \rangle$ (with interfaces X and Y) consists of a plain rule $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ with left application condition ac and two injective morphisms $x: X \hookrightarrow L$, $y: Y \hookrightarrow R$, called the *left and right interface*.



An interface is *empty* if the domain of the interface morphism is empty.

A triple $\langle g, h, i \rangle$ with partial morphism $i = y^{-1} \circ r \circ l^{-1} \circ x$ is in the semantics of ϱ , denoted by $\llbracket \varrho \rrbracket$, if there is an injective morphism $g': L \hookrightarrow G$ such that $g = g' \circ x$ and $g' \models \text{ac}$, $G \Rightarrow_{p, g', h'} H$, and $h = h' \circ y$. This is denoted by $G \Rightarrow_{\varrho, g, h, i} H$ or short $G \Rightarrow_{\varrho} H$, and called *direct transformation*. Given graphs G, H and a finite set \mathcal{R} of rules, G *derives* H by \mathcal{R} if $G \cong H$ or there is a sequence of direct transformations $G = G_0 \Rightarrow_{\varrho_1} \dots \Rightarrow_{\varrho_n} G_n = H$ with $\varrho_1, \dots, \varrho_n \in \mathcal{R}$. In this case, we write $G \Rightarrow_{\mathcal{R}}^* H$ or just $G \Rightarrow^* H$.

The application of such a rule to a graph amounts to the following steps: Let $g: X \hookrightarrow G$ be given.

- (1) Select a match $g': L \hookrightarrow G$ such that $g = g' \circ x$ and $g' \models \text{ac}$.
- (2) Apply the plain rule p at g' (possibly yielding a comatch $h': R \hookrightarrow H$).
- (3) Unselect $h: Y \hookrightarrow H$, i.e., define $h = h' \circ y$.

The application of a plain rule is as in the double-pushout approach [EEPT06].

Notation. Empty interfaces and **true** application conditions are not written. If both interfaces of $\varrho = \langle x, p, \text{ac}, y \rangle$ are empty, we write $\varrho = \langle p, \text{ac} \rangle$. If additionally $\text{ac} = \text{true}$, we write $\varrho = \langle p \rangle$ or short p and speak of the underlying *plain* rule. A plain rule $p = \langle L \leftarrow K \hookrightarrow R \rangle$ sometimes is denoted by $L \Rightarrow R$ where indexes in L and R refer to the corresponding nodes. The rules $\varrho = \langle x, \text{ac} \rangle$ and $\langle y \rangle$ are denoted by $\text{Sel}(x, \text{ac})$ (selection of additional elements) and $\text{Uns}(y)$ (unselection of selected elements). Additionally, $\text{Sel}(x)$ abbreviates $\text{Sel}(x, \text{true})$.

With every transformation $t: G \Rightarrow^* H$ a partial track morphism can be associated that “follows” the items of G through the transformation: this morphism is undefined for all items in G that are removed by t , and maps all other items to the corresponding items in H .

Definition 7 (track morphism). The *track morphism* $\text{tr}_{G \Rightarrow H}$ from G to H is the partial morphism² defined by $\text{tr}_{G \Rightarrow H}(x) = \text{inc}_H(\text{inc}_G^{-1}(x))$ if $x \in D$ and *undefined* otherwise, where $\text{inc}_G = \text{inc} \circ \text{inc}_{G'}$ and $\text{inc}_G^{-1}: \text{inc}_G(D) \hookrightarrow D$ is the inverse of inc_G . Given a transformation $G \Rightarrow^* H$, $\text{tr}_{G \Rightarrow^* H}$ is defined by induction on the length of the transformation: $\text{tr}_{G \Rightarrow^* H} = \text{iso}$ for an isomorphism $\text{iso}: G \xrightarrow{\sim} H$, and $\text{tr}_{G \Rightarrow^* H} = \text{tr}_{G' \Rightarrow H} \circ \text{tr}_{G \Rightarrow^* G'}$ for $G \Rightarrow^* H = G \Rightarrow^* G' \Rightarrow H$.

Conditions can be “shifted” over morphisms and rules.

Lemma 1 (Shift, Left [HP09]). There are constructions Shift and Left, such that the following holds. For each condition d over A and injective morphisms $b: A \hookrightarrow R, n: R \hookrightarrow H, n \circ b \models d \iff n \models \text{Shift}(b, d)$. For each rule $p = \langle L \leftarrow K \hookrightarrow R \rangle$ and each condition ac over R , for each $G \Rightarrow_{p,g,h} H, g \models \text{Left}(p, \text{ac}) \iff h \models \text{ac}$.

Construction 1. The construction is as follows.

$$\begin{array}{ll}
\begin{array}{c} A \xhookrightarrow{b} R \\ a \downarrow \quad (0) \quad \downarrow a' \\ C \xhookrightarrow{b'} R' \\ \Delta_d \end{array} & \begin{array}{l} \text{Shift}(b, \text{true}) := \text{true}. \\ \text{Shift}(b, \exists(a, d)) := \bigvee_{(a', b') \in \mathcal{F}} \exists(a', \text{Shift}(b', d)) \text{ where} \\ \mathcal{F} = \{(a', b') \mid b' \circ a = a' \circ b, a', b' \text{ inj, } (a', b') \text{ jointly surjective}^3\} \\ \text{Shift}(b, \neg d) := \neg \text{Shift}(b, d), \text{ Shift}(b, \bigwedge_{i \in I} d_i) := \bigwedge_{i \in I} \text{Shift}(b, d_i). \end{array} \\
\\
\begin{array}{c} R \xleftarrow{\quad} K \xrightarrow{\quad} L \\ a \downarrow \quad (1) \quad \downarrow \quad (2) \quad \downarrow a' \\ R' \xleftarrow{\quad} K' \xrightarrow{\quad} L' \\ \Delta_{\text{ac}} \end{array} & \begin{array}{l} \text{Left}(p, \text{true}) := \text{true}. \\ \text{Left}(p, \exists(a, \text{ac})) := \exists(a', \text{Left}(p', \text{ac})) \text{ if } p^{-1} \text{ }^4 \text{ is applicable} \\ \text{w.r.t. the morphism } a, p' := \langle L' \leftarrow K' \hookrightarrow R' \rangle \text{ is the } \textit{derived} \\ \text{rule, and } \text{false, otherwise.} \\ \text{Left}(p, \neg \text{ac}) := \neg \text{Left}(p, \text{ac}), \quad \text{Left}(p, \bigwedge_{i \in I} \text{ac}_i) := \\ \bigwedge_{i \in I} \text{Left}(p, \text{ac}_i). \end{array}
\end{array}$$

² A *partial* graph morphism $G \rightarrow H$ is an injective morphism $S \hookrightarrow H$ such that $S \subseteq G$.

³ A pair (a', b') is *jointly surjective* if for each $x \in R'$ there is a preimage $y \in R$ with $a'(y) = x$ or $z \in C$ with $b'(z) = x$.

Graph programs [HP01] are made of sets of rules with application conditions, sequential composition, and as-long-as possible iteration. More specifically, we consider graph programs with interface [Pen09] that are capable of handling over a selection of elements between computation steps.

Definition 8 (graph programs with interface). The set of (*graph*) *programs with interface* X , $\text{Prog}(X)$, is defined inductively:

- (1) Every rule with interface X is in $\text{Prog}(X)$.
- (2) If $P \in \text{Prog}(X)$ and $Q \in \text{Prog}^5$, then $\langle P; Q \rangle \in \text{Prog}(X)$.
- (3) If $P, Q \in \text{Prog}(X)$, then $\{P, Q\}$, $P \downarrow$, and $\text{try } P$ are in $\text{Prog}(X)$.

Definition 9 (semantic of programs with interface). The *semantics* of a program P with interface X , denoted by $\llbracket P \rrbracket$, is a set of triples such that - for all $\langle g, h, i \rangle \in \llbracket P \rrbracket$, $X = \text{dom}(g) = \text{dom}(i)$ ⁶ and $\text{dom}(h) = \text{ran}(i)$ - and is defined as follows:

$$\begin{aligned} \llbracket \langle P; Q \rangle \rrbracket &= \{ \langle g_1, h_2, i_2 \circ i_1 \rangle \mid \langle g_1, h_1, i_1 \rangle \in \llbracket P \rrbracket, \langle g_2, h_2, i_2 \rangle \in \llbracket Q \rrbracket \text{ and } h_1 = g_2 \} \\ \llbracket \{P, Q\} \rrbracket &= \llbracket P \rrbracket \cup \llbracket Q \rrbracket \\ \llbracket P \downarrow \rrbracket &= \{ \langle g, h, \text{id} \rangle \in P^* \mid \nexists h'. \langle h, h', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket \} \end{aligned}$$

where $P^0 = \text{Skip}$, $P^j = \langle \text{Fix}(P); P^{j-1} \rangle$ with $j > 0$, $\llbracket P^* \rrbracket = \bigcup_{j=0}^{\infty} \llbracket P^j \rrbracket$, and $\llbracket \text{Fix}(P) \rrbracket = \{ \langle g, h \circ i, \text{id} \rangle \mid \langle g, h, i \rangle \in \llbracket P \rrbracket \}$. If $\langle g, h, i \rangle \in \llbracket P \rrbracket$ with $g: X \hookrightarrow G$, $h: Y \hookrightarrow H$, and partial morphism $i: X \hookrightarrow Y$, we write $G \Rightarrow_{P, g, h, i} H$ or short $G \Rightarrow_P H$.

Additionally, we introduce the **try** statement as in [PP13]. Intuitively, **try** keeps the changes of the program, if the execution is successful, and discards them, otherwise. The semantics of **try** is given in the style of structural operational semantics, and can be found in [PP13]. The statement **Skip** is the identity element $\text{Sel}(\text{id}, \text{true})$ of sequential composition.

In the double-pushout approach, the *dangling condition* for a rule $\varrho = \langle L \Rightarrow R \rangle$ and an injective morphism $g: L \hookrightarrow G$ requires: “No edge in $G - g(L)$ is incident to a node in $g(L - K)$ ”. For this condition, there is a program such that after application the dangling condition is satisfied.

Fact 3 (program for deleting dangling edges [HS18]). For every node-deleting⁷ rule ϱ with left-hand side L and every match $g: L \hookrightarrow G$, there is an edge-deleting program $P_{\text{dg}}(\varrho)$, such that $\forall G \Rightarrow_{P_{\text{dg}}(\varrho)} H$, with partial track morphism $\text{tr}: G \hookrightarrow H$, the rule ϱ becomes applicable at $\text{tr} \circ g$.

⁴ For a rule $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$, $p^{-1} = \langle R \hookleftarrow K \hookrightarrow L \rangle$ denotes the *inverse* rule. For $L' \Rightarrow_p R'$ with intermediate graph K' , $\langle L' \hookleftarrow K' \hookrightarrow R' \rangle$ is the *derived* rule.

⁵ Programs with arbitrary interface are denoted by Prog .

⁶ For a morphism m , $\text{dom}(m)$ denotes the domain, $\text{ran}(m)$ denotes the codomain of m .

⁷ A rule $p = \langle L \hookleftarrow K \hookrightarrow R \rangle$ is *node-preserving* if $|V_L| = |V_K|$. It is *node-deleting* if it is not node-preserving, i.e. $|V_L| > |V_K|$.

Definition 10 (rule-based programs). Given a set of rules \mathcal{R} , a program P is \mathcal{R} -based, if all rules in the program are rules in \mathcal{R} equipped with context, application condition, and interface. Additionally, the empty program **Skip** is \mathcal{R} -based.

Example 2. The rule **Build** = $\langle \text{ } \circ \quad \circ_2 \leftarrow \circ \quad \circ_2 \hookrightarrow \circ \text{---} \circ_2 \text{ } \rangle$ equipped with context $k': \circ \quad \circ \hookrightarrow \text{ } \circ \text{---} \circ \text{ } \circ$, and application condition $\text{ac} = \# \text{ } \circ \text{---} \circ \wedge \exists \text{ } \circ \text{---} \circ$, yields to the \mathcal{R} -based program $P = \text{try Build2}$, where

$$\text{Build2} = \langle \text{ } \circ \text{---} \circ \leftarrow \text{ } \circ \text{---} \circ \hookrightarrow \text{ } \circ \text{---} \circ, \# \text{ } \circ \text{---} \circ \wedge \exists \text{ } \circ \text{---} \circ \text{ } \rangle.$$

The program **Delete** \downarrow is not $\{\text{Move}\}$ -based.

Remark. For every transformation $G \Rightarrow_P H$ by an \mathcal{R} -based program P , there is a transformation $G \Rightarrow_{\mathcal{R}}^* H$.

3 Rule-based graph repair

A repair program for a constraint is a graph program with the property that any application to a graph yields a graph satisfying the constraint. More generally, we consider repair programs for conditions.

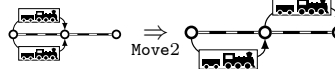
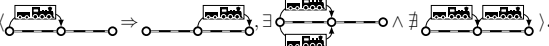
Definition 11 (repair programs). Given a constraint d , a program P with empty interface is a *repair program* for d if, for all transformations $G \Rightarrow_P H$, $H \models d$. Given a condition ac over A , a program P with interface A is a *repair program* for ac if, for all transformations $G \Rightarrow_{P,g,h,i} H$, i preserves A , i.e. $i(A) = A$, and $h \circ i \models \text{ac}$.


Remark. There is a close relationship to Hoare triples. A repair program for constraints c and d is a program such that, for all $G \Rightarrow_P H$, $G \models c$ implies $H \models d$. Then, P is a repair program for c, d if and only if $\{c\}P\{d\}$.

Definition 12 (ac-resulting transformations). Given a condition ac over A , a transformation $t: G_0 \Rightarrow_{\mathcal{R}}^* H_0$ with $x: A \hookrightarrow G_0$ is *ac-resulting* if the transformation t *preserves* A , i.e., $\text{tr}_0(x(A)) = A$, and $\text{tr}_0 \circ x \models \text{ac}$ ⁸. It is *minimal* if there is no ac-resulting transformation $t': G'_0 \Rightarrow_{\mathcal{R}}^* H'_0$ with $G'_0 \subset G_0$.

Given a condition ac over A , and an ac-resulting transformation, we construct a repair program inductively as follows: If the transformation step yields to the satisfaction of ac , we equip the rule ρ with an application condition ac_0 , such that ρ becomes applicable provided there is a violation of ac , and the transformation step is ac-guaranteeing. If the transformation step does not directly yield to the

⁸ $\text{tr}_0: G_0 \hookrightarrow H_0$ denotes the partial track morphism from G_0 to H_0 .

For the rule set $\mathcal{R}_2 = \{\text{Move}, \text{Delete}\}$, and the constraint **NoTwo**, there is a **NoTwo**-resulting transformation  via the \mathcal{R}_2 -based program **Move2** : .

Example 4 (no railroad repair). Consider the railroad system in Example 1 with the rules **Move** and **Delete** and the constraint **Station** = \exists  (there exists a train station). Whenever the start graph has no station, then no station can be created by a $\{\text{Move}, \text{Delete}\}$ -based program. The reason is that the labels of the constraint do not occur in the right-hand sides of the rules.

Proof. Let $t : G_0 \Rightarrow_{\mathcal{R}}^* H_0$ be ac-resulting with match $x : A \hookrightarrow G_0$ and condition $\text{ac} = \exists a (\nexists a)$ with $a : A \hookrightarrow C$.

1. Since \mathcal{R} is closed under edge deletion, P_{dg} is \mathcal{R} -based, and, by Construction, $P(t)$ is an \mathcal{R} -based program. $P(t)$ is also a repair program for ac : Let $G \Rightarrow_{P(t), g, h, i} H$ be an arbitrary transformation. Since t is minimal, there is no transformation $t' : G'_0 \Rightarrow_{\mathcal{R}}^* H'_0$, thus, there exists a morphism $g' : G_0 \hookrightarrow G$. By the definition of the application of a rule with interface (Definition 6), $h \circ i \models \text{ac}_0$. By Lemma 1, and the fact that $\text{tr}_0 \circ x = y \circ i$, $h = h' \circ y$, $g' \models \text{Left}(\bar{\varphi}, \text{Shift}(\text{tr}_0 \circ x, \text{ac}_0)) \Leftrightarrow h' \models \text{Shift}(\text{tr}_0 \circ x, \text{ac}_0) \Leftrightarrow h' \circ \text{tr}_0 \circ x \models \text{ac} \Leftrightarrow h' \circ y \circ i \models \text{ac} \Leftrightarrow h \circ i \models \text{ac}$.

$$\begin{array}{ccccc}
 & L & \Longrightarrow & R & \\
 & \downarrow & & \downarrow & \\
 A & \xrightarrow{x} & G_0 & \xRightarrow{\text{tr}_0} & H_0 & \xleftarrow{y} & A \\
 & \searrow g & \downarrow g' & & \downarrow h' & \swarrow h & \\
 & & G & \xRightarrow{\text{tr}} & H & &
 \end{array}$$

2. Let $t_{n+1} : G_0 \Rightarrow_{\mathcal{R}}^{n+1} G_{n+1}$ be an ac-resulting transformation with match $x : A \hookrightarrow G_0$ with subtransformations $t_n : G_0 \Rightarrow_{\mathcal{R}}^n G_n$ and $t : G_n \Rightarrow_{\mathcal{R}} G_{n+1}$ with match $x_2 : \text{ran}(\text{tr}_0 \circ x) \hookrightarrow G_n$. Then, there are two cases:

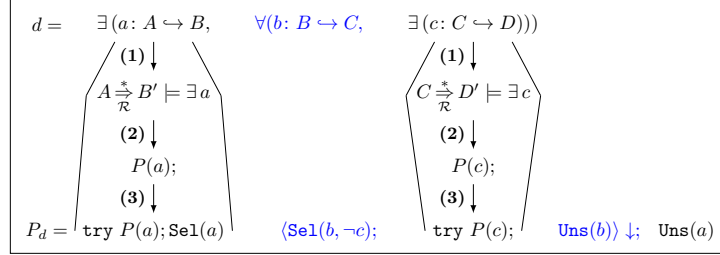
- (A) t_n is \neg ac-resulting. Then t is ac-resulting and, by inductive hypothesis, there are repair programs $P(t_n)$ with interface A for \neg ac and $P(t)$ for ac.
- (B) t_n is ac-resulting. Then t is ac-resulting and, by inductive hypothesis, there are repair programs $P(t_n)$ and $P(t)$ for ac.

Then $P(t_{n+1}) := P(t_n); \text{Uns}(G_n \hookleftarrow A); \text{Sel}(A \hookrightarrow G_{n+1}); P(t)$ is an \mathcal{R} -based repair program for ac . \square

To get a repair program for proper conditions, we automatically break the condition into conditions of the form $\exists (A \hookrightarrow C)$ and $\nexists (A \hookrightarrow C)$.

- (1) Try to find ac-resulting transformations via the rule set for the small conditions if this is possible, i.e., the rule set is “compatible” with the condition.

- (2) Construct a \mathcal{R} -based program $P(a)$, according to Lemma 2.
- (3) Combine the programs by the construction of interfaces to a program for the condition d (see figure below).



Given a condition d , $\mathcal{M}_{\exists}(d)$ ($\mathcal{M}_{\#}(d)$) denotes the sets of injective morphisms in d occurring behind the existential (negated existential) quantifier and $\mathcal{M}(d)$ their union.

Definition 14 (compatibility). Given a set of rules \mathcal{R} and a constraint d , \mathcal{R} is d -compatible if, for all morphisms $a: A \hookrightarrow C$ in $\mathcal{M}(d)$, there is a set of ac-resulting transformations $\mathcal{T}(a)$ via rules of \mathcal{R} . If \mathcal{R} consists of a single rule ϱ , we say that ϱ is d -compatible instead of \mathcal{R} is d -compatible.

Theorem 1 (Repair). For proper conditions d and a rule set \mathcal{R} compatible with d , an \mathcal{R} -based repair program for d can be constructed.

Construction 3. The program P_d for d is constructed inductively as follows.

- (1) For $d = \text{true}$, $P_d = \text{Skip}$.
- (2) For $d = \exists a$, $P_d = \text{try } P(a)$.
- (3) For $d = \#a$, $P_d = P(a) \downarrow$.
- (4) For $d = \exists(a, c)$, $P_d = \langle \text{try } P(a); \text{Sel}(a); P_c; \text{Uns}(a) \rangle$.
- (5) For $d = \forall(a, c)$, $P_d = \langle \text{Sel}(a, \neg c); P_c; \text{Uns}(a) \rangle \downarrow$

where $P(a) = P(t(a))$ and P_c denotes the \mathcal{R} -based repair program for c .

Example 5. Consider the condition $d = \forall(\text{train on track}, \exists(\text{train on track}))$, meaning that each train occupies a track, i.e. there are no derailed trains.

For the rule set $\mathcal{R} = \{\text{Build} = \langle \text{train on track} \hookrightarrow \text{train on track} \quad \text{train on track} \hookrightarrow \text{train on track} \rangle\}$, and the subcondition $\exists(a: \text{train on track} \hookrightarrow \text{train on track})$ there is a $\exists a$ -resulting transformation $\text{train on track} \xRightarrow{\text{Build}} \text{train on track}$. The rule Build is equipped with the context $k': \text{train on track} \hookrightarrow \text{train on track}$, and application condition $\text{ac}_0 = \#(\text{train on track}) \wedge \exists(\text{train on track})$, yielding to the \mathcal{R} -based repair program $P_{\exists a} = \text{try Build2}$ for $\exists a$, where $\text{Build2} = \langle \text{train on track} \hookrightarrow \text{train on track} \hookrightarrow \text{train on track} \hookrightarrow \#(\text{train on track}) \wedge \exists(\text{train on track}) \rangle$.

The \mathcal{R} -based repair program for d is $P_d = \langle \text{Sel}(a_1, \#a); P_{\exists a}; \text{Uns}(a_1) \rangle \downarrow$ with $a_1 = \emptyset \hookrightarrow \text{[Diagram: a node with a train icon and a self-loop arrow]} \leftarrow \text{[Diagram: a node with a self-loop arrow]}$, which intuitively works as follows: select a train, for which there is no track, then add a track, provided there does not exist one, and unselect the train. This is done as long as possible.

Remark. The statement holds for proper conditions: $\forall (\circ_1, \exists \circ_1 \xrightarrow{\circ_2} \circ_2, \# \circ_1 \xrightarrow{\circ_2} \circ_2 \leftarrow \circ)$ is satisfiable. By Construction 3, we do not get a repair program for it. This may be handled by the deletion of nodes.

Fact 4 (repair).

- (1) If $P_{\exists a}$ is a repair program for $\exists a$, P_c a repair program for c , then $\langle P_{\exists a}; (P_c)_a \rangle$ is a repair program for $\exists(a, c)$.
- (2) If P_c is a repair program for c , then $(P'_c)_a \downarrow$ is a repair program for $\forall(a, c)$.

where $(P_c)_a = \langle \text{Sel}(a); P_c; \text{Uns}(a) \rangle$ and $(P'_c)_a = \langle \text{Sel}(a, \neg c); P_c; \text{Uns}(a) \rangle$ are programs with interface A .

Proof. Let d be a proper condition, \mathcal{R} be a rule set compatible with d , and P_d the program as in Construction 3. By Construction 3, P_d is \mathcal{R} -based. By induction on the structure of d , we show that P_d is a repair program for d :

- (1) Let $d = \text{true}$. By the semantics of **Skip**, for every transformation $G \Rightarrow_{\text{Skip}} H \cong G \models \text{true}$, i.e., **Skip** is a repair program for d .
- (2) Let $d = \exists a$ and $G \Rightarrow_{\text{try } P(a)} H$ be an arbitrary transformation. By compatibility, there is a $\exists a$ -resulting transformation t . (a) If $G \models \#a$, then, by Lemma 2 $H \models \exists a$. (b) If $G \models \exists a$, then $G \Rightarrow_{\text{try } P(a)} H \cong G \models \exists a$. Consequently, P_d is a repair program for $\exists a$.
- (3) Let $d = \#a$ and $G \Rightarrow_{P(a) \downarrow} H$ be an arbitrary transformation. By compatibility, there is a $\#a$ -resulting transformation t . By the semantics of \downarrow , $P(a)$ is not applicable to H , and by Lemma 2, $H \models \#a$, i.e., P_d is a repair program for $\#a$.
- (4) Let $d = \exists(a, c)$ and $G \Rightarrow_{P_d} H$ be an arbitrary transformation. By step (2), $\text{try } P(a)$ is a repair program for $\exists a$ with interface A . If $c = \#b$, then, by step (3), $P_c = P(b) \downarrow$ is a repair program for c . If $c \neq \#b$, then c is proper and, by induction hypothesis, P_c is a repair program for c . In both cases, by Fact 4, P_d is a repair program for $\exists(a, c)$.
- (5) Let $d = \forall(a, c)$ and $G \Rightarrow_{P_d} H$ an arbitrary transformation. Since d is proper, c is proper. By induction hypothesis, P_c is a repair program for c . By Fact 4, P_d is a repair program for $\forall(a, c)$. By the semantics of \downarrow , no rule is applicable to H , i.e., $H \models \#(a, \neg c) \equiv \forall(a, c)$. \square

It remains the question, whether compatibility is decidable.

Unfortunately, the compatibility problem is undecidable. This follows immediately from the undecidability of the coverability problem for non-deleting rule sets [BDK⁺12].

Lemma 3 (undecidability of compatibility). The compatibility problem is undecidable for non-deleting rule sets and positive constraints⁹.

Proof. The statement follows immediately from the undecidability of the coverability problem for non-deleting rule sets [BDK⁺12]. Assume the compatibility problem for non-deleting rule sets \mathcal{R} and positive constraints $\exists C$ is decidable. Then, the coverability problem for non-deleting rule sets \mathcal{R} , the empty start graph and final graph C is decidable: By definition of compatibility and covering, \mathcal{R} is $\exists C$ -compatible \iff there is a transformation $\emptyset \Rightarrow_{\mathcal{R}}^* C'.C' \models \exists C \iff$ there is a transformation $\emptyset \Rightarrow_{\mathcal{R}}^* C'.C' \sqsupseteq C$ ¹⁰ i.e., there is a covering. \square

Fortunately, the compatibility problem is semi-decidable for arbitrary rule sets and arbitrary constraints.

Lemma 4 (semi-decidability of compatibility). For every rule set \mathcal{R} and every condition d , \mathcal{R} -compatibility of d is semi-decidable.

⁹ A rule $\varrho = \langle p, \text{ac} \rangle$ with plain rule $p = \langle L \leftrightarrow K \hookrightarrow R \rangle$ is *non-deleting* if $L \cong K$. A condition of the form $\exists C$ is *positive*.

¹⁰ $H \sqsupseteq G$ if there is an injective morphism from G to H . Obviously, $H \sqsupseteq G$ iff $H \models \exists G$.

Construction 4.

```

Input: rule set  $\mathcal{R}$ , proper condition  $d$ ,  $maxRound \in \mathbb{N} \cup \{\infty\}$ 
Output: true,  $\mathcal{R}$  is  $d$ -compatible throws UndecidedException;
 $\mathcal{T}(a, 0) \leftarrow \emptyset, \quad i, j \leftarrow 1; \quad \quad \quad // \text{ initialize}$ 
/* Construct  $\mathcal{M}_{\exists}(d), \mathcal{M}_{\nexists}(d), \mathcal{M}(d)$  */
for  $a \in \mathcal{M}_{\exists}(d)$  do
  repeat
    /* Construct the set of transformations  $A \Rightarrow C'$  of length  $i$  */
     $t(a, i) \leftarrow \text{constructTrafo}(a, \mathcal{R}, i);$ 
     $\mathcal{T}(a, i) \leftarrow \mathcal{T}(a, i-1) \cup t(a, i);$ 
     $i \leftarrow i + 1;$ 
    if  $i \geq maxRound$  then
      return throws UndecidedException;
      /* If termination is requested, we cannot decide */
    end
  until  $C' \models \exists a;$ 
  /* until a repairing transformation is found */
end
for  $a \in \mathcal{M}_{\nexists}(d)$  do
  repeat
    /* .... */
  until  $A' \models \nexists a;$ 
end
return true;
/* for all  $a \in \mathcal{M}(d)$  a repairing transformation was found */

```

Proof. The algorithm either returns **true**, provided \mathcal{R} is d -compatible, or terminates with an exception, i.e. it is semi-decidable. \square

4 Related concepts

In this section, we present some related concepts on rule-based graph repair. For the related problem of model repair, there is a wide variety of different approaches. We consider only selected model repair approaches. For a more sophisticated survey on different model repair techniques, and a feature-based classification of these approaches, see [MTC17].

The notion rule-based repair is used in different meanings. In most cases [NKR17], [HS18], a rule set is derived from (a set of) constraint(s) and a repair algorithm/program is constructed from the rule set. In some cases (e.g., this paper), a rule set and a constraint are given as input and a repair program is constructed from the rule set.

In [NKR17], a rule-based approach to support the modeler in automatically trimming and completing EMF models is presented. For that, repair rules are automatically generated from multiplicity constraints imposed by a given meta-model. The rule schemes are carefully designed to consider the EMF model constraints defined in [BET12].

In [HS18], given a graph constraint, we derive a rule set $\mathcal{R}(d)$ from the constraint d and try to find a repair program using this rule set. The repair program is constructed, to repair all graphs. In this paper, we use programs with interface [Pen09] with selection and unselection of parts, instead of markings as in [HS18]. In this paper, given a rule set and graph constraint, we try to find a repair program based on the rule set. The repair program in [HS18], and an $\mathcal{R}(d)$ -based repair program for d are equal.

In [SLO19], a logic-based incremental approach to graph repair is presented, generating a sound and complete (upon termination) overview of least changing repairs. The state-based graph repair algorithm takes a graph and a graph constraint as inputs and returns a set of graph repairs. In [SLO19], one has similar repairs for similar graphs. But for each graph, one has to compute the repair.

In [TOLR17], a designer can specify a set of so-called change-preserving rules, and a set of edit rules. Each edit rule, which yields to an inconsistency, is then repaired by a set of repair rules. The construction of the repair rules is based on the complement construction. It is shown, that a consistent graph is obtained by the repair program, provided that each repair step is sequentially independent from each following edit step, and each edit step can be repaired. The repaired models are not necessarily as close as possible to the original model.

In [CCYW18], a rule-based approach for graph repair is presented. Given a set of rules, and a graph, they use this set of rules, to handle different kinds of conditions, i.e., incompleteness, conflicts and redundancies. The rules are based on seven different operations, and are not defined in the framework of the DPO-approach. They look for the best repair, which is based on the “graph edit distance”.

5 Conclusion

In [HS18], the repair programs are formed from rules derived from the given condition. They were constructed to be maximally preserving, i.e. to preserve nodes as much as possible. In this paper, we consider rule-based repair where the repair programs are constructed from a given set of small rules and a given condition. In general, we do not get maximally preserving repair programs, because the given rules may be not maximally preserving. The main problem is to restrict rule application to a certain context. This is done by programs with interface [Pen09].

It depends on the given set of rules whether we obtain a repair program.

- (1) Whenever the rule set is compatible with the proper condition, a rule-based repair program can be constructed.
- (2) The compatibility problem is undecidable for non-deleting rule sets and positive constraints and semi-decidable for arbitrary rule sets and arbitrary constraints.

Further topics are

- (1) Rule-based repair programs for all satisfiable conditions, i.e. conditions with conjunctions and disjunctions.
- (2) Characterization of rule-based repair programs. We will look for rule-based repair programs (for classes of graphs) with some quality metrics for the repair programs, e.g. minimal number of repair steps, maximal preservation of nodes and edges, minimal number of deletions / changes, For this purpose, we have to restrict the classes of graphs in consideration, e.g., to graphs with bounded node degree.
- (3) Rule-based repair programs for typed attributed graphs and EMF-models, i.e., typed, attributed graphs satisfying some constraints [NKR17].
- (4) An implementation of the rule-based graph repair approach.

Acknowledgements. We are grateful to Marius Hubatschek, Jens Kosiol, and the anonymous reviewers for their helpful comments to this paper.

References

- BDK⁺12. Nathalie Bertrand, Giorgio Delzanno, Barbara König, Arnaud Sangnier, and Jan Stückrath. On the decidability status of reachability and coverability in graph transformation systems. In *Rewriting Techniques and Applications (RTA'12)*, volume 15 of *LIPICs*, pages 101–116, 2012. Long version: Technical Report DISI-TR-11-04, Università di Genova, 2012.
- BET12. Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent emf model transformations by algebraic graph transformation. *Software and System Modeling*, 11(2):227–250, 2012.
- CCYW18. Yurong Cheng, Lei Chen, Ye Yuan, and Guoren Wang. Rule-based graph repairing: Semantic and efficient repairing methods. In *34th IEEE International Conference on Data Engineering, ICDE 2018*,, pages 773–784, 2018.
- EEGH15. Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. Springer, 2015.
- EEPT06. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer, 2006.
- HP01. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245, 2001.

- HP09. Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19:245–296, 2009.
- HS18. Annegret Habel and Christian Sandmann. Graph repair by graph programs. In *Graph Computation Models (GCM 2018)*, volume 11176 of *Lecture Notes in Computer Science*, pages 431–446, 2018.
- MTC17. Nuno Macedo, Jorge Tiago, and Alcino Cunha. A feature-based classification of model repair approaches. *IEEE Trans. Software Eng.*, 43(7):615–640, 2017.
- NEF03. Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Software Engineering*, pages 455–464. IEEE Computer Society, 2003.
- NKR17. Nebras Nassar, Jens Kosiol, and Hendrik Radke. Rule-based repair of emf models: Formalization and correctness proof. In *Graph Computation Models (GCM 2017)*, 2017. <https://www.uni-marburg.de/fb12/arbeitsgruppen/swt/forschung/publikationen/2017/NKR17.pdf>.
- Pen09. Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.
- PP13. Christopher M. Poskitt and Detlef Plump. Verifying total correctness of graph programs. *Electronic Communications of the EASST*, 61, 2013.
- RAB⁺18. Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Gabriele Taentzer. Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. *Science of Computer Programming*, 152:38–62, 2018.
- SLO19. Sven Schneider, Leen Lambers, and Fernando Orejas. A logic-based incremental approach to graph repair. In *Fundamental Approaches to Software Engineering - (FASE 2019)*, volume 11424 of *Lecture Notes in Computer Science*, pages 151–167, 2019.
- TOLR17. Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. Change-preserving model repair. In *Fundamental Approaches to Software Engineering (ETAPS 2017)*, volume 10202 of *Lecture Notes in Computer Science*, pages 283–299, 2017.

Sesqui-Pushout Rewriting: Concurrency, Associativity and Rule Algebra Framework^{*}

Nicolas Behr^[0000–0002–8738–5040]

Université de Paris, IRIF, CNRS, F-75013 Paris, France

`nicolas.behr@irif.fr`

`http://nicolasbehr.com`

Abstract. Sesqui-pushout (SqPO) rewriting is a variant of transformations of graph-like and other types of structures that fit into the framework of adhesive categories where deletion in unknown context may be implemented. We provide the first account of a concurrency theorem for this important type of rewriting, and we demonstrate the additional mathematical property of a form of associativity for these theories. Associativity may then be exploited to construct so-called rule algebras (of SqPO type), based upon which in particular a universal framework of continuous-time Markov chains for stochastic SqPO rewriting systems may be realized.

Keywords: Sesqui-Pushout rewriting · adhesive categories · rule algebras · stochastic rewriting systems · continuous-time Markov chains.

1 Motivation and relation to previous works

The framework of *Sesqui-Pushout (SqPO) rewriting* has been introduced relatively recently in [16] as a novel alternative to the pre-existing algebraic graph transformation frameworks known as *Double-Pushout (DPO)* [23,25,17,34] and *Single-Pushout (SPO) rewriting* [32,35,38]. In the setting of the rewriting of graph-like structures, the distinguishing feature of the aforementioned DPO-type rewriting is that the deletion of vertices with incident edges is only possible if the incident edges are explicitly deleted via the application of the rewriting rule. In contrast, in both the SqPO and the SPO rewriting setups, “*deletion in unknown context*” is implementable. Thus for practical applications of rewriting, in particular in view of the modeling of stochastic rewriting systems, the S(q)PO rewriting semantics provide an important additional option for the practitioners, and will thus in particular complement the existing DPO-type associative rewriting and rule algebra framework as introduced in [3,7]. Referring the interested readers to [36] for a recent review and further conceptual details of the three approaches, suffice it here to quote that SqPO and SPO rewriting via linear

^{*} This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 753750.

rules (to be defined in the following) and along monomorphic matches effectively encode the same semantics of rewriting. We chose (by the preceding argument without loss of expressivity) to develop the theory of associative rewriting within the SqPO rather than the SPO setting, since the SqPO framework bears certain close technical similarities to the DPO rewriting framework, which proved crucial in finding a strategy for the highly intricate proofs of the concurrency and associativity theorems presented in this paper. While it is well-known (see e.g. Section 5.1 of [16]) that DPO- and SqPO-type semantics coincide for certain special classes of linear rules (essentially rules that do not delete vertices), and while these cases might provide some valuable cross-checks of technical results to the experts, SqPO-type semantics is in its full generality a considerably more intricate variant of semantics due to its inherent “mixing” of pushouts with final pullback complements. It should further be noted that we must impose a set of additional assumptions on the underlying adhesive categories (see Assumption 1) in order to ensure certain technical properties necessary for our concurrency and associativity theorems to hold. To the best of our knowledge, apart from some partial results in the direction of developing a concurrency theorem for SqPO-type rewriting in [16,36,15], prior to this work neither of the aforementioned theorems had been available in the SqPO framework.

Associativity of SqPO rewriting theories plays a pivotal role in our development of a novel form of concurrent semantics for these theories, the so-called *SqPO-type rule algebras*. Previous work on associative DPO-type rewriting theories [3,5,7] (see also [8]) has led to a category-theoretical understanding of associativity that may be suitably extended to the SqPO setting. In contrast to the traditional and well-established formalisms of concurrency theory for rewriting systems (see e.g. [42,26,24,15] for DPO-type semantics and [16,15] for a notion of parallel independence and a Local Church-Rosser theorem for SqPO-rewriting of graphs), wherein the focus of the analysis is mostly on *derivation traces* and their sequential independence and parallelism properties, the focus of our rule-algebraic approach differs significantly: we propose instead to put *sequential compositions of linear rules* at the center of the analysis (rather than the derivation traces), and moreover to employ a vector-space based semantics in order to encode the non-determinism of such rule compositions. It is for this reason that the concurrency theorem plays a quintessential role in our rule algebra framework, in that it encodes the relationship between sequential compositions of linear rules and derivation traces, which in turn gives rise to the so-called *canonical representations* of the rule algebras (see Section 4). This approach in particular permits to uncover certain combinatorial properties of rewriting systems that would otherwise not be accessible. While undoubtedly not a standard technique in the realm of theoretical computer science, certain special examples of rule algebras are ubiquitous in many areas of applied mathematics and theoretical physics. The most famous such example concerns the so-called *Heisenberg-Weyl algebra* (see e.g. [9,10,11]), which is well-known to possess a representation in terms of the formal multiplication operator \hat{x} and the differentiation operator ∂_x

on formal power series in the formal variable x , with $\hat{x} x^n := x^{n+1}$ and ∂_x acting as the derivative. Referring the interested readers to Example 2 (see also [7,4]) for the precise details, it transpires that the monomials x^n (for n a non-negative integer) are found to be in one-to-one correspondence with *graph states* associated to n -vertex discrete graphs, while \hat{x} and ∂_x may be understood as the *canonical representations* of the discrete graph rewriting rules of creation and deletion of vertices. It will thus come as no surprise that considering more general rewriting rules than those of discrete graphs will lead to a very substantial generalization of these traditional results and techniques.

From the very beginning of the development of the rule algebra framework [3], one of our main motivations has been the study of stochastic rewriting systems, whence of continuous-time Markov chains (CTMCs) based upon DPO- or SqPO-type rewriting rules. While previously in particular applications of stochastic SqPO-type rewriting systems have played a role predominantly in highly specialized settings such as e.g. the formulation of the biochemical reaction system framework known as KAPPA [20,21,19,18], our novel approach of formulating such systems in terms of associative unital rule algebras may very well open this versatile modeling technique to many other areas of applied research. In conjunction with our previously developed DPO-type framework in [7], one could argue that our *stochastic mechanics frameworks* are in a certain sense a *universal construction*, in that once a semantics for associative unital rewriting is provided, the steps necessary to obtain the associated CTMCs are clearly formalized. It is interesting to compare the traditional approaches to stochastic rewriting systems with CTMC semantics such as [31,30] in the DPO- and [29] in the SPO-settings to our present reformulation in terms of rule algebras. The former approaches yet again tend to focus on derivation traces of stochastic rewriting systems, while our rule-algebraic approach aims to extract dynamical information from stochastic rewriting systems via analysis of certain combinatorial relationships (so-called nested commutators) of the infinitesimal generator of the CTMC with the (pattern-counting) observables of the system. It is via these relations that one may in certain cases obtain *exact closed-form solutions* for such dynamical data (see e.g. Section 6). It would nevertheless be an intriguing avenue for future research to understand better the finer points of the “traditional” stochastic rewriting frameworks (which also feature sophisticated developments in terms of probabilistic model-checking and various types of stochastic logics), and furthermore whether or not rule-algebraic techniques might be of interest also in more general stochastic rewriting semantics such as probabilistic (timed) graph transformations [33,40].

Structure of the paper: In Section 2, some category-theoretical background material is provided. The key results of associativity and concurrency of SqPO rewriting are presented in Section 3, followed by the construction of SqPO-type rule algebras in Section 4. The second part of the paper contains the stochas-

tic mechanics framework (Section 5) as well as a practical application example (Section 6). Technical proofs are situated in the Appendix.

2 Background: adhesive categories and final pullback complements

We recall some of the elementary definitions and properties related to the notions of adhesive categories, upon which our framework will rely.

Definition 1 ([34]). *A category \mathbf{C} is said to be **adhesive** if*

- (i) \mathbf{C} has pushouts along monomorphisms,
- (ii) \mathbf{C} has pullbacks,
- (iii) pushouts along monomorphisms are van Kampen (VK) squares.

The last property entails that in a commutative cube as in (1) on the left where the bottom square is a pushout, this square is a VK square if and only if whenever the back and right vertical faces are pullbacks, then the top square is a pushout if and only if the front and left vertical squares are pullbacks.

Definition 2 (Finitary categories [12]). *A category \mathbf{C} is said to be finitary if every object $X \in \text{obj}(\mathbf{C})$ has only finitely many subobjects (i.e. if there only exist finitely many monomorphisms $Y \rightarrow X$ up to isomorphism for every $X \in \text{obj}(\mathbf{C})$). For every adhesive category \mathbf{C} , the restriction to finite objects of \mathbf{C} defines a full subcategory \mathbf{C}_{fin} called the finitary restriction of \mathbf{C} .*

Theorem 1 (Finitary restrictions; [12], Thm. 4.6). *The finitary restriction \mathbf{C}_{fin} of any adhesive category \mathbf{C} is a finitary adhesive category.*

Adhesive categories have been introduced and advocated in [34] as a framework for rewriting due to their numerous useful properties, some of which are listed in Appendix A for the reader's convenience. One of the central concepts in the theory of SqPO rewriting is the following:

Definition 3 (Final Pullback Complement (FPC); [16,36]). *Let \mathbf{C} be a category. Given a commutative diagram as in (1) on the right, a pair of morphisms (d, b) is a final pullback complement (FPC) of a pair (c, a) if (i) (a, b) is a pullback of (c, d) (i.e. if the square marked (B) is a pullback square), and (ii) for each collection of morphisms (x, y, z, w) as in (1) on the right, where (x, y) is pullback of (c, z) and where $a \circ w = x$, there exists a unique morphism w^* with $d \circ w^* = z$ and $w^* \circ y = b \circ w$.*

For our associative rewriting framework, it will be crucial to work with a category in which (i) FPCs are guaranteed to exist when constructing them for composable pairs of monomorphisms, and (ii) monomorphisms are stable under FPCs, i.e. FPCs of pairs of monomorphisms are given by pairs of monomorphisms (cf. Lemma 3 of Appendix A). To the best of our knowledge, the question of which categories possess this property has not yet been investigated to quite the level of generality as analogous classification problems in the case of DPO rewriting, even though there does exist a large body of work on classes of categories that admit SqPO constructions (some of which coincide with adhesive categories) [16,39,36,14,37]. Within these classes, according to [13,14] guarantees for the existence of FPCs may be provided for categories that possess a so-called \mathcal{M} -partial map classifier. However, it appears to be an open question of whether the statement of Lemma 3 on stability of monomorphisms under FPCs may be generalized to the setting of \mathcal{M} -adhesive categories, where \mathcal{M} is a class of monomorphisms. Relaying such questions to future work, we refer to Lemma 4 of Appendix A for a well-known instantiation of a suitable categorical setting from the SqPO literature in the form of the category **FinGraph** of finite directed multigraphs, which also serves to illustrate the FPC construction.

Assumption 1. *\mathbf{C} is an adhesive category in which all FPCs along monomorphisms exist, and in which monomorphisms are stable under FPCs.*

3 Sesqui-Pushout rewriting

We will now develop a framework for *Sesqui-Pushout (SqPO) rewriting* in the setting of a category \mathbf{C} satisfying Assumption 1, in close analogy to the framework of associative Double-Pushout (DPO) rewriting as introduced in [7,8]. Unlike in the general setting of SqPO rewriting, we will thus be able to not only prove a *concurrency theorem* (Section 3.1), but also an *associativity property* of the SqPO-type rule composition (Section 3.2).

3.1 Concurrent composition and concurrency theorem

For reasons that will become more transparent when introducing the SqPO-type rule algebra framework starting from Section 4, we opt for a non-standard convention of reading spans of monomorphisms “from right to left” (rather than the traditional “left to right”), which is why we will speak of “input” and “output” of rules rather than “left-” and “right hand sides” to avoid confusion.

Definition 4 (SqPO-type rewriting; compare [16], Def. 4). *Let \mathbf{C} be an adhesive category satisfying Assumption 1. Denote by $\text{Lin}(\mathbf{C})$ the set of (isomorphism classes¹ of) so-called linear productions, defined as the set of spans of*

¹ Two productions $O \leftarrow K \rightarrow I$ and $O' \leftarrow K' \rightarrow I'$ are defined to be isomorphic if there exist isomorphisms $I \rightarrow I'$, $K \rightarrow K'$ and $O \rightarrow O'$ that make the obvious diagram commute; we will not distinguish between isomorphic productions. As natural in this category-theoretical setting, the constructions presented in the following are understood as defined up to such isomorphisms.

monomorphisms,

$$\text{Lin}(\mathbf{C}) := \{p \equiv (O \xleftarrow{o} K \xrightarrow{i} I) \mid o, i \in \text{mono}(\mathbf{C})\} / \cong. \quad (2)$$

Given an object $X \in \text{obj}(\mathbf{C})$ and a linear production $p \in \text{Lin}(\mathbf{C})$, we denote the set of SqPO-admissible matches $M_p^{sq}(X)$ as the set of monomorphisms $m : I \rightarrow X$. Then the diagram below is constructed by taking the final pullback complement marked FPC followed by taking the pushout marked PO:

$$\begin{array}{ccccc} O & \xleftarrow{o} & K & \xrightarrow{i} & I \\ m^* \downarrow & \text{PO} & k \downarrow & \text{FPC} & \downarrow m \\ X' & \xleftarrow{o'} & \bar{K} & \xrightarrow{i'} & X \end{array} \quad (3)$$

We write $p_m(X) := X'$ for the object “produced” by the above diagram. The process is called (SqPO-) derivation of X along production p and admissible match m , and denoted $p_m(X) \xleftarrow[p, m]{\text{SqPO}} X$.

Next, a notion of sequential composition of productions is introduced:

Definition 5 (SqPO-type concurrent composition). Let $p_1, p_2 \in \text{Lin}(\mathbf{C})$ be two linear productions. Then an overlap of the output object O_1 of p_1 with the input object I_2 of p_2 , encoded as a span $\mathbf{m} = (I_2 \xleftarrow{m_2} M_{21} \xrightarrow{m_1} O_1)$ with $m_1, m_2 \in \text{mono}(\mathbf{C})$, is called an SqPO-admissible match of p_2 into p_1 , denoted $\mathbf{m} \in M_{p_2}^{sq}(p_1)$, if the square marked POC in (4) is constructible as a pushout complement (with the cospan $I_2 \xrightarrow{n_2} N_{21} \xleftarrow{n_1} O_1$ obtained by taking the pushout marked PO). In this case, the remaining parts of the diagram are formed by taking the final pullback complement marked FPC and the pushouts marked PO:

$$\begin{array}{ccccccc} O_2 & \xleftarrow{o_2} & K_2 & \xrightarrow{i_2} & I_2 & \xleftarrow{m_2} & M_{21} & \xrightarrow{m_1} & O_1 & \xleftarrow{o_1} & K_1 & \xrightarrow{i_1} & I_1 \\ n_2^* \downarrow & \text{PO} & k_2 \downarrow & \text{FPC} & \downarrow n_2 & \text{PO} & \downarrow n_1 & \text{POC} & \downarrow k_1 & \text{PO} & \downarrow n_1^* \\ O_{21} & \xleftarrow{o'_2} & \bar{K}_2 & \xrightarrow{i'_2} & N_{21} & \xleftarrow{o'_1} & \bar{K}_1 & \xrightarrow{i'_1} & I_{21} \\ & \text{PB} & & & & & & & & & \\ & \text{PB} & & & & & & & & & \\ & \text{PB} & & & & & & & & & \end{array} \quad (4)$$

$\xrightarrow{o_{21}=o'_2 \circ i'_2} \quad \xrightarrow{i''_2} \quad \xrightarrow{o''_1} \quad \xrightarrow{i_{21}=o'_1 \circ i'_1}$

If $\mathbf{m} \in M_{p_2}^{sq}(p_1)$, we write $p_2 \triangleleft^{\mathbf{m}} p_1 \in \text{Lin}(\mathbf{C})$ for the composite of p_2 with p_1 along the admissible match \mathbf{m} , defined as

$$p_2 \triangleleft^{\mathbf{m}} p_1 \equiv (O_{21} \xleftarrow{o_{21}} K_{21} \xrightarrow{i_{21}} I_{21}). \quad (5)$$

Due to stability of monomorphisms under pushouts, pullbacks and FPCs in the setting of a category satisfying Assumption 1, all morphisms in Definitions 4 and 5 are guaranteed to be monomorphisms, whence in particular the span $p_2 \triangleleft^{\mathbf{m}} p_1$ is a span of monomorphisms and thus indeed an element of $\text{Lin}(\mathbf{C})$.

At first sight, it might appear irritating that in the definition of the SqPO-type rule composition, the right hand part of (4) involves a *pushout complement* (marked POC), while the left hand part of the diagram in (4) features a *final pullback complement* (marked FPC). Intuitively, considering the case of graph rewriting for concreteness, in a given sequential application of two productions, while the application of the first production may lead to implicit edge deletions, the second production is incapable of having any causal interaction with edges deleted by the first production. In contrast, the second production may in a given sequential application very well implicitly delete edges present in the output object of the first production, which explains the presence of the FPC in the defining equation (4). We refer the interested readers to [5] for further intuitions attainable in terms of so-called rule diagrams for presenting rule compositions. The justification for Definition 5 in the general case is provided via the following *concurrency theorem*. Even though at least in certain specialized settings the “synthesis” part of this theorem has been foreseen already in [36] (where it is also commented that a full concurrency theorem for SqPO rewriting might be attainable), the following result appears to be new.

Theorem 2 (SqPO-type Concurrency Theorem). *Let \mathbf{C} be an adhesive category satisfying Assumption 1. Let $p_1, p_2 \in \text{Lin}(\mathbf{C})$ be two linear rules and $X_0 \in \text{ob}(\mathbf{C})$ an object.*

– **Synthesis:** *Given a two-step sequence of SqPO derivations*

$$X_2 \xleftarrow[p_2, m_2]{SqPO} X_1 \xleftarrow[p_1, m_1]{SqPO} X_0,$$

with $X_1 := p_{1_{m_1}}(X_0)$ and $X_2 := p_{2_{m_2}}(X_1)$, there exists a SqPO-composite rule $q = p_2 \overset{\mathbf{n}}{\triangleleft} p_1$ for a unique $\mathbf{n} \in \mathbf{M}_{p_2}^{sq}(p_1)$, and a unique SqPO-admissible match $n \in \mathbf{M}_q^{sq}(X)$, such that

$$q_n(X) \xleftarrow[q, n]{SqPO} X_0 \quad \text{and} \quad q_n(X_0) \cong X_2.$$

– **Analysis:** *Given an SqPO-admissible match $\mathbf{n} \in \mathbf{M}_{p_2}^{sq}(p_1)$ of p_2 into p_1 and an SqPO-admissible match $n \in \mathbf{M}_q^{sq}(X)$ of the SqPO-composite $q = p_2 \overset{\mathbf{n}}{\triangleleft} p_1$ into X , there exists a unique pair of SqPO-admissible matches $m_1 \in \mathbf{M}_{p_1}^{sq}(X_0)$ and $m_2 \in \mathbf{M}_{p_2}^{sq}(X_1)$ with $X_1 := p_{1_{m_1}}(X_0)$ such that*

$$X_2 \xleftarrow[p_2, m_2]{SqPO} X_1 \xleftarrow[p_1, m_1]{SqPO} X_0 \quad \text{and} \quad X_2 \cong q_n(X).$$

Proof. See Appendix B.1.

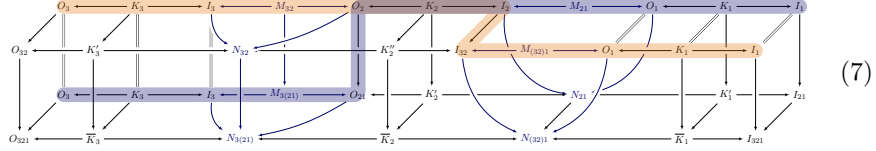
3.2 Composition and associativity

The following theorem establishes that in analogy to the DPO rewriting setting of [7], also the sesqui-pushout variant of rule compositions possesses a form of associativity property.

Theorem 3 (SqPO-type associativity theorem). *Let \mathbf{C} be an adhesive category satisfying Assumption 1. Then the SqPO-composition operation $\cdot \triangleleft \cdot$ on linear productions of \mathbf{C} is associative in the following sense: given linear productions $p_1, p_2, p_3 \in \text{Lin}(\mathbf{C})$, there exists a bijective correspondence between pairs of SqPO-admissible matches $(\mathbf{m}_{21}, \mathbf{m}_{3(21)})$ and $(\mathbf{m}_{32}, \mathbf{m}_{(32)1})$ such that*

$$p_3 \overset{\mathbf{m}_{3(21)}}{\triangleleft} \left(p_2 \overset{\mathbf{m}_{21}}{\triangleleft} p_1 \right) \cong \left(p_3 \overset{\mathbf{m}_{32}}{\triangleleft} p_2 \right) \overset{\mathbf{m}_{(32)1}}{\triangleleft} p_1. \quad (6)$$

Proof. Intuitively, the associativity property in the SqPO case manifests itself in a form entirely analogous to the DPO case [7], whereby the data provided along the path highlighted in orange below permits to uniquely compute the data provided along the path highlighted in blue and vice versa (with both sets of overlaps computing the same “triple composite” production that is encoded as the composition of the three spans in the bottom front row):



See Appendix B.2 for the precise technical details of the proof.

We invite the interested readers to compare the SqPO-type constructions presented here against those contained in the extended journal version [8] of [7] for the DPO framework, since this might lend some intuitions on the otherwise very abstract nature of the proofs to the experts.

4 From associativity to SqPO-type rule algebras

For the rule algebra constructions, we will require an additional structure:

Definition 6 (Initial objects). *An object $\emptyset \in \text{obj}(\mathbf{C})$ of some category \mathbf{C} is said to be a strict initial object if for every object $X \in \text{obj}(\mathbf{C})$, there exists a unique morphism $\emptyset \rightarrow X$, and if any morphism $X \rightarrow \emptyset$ must be an isomorphism.*

For example, the category **Graph** possesses a strict initial object (the empty graph). For the experts, it appears worthwhile noting the following result:

Lemma 1 (Extensive categories; [34], Lem. 4.1). *An adhesive category \mathbf{C} is an extensive category² if and only if it possesses a strict initial object.*

Assumption 2 (Prerequisites for SqPO-type rule algebras). *We assume that \mathbf{C} is an adhesive category satisfying Assumption 1, and which is in addition finitary and possesses a strict initial object $\emptyset \in \text{obj}(\mathbf{C})$.*

² For the purposes of this paper, it suffices to consider the “if” direction as a definition of extensivity, since the relevant structure to our constructions is that of having a strict initial object (see e.g. [34] for the precise definition of extensivity).

Definition 7 (SqPO-type rule algebras). Let $\delta : \text{Lin}(\mathbf{C}) \rightarrow \mathcal{R}_{\mathbf{C}}$ be defined as an isomorphism from $\text{Lin}(\mathbf{C})$ to the basis of a free \mathbb{R} -vector space $\mathcal{R}_{\mathbf{C}} \equiv (\mathcal{R}_{\mathbf{C}}, +, \cdot)$, such that³

$$\mathcal{R}_{\mathbf{C}} := \text{span}_{\mathbb{R}}(\{\delta(p) \mid p \in \text{Lin}(\mathbf{C})\}). \quad (8)$$

In order to clearly distinguish between elements of $\text{Lin}(\mathbf{C})$ and basis vectors of $\mathcal{R}_{\mathbf{C}}$, we introduce the notation

$$(O \stackrel{p}{\leftarrow} I) := \delta \left(O \stackrel{o}{\leftarrow} K \stackrel{i}{\rightarrow} I \right). \quad (9)$$

Define the SqPO rule algebra product $\odot_{\mathcal{R}_{\mathbf{C}}}$ on a category \mathbf{C} that satisfies Assumption 2 as the binary operation

$$\odot_{\mathcal{R}_{\mathbf{C}}} : \mathcal{R}_{\mathbf{C}} \times \mathcal{R}_{\mathbf{C}} \rightarrow \mathcal{R}_{\mathbf{C}} : (R_1, R_2) \mapsto R_1 \odot_{\mathcal{R}_{\mathbf{C}}} R_2, \quad (10)$$

where for two basis vectors $R_i = \delta(p_i)$ encoding the linear rules $p_i \in \text{Lin}(\mathbf{C})$ ($i = 1, 2$),

$$R_2 \odot_{\mathcal{R}_{\mathbf{C}}} R_1 := \sum_{\mathbf{m} \in \mathbf{M}_{p_2}^{sq}(p_1)} \delta \left(p_2 \stackrel{\mathbf{m}}{\triangleleft} p_1 \right). \quad (11)$$

The definition is extended to arbitrary (finite) linear combinations of basis vectors by bilinearity, whence for $p_i, p_j \in \text{Lin}(\mathbf{C})$ and $\alpha_i, \beta_j \in \mathbb{R}$,

$$\left(\sum_i \alpha_i \cdot \delta(p_i) \right) \odot_{\mathcal{R}_{\mathbf{C}}} \left(\sum_j \beta_j \cdot \delta(p_j) \right) := \sum_{i,j} (\alpha_i \cdot \beta_j) \cdot (\delta(p_i) \odot_{\mathcal{R}_{\mathbf{C}}} \delta(p_j)). \quad (12)$$

We call $\mathcal{R}_{\mathbf{C}}^{sq} \equiv (\mathcal{R}_{\mathbf{C}}, \odot_{\mathcal{R}_{\mathbf{C}}})$ the **SqPO-type rule algebra** over the finitary adhesive and extensive category \mathbf{C} .

Example 1. Let $\mathbf{C} = \mathbf{FinGraph}$ be the category of finite directed multigraphs, with \emptyset the empty graph. Then with $\odot \equiv \odot_{\mathcal{R}_{\mathbf{C}}}$, we find for example

$$\begin{aligned} & \delta(\emptyset \leftarrow \emptyset \hookrightarrow \bullet) \odot \delta(\bullet \leftarrow \emptyset \hookrightarrow \emptyset) \\ &= \sum_{\mathbf{m} \in \{(\bullet \leftarrow \emptyset \hookrightarrow \bullet, \bullet), (\bullet \leftarrow \bullet \hookrightarrow \bullet, \bullet), (\bullet \leftarrow \bullet \hookrightarrow \bullet, \bullet)\}} \delta \left((\emptyset \leftarrow \emptyset \hookrightarrow \bullet) \stackrel{\mathbf{m}}{\triangleleft} (\bullet \leftarrow \emptyset \hookrightarrow \emptyset) \right) \\ &= \delta(\bullet \leftarrow \emptyset \hookrightarrow \bullet) + 2\delta(\bullet \leftarrow \emptyset \hookrightarrow \emptyset). \end{aligned} \quad (13)$$

The result of the composition thus captures the combinatorial insight that there are two contributions that evaluate to an isomorphic rule algebra element. More

³ Recall that for a set A , the notation $\text{span}_{\mathbb{R}}(\{e(a) \mid a \in A\})$ entails to “take the \mathbb{R} -span over basis vectors $e(a)$ indexed by elements of A ”, i.e. elements of the resulting \mathbb{R} -vector space are (finite) linear combinations of the basis vectors $e(a)$ with real coefficients.

generally, one finds the following structure of compositions of rule algebra elements based upon “discrete” graph rewriting rules: letting $\bullet^{\uplus n}$ denote the n -vertex graph without edges (for $n \geq 0$), one finds (for $p, q, r, s \geq 0$)

$$\begin{aligned} & \delta(\bullet^{\uplus p} \leftarrow \emptyset \hookrightarrow \bullet^{\uplus q}) \odot \delta(\bullet^{\uplus r} \leftarrow \emptyset \hookrightarrow \bullet^{\uplus s}) \\ &= \sum_{k=0}^{\min(q,r)} k! \binom{q}{k} \binom{r}{k} \delta(\bullet^{\uplus(p+r-k)} \leftarrow \emptyset \hookrightarrow \bullet^{\uplus(q+k)}). \end{aligned} \quad (14)$$

This result is further interpreted in Example 2.

Theorem 4 (Properties of $\mathcal{R}_{\mathbf{C}}^{sq}$). *For every category \mathbf{C} satisfying Assumption 2, the associated SqPO-type rule algebra $\mathcal{R}_{\mathbf{C}}^{sq} \equiv (\mathcal{R}_{\mathbf{C}}, \odot_{\mathcal{R}_{\mathbf{C}}})$ is an associative unital algebra, with unit element $R_{\emptyset} := (\emptyset \leftarrow \emptyset)$. (Proof: Appendix B.3)*

For the unital and associative SqPO-type rule algebras, one may provide a notion of *representations* in analogy to the DPO-type case (compare [3,7]):

Definition 8 (Canonical representation of $\mathcal{R}_{\mathbf{C}}^{sq}$). *Let \mathbf{C} be a category satisfying Assumption 2, with a strict initial object $\emptyset \in \text{ob}(\mathbf{C})$, and let $\mathcal{R}_{\mathbf{C}}^{sq}$ be its associated rule algebra of SqPO type. Denote by $\hat{\mathbf{C}}$ the free \mathbb{R} -vector space spanned by basis vectors $|X\rangle$ indexed by isomorphism classes of objects,*

$$\hat{\mathbf{C}} := \text{span}_{\mathbb{R}}(\{|X\rangle \mid X \in \text{obj}(\mathbf{C})_{\cong}\}) \equiv (\hat{\mathbf{C}}, +, \cdot). \quad (15)$$

Then the canonical representation $\rho_{\mathbf{C}}^{sq} : \mathcal{R}_{\mathbf{C}}^{sq} \rightarrow \text{End}_{\mathbb{R}}(\hat{\mathbf{C}})$ of $\mathcal{R}_{\mathbf{C}}^{sq}$ is defined as a morphism from the SqPO-type rule algebra $\mathcal{R}_{\mathbf{C}}^{sq}$ to endomorphisms of $\hat{\mathbf{C}}$, with

$$\rho_{\mathbf{C}}^{sq}(\delta(p)) |X\rangle := \begin{cases} \sum_{m \in M_p^{sq}(X)} |p_m(X)\rangle & \text{if } M_p^{sq}(X) \neq \emptyset \\ 0_{\hat{\mathbf{C}}} & \text{otherwise,} \end{cases} \quad (16)$$

and extended to arbitrary elements of $\mathcal{R}_{\mathbf{C}}^{sq}$ and of $\hat{\mathbf{C}}$ by linearity.

Example 2. Extending Example 1, letting $\rho \equiv \rho_{\mathbf{FinGraph}}^{sq}$, note first that by definition for all $G \in \text{obj}(\mathbf{FinGraph})$, $|G\rangle = \rho(\delta(G \leftarrow \emptyset \hookrightarrow \emptyset)) |\emptyset\rangle$. With

$$\hat{D} := \rho(\delta(\emptyset \leftarrow \emptyset \hookrightarrow \bullet)), \quad \hat{X} := \rho(\delta(\bullet \leftarrow \emptyset \hookrightarrow \emptyset)), \quad |n\rangle := |\bullet^{\uplus n}\rangle \quad (n \geq 0), \quad (17)$$

as a consequence of (14) of Example 1 one may verify that

$$\hat{D} |0\rangle = 0_{\widehat{\mathbf{FinGraph}}}, \quad \hat{D} |n\rangle = n |n-1\rangle \quad (n > 0), \quad \hat{X} |n\rangle = |n+1\rangle. \quad (18)$$

In other words, the data of (17) and (18) furnishes a representation of the famous *Heisenberg-Weyl algebra* that is of fundamental importance in combinatorics and physics (see e.g. [9,10,11]). An alternative such representation is given by the linear operators \hat{x} (multiplication by x) and ∂_x (derivation by x) acting on the \mathbb{R} -vector space spanned by monomials x^n , which reproduces equations isomorphic to (17) and (18), with $\partial_x x^n = n x^{n-1}$ and $\hat{x} x^n = x^{n+1}$. However, the action of \hat{D} and \hat{X} is of course defined on *all* states $|G\rangle$ with $G \in \text{obj}(\mathbf{FinGraph})$, so that we may e.g. compute the following “derivative of a graph”:

$$\hat{D} |\bullet \rightarrow \bullet \rightarrow \bullet\rangle = 2 |\bullet \rightarrow \bullet\rangle + |\bullet \quad \bullet\rangle \quad (19)$$

The following theorem states that ρ_C^{sq} as given in Definition 8 is indeed a homomorphism (and thus qualifies as a representation of \mathcal{R}_C^{sq}).

Theorem 5 (SqPO-type canonical representation). *For a category \mathbf{C} satisfying Assumption 2, $\rho_C^{sq} : \mathcal{R}_C^{sq} \rightarrow \text{End}(\hat{\mathbf{C}})$ of Definition 8 is a homomorphism of unital associative algebras. (Proof: Appendix B.4)*

5 Applications of SqPO-type rule algebras to stochastic mechanics

In practical applications of stochastic rewriting systems, the type of rewriting semantics presents one of the key design choices. For example, if in a given situation a stochastic graph rewriting system should be implemented, choosing DPO- vs. SqPO-type rewriting entails two entirely different semantics in terms of the behavior of vertex deletion rules: in the former case, vertices may only be deleted if also all its incident edges are explicitly deleted as well, while in the latter case no such restriction applies (i.e. an application of a vertex deletion rule “automatically” leads to the deletion of all incident edges). Evidently, such fundamentally different behavior at the level of rewriting rules will also have strong influence on the dynamical behavior of the associated stochastic rewriting systems, whence it is of considerable practical interest to have a universal implementation of such systems available in both formalisms. We begin by specializing the general definition of continuous-time Markov chains (see e.g. [41,1]) to the setting of SqPO-type rewriting systems in close analogy to [3,6,7].

Definition 9 (Continuous-time Markov Chains (CTMCs); compare [7], Def. 7.1). *Let \mathbf{C} be a category satisfying Assumption 2, and which in addition possesses a countable set of isomorphism classes of objects $\text{obj}(\mathbf{C})_{\cong}$. Let $\hat{\mathbf{C}}$ denote the free \mathbb{R} -vector space introduced in Definition 8. We define the space $\text{Prob}(\mathbf{C})$ as the space of sub-probability distributions in the following sense:*

$$\text{Prob}(\mathbf{C}) := \left\{ |\Psi\rangle = \sum_{o \in \text{obj}(\mathbf{C})_{\cong}} \psi_o |o\rangle \mid \forall o \in \text{obj}(\mathbf{C})_{\cong} : \psi_o \in \mathbb{R}_{\geq 0} \wedge \sum_{o \in \text{obj}(\mathbf{C})_{\cong}} \psi_o \leq 1 \right\} \quad (20)$$

Let $\text{Stoch}(\mathbf{C}) := \text{End}_{\mathbb{R}}(\text{Prob}(\mathbf{C}))$ be the space of endomorphisms of $\text{Prob}(\mathbf{C})$, with elements referred to as sub-stochastic operators. Then a **continuous-time Markov chain (CTMC)** is specified in terms of a tuple of data $(|\Psi(0)\rangle, H)$, where $|\Psi(0)\rangle \in \text{Prob}(\mathbf{C})$ is the initial state, and where $H \in \text{End}_{\mathbb{R}}(\mathcal{S}_{\mathbf{C}})$ is the infinitesimal generator or Hamiltonian of the CTMC (with $\mathcal{S}_{\mathbf{C}}$ the space of real-valued sequences indexed by elements of $\text{obj}(\mathbf{C})_{\cong}$ and with finite coefficients). H is required to be an infinitesimal (sub-)stochastic operator, which entails that for $H \equiv (h_{o,o'})_{o,o' \in \text{obj}(\mathbf{C})_{\cong}}$ and for all $o, o' \in \text{obj}(\mathbf{C})_{\cong}$,

$$(i) \ h_{o,o} \leq 0, \ (ii) \ \forall o \neq o' : h_{o,o'} \geq 0, \ (iii) \ \sum_{o'} h_{o,o'} = 0. \quad (21)$$

Then this data encodes the evolution semi-group $\mathcal{E} : \mathbb{R}_{\geq 0} \rightarrow \text{Stoch}(\mathbf{C})$ as the (point-wise minimal non-negative) solution of the Kolmogorov backwards or master equation:

$$\frac{d}{dt}\mathcal{E}(t) = H\mathcal{E}(t), \mathcal{E}(0) = \mathbb{1}_{\text{Stoch}(\mathbf{C})} \Rightarrow \forall t, t' \in \mathbb{R}_{\geq 0} : \mathcal{E}(t)\mathcal{E}(t') = \mathcal{E}(t+t') \quad (22)$$

Consequently, the time-dependent state $|\Psi(t)\rangle$ of the system is given by

$$\forall t \in \mathbb{R}_{\geq 0} : |\Psi(t)\rangle = \mathcal{E}(t) |\Psi(0)\rangle. \quad (23)$$

An important technical aspect of the above definition of CTMCs is the definition of the relevant space of (sub-)probability distributions in interaction with the definition of the infinitesimal generator H and of the space $\mathcal{S}_{\mathbf{C}}$. Some remarks on this interaction and a short explanation of the relevant mathematical concepts are provided in Appendix B.5.

Our main approach in studying CTMCs based on rewriting systems will consist in analyzing the dynamical statistical behavior of so-called observables:

Definition 10 (Observables; [7], Def. 7.1). Let $\mathcal{O}_{\mathbf{C}} \subset \text{End}_{\mathbb{R}}(\mathcal{S}_{\mathbf{C}})$ denote the space of observables, defined as the space of diagonal operators⁴,

$$\mathcal{O}_{\mathbf{C}} := \{O \in \text{End}_{\mathbb{R}}(\mathcal{S}_{\mathbf{C}}) \mid \forall X \in \text{obj}(\mathbf{C})_{\cong} : O|X\rangle = \omega_O(X)|X\rangle, \omega_O(X) \in \mathbb{R}\}. \quad (24)$$

We furthermore define the so-called projection operation $\langle | : \mathcal{S}_{\mathbf{C}} \rightarrow \mathbb{R}$ via extending by linearity the definition of $\langle |$ acting on basis vectors of $\hat{\mathbf{C}}$,

$$\forall X \in \text{obj}(\mathbf{C})_{\cong} : \langle |X\rangle := 1_{\mathbb{R}}. \quad (25)$$

These definitions induce a notion of correlators of observables (also referred to as (mixed) moments), defined for $O_1, \dots, O_n \in \mathcal{O}_{\mathbf{C}}$ and $|\Psi\rangle \in \text{Prob}(\mathbf{C})$ as

$$\langle O_1, \dots, O_n \rangle_{|\Psi\rangle} := \langle | O_1, \dots, O_n |\Psi\rangle = \sum_{X \in \text{obj}(\mathbf{C})_{\cong}} \psi_X \cdot \omega_{O_1}(X) \cdots \omega_{O_n}(X). \quad (26)$$

The precise relationship between the notions of CTMCs and SqPO-type rewriting rules as encoded in the corresponding SqPO-type rule algebra formalism is established in the form of the following theorem, where in particular the notion of observables is quite different in nature to the DPO-type analogon (compare Thm. 7.12 of [7]). This result is the first-of-its-kind *universal* definition of SqPO-type stochastic rewriting systems with “mass-action semantics” (where activities of productions are proportional to their number of admissible matches in a given system state).

Theorem 6 (SqPO-type stochastic mechanics framework). Let \mathbf{C} be a category satisfying Assumption 2. Let $\{(O_j \xleftarrow{p_j} I_j) \in \mathcal{R}_{\mathbf{C}}^{sq}\}_{j \in \mathcal{J}}$ be a (finite) set of

⁴ Depending on the concrete case, the eigenvalue $\omega_O(X)$ in $O|X\rangle = \omega_O(X)|X\rangle$ may e.g. coincide with the number of occurrences of a pattern in the object X (see also Appendix B.5).

rule algebra elements, and $\{\kappa_j \in \mathbb{R}_{\geq 0}\}_{j \in \mathcal{J}}$ a collection of non-zero parameters (called base rates). Then one may construct the Hamiltonian H of the associated CTMC from this data according to

$$H := \hat{H} + \bar{H}, \quad \hat{H} := \sum_{j \in \mathcal{J}} \kappa_j \cdot \rho_{\mathbf{C}}^{sq} \left(O_j \xleftarrow{p_j} I_j \right), \quad \bar{H} := - \sum_{j \in \mathcal{J}} \kappa_j \cdot \mathbb{O}_{I_j}^{sq}. \quad (27)$$

Here, the notation \mathbb{O}_M^{sq} for arbitrary objects $M \in \mathbf{obj}(\mathbf{C})$ denotes the observables (sometimes referred to as motif counting observables) for the resulting CTMC of SqPO-type, with

$$\mathbb{O}_M^{sq} := \rho_{\mathbf{C}}^{sq} \left(\delta \left(M \xleftarrow{id_M} M \xrightarrow{id_M} M \right) \right). \quad (28)$$

We furthermore have the SqPO-type jump-closure property, whereby for all $(O \xleftarrow{p} I) \in \mathcal{R}_{\mathbf{C}}^{sq}$

$$\langle | \rho_{\mathbf{C}}^{sq}(O \xleftarrow{p} I) = \langle | \mathbb{O}_I^{sq}. \quad (29)$$

Proof. See Appendix B.5.

6 Application example: a dynamical random graph model

In order to illustrate our novel SqPO-type stochastic mechanics framework, let us consider a dynamical system evolving on the space of finite directed multigraphs.

Example 3. Let **FinGraph** be the finitary restriction of the category **Graph** (see also Lemma 4), and denote by $\emptyset \in \mathbf{FinGraph}$ the strict initial object (the empty graph). We define a stochastic SqPO rewriting system based upon rules encoding *vertex creation/deletion* (v_{\pm}) and *edge creation/deletion* (e_{\pm}):

$$\begin{aligned} v_+ &:= (\bullet \leftarrow \emptyset \rightarrow \emptyset) & v_- &:= (\emptyset \leftarrow \emptyset \rightarrow \bullet) \\ e_+ &:= (\bullet \rightarrow \bullet \leftarrow \bullet \rightarrow \bullet \rightarrow \bullet) & e_- &:= (\bullet \leftarrow \bullet \leftarrow \bullet \rightarrow \bullet \rightarrow \bullet) \end{aligned} \quad (30)$$

Together with a choice of *base rates* $\nu_{\pm}, \varepsilon_{\pm} \in \mathbb{R}_{\geq 0}$ and an initial state $|\Psi(0)\rangle \in \text{Prob}(\mathbf{FinGraph})$, this data defines a stochastic rewriting system with Hamiltonian $H := \hat{H} + \bar{H}$,

$$\begin{aligned} \hat{H} &= \nu_+ V_+ + \nu_- V_- + \varepsilon_+ E_+ + \varepsilon_- E_- \\ \bar{H} &= -\nu_+ \mathbb{O}_{\emptyset} - \nu_- \mathbb{O}_{\bullet} - \varepsilon_+ \mathbb{O}_{\bullet\bullet} - \varepsilon_- \mathbb{O}_{\bullet\rightarrow\bullet}, \end{aligned} \quad (31)$$

where $V_{\pm} := \rho_{\mathbf{FinGraph}}^{sq}(\delta(v_{\pm}))$ and $E_{\pm} := \rho_{\mathbf{FinGraph}}^{sq}(\delta(e_{\pm}))$.

Despite the apparent simplicity of this model (which might be seen as a paradigmatic example of a *random graph model*), the explicit analysis via the stochastic mechanics framework will uncover a highly non-trivial interaction of the dynamics of the vertex- and of the edge-counting observables. Intuitively, since in SqPO-rewriting no conditions are posed upon vertices that are to be

deleted, the model is expected to possess a vertex dynamics that is the one of a so-called *birth-death process*. If it were not for the vertex deletions, one would find a similar dynamics for the edge-counting observables (compare e.g. the DPO-type rewriting model considered in [7]). However, since deletion of vertices deletes all incident edges, the dynamics of the edge-counting observable is rendered considerably more complicated, and in particular much less evident to foresee by heuristic arguments.

In order to compute the dynamics of the vertex counting observable $O_V := \mathbb{O}_\bullet$, we follow the approach of *exponential moment-generating functions* put forward in [3,6,4] and define

$$M_V(t; \lambda) := \langle | e^{\lambda O_V} | \Psi(t) \rangle, \quad (32)$$

with λ a formal variable. $M_V(t; \lambda)$ encodes the moments of the observable O_V , in that taking the n -th derivative of $M_V(t; \lambda)$ w.r.t. λ followed by setting $\lambda \rightarrow 0$ yields the n -th moment of O_V . Note that we must assume the *finiteness* of all statistical moments as standard in the probability theory literature in order for $M_V(t; \lambda)$ to be well-posed, a property that we will in the case at hand indeed derive explicitly. Referring the interested readers to [8] for further details, suffice it here to recall the following variant of the BCH formula (see e.g. [28], Prop. 3.35), for λ a formal variable and A, B two composable linear operators,

$$e^{\lambda A} B e^{-\lambda A} = e^{ad_{\lambda A}} B = \sum_{n \geq 0} \frac{\lambda^n}{n!} ad_A^n(B), \quad ad_A(B) := AB - BA \equiv [A, B], \quad (33)$$

with the convention that $ad_A^{00}(B) := B$. The operation $[.,.]$ is typically referred to as the *commutator*. We may then derive the *formal evolution equation* for $M_V(t; \lambda)$:

$$\begin{aligned} \frac{\partial}{\partial t} M_V(t; \lambda) &= \langle | e^{\lambda O_V} H | \Psi(t) \rangle = \langle | (e^{\lambda O_V} H e^{-\lambda O_V}) e^{\lambda O_V} | \Psi(t) \rangle \\ &= \langle | (e^{ad_{\lambda O_V}} H) e^{\lambda O_V} | \Psi(t) \rangle. \end{aligned} \quad (34)$$

Since by definition $\langle | H = 0$, it remains to compute the adjoint action $ad_{O_V}(H)$ of O_V on H :

$$\begin{aligned} ad_{O_V}(H) &= \nu_+[O_V, V_+] + \nu_-[O_V, V_-] + \varepsilon_+[O_V, E_+] + \varepsilon_-[O_V, E_-] \\ &= \nu_+ V_+ - \nu_- V_- \end{aligned} \quad (35)$$

Here, the result that $[O_V, E_\pm] = 0$ has a very simple intuitive meaning: in applications of the linear rules e_\pm , the number of vertices remains unchanged, whence the vanishing of the commutator. Combining these results with the SqPO-type *jump-closure property* (cf. Theorem 6), we finally arrive at the following *formal evolution equation* for $M_V(t; \lambda)$:

$$\begin{aligned} \frac{\partial}{\partial t} M_V(t; \lambda) &= \nu_+ (e^\lambda - 1) \langle | V_+ e^{\lambda O_V} | \Psi(t) \rangle + \nu_- (e^{-\lambda} - 1) \langle | V_- e^{\lambda O_V} | \Psi(t) \rangle \\ &\stackrel{(29)}{=} \nu_+ (e^\lambda - 1) \langle | e^{\lambda O_V} | \Psi(t) \rangle + \nu_- (e^{-\lambda} - 1) \langle | O_V e^{\lambda O_V} | \Psi(t) \rangle \\ &= (\nu_+ (e^\lambda - 1) + \nu_- (e^{-\lambda} - 1) \frac{\partial}{\partial \lambda}) M_V(t; \lambda). \end{aligned} \quad (36)$$

Supposing for simplicity an initial state $|\Psi(0)\rangle = |G_0\rangle$ (for $G_0 \in \text{obj}(\mathbf{Graph}_{fin})$ some graph with N_V vertices and N_E edges), we find that $M_V(0; \lambda) = \exp(\lambda N_V)$. The resulting initial value problem may be solved in closed-form via *semi-linear normal-ordering* techniques known from the combinatorics literature [22,9,11,6] (see also [8,4]), and we obtain (for $t \geq 0$)

$$M_V(t; \lambda) = \exp\left(\frac{\nu_+}{\nu_-}(e^\lambda - 1)(1 - e^{-\nu-t})\right) (1 + (e^\lambda - 1)e^{-\nu-t})^{N_V}. \quad (37)$$

In the limit $t \rightarrow \infty$, the moment-generating function becomes that of a *Poisson-distribution* (of parameter ν_+/ν_-), thus confirming the aforementioned intuition that the vertex-counting observable has the dynamical behavior of a so-called *birth-death process* (see e.g. [6]).

Let us consider next the dynamics of the edge-counting observable $O_E := \bullet \text{---} \bullet$, where for brevity we will only consider the evolution of the mean edge count. The calculation of the evolution equation for the expectation value of O_E simplifies to the analogue of the so-called *Ehrenfest equation*,

$$\frac{\partial}{\partial t} \langle |O_E| \Psi(t) \rangle = \langle |O_E H| \Psi(t) \rangle = \langle | (H O_E + [O_E, H]) | \Psi(t) \rangle. \quad (38)$$

Recalling that $\langle |H = 0$, it remains to compute the commutator $[O_E, H]$:

$$\begin{aligned}
[O_E, H] &= \nu_+[O_E, V_+] + \nu_-[O_E, V_-] + \varepsilon_+[O_E, E_+] + \varepsilon_-[O_E, V_-] \\
&= \nu_+ \cdot 0 - \nu_- (E_-^{0,1} + E_-^{1,0}) + \varepsilon_+ E_+ - \varepsilon_- E_- \\
E_-^{0,1} &= \rho_{\mathbf{FinGraph}}^{sq} \left(\delta \left(\begin{array}{c} \bullet \leftarrow \bullet \rightarrow \bullet \rightleftarrows \bullet \\ \text{\tiny b} \qquad \text{\tiny b} \qquad \text{\tiny a} \qquad \text{\tiny b} \end{array} \right) \right) \\
E_-^{1,0} &= \rho_{\mathbf{FinGraph}}^{sq} \left(\delta \left(\begin{array}{c} \bullet \leftarrow \bullet \rightarrow \bullet \rightleftarrows \bullet \\ \text{\tiny a} \qquad \text{\tiny a} \qquad \text{\tiny a} \qquad \text{\tiny b} \end{array} \right) \right).
\end{aligned} \tag{39}$$

This calculation is a representative example of various effects that may occur in rule-algebraic commutation relations: we find a zero commutator $[O_E, V_+]$, indicating the fact that application of the vertex creation rule V_+ does not influence the edge count. The commutators $[O_E, E_\pm] = \pm E_\pm$ encode that application of the edge creation/deletion rules leads to positive/negative contributions to the edge count. Finally, the contribution of the commutator $[O_E, V_-] = -E_-^{0,1} - E_-^{1,0}$ is given by the representations of two rule algebra elements not originally present in the Hamiltonian H , with the structure of the underlying linear rules indicated by the labels a and b on the vertices (as customary in the rewriting literature). It then remains to apply the jump-closure property (Theorem 6) together with the identity $\mathbb{O}_{\bullet\bullet} = O_V(O_V - 1)$ in order to obtain the *evolution equation*

$$\frac{\partial}{\partial t} \langle O_E | \Psi(t) \rangle = \varepsilon_+ \langle O_V (O_V - 1) | \Psi(t) \rangle - (\varepsilon_- + 2\nu_-) \langle O_E | \Psi(t) \rangle. \quad (40)$$

Together with an initial condition such as e.g. $|\Psi(0)\rangle = |G_0\rangle$ for some (finite) directed graph G_0 with N_V vertices and N_E edges, and computing the closed-form expression for the first contribution in (40) from our previous solution (37) (as $\partial_\lambda(\partial_\lambda - 1)M_V(t; \lambda)$ followed by setting $\lambda \rightarrow 0$), the initial value problem for

the mean edge count evolution may be easily solved in closed form via the use of a computer algebra software such as MAPLE, MATHEMATICA or SAGE. It is also straightforward to verify that for an arbitrary initial state $|\Psi(0)\rangle = |G_0\rangle$, the limit value of the mean edge count for $t \rightarrow \infty$ reads

$$\lim_{t \rightarrow \infty} \langle |O_E| \Psi(t) \rangle = \frac{\nu_+^2 \varepsilon_+}{\nu_-^2 (2\nu_- + \varepsilon_-)} . \quad (41)$$

Since the rates ν_{\pm} and ε_{\pm} are free parameters, the above result entails that in this model one may freely adjust the limit value of the average vertex count as encoded in (36) (whence ν_+/ν_-) as well as the limit value of the average edge count via suitable choices of the parameters ε_{\pm} . For illustration, we present some plots of the mean edge count evolution for the case $|\Psi(0)\rangle = |\emptyset\rangle$ and various choices of parameters in Figure 1.

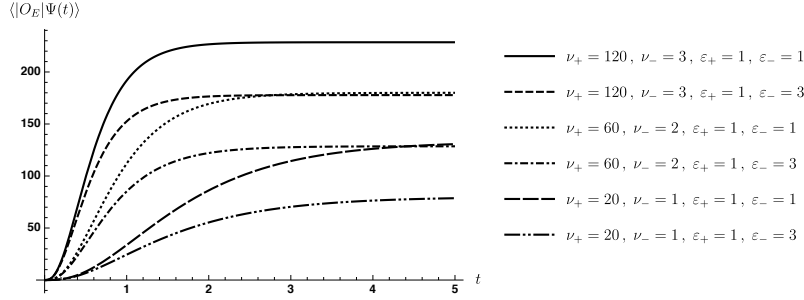


Fig. 1. Time-evolution of $\langle |O_E| \Psi(t) \rangle$ for $|\Psi(0)\rangle = |\emptyset\rangle$.

7 Conclusion and Outlook

Extending our previous work on Double-Pushout (DPO) rewriting theories as presented in [3,7,4] to the important alternative setting of Sesqui-Pushout (SqPO) rewriting, we provide a number of original results in the form of *concurrency* and *associativity* theorems for SqPO rewriting theories on adhesive categories. These fundamental results in turn permit us to formulate so-called *SqPO-type rule algebras*, which play a central role in our novel *universal stochastic mechanics framework*. We strongly believe that these contributions will provide fruitful grounds for further developments both in theory and practice of rewriting beyond the specialists' communities, especially in view of static analysis techniques [2].

References

1. Anderson, W.J.: Continuous-Time Markov Chains. Springer New York (1991). <https://doi.org/10.1007/978-1-4612-3038-0>
2. Behr, N.: Tracelets and Tracelet Analysis Of Compositional Rewriting Systems. arXiv preprint arXiv:1904.12829 (April 2019)
3. Behr, N., Danos, V., Garnier, I.: Stochastic mechanics of graph rewriting. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '16. ACM Press (2016). <https://doi.org/10.1145/2933575.2934537>
4. Behr, N., Danos, V., Garnier, I.: Combinatorial Conversion and Moment Bisimulation for Stochastic Rewriting Systems. arXiv preprint 1904.07313 (2019)
5. Behr, N., Danos, V., Garnier, I., Heindel, T.: The algebras of graph rewriting. arXiv:1612.06240 (2016)
6. Behr, N., Duchamp, G.H., Penson, K.A.: Combinatorics of Chemical Reaction Systems. arXiv:1712.06575 (2017)
7. Behr, N., Sobocinski, P.: Rule Algebras for Adhesive Categories. In: Ghica, D., Jung, A. (eds.) 27th EACSL Annual Conference on Computer Science Logic (CSL 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 119, pp. 11:1–11:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). <https://doi.org/10.4230/LIPIcs.CSL.2018.11>
8. Behr, N., Sobocinski, P.: Rule Algebras for Adhesive Categories (invited extended journal version). arXiv preprint 1807.00785v2 (2019)
9. Blasiak, P., Duchamp, G.H., Horzela, A., Penson, K.A., Solomon, A.I.: Boson normal ordering via substitutions and Sheffer-Type Polynomials. *Physics Letters A* **338**(2), 108–116 (2005). <https://doi.org/10.1016/j.physleta.2005.02.028>
10. Blasiak, P., Duchamp, G.H., Horzela, A., Penson, K.A., Solomon, A.I.: Combinatorial Algebra for second-quantized Quantum Theory. *Advances in Theoretical and Mathematical Physics* **14**(4), 1209–1243 (2010). <https://doi.org/10.4310/atmp.2010.v14.n4.a5>
11. Blasiak, P., Flajolet, P.: Combinatorial Models of Creation-Annihilation. *Séminaire Lotharingien de Combinatoire* **65**(B65c), 1–78 (2011)
12. Braatz, B., Ehrig, H., Gabriel, K., Golas, U.: Finitary \mathcal{M} -Adhesive Categories. *Lecture Notes in Computer Science* pp. 234–249 (2010). https://doi.org/10.1007/978-3-642-15928-2_16
13. Cockett, J., Lack, S.: Restriction categories II: partial map classification. *Theoretical Computer Science* **294**(1-2), 61–102 (feb 2003). [https://doi.org/10.1016/s0304-3975\(01\)00245-6](https://doi.org/10.1016/s0304-3975(01)00245-6)
14. Corradini, A., Duval, D., Echahed, R., Prost, F., Ribeiro, L.: AGREE – Algebraic Graph Rewriting with Controlled Embedding. In: *Graph Transformation*, pp. 35–51. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-21145-9_3
15. Corradini, A., Duval, D., Löwe, M., Ribeiro, L., Machado, R., Costa, A., Azzi, G.G., Bezerra, J.S., Rodrigues, L.M.: On the Essence of Parallel Independence for the Double-Pushout and Sesqui-Pushout Approaches. In: *Graph Transformation, Specifications, and Nets*, pp. 1–18. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-75396-6_1
16. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-Pushout Rewriting. In: *Lecture Notes in Computer Science*, pp. 30–45. Springer Berlin Heidelberg (2006). https://doi.org/10.1007/11841883_4

17. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. pp. 163–246. World Scientific (1997). https://doi.org/10.1142/9789812384720_0003
18. Danos, V., Feret, J., Fontana, W., Harmer, R., Hayman, J., Krivine, J., Thompson-Walsh, C., Winskel, G.: Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models. In: D’Souza, D., Kavitha, T., Radhakrishnan, J. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012). Leibniz International Proceedings in Informatics (LIPIcs), vol. 18, pp. 276–288. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2012). <https://doi.org/10.4230/LIPIcs.FSTTCS.2012.276>
19. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-based modelling, symmetries, refinements. In: Fisher, J. (ed.) Formal Methods in Systems Biology. pp. 103–122. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68413-8_8
20. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Abstracting the differential semantics of rule-based models: Exact and automated model reduction. In: 2010 25th Annual IEEE Symposium on Logic in Computer Science. IEEE (jul 2010). <https://doi.org/10.1109/lics.2010.44>
21. Danos, V., Laneve, C.: Formal molecular biology. Theoretical Computer Science **325**(1), 69–110 (2004). <https://doi.org/10.1016/j.tcs.2004.03.065>
22. Dattoli, G., Ottaviani, P.L., Torre, A., Vazquez, L.: Evolution operator equations: Integration with algebraic and finite-difference methods: Applications to physical problems in classical and quantum mechanics and quantum field theory. Riv. Nuovo Cim. **20N2**, 1–133 (1997). <https://doi.org/10.1007/BF02907529>
23. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: 14th Annual Symposium on Switching and Automata Theory (swat 1973). IEEE (Oct 1973). <https://doi.org/10.1109/swat.1973.11>
24. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: \mathcal{M} -adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation. Mathematical Structures in Computer Science **24**(04) (2014). <https://doi.org/10.1017/s0960129512000357>
25. Ehrig, H., Habel, A., Kreowski, H.J., Parisi-Presicce, F.: Parallelism and concurrency in high-level replacement systems. Mathematical Structures in Computer Science **1**(3), 361–404 (Nov 1991). <https://doi.org/10.1017/s0960129500001353>
26. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive High-Level Replacement Categories and Systems. In: Lecture Notes in Computer Science, pp. 144–160. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_12
27. Golas, U., Habel, A., Ehrig, H.: Multi-amalgamation of rules with application conditions in \mathcal{M} -adhesive categories. Mathematical Structures in Computer Science **24**(04) (jun 2014). <https://doi.org/10.1017/s0960129512000345>
28. Hall, B.C.: Lie Groups, Lie Algebras, and Representations. Springer International Publishing (2015). <https://doi.org/10.1007/978-3-319-13467-3>
29. Heckel, R.: Stochastic Analysis of Graph Transformation Systems: A Case Study in P2P Networks. In: Theoretical Aspects of Computing – ICTAC 2005, pp. 53–69. Springer Berlin Heidelberg (2005). https://doi.org/10.1007/11560647_4
30. Heckel, R., Ehrig, H., Golas, U., Hermann, F.: Parallelism and Concurrency of Stochastic Graph Transformations. In: Lecture Notes in Computer Science, pp. 96–110. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_7

31. Heckel, R., Lajos, G., Menge, S.: Stochastic Graph Transformation Systems. In: Lecture Notes in Computer Science, pp. 210–225. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_16
32. Kennaway, R.: Graph rewriting in some categories of partial morphisms. In: Lecture Notes in Computer Science, pp. 490–504. Springer-Verlag (1990). <https://doi.org/10.1007/bfb0017408>
33. Krause, C., Giese, H.: Probabilistic Graph Transformation Systems. In: Lecture Notes in Computer Science, pp. 311–325. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_21
34. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. *RAIRO - Theoretical Informatics and Applications* **39**(3), 511–545 (2005). <https://doi.org/10.1051/ita:2005028>
35. Löwe, M.: Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science* **109**(1-2), 181–224 (mar 1993). [https://doi.org/10.1016/0304-3975\(93\)90068-5](https://doi.org/10.1016/0304-3975(93)90068-5)
36. Löwe, M.: Polymorphic Sesqui-Pushout Graph Rewriting. In: Graph Transformation, pp. 3–18. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-21145-9_1
37. Löwe, M.: Characterisation of Parallel Independence in AGREE-Rewriting. In: Graph Transformation, pp. 118–133. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-92991-0_8
38. Löwe, M., König, H., Schulz, C.: Polymorphic Single-Pushout Graph Transformation. In: Fundamental Approaches to Software Engineering, pp. 355–369. Springer Berlin Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_25
39. Löwe, M., Tempelmeier, M.: Single-Pushout Rewriting of Partial Algebras. *GCM 2015 Graph Computation Models* p. 82 (2015)
40. Maximova, M., Giese, H., Krause, C.: Probabilistic timed graph transformation systems. *Journal of Logical and Algebraic Methods in Programming* **101**, 110–131 (Dec 2018). <https://doi.org/10.1016/j.jlamp.2018.09.003>
41. Norris, J.R.: Markov Chains. Cambridge University Press (1997). <https://doi.org/10.1017/cbo9780511810633>
42. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997). <https://doi.org/10.1142/9789812384720>

A A collection of useful technical results on adhesive categories and final pullback complements

Notational convention: Here and as throughout this paper, while evidently category-theoretical constructions such as pushouts are only unique up to isomorphisms, we will typically nevertheless pick convenient representatives of the respective isomorphism classes to simplify our notations. As standard practice in the literature, we will thus e.g. fix the convention as in (42) to choose representatives appropriately to label the pushout along an isomorphism with “equality arrows” (rather than keeping object labels generic and decorating the relevant arrow with a “ \cong ” symbol).

Lemma 2. *Let \mathbf{C} be a category.*

- (i) “Single-square” lemmata (see e.g. [8], Lem. 1.7): *In any category, given commutative diagrams of the form*

$$\begin{array}{ccc} A \xrightarrow{f} B & A \xlongequal{\quad} A & A \xrightarrow{f} B \\ \parallel (A) \parallel & \parallel (B) \downarrow g & \parallel (C) \downarrow g \\ A \xrightarrow{f} B & A \xrightarrow{g} B & A \xrightarrow{g \circ f} C \end{array}, \quad (42)$$

- (a) (A) is a pushout for arbitrary morphisms f ,
(b) (B) is a pullback if and only if the morphism g is a monomorphism, and
(c) (C) is a pullback for arbitrary morphisms f if g is a monomorphism.
(ii) special adhesivity corollaries (cf. e.g. [24], Lemma 2.6): *in any adhesive category,*
(a) pushouts along monomorphisms are also pullbacks, and
(b) (uniqueness of pushout complements) *given a monomorphism $A \hookrightarrow C$ and a generic morphism $C \rightarrow D$, the respective pushout complement $A \rightarrow B \xrightarrow{b} D$ (if it exists) is unique up to isomorphism, and with $b \in \text{mono}(\mathbf{C})$ (due to stability of monomorphisms under pushouts).*
(iii) “Double-square lemmata”: *given commutative diagrams of the shapes*

$$\begin{array}{ccc} A \xleftarrow{d} B \xleftarrow{e} C & & Z \xleftarrow{z} Z' \\ a \downarrow (1) \downarrow b \downarrow (2) \downarrow c & & w \downarrow (3) \downarrow w' \\ A' \xleftarrow{d'} B' \xleftarrow{e'} C' & & Y \xleftarrow{y} Y' \\ v \downarrow (4) \downarrow v' & & \\ X \xleftarrow{x} X' & & \end{array} \quad (43)$$

then in any category \mathbf{C} (cf. e.g. [34]):

- (a) Pullback-pullback (de-)composition: *If (1) is a pullback, then (1) + (2) is a pullback if and only if (2) is a pullback.*
(b) Pushout-pushout (de-)composition: *If (2) is a pushout, then (1) + (2) is a pushout if and only if (1) is a pushout.*

If the category is adhesive:

- (c) pushout-pullback decomposition ([24], Lemma 2.6): If (1) + (2) is a pushout, (1) is a pullback, and if $d' \in \mathbf{mono}(\mathbf{C})$ and ($c \in \mathbf{mono}(\mathbf{C})$ or $e \in \mathbf{mono}(\mathbf{C})$), then (1) and (2) are both pushouts (and thus also pullbacks).
- (d) pullback-pushout decomposition ([27], Lem. B.2): if (1) + (2) is a pullback, (2) a pushout, (1) commutes and $a \in \mathbf{mono}(\mathbf{C})$, then (1) is a pullback.
- (e) Horizontal FPC (de-)composition (cf. [16], Lem. 2 and Lem. 3, compare [36], Prop. 36).⁵ If (1) is an FPC (i.e. if (d', b) is FPC of (a, d)), then (1) + (2) is an FPC if and only if (2) is an FPC.
- (f) Vertical FPC (de-)composition (ibid): if (3) is an FPC (i.e. if (v, w') is FPC of (w, z)), then
 - i. if (4) is an FPC (i.e. if (x, v') is FPC of (v, y)), then (3) + (4) is an FPC (i.e. $(x, v' \circ w')$ is FPC of $(v \circ w, z)$);
 - ii. if (3) + (4) is an FPC (i.e. if $(x, v' \circ w')$ is FPC of $(v \circ w, z)$), and if (4) is a pullback, then (4) is an FPC (i.e. (x, v') is FPC of (v, y)).
- (g) Vertical FPC-pullback decomposition (compare [36], Lem. 38): If $v \in \mathbf{mono}(\mathbf{C})$, if (4) is a pullback and if (3) + (4) is an FPC (i.e. if $(x, v' \circ w')$ is FPC of $(v \circ w, z)$), then (3) and (4) are FPCs.
- (h) Vertical FPC-pushout decomposition⁶: If all morphisms of the squares (3) and (4) except v are in $\mathbf{mono}(\mathbf{C})$, if $v \circ w \in \mathbf{mono}(\mathbf{C})$, if (3) is a pushout and if (3) + (4) is an FPC (i.e. if $(x, v' \circ w')$ is FPC of $(v \circ w, z)$), then (4) is an FPC and $v \in \mathbf{mono}(\mathbf{C})$.

Proof. Referring to the references above for the proofs of the (well-known) statements (where necessary by specializing the more general case of \mathcal{M} -adhesive categories to the case of adhesive categories via setting \mathcal{M} to the class of all monomorphisms), it remains to prove our novel vertical FPC-pushout decomposition result. To this end, we first invoke pullback-pushout decomposition (Lemma 2(3d)) in order to demonstrate that since (3) + (4) is an FPC and thus also a pullback, and since (3) is a pushout and since $x \in \mathbf{mono}(\mathbf{C})$, (4) is a pullback. By applying vertical FPC-pullback decomposition, we may conclude that (4) is an FPC. In order to demonstrate that $v \in \mathbf{mono}(\mathbf{C})$, construct the commutative

⁵ It is worthwhile emphasizing that in these FPC-related lemmata, the “orientation” of the diagrams plays an important role. Moreover, the precise identity of the pair of morphisms that plays the role of the final pullback complement in a given square may be inferred from the “orientation” specified in the condition part of each statement.

⁶ We invite the interested readers to compare the precise formulation of the vertical FPC-pushout decomposition result to its concrete applications in the setting of the proof of the concurrency theorem, for which it has been specifically developed.

cube below left:

$$\begin{array}{ccc}
 & Z & \xleftarrow{z} Z' \\
 w \swarrow & \parallel & \swarrow w' \\
 Y & \xleftarrow{y} Y' & \\
 v \downarrow & \parallel & \downarrow v' \\
 & Z & \xleftarrow{z} Z' \\
 x \swarrow & & \swarrow \\
 X & \xleftarrow{x} X' &
 \end{array}
 \qquad
 \begin{array}{ccc}
 & Z & \xleftarrow{z} Z' \\
 w \downarrow & (3) & \downarrow w' \\
 Y & \xleftarrow{y} Y' & \\
 v \swarrow & & \swarrow v \circ y \\
 X & \xleftarrow{v \circ w} &
 \end{array}
 \tag{44}$$

Since the bottom square is the FPC (and thus pullback) (3) + (4), and since the right square is a pullback via Lemma 2(1c) (because $v' \in \mathbf{mono}(\mathbf{C})$), by pullback composition the square $\square(Z', Z, X, Y')$ (the right plus the bottom square) is a pullback. Thus assembling the commutative diagram as shown above right, since by assumption (3) is a pushout and all arrows except v are monomorphisms, invoking Theorem 7 permits to prove that also $v \in \mathbf{mono}(\mathbf{C})$.

Next, let us highlight one of the quintessential properties of adhesive categories in view of associative rewriting theories:

Theorem 7 (Effective unions; [34], Thm. 5.1). *In an adhesive category \mathbf{C} , given a commutative diagram such as in the middle of (1), if all morphisms except the morphism x are monomorphisms, if the square marked (A) is a pushout and if the exterior square is a pullback, then x is also a monomorphism.*

The following result provides several important facts on FPCs.

Lemma 3 (cf. [36], Fact 2, and [16], Lemma 2 and Proposition 2). *Let \mathbf{C} be adhesive. For an arbitrary morphism $f : A \rightarrow B$, (id_B, f) is an FPC of (f, id_A) and vice versa. Moreover, every pushout square along monomorphisms is also an FPC square. FPCs are unique up to isomorphism and preserve monomorphisms. The latter property entails that if $C \xleftarrow{d} D \xleftarrow{b} A$ is the FPC of $C \xleftarrow{c} B \xleftarrow{a} A$ and if $a \in \mathbf{mono}(\mathbf{C})$, then also $d \in \mathbf{mono}(\mathbf{C})$ and vice versa (while $c \in \mathbf{mono}(\mathbf{C})$ entails that $b \in \mathbf{mono}(\mathbf{C})$ by stability of monomorphisms under pullbacks in an adhesive category \mathbf{C}).*

For concreteness, we quote the following explicit construction of FPCs in the category **Graph** of directed multigraphs:

Lemma 4 (FPCs for graphs; [16], Sec. 4.1 and Construction 5). *Let **Graph** denote the adhesive category of directed multigraphs, with*

- $\mathbf{obj}(\mathbf{Graph})$: (multi-)graphs, i.e. tuples $G = (V_G, E_G, \mathbf{src}_G : E_G \rightarrow V_G, \mathbf{trg}_G : E_G \rightarrow V_G)$, with V_G the set of vertices, E_G the set of edges (with $V_G \cap E_G = \emptyset$), \mathbf{src}_G the source and \mathbf{trg}_G the target maps

- **mor(Graph)**: graph homomorphisms $f : G \rightarrow H$, specified in terms of pairs of morphisms $(f_V : V_G \rightarrow V_H, f_E : E_G \rightarrow E_H)$ such that $\text{src}_H \circ f_E = f_V \circ \text{src}_G$ and $\text{trg}_H \circ f_E = f_V \circ \text{trg}_G$.

Let **mono(Graph)** denote the class of all injective graph morphisms. Then for every composable pair of monomorphisms $K \xrightarrow{i} I \xrightarrow{m} X$, the FPC exists and is constructed explicitly as⁷ $K \xrightarrow{m|_K} \bar{K} \xrightarrow{\subseteq} X$, where $\xrightarrow{\subseteq}$ denotes an inclusion morphism, and where the graph \bar{K} reads

$$\begin{aligned} V_{\bar{K}} &= V_X \setminus m[V_I \setminus V_K] \\ E_{\bar{K}} &= \{e \in E_X \setminus m[E_I \setminus E_K] \mid \text{src}_X(e) \in V_{\bar{K}} \wedge \text{trg}_X(e) \in V_{\bar{K}}\}. \end{aligned} \quad (45)$$

B Proofs

B.1 Proof of the SqPO concurrency theorem

Proof. Throughout this proof, in each individual constructive step it may be verified that due to the stability of monomorphisms under pullbacks and pushouts, due to the various decomposition lemmata provided in the form of Lemma 2, and on occasion due to Theorem 7 on effective unions in adhesive categories, all morphisms induced in the “Synthesis” and “Analysis” steps are in fact monomorphisms. For better readability, we will not explicitly mention the individual reasoning steps on this point except for a few intricate sub-steps, since they may be recovered in a straightforward manner.

— **Synthesis:** Consider the setting presented in (46a). Here, we have obtained the candidate match $\mathbf{n} = (I_2 \leftarrow M_{21} \rightarrow O_1)$ via pulling back the cospan $(I_2 \rightarrow X_1 \leftarrow O_1)$. Next, we construct N_{21} via taking the pushout of \mathbf{n} , which induces a unique arrow $N_{21} \rightarrow X_1$ that is according to Theorem 7 a monomorphism. The diagram in (46b) is obtained by taking the pullbacks of the spans $\bar{K}_i \rightarrow X_1 \leftarrow N_{21}$ (obtaining the objects K'_i , for $i = 1, 2$). By virtue of pushout-pullback decomposition (Lemma 2(3c)), the squares $\square(K'_1, \bar{K}_1, X_1, N_{21})$ and $\square(K_1, K'_1, N_{21}, O_1)$ are pushouts. Invoking vertical FPC-pullback decomposition (Lemma 2(3g)), the squares $\square(K'_2, N_{21}, X_1, \bar{K}_2)$ and $\square(K_2, I_2, N_{21}, K'_2)$ are FPCs. Next, letting $O_{21} := \text{PO}(O_2 \leftarrow K_2 \rightarrow K'_2)$ and $I_{21} := \text{PO}(O_1 \leftarrow K_1 \rightarrow K'_1)$, we have via vertical FPC-pushout decomposition (Lemma 2(3h)) that the resulting two squares on the very right (the ones involving I_{21}) are FPCs and that the arrow $I_{21} \rightarrow X_0$ is a monomorphism, while pushout-pushout decomposition (Lemma 2(3b)) entails that the two newly formed squares on the very left (the ones involving O_{21}) are pushouts.

The final step as depicted in (46c) consists in constructing $K_{21} = \text{PB}(K'_2 \rightarrow N_{21} \leftarrow K'_1)$ and $\bar{K}_{21} = \text{PB}(\bar{K}_2 \rightarrow X_1 \leftarrow \bar{K}_1)$, which by universality of pullbacks

⁷ The quoted Construction 5 of [16] is slightly more general, in that the morphism m may be permitted to not be a monomorphism; we will however have no application for such a generalization in our framework.

induces a unique arrow $K_{21} \rightarrow \overline{K}_{21}$. By invoking pullback-pullback decomposition (Lemma 2(3a)), one may demonstrate that the squares $\square(K_{21}, \overline{K}_{21}, \overline{K}_i, K'_i)$ (for $i = 1, 2$) are pullbacks. Since the square $\square(K'_1, \overline{K}_1, X_1, N_{21})$ is a pushout, via the van Kampen property (cf. Def. 1) the square $\square(K_{21}, \overline{K}_{21}, \overline{K}_2, K'_2)$ is a pushout. Since according to Lemma 3 pushouts are also FPCs, it follows via horizontal composition of FPCs (Lemma 2(3e)) that the square $\square(K_{21}, \overline{K}_{21}, X_1, N_{21})$ is an FPC. Noting that the pushout square $\square(K'_1, \overline{K}_1, X_1, N_{21})$ is an FPC as well, it follows via horizontal decomposition of FPCs (Lemma 2(3e)) that $\square(K_{21}, \overline{K}_{21}, \overline{K}_1, K'_1)$ is an FPC. Thus the claim follows by invoking pushout composition (Lemma 2(3b)) and horizontal FPC composition (Lemma 2(3e)) in order to obtain the pushout square $\square(K_{21}, \overline{K}_{21}, X_2, O_{21})$ and the FPC square $\square(K_{21}, \overline{K}_{21}, X_0, I_{21})$.

— **Analysis:** Given the setting as depicted in (47a), where the top row has the structure of an SqPO-composition (compare (4)), where the square $\square(K_{21}, K'_1, N_{21}, K'_2)$ is a pullback, the left “curvy” bottom square a pushout and the right “curvy” bottom square an FPC, we may obtain the configuration of (47c) as follows: construct⁸ \overline{K}_1 via taking the final pullback complement of $K'_1 \rightarrow I_{21} \rightarrow X_0$ (which implies the existence of an arrow $\overline{K}_{21} \rightarrow \overline{K}_1$ via the FPC property). Note in particular that according to Lemma 3, both arrows constructed via forming the aforementioned FPC are monomorphisms, and thus by stability of monomorphisms in an adhesive category (compare Definition 1 and Lemma 2(1c)), the arrow $\overline{K}_{21} \rightarrow \overline{K}_1$ is a monomorphism as well. Next, take the pushout $X_1 = \text{PO}(\overline{K}_1 \leftarrow K'_1 \rightarrow N_{21})$, followed by constructing \overline{K}_2 as the final pullback complement of $K'_2 \rightarrow N_{21} \rightarrow X_1$ (which implies due to the FPC property of the resulting square $\square(K'_2, \overline{K}_2, X_1, N_{21})$ the existence of an arrow $\overline{K}_{21} \rightarrow \overline{K}_2$). Invoking pullback-pullback decomposition (Lemma 2(3a)) twice, followed by the van Kampen property (Def. 1), we may conclude that the square $\square(K_{21}, \overline{K}_{21}, \overline{K}_2, K'_2)$ is a pushout. Thus invoking pushout-pushout decomposition (Lemma 2(3b)), we find that also $\square(K'_2, \overline{K}_2, X_2, O_{21})$ is a pushout. We finally arrive at the configuration in (47d) via composition of pushout and FPC squares, respectively, thus concluding the proof.

B.2 Proof of the SqPO associativity theorem

Our proof strategy will be closely related to the one presented in [7] (with full technical details provided in [8]) for the analogous associativity theorem in the DPO-rewriting case. However, the SqPO-type case poses considerable additional challenges, since this rewriting semantics yields diagrams of a rather heterogeneous nature (including pullbacks, pushouts, pushout complements and FPCs) as compared to the DPO case, and in addition unlike DPO-type rule compositions, SqPO-type compositions are not reversible in general, which necessitates an independent proof of both directions of the bijective correspondence.

⁸ Note that it is precisely in this step and the following step that we require the existence of FPCs for arbitrary pairs of monomorphisms as per Assumption 1.

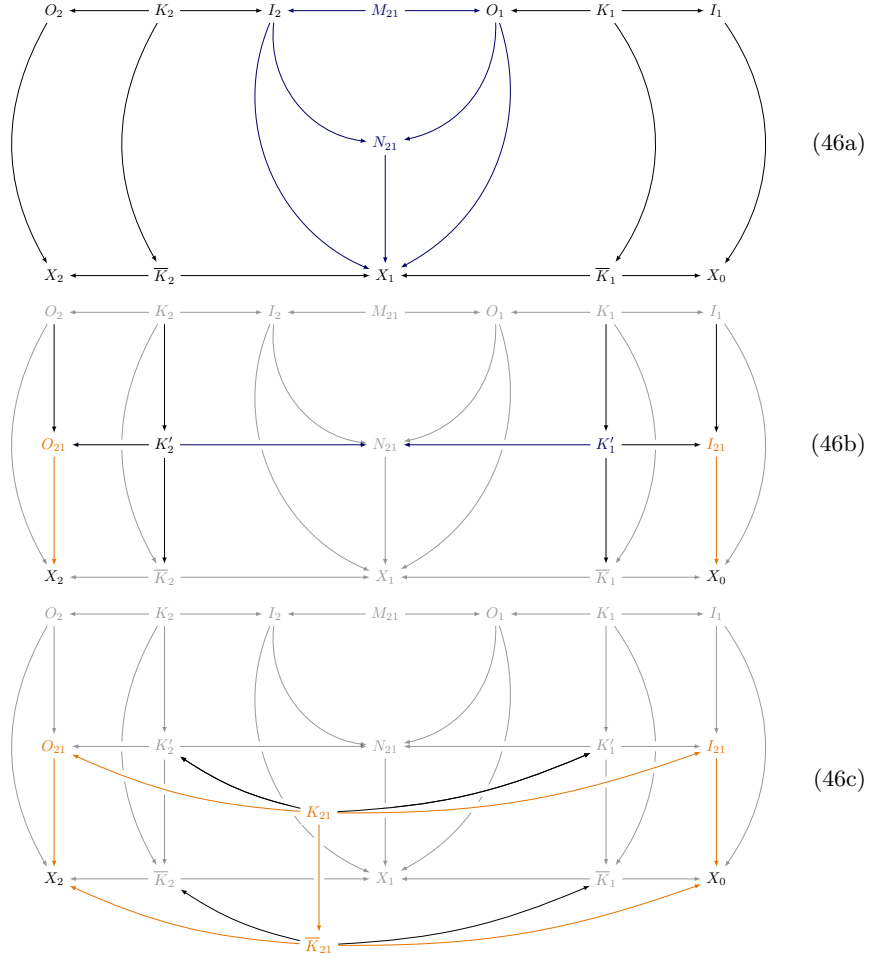


Fig. 2. *Synthesis* part of the concurrency theorem.

Proof. We first prove the claim in the “ \Rightarrow ” direction, i.e. starting from the set of data

$$\begin{aligned}
 \mathbf{m}_{21} &= (O_2 \leftarrow M_{21} \rightarrow I_1) \in \mathbf{M}_{p_2}^{sq}(p_1) \\
 \mathbf{m}_{3(21)} &= (O_3 \leftarrow M_{3(21)} \rightarrow I_{21}) \in \mathbf{M}_{p_3}^{sq}(p_{21}), \quad p_{21} = p_2 \stackrel{\mathbf{m}_{21}}{\triangleleft} p_1,
 \end{aligned} \tag{48}$$

we have to demonstrate that one may uniquely (up to isomorphisms) construct from this information a pair of admissible matches

$$\begin{aligned} \mathbf{m}_{32} &= (O_3 \leftarrow M_{32} \rightarrow I_2) \in \mathbf{M}_{p_3}^{sq}(p_2) \\ \mathbf{m}_{(32)1} &= (O_{32} \leftarrow M_{(32)1} \rightarrow I_1) \in \mathbf{M}_{p_{32}}^{sq}(p_1), \quad p_{32} = p_3 \stackrel{\mathbf{m}_{32}}{\triangleleft} p_2, \end{aligned} \quad (49)$$

and such that the property described in (6) holds. We begin by forming the SqPO-composite rule $p_{3(21)} = p_3 \stackrel{\mathbf{m}_{3(21)}}{\triangleleft} p_{21}$, which results in the diagram

by virtue of invoking SqPO-composition twice. For the remainder of the proof, it is very important to precisely determine the nature of each of the squares in this diagram:

- To clarify the structure of the the rightmost four squares at the bottom, consider the setting presented in (47a): since in the definition of the SqPO-composition as presented in (5) the nature of the squares to the right is that of pushout complement and pushout, respectively, it may be verified that applying the analysis procedure to the diagram in (47a) with thus the “curvy” front and right bottom faces both pushouts, one eventually arrives (by virtue of pushout-pushout and pushout-pullback decomposition) at the setting depicted in (47c) with all squares in the bottom row being pushouts. Thus the rightmost four squares at the bottom of (50) are all pushouts.
- By virtue of the definition of SqPO-composition, all vertical squares in the back of (47c) are pushouts, except for the square $\square(K_2, K'_2, N_{21}, I_2)$, which is an FPC. Analogously, the bottom leftmost three squares are (in order from left to right) a pushout, an FPC and a pushout.

Constructing the pullback $M_{32} = \text{PB}(M_{3(21)} \rightarrow O_{21} \leftarrow O_2)$ (which by universality of pullbacks also leads to an arrow $M_{32} \rightarrow I_3$) and forming the three additional pullbacks on the far left in the evident fashion in the diagram below

allows us to construct $N_{32} = \text{PO}(I_3 \leftarrow M_{32} \rightarrow O_2)$, which in turn via universality of pushouts uniquely induces an arrow $N_{32} \rightarrow N_{3(21)}$:

Here, the rightmost three squares on the top are formed in the evident fashion (and are pushouts by universal category theory), while the other arrows of the above diagram are constructed as follows:

$$\begin{aligned} K'_3 &= \text{PB}(\overline{K}_3 \rightarrow N_{3(21)} \leftarrow N_{32}), & O_{32} &= \text{PO}(K'_3 \leftarrow K_3 \rightarrow O_3) \\ K''_2 &= \text{PB}(N_{32} \rightarrow N_{3(21)} \leftarrow \overline{K}_2), & I_{32} &= \text{PO}(K''_2 \leftarrow K_2 \rightarrow I_2) \end{aligned} \quad (53)$$

Invoking pushout-pullback, pushout-pushout and vertical FPC-pullback decompositions, it may be verified that (describing positions of front and top square pairs by the position of the respective front square, from left to right)

- the leftmost front and top squares are pushouts,
- the second from the left front and top squares are FPCs,
- the third from the left front and top squares are pushouts,
- in the next adjacent pair, the front square is an FPC and the top square a pushout,
- the second from the right front and top squares are pushouts, and
- the rightmost front and top squares are pushouts.

Defining the pullback object $M_{(32)1} = \text{PB}(I_{32} \rightarrow N_{3(21)} \leftarrow O_1)$, thus inducing an arrow $M_{21} \rightarrow M_{3(21)}$,

it remains to verify that the square $\square(M_{3(21)}, I_{32}, N_{3(21)}, O_1)$ is not only a pullback, but also a pushout square. This part of the proof requires a somewhat intricate diagram chase; since the required arguments are identical⁹ to the “ \Rightarrow ” part of proof of DPO-type associativity as presented in [8], we omit this part of

⁹ More precisely, the only difference in the structure of the relevant sub-diagram compared to the DPO case resides in the two FPC squares in the front and back in fourth position from the left (which happen to be pushout squares in the corresponding DPO-type proof), but the structure of this part of the diagram is not explicitly used in the proof in the DPO variant of the theorem, whence the claim follows.

the proof here in the interest of brevity.

It thus remains to prove the claim in the “ \Leftarrow ” direction, i.e. starting from the set of data

$$\begin{aligned} \mathbf{m}_{32} &= (O_3 \leftarrow M_{32} \rightarrow I_2) \in \mathbf{M}_{p_3}^{sq}(p_2) \\ \mathbf{m}_{(32)1} &= (O_{32} \leftarrow M_{(32)1} \rightarrow I_1) \in \mathbf{M}_{p_{32}}^{sq}(p_1), \quad p_{32} = p_3 \triangleleft^{\mathbf{m}_{32}} p_2, \end{aligned} \quad (55)$$

we need to demonstrate that one may uniquely (up to isomorphisms) construct from this information a pair of admissible matches

$$\begin{aligned} \mathbf{m}_{21} &= (O_2 \leftarrow M_{21} \rightarrow I_1) \in \mathbf{M}_{p_2}^{sq}(p_1) \\ \mathbf{m}_{3(21)} &= (O_3 \leftarrow M_{3(21)} \rightarrow I_{21}) \in \mathbf{M}_{p_3}^{sq}(p_{21}), \quad p_{21} = p_2 \triangleleft^{\mathbf{m}_{21}} p_1, \end{aligned} \quad (56)$$

and such that the property described in (6) holds. We begin by forming the SqPO-composite rule $p_{(32)1} = p_{32} \triangleleft^{\mathbf{m}_{(32)1}} p_1$, which results in the diagram

(57)

by virtue of invoking SqPO-composition twice. A careful inspection of the definition of the SqPO-composition and of the analysis part of the SqPO concurrency theorem permit to verify that the nature of all squares thus constructed coincides precisely with the nature of the corresponding squares in the “ \Rightarrow ” part of the proof.

Constructing the pullback $M_{21} = \text{PB}(I_2 \rightarrow I_{32} \leftarrow M_{(32)1})$ (which by the universal property of pullbacks also leads to an arrow $M_{21} \rightarrow O_1$) and forming the three additional vertical squares on the far right in the evident fashion in the diagram below

(58)

allows us to construct $N_{21} = \text{PO}(I_2 \leftarrow M_{21} \rightarrow O_1)$, which in turn via the universal property of pushouts induces an arrow $N_{21} \rightarrow N_{(32)1}$:

(59)

The remaining new squares of the above diagram are constructed as follows:

$$K'_2 = \text{PB}(\overline{K}_2 \rightarrow N_{(32)1} \leftarrow N_{21}) \quad O_{211} = \text{PO}(O_2 \leftarrow K_2 \rightarrow K'_2). \quad (60)$$

Moreover, by virtue of vertical FPC composition, the square $\square(K_3, \overline{K}_3, N_{3(21)}, I_3)$ is an FPC, while via pushout composition the square $\square(K'_3, \overline{K}_3, O_{321}, O_3)$ is a pushout.

Again, the nature of all squares constructed thus far coincides precisely with the structure as presented in the “ \Rightarrow ” part of the proof, with one notable exception: by virtue of vertical FPC-pullback decomposition, we may only conclude that the square $\square(K'_2, \overline{K}_2, N_{(32)1}, N_{21})$ is an FPC (but at this point we do *not* know whether it is also a pushout as in the analogous part of the diagram in the “ \Rightarrow ” part of the proof). However, an auxiliary calculation demonstrates that this square is in fact a pushout in disguise — consider the following “splitting” of the relevant sub-part of the diagram as shown below:

$$(61)$$

The precise steps are as follows: the front left square in the diagram above left is an FPC; thus if one takes the pushout $N_{21}' = \text{PO}(\overline{K}_2 \leftarrow K'_2 \rightarrow N_{21})$ as well as in the bottom back the pushout along the isomorphism of K'_2 as displayed (yielding the arrows in the middle), followed by taking the pullback $O'_1 = \text{PB}(N_{21}', N_{(32)1}, O_1)$ (which entails that the arrow $M_{3(21)} \rightarrow O'_1$ exists), it is straightforward to verify that $O'_1 \cong O_1$. Invoking the van Kampen property (recalling that by definition the right square on the bottom is a pushout), we find that $\square(M_{3(21)}, O'_1, N_{21}', I_{32})$ is a pushout. Thus by pushout-pushout decomposition, the square $\square(O'_1, O_1, N_{(32)1}, N_{21}')$ is a pushout, whence $N_{21}' \cong N_{21}$. This in summary entails¹⁰ that the square $\square(K'_2, \overline{K}_2, N_{(32)1}, N_{21})$ is not only an FPC, but in fact also a pushout.

¹⁰ Coincidentally, at this point we are back into full structural analogy to the “ \Rightarrow ” part of the proof, a necessary prerequisite for completing this part of the proof as it will turn out.

Back to the main proof, defining the pullback object

$$M_{3(21)} = \text{PB}(I_3 \rightarrow N_{(32)1} \leftarrow O_{21}),$$

thus inducing an arrow $M_{32} \rightarrow M_{(32)1}$,

it remains to verify that the square $\square(M_{(32)1}, O_{21}, N_{(32)1}, I_3)$ is not only a pullback, but also a pushout square. Let us construct the auxiliary diagram as depicted in Figure 4, with objects obtained via taking suitable pullbacks as indicated. The four cubes that are drawn separately are the top, back, bottom and front cubes induced via the newly constructed arrows, and are oriented such that one may easily apply the van Kampen property in the next step of the proof (which in most cases requires a suitable 3d-rotation).

Invoking pullback-pullback decomposition and the van Kampen property repeatedly, it may be verified that in the relevant sub-diagram as presented below

we find the following structure of the squares:

- All squares on the top are pushouts, except the second one from the right (which is a pullback).
- The second and third square from the left in the back of the diagram are pushouts, the other two back squares are pullbacks, with the same structure for the front squares.
- Counting from left to right, the second and fourth square on the bottom are pushouts, the other two are pullbacks.

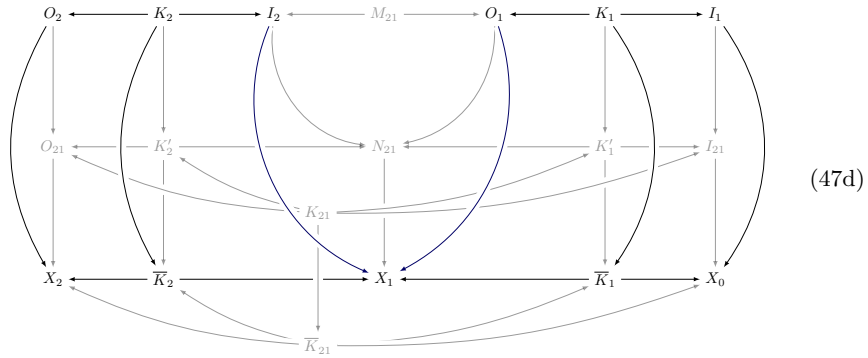
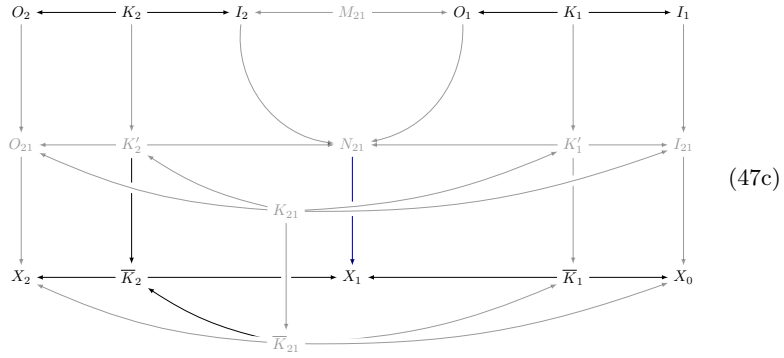
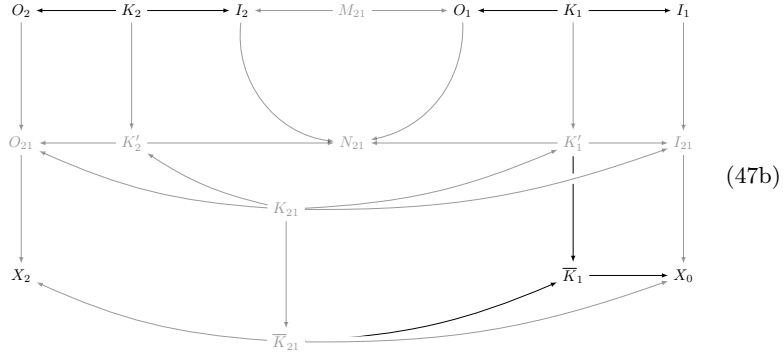
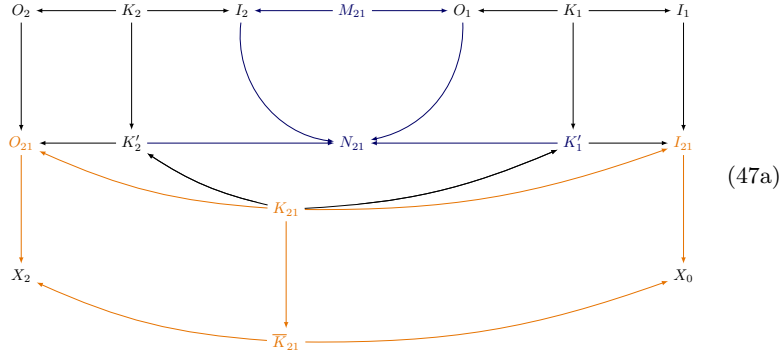


Fig. 3. Analysis part of the concurrency theorem.

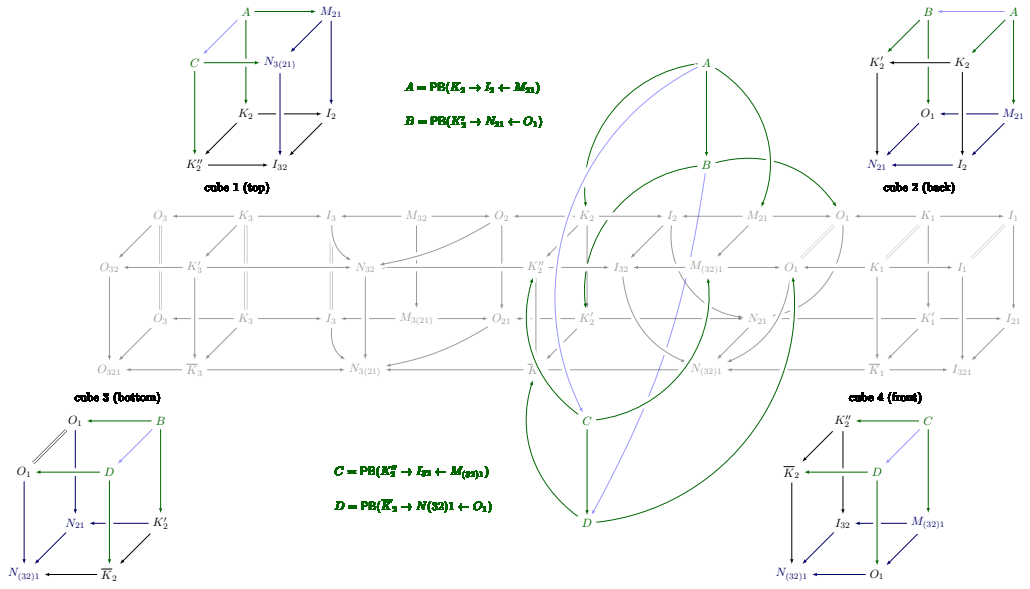


Fig. 4. Auxiliary diagram for the second part of the SqPO associativity proof.

In particular, as indicated this entails that $D \cong B$. We proceed by performing the following “splitting” of the diagram:

Start the “splitting” via taking the pushouts $N'_{32} = \text{PO}(I_3 \leftarrow I_3 \rightarrow N_{32})$ (which entails that $N'_{32} \cong N_{32}$) and $O'_2 = \text{PO}(M_{3(21)} \leftarrow M_{32} \rightarrow O_2)$. By pullback-pullback decomposition followed by pushout-pullback decomposition, we may conclude that the resulting square $\square(O'_2, N'_{32}, I_3, M_{3(21)})$ is a pushout. Note also that all vertical squares in the bottom left part of the diagram thus constructed are pullbacks (by virtue of suitable pullback decompositions).

Next, construct the pullbacks $\bar{K}'_2 = \text{PB}(N'_{21} \rightarrow N_{3(21)} \leftarrow \bar{K}_2)$ and $K'''_2 = \text{PB}(O'_2 \rightarrow O_{21} \leftarrow K'_2)$. By pushout-pullback decomposition, in the diagram on the right the top and bottom front squares and back squares in the second column are pushouts, while the square $\square(K'''_2, K'_2, \bar{K}_2, \bar{K}'_2)$ is a pullback. Performing the precise same steps in the next column, i.e. via taking the pullbacks $D' = \text{PB}(\bar{K}'_2 \rightarrow \bar{K}_2 \leftarrow D)$ and $B' = \text{PB}(K'''_2 \rightarrow K'_2 \leftarrow B)$, we obtain via pushout-pullback decomposition that the top and bottom front and top and bottom back squares in the third column are pushouts, while the square $\square(B', B, D, D')$ is a pullback. But since $D \cong B$ and since by universal category theory isomorphisms are stable under pullback, we conclude that $D' \cong B'$, and thus that the square $\square(B', B, D, D')$ is in fact a pushout.

By pushout composition, we may thus conclude that $\square(K'''_2, K'_2, \bar{K}_2, \bar{K}'_2)$ is a pushout, whence $\square(O'_2, O_{21}, N_{3(21)}, N'_{32})$ is a pushout, which finally allows us to conclude that $\square(M_{3(21)}, O_{21}, N_{3(21)}, I_3)$ is a pushout. This concludes the proof of the SqPO-type associativity theorem.

B.3 Proof of the theorem on SqPO rule algebra properties

Proof. Associativity of $\odot_{\mathcal{R}_C}$ follows from the associativity of the operation $\cdot \triangleleft \cdot$ proved in Theorem 3. The claim that $R_\emptyset = \delta(\emptyset \leftarrow \emptyset \rightarrow \emptyset)$ is the unit element of the rule algebra \mathcal{R}_C^{sq} follows directly from the definition of the rule algebra

product for $R_{\emptyset} \odot_{\mathcal{R}_{\mathbf{C}}} R$ and $R \odot_{\mathcal{R}_{\mathbf{C}}} R_{\emptyset}$ for $R \in \mathcal{R}_{\mathbf{C}}^{sq}$. More concretely, we present below the category-theoretic composition calculation that underlies the equation $R_{\emptyset} \odot_{\mathcal{R}_{\mathbf{C}}} R = R$:

$$\begin{array}{ccccccc}
 \emptyset & \xleftarrow{\quad} & \emptyset & \xrightarrow{\quad} & \emptyset & \xleftarrow{\quad} & \emptyset & \xrightarrow{\quad} & O & \xleftarrow{o} & K & \xrightarrow{i} & I \\
 \downarrow & \text{PO} & \downarrow & \text{FPC} & \downarrow & \text{PO} & \downarrow & \text{POC} & \downarrow & \text{PO} & \downarrow & \text{PO} & \downarrow \\
 O & \xleftarrow{\quad} & O & \xrightarrow{\quad} & O & \xleftarrow{o} & K & \xrightarrow{i} & I
 \end{array} \quad (65)$$

Here, it is important to note that the pushout complement used to construct the square marked POC always exists (see Lemma 2(1a)), whence the claim follows.

B.4 Proof of the SqPO canonical representation theorem

Proof. In order for $\rho_{\mathbf{C}}^{sq}$ to qualify as an algebra homomorphism (of unital associative algebras $\mathcal{R}_{\mathbf{C}}^{sq}$ and $End(\hat{\mathbf{C}})$), we must have (with $R_{\emptyset} = \delta(p_{\emptyset})$, $p_{\emptyset} = (\emptyset \leftarrow \emptyset \rightarrow \emptyset)$)

$$\begin{aligned}
 (i) \quad & \rho_{\mathbf{C}}^{sq}(R_{\emptyset}) = \mathbb{1}_{End(\hat{\mathbf{C}})} \\
 (ii) \quad & \forall R_1, R_2 \in \mathcal{R}_{\mathbf{C}}^{sq} : \rho_{\mathbf{C}}^{sq}(R_1 \odot_{\mathcal{R}_{\mathbf{C}}} R_2) = \rho_{\mathbf{C}}^{sq}(R_1) \rho_{\mathbf{C}}^{sq}(R_2) .
 \end{aligned}$$

Due to linearity, it suffices to prove the two properties on basis elements $\delta(p), \delta(q)$ of $\mathcal{R}_{\mathbf{C}}^{sq}$ (for $p, q \in Lin(\mathbf{C})$) and on basis elements $|X\rangle$ of $\hat{\mathbf{C}}$. Property (i) follows directly from the definition,

$$\forall X \in \text{obj}(\mathbf{C})_{\cong} : \quad \rho_{\mathbf{C}}^{sq}(R_{\emptyset}) |X\rangle \stackrel{(16)}{=} \sum_{m \in \mathbf{M}_{r_{\emptyset}}^{sq}(X)} |(r_{\emptyset})_m(X)\rangle = |X\rangle .$$

Property (ii) follows from Theorem 2 (the SqPO-type concurrency theorem): for all basis elements $\delta(p), \delta(q) \in \mathcal{R}_{\mathbf{C}}^{sq}$ (with $p, q \in Lin(\mathbf{C})$) and for all $X \in \text{obj}(\mathbf{C})$,

$$\begin{aligned}
 \rho_{\mathbf{C}}^{sq}(\delta(q) \odot_{\mathbf{C}} \delta(p)) |X\rangle & \stackrel{(11)}{=} \sum_{\mathbf{d} \in \mathbf{M}_q^{sq}(p)} \rho_{\mathbf{C}}^{sq} \left(\delta \left(q \overset{\mathbf{d}}{\triangleleft} p \right) \right) |X\rangle \\
 & \stackrel{(16)}{=} \sum_{\mathbf{d} \in \mathbf{M}_q^{sq}(p)} \sum_{e \in \mathbf{M}_{r_{\mathbf{d}}}^{sq}(X)} |(r_{\mathbf{d}})_e(X)\rangle \quad (r_{\mathbf{d}} = q \overset{\mathbf{d}}{\triangleleft} p) \\
 & = \sum_{m \in \mathbf{M}_p^{sq}(X)} \sum_{n \in \mathbf{M}_q^{sq}(p_m(X))} |q_n(p_m(X))\rangle \quad (\text{via Thm. 2}) \\
 & \stackrel{(16)}{=} \sum_{m \in \mathbf{M}_p^{sq}(X)} \rho_{\mathbf{C}}^{sq}(\delta(q)) |p_m(X)\rangle \\
 & \stackrel{(16)}{=} \rho_{\mathbf{C}}^{sq}(\delta(q)) \rho_{\mathbf{C}}^{sq}(\delta(p)) |X\rangle .
 \end{aligned}$$

B.5 Proof of the SqPO stochastic mechanics framework theorem

Proof. By definition, the SqPO-type canonical representation of a generic rule algebra element $(O \xleftarrow{p} I) \in \mathcal{R}_{\mathbf{C}}$ is a row-finite linear operator, since by virtue of the finitariness of objects according to Assumption 2, for every object $X \in \text{obj}(\mathbf{C})$ the set of SqPO-admissible matches $M_p^{sq}(X)$ of the associated linear rule $p = (O \xleftarrow{o} K \xrightarrow{i} I)$ is finite. We may thus verify that the linear operator H possesses all the required properties of a so-called Q -matrix (or infinitesimal generator) of a CTMC [41,1], i.e. its non-diagonal entries are non-negative, its diagonal entries are finite, and furthermore the row sums of H are zero (whence H constitutes a conservative and stable Q -matrix; compare (21) of Definition 9). It is crucial to note that while originally H as a linear combination of representations of rule algebra elements is only defined to act on *finite* linear combinations of basis vectors $|X\rangle$ of $\hat{\mathbf{C}}$, an important mathematical result from the theory of CTMCs entails that if a row-finite linear operator such as H is a stable and conservative Q -matrix, it extends to a linear operator on *infinitely supported distributions* (here over basis vectors of $\hat{\mathbf{C}}$) with finite real coefficients (see e.g. [1], Chapters 1 and 2). Moreover, the property $\langle |H = 0$ follows directly from the defining equations (21) of Definition 9.

Let us prove next the claim on the precise structure of observables. Recall that according to Definition 10, an observable $O \in \mathcal{O}_{\mathbf{C}}$ must be a linear operator in $\text{End}(\mathcal{S}_{\mathbf{C}})$ that acts diagonally on basis states $|X\rangle$ (for $X \in \text{obj}((\mathbf{C})_{\cong})$), whence that satisfies for all $X \in \text{obj}(\mathbf{C})_{\cong}$

$$O |X\rangle = \omega_O(X) |X\rangle \quad (\omega_O(X) \in \mathbb{R}).$$

Comparing this equation to the definition of the SqPO-type canonical representation (Definition 8) of a generic rule algebra basis element $\delta(p) \in \mathcal{R}_{\mathbf{C}}^{sq}$ (for $p \equiv (O \xleftarrow{o} K \xrightarrow{i} I) \in \text{Lin}(\mathbf{C})$),

$$\rho_{\mathbf{C}}^{sq}(\delta(p)) |X\rangle := \begin{cases} \sum_{m \in M_p^{sq}(X)} |p_m(X)\rangle & \text{if } M_p^{sq}(X) \neq \emptyset \\ 0_{\hat{\mathbf{C}}} & \text{else,} \end{cases}$$

we find that in order for $\rho_{\mathbf{C}}^{sq}(\delta(p))$ to be diagonal we must have

$$\forall X \in \text{obj}(\mathbf{C}) : \forall m \in M_p^{sq}(X) : \quad p_m(X) \cong X.$$

But by definition of SqPO-type derivations of objects along admissible matches (Definition 4), the only linear rules $p \in \text{Lin}(\mathbf{C})$ that have this special property are precisely the rules of the form

$$p_{id_M} = (M \xleftarrow{id_M} M \xrightarrow{id_M} M).$$

In particular, defining $O_M^{sq} := \rho_{\mathbf{C}}^{sq}(\delta(p_{id_M}))$, we find that the eigenvalue $\omega_{O_M^{sq}}(X)$ coincides with the cardinality of the set $M_{p_{id_M}}^{sq}(X)$ of SqPO-admissible matches,

$$\forall X \in \text{ob}(\mathbf{C}) : \quad O_M^{sq} |X\rangle = |M_{p_{id_M}}^{sq}(X)| \cdot |X\rangle.$$

This proves that the operators O_M^{sq} form a basis of diagonal operators on $End(\hat{\mathbf{C}})$ (and thus on $End(\mathcal{S}_{\mathbf{C}})$) that can arise as linear combinations of representations of rule algebra elements.

To prove the jump-closure property, note that it follows from Definition 4 that for an arbitrary *linear* rule $p \equiv (O \xleftarrow{o} K \xrightarrow{i} I) \in \text{Lin}(\mathbf{C})$, a generic object $X \in \text{obj}(\mathbf{C})$ and a monomorphism $m : I \rightarrow X$, m is according to Definition 4 both a match of the rule p as well as of the rule p_{id_I} . Evidently, the application of the rule p to X along the match m produces an object $p_m(X)$ that is in general different from the object $p_{id_{I_m}}(X)$ produced by application of the rule p_{id_I} to X along the match m . But by definition of the projection operator $\langle |$ (Definition 10),

$$\forall X \in \text{obj}(\mathbf{C})_{\cong} : \quad \langle | X \rangle := 1_{\mathbb{R}},$$

we find that

$$\langle | p_m(X) \rangle = \langle | p_{id_{I_m}}(X) \rangle = 1,$$

whence we may prove the claim of the SqPO-type jump-closure property via verifying it on arbitrary basis elements (with notations as above):

$$\langle | \rho_{\mathbf{C}}^{sq}(\delta(p)) | X \rangle = | \mathbf{M}_p^{sq}(X) | = | \mathbf{M}_{p_{id_I}}^{sq}(X) | = \langle | \rho_{\mathbf{C}}^{sq}(\delta(p_{id_I})) | X \rangle.$$

Since $X \in \text{obj}(\mathbf{C})_{\cong}$ was chosen arbitrarily, we thus have indeed that

$$\langle | \rho_{\mathbf{C}}^{sq}(\delta(p)) \rangle = \langle | \rho_{\mathbf{C}}^{sq}(\delta(p_{id_I})) \rangle.$$

This concludes the proof that our definition of continuous-time Markov chains based upon SqPO-type rewriting rules is well-posed and yields all the requisite properties.

Metric Temporal Graph Logic over Typed Attributed Graphs (Short Version)

Holger Giese, Maria Maximova, Lucas Sakizoglou, and Sven Schneider

Hasso Plattner Institut, University of Potsdam, Germany

Discrete event dynamic systems generate possibly infinite state sequences. We consider systems generating *timed graph sequences* where a non-constant amount of time in \mathbf{R}^+ elapses between two successive states and where these states are given by graphs. Metric temporal logics can then be employed to characterize the subset of all well-behaving dynamic timed systems specifying all timed graph sequences that are admissible.

For this graph-based setting, we rely on the logic of nested graph conditions [5] to specify a single graph occurring in a state sequence. We define nested graph conditions over symbolic graphs, which represent node/edge attributions by containing (a) node/edge attributes and variables as elements, (b) mappings for each of these attributes to a node/edge and to a variable, and (c) constraints over the set of variables contained in the graph. The following example shows that the usage of attribute constraints is necessary for specifying symbolic graphs when attribute values may range over an infinite set of values.

Example 1 (Nested Graph Condition over Symbolic Graphs). Let ϕ be the nested graph condition that states the existence of a node of type $:A$ with an attribute *value* = x and an attribute constraint $x \geq 3$. Let G be the graph that contains a node of type $:A$ with an attribute *value* = y and an attribute constraint $y = 5$. The satisfaction of ϕ by G is then proven using the monomorphism m that maps the variable x to the variable y for which the implication $y = 5 \rightarrow m(x \geq 3)$ is a tautology as can be decided by SMT solvers when the attribute constraints range over booleans, integers, reals, and strings with their usual operations.

Nested graph conditions can describe invariants [3]. Metric temporal logics such as MTL [6] support further operators such as the *until* operator to describe causal dependencies of occurrences of graphs also incorporating an interval over \mathbf{R}_0^+ to describe *when* graphs must occur in a timed graph sequence. However, properties such as “every task that is started is completed within 10 timeunits” can not be expressed in MTL when there is no upper bound on the number of tasks. In this property, tasks have individual deadlines and task completion events must refer to the corresponding task creation events to prevent confusion of these deadlines. Consequently, an unbounded number of tasks can then not be expressed using a finite MTL formula.

In our paper [4], we extend the logic of nested graph conditions from above, which supports a suitable binding mechanism via quantification, with the metric temporal operator *until* resulting in the Metric Temporal Graph Logic (MTGL).

With this logic, we express properties on the structure and attributes of states as well as on the occurrence of states over time that are related by their inner structure, which no formal logic over graphs concisely accomplishes so far. For example, we can express properties such as “every started task is completed within 10 timeunits” using MTGL even when the number of concurrent tasks is unbounded.

We provide fully-automatic support for MTGL in the form of the tool **AUTOGRAPH** in which we have implemented the following procedure for deciding whether a timed graph sequence satisfies a given MTGL condition. Checking satisfaction directly for timed graph sequences and MTGL conditions is difficult when the MTGL condition contains nested occurrences of the *until* operator. We therefore developed an approach allowing us to reduce the MTGL satisfaction problem to the satisfaction problem for nested graph conditions that is well-supported in **AUTOGRAPH**. For this purpose, we (a) merge all information contained in a timed graph sequence π into a single *graph with history* G where creation and deletion times of graph elements are represented by additional attributes and (b) convert the MTGL condition ϕ into a nested graph condition ϕ' replacing recursively MTGL operators by nested graph conditions that encode the semantics of these operators. This conversion procedure uses attribute constraints to express the metric aspects related to intervals used in the *until* operator as well as references to the creation and deletion time attributes. As a consequence, we then equivalently check using **AUTOGRAPH** whether the obtained graph G satisfies the nested graph condition ϕ' instead of checking whether the timed graph sequence π satisfies the MTGL condition ϕ .

The proposed MTGL logic can be used (a) to specify timed graph sequences as generated by timed graph transformation systems [2] and (b) in the field of runtime monitoring where violations of temporal properties are to be detected in timed sequences of states. Another logic used for runtime monitoring is MFOTL [1], which assumes that states are given by sets of relations. Compared to MFOTL, MTGL has distinct benefits such as its continuous semantics that also reveals missing events and the direct support of binding of graph elements as a natural extension of nested graph conditions.

References

1. D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. (*JACM*), 62(2):15, 2015.
2. B. Becker and H. Giese. On safe service-oriented real-time coordination for autonomous vehicles. *ISORC 2008*, pages 203–210.
3. J. Dyck and H. Giese. K-inductive invariant checking for graph transformation systems. *ICGT, LNCS*, 10373:142–158.
4. H. Giese, M. Maximova, L. Sakizoglou, and S. Schneider. Metric temporal graph logic over typed attributed graphs. *FASE 2019, LNCS*, 11424:282–298.
5. A. Habel and K. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *MSCS*, 19(2):245–296, 2009.
6. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

Automated Reasoning for Attributed Graph Properties (Short Version)

Sven Schneider¹, Leen Lambers¹, and Fernando Orejas²

Hasso Plattner Institut, University of Potsdam, Germany¹
Dpto de L.S.I., Universitat Politècnica de Catalunya, Barcelona, Spain²

The logic of nested graph conditions (called conditions) [4] is applicable in various application scenarios in which its conditions are used to specify sets of graphs.

We define conditions over typed symbolic graphs, which represent node/edge attributions by containing (a) node/edge attributes and variables as elements, (b) mappings for each of these attributes to a node/edge and to a variable, and (c) constraints over the set of variables contained in the graph. The following example shows that the usage of attribute constraints is necessary for specifying symbolic graphs when attribute values may range over an infinite set of values.

Example 1 (Condition over Symbolic Graphs). If the condition ϕ states that an attribute $value = x$ must satisfy $x \geq 3$, then ϕ is satisfied by a graph G where an attribute $value = y$ is restricted to $y = 5$. This satisfaction is proven using the monomorphism m that maps x to y for which $y = 5 \rightarrow m(x \geq 3)$ is a tautology as can be decided by SMT solvers. The satisfaction problem is decidable for conditions when the satisfiability problem is decidable for the underlying data algebra, which is the case when the attribute constraints range over booleans, integers, reals, and strings with their usual operations.

The satisfiability problem is semi-decidable but undecidable for conditions as they are equally expressive to first-order logic on graphs as introduced in [2]. The procedure SEEKSAT [6] terminates and gives the correct answer true, for every satisfiable condition ϕ . The sound resolution-based procedure PROCON [6] targeted tautological conditions and was adapted in [5] into a tableau-based procedure that also terminates when the condition ϕ is not satisfiable, which means that the latter procedure is refutationally complete.

The model generation problem requires the generation of satisfying instances. The procedure SEEKSAT from above also returns such a graph upon termination. In translation-based approaches, conditions (for example from OCL) are translated to SAT or SMT solvers such as KODKOD and ALLOY [3,1], which are then used for model generation. In approaches based on model rewriting, models are generated by modification steps from initially given representations as in [11].

In our paper [9], we extend [5,8] (by adding support for attributes and disambiguation as explained below) to an automatic reasoning procedure \mathcal{A} for conditions implemented in the tool AUTOGRAPH. Provided termination of the procedure \mathcal{A} for a given condition ϕ , we obtain an equivalent rewriting in the form of a finite disjunction of a set \mathcal{S} of conditions $\psi_i = \exists(G_i, \psi'_i)$ where every G_i is a minimal graph satisfying the provided condition ϕ . We call the conditions ψ_i *symbolic* models as each ψ'_i describes how the symbolic graph G_i can be extended

to further satisfying graphs. Moreover, \mathcal{A} has the following properties supporting an extended form of model generation.

- \mathcal{S} jointly covers all graphs G satisfying the condition ϕ (*completeness of \mathcal{S}*),
- \mathcal{S} does not cover any graph G violating the condition ϕ (*soundness of \mathcal{S}*),
- \mathcal{S} has no strict complete subset (*compactness of \mathcal{S}*),
- \mathcal{S} allows for each of its symbolic models the immediate extraction of a finite minimal graph G that satisfies the condition ϕ (*minimal representable \mathcal{S}*),
- \mathcal{S} allows for an enumeration of further finite graphs G satisfying the condition ϕ (*explorable \mathcal{S}*).

While \mathcal{A} does not terminate in general, it is refutationally complete and gradually generates \mathcal{S} , which can be exploited in cases where \mathcal{A} does not terminate, does not terminate in a given duration, or where only a certain number of minimal models are to be obtained. Finally, we developed additional operations adapting \mathcal{S} such that the resulting symbolic models describe disjoint sets of graphs enforcing a notion of nonambiguity among symbolic models.

Besides the refutationally complete check for satisfiability, the rewriting employed in \mathcal{A} can be understood to result in a more explicit representation that may ease subsequent operations due to its restricted syntactical form and that may be more comprehensible supporting the validation of the original condition ϕ . Moreover, the procedure \mathcal{A} has been applied recently in a graph repair approach in [10] where repairs are generated from the minimal models generated here. The generation of complete sets of minimal models is a distinguishing feature when comparing to the previously mentioned related approaches.

References

1. K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski. Clafer: unifying class and feature modeling. *Software and System Modeling*, 15(3):811–845, 2016.
2. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [7], pp. 313–400.
3. M. Gogolla and F. Hilken. Model validation and verification options in a contemporary UML and OCL analysis tool. In *Modellierung 2016, LNI 254*:205–220.
4. A. Habel and K. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *MSCS*, 19(2):245–296, 2009.
5. L. Lambers and F. Orejas. Tableau-based reasoning for graph properties. ICGT 2014, *LNCS 8571*:17–32.
6. K. Pennemann. *Development of Correct Graph Transformation Systems, PhD Thesis*. Dept. Informatik, Univ. Oldenburg, 2009.
7. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
8. S. Schneider, L. Lambers, and F. Orejas. Symbolic model generation for graph properties. FASE 2017, *LNCS 10202*:226–243.
9. S. Schneider, L. Lambers, and F. Orejas. Automated reasoning for attributed graph properties. *STTT*, 20(6):705–737, 2018.
10. S. Schneider, L. Lambers, and F. Orejas. A logic-based incremental approach to graph repair. FASE 2019, *LNCS 11424*:151–167.
11. O. Semeráth, A. Vörös, and D. Varró. Iterative and incremental model generation by logic solvers. FASE 2016, *LNCS 9633*:87–103.

Analysis of Graph Transformation Systems: Native vs Translation-based Approaches

Reiko Heckel¹, Leen Lambers², and Maryam Ghaffari Saadat¹

¹ Department of Informatics, University of Leicester, UK, {rh122, mgs17}@le.ac.uk

² Hasso-Plattner-Institut Potsdam, Germany, leen.lambers@hpi.de

Logic-based methods have come a long way over recent years. Improvements in the usability and scalability of tools have led to significant advances in the automation of hard computational problems in software engineering. Automated, formal verification, design space exploration, among others, requiring scalable solutions to constraint satisfaction or optimisation problems.

The analysis of graph transformation systems involves a variety of hard computational problems, including dynamic techniques such as execution, simulation or unfolding of systems, the membership problem for graph grammars, reachability and model checking problems, as well as static techniques such as the analyses of critical pairs, the verification and enforcement of graph constraints as invariants and the verification of systems based on a calculus of weakest preconditions. Many of these problems also arise in other contexts where state- and rule-based models or programs are analysed. One might expect that solutions adopted in software engineering more widely are also applicable to graph transformation systems, despite the fact that they are not particularly designed for our domain. This suggests a *translation-based approach* where (typically logic-based) specifications are extracted and analysed in their own domain.

On the other hand, solutions to our analysis problems are often based on theoretical results that take into account the specific features of graph transformations, such as their inherent non-determinism and concurrency, the complex non-linear structure of graphs, the properties of particular approaches and formalisations, and restrictions including context freeness or monotonicity. As a consequence, the majority of existing solutions are *native* ones, providing bespoke analysis tools for graph transformation systems and grammars.

The aim of this session is two-fold, to discuss the pros and cons of native vs translation-based approaches to the analysis of graph transformation systems and, for the latter, understand some of the design choices such as selecting the right logics and tools, choosing an appropriate encoding, etc.

We start the session with an overview of a range of analysis problems and solutions to establish the state of the art of this area as well as its open problems.

Then we review and compare native vs. translation-based solutions to analysis problems from the literature and discuss reported experiments aimed at evaluating these solutions. We will consider, in particular, the literature on the use of SAT and SMT solvers, which have seen some of the most impressive recent advances in technology, for solving analysis problems of graph transformations. We conclude with short contributions from the audience and a general panel discussion.