Leen Lambers
Jens Weber (Eds.)

LNCS 10887

Graph Transformation

11th International Conference, ICGT 2018
Held as Part of STAF 2018
Toulouse, France, June 25–26, 2018, Proceedings

Springer

Leen Lambers · Jens Weber (Eds.)

Graph Transformation

11th International Conference, ICGT 2018
Held as Part of STAF 2018
Toulouse, France, June 25–26, 2018
Proceedings

Springer

Editors

Leen Lambers
University of Potsdam
Potsdam
Germany

Jens Weber
University of Victoria
Victoria, BC
Canada

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-92990-3 ISBN 978-3-319-92991-0 (eBook)
https://doi.org/10.1007/978-3-319-92991-0

Library of Congress Control Number: 2018944418

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG
part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

Software Technologies: Applications and Foundations (STAF) is a federation of leading conferences on software technologies. It provides a loose umbrella organization with a Steering Committee that ensures continuity. The STAF federated event takes place annually. The participating conferences and workshops may vary from year to year, but they all focus on foundational and practical advances in software technology. The conferences address all aspects of software technology, from object-oriented design, testing, mathematical approaches to modeling and verification, transformation, model-driven engineering, aspect-oriented techniques, and tools. STAF was created in 2013 as a follow-up to the TOOLS conference series that played a key role in the deployment of object-oriented technologies. TOOLS was created in 1988 by Jean Bézivin and Bertrand Meyer and STAF 2018 can be considered as its 30th birthday.

STAF 2018 took place in Toulouse, France, during June 25–29, 2018, and hosted: five conferences, ECMFA 2018, ICGT 2018, ICMT 2018, SEFM 2018, TAP 2018, and the Transformation Tool Contest TTC 2018; eight workshops and associated events. STAF 2018 featured seven internationally renowned keynote speakers, welcomed participants from all around the world, and had the pleasure to host a talk by the founders of the TOOLS conference Jean Bézivin and Bertrand Meyer.

The STAF 2018 Organizing Committee would like to thank (a) all participants for submitting to and attending the event, (b) the Program Committees and Steering Committees of all the individual conferences and satellite events for their hard work, (c) the keynote speakers for their thoughtful, insightful, and inspiring talks, and (d) the École Nationale Supérieure d'Électrotechnique, d'Électronique, Hydraulique et des Télécommunications (ENSEEIHT), the Institut National Polytechnique de Toulouse (Toulouse INP), the Institut de Recherche en Informatique de Toulouse (IRIT), the région Occitanie, and all sponsors for their support. A special thanks goes to all the members of the Software and System Reliability Department of the IRIT laboratory and the members of the INP-Act SAIC, coping with all the foreseen and unforeseen work to prepare a memorable event.

June 2018

Marc Pantel
Jean-Michel Bruel

Preface

This volume contains the proceedings of ICGT 2018, the 11th International Conference on Graph Transformation held during June 25–26, 2018 in Toulouse, France. ICGT 2018 was affiliated with STAF (Software Technologies: Applications and Foundations), a federation of leading conferences on software technologies. ICGT 2018 took place under the auspices of the European Association of Theoretical Computer Science (EATCS), the European Association of Software Science and Technology (EASST), and the IFIP Working Group 1.3, Foundations of Systems Specification.

The aim of the ICGT series is to bring together researchers from different areas interested in all aspects of graph transformation. Graph structures are used almost everywhere when representing or modeling data and systems, not only in computer science, but also in the natural sciences and in engineering. Graph transformation and graph grammars are the fundamental modeling paradigms for describing, formalizing, and analyzing graphs that change over time when modeling, e.g., dynamic data structures, systems, or models. The conference series promotes the cross-fertilizing exchange of novel ideas, new results, and experiences in this context among researchers and students from different communities.

ICGT 2018 continued the series of conferences previously held in Barcelona (Spain) in 2002, Rome (Italy) in 2004, Natal (Brazil) in 2006, Leicester (UK) in 2008, Enschede (The Netherlands) in 2010, Bremen (Germany) in 2012, York (UK) in 2014, L'Aquila (Italy) in 2015, Vienna (Austria) in 2016, and Marburg (Germany) in 2017, following a series of six International Workshops on Graph Grammars and Their Application to Computer Science from 1978 to 1998 in Europe and in the USA.

This year, the conference solicited research papers describing new unpublished contributions in the theory and applications of graph transformation as well as tool presentation papers that demonstrate main new features and functionalities of graph-based tools. All papers were reviewed thoroughly by at least three Program Committee members and additional reviewers. We received 16 submissions, and the Program Committee selected nine research papers and two tool presentation papers for publication in these proceedings, after careful reviewing and extensive discussions. The topics of the accepted papers range over a wide spectrum, including advanced concepts and tooling for graph language definition, new graph transformation formalisms fitting various application fields, theory on conflicts and parallel independence for different graph transformation formalisms, as well as practical approaches to graph transformation and verification. In addition to these paper presentations, the conference program included an invited talk, given by Olivier Rey (GraphApps, France).

We would like to thank all who contributed to the success of ICGT 2018, the invited speaker Olivier Rey, the authors of all submitted papers, as well as the members of the Program Committee and the additional reviewers for their valuable contributions to the selection process. We are grateful to Reiko Heckel, the chair of the Steering Committee of ICGT for his valuable suggestions; to Marc Pantel and Jean-Michel Bruel,

the organization co-chairs of STAF; and to the STAF federation of conferences for hosting ICGT 2018. We would also like to thank EasyChair for providing support for the review process.

June 2018

Leen Lambers
Jens Weber

Organization

Steering Committee

Program Committee

Berthold Hoffmann	University of Bremen, Germany
Dirk Janssens	University of Antwerp, Belgium
Barbara König	University of Duisburg-Essen, Germany
Leen Lambers (Co-chair)	Hasso Plattner Institute at the University of Potsdam, Germany
Yngve Lamo	Bergen University College, Norway
Mark Minas	Bundeswehr University Munich, Germany
Mohamed Mosbah	LaBRI, University of Bordeaux, France
Fernando Orejas	Technical University of Catalonia, Spain
Francesco Parisi-Presicce	Sapienza University of Rome, Italy
Detlef Plump	University of York, UK
Arend Rensink	University of Twente, The Netherlands
Leila Ribeiro	University Federal do Rio Grande do Sul, Brazil
Andy Schürr	Technical University of Darmstadt, Germany
Gabriele Taentzer	Philipps University of Marburg, Germany
Jens Weber (Co-chair)	University of Victoria, Canada
Bernhard Westfechtel	University of Bayreuth, Germany
Albert Zündorf	University of Kassel, Germany

Additional Reviewers

Atkinson, Timothy
Azzi, Guilherme
Dyck, Johannes
Farkas, Rebeka
Kluge, Roland
Nolte, Dennis
Peuser, Christoph
Sakizloglou, Lucas
Semeráth, Oszkár

Introduction to Graph-Oriented Programming (Keynote)

Olivier Rey

GraphApps, France
rey.olivier@gmail.com
orey.github.io/papers

Abstract. Graph-oriented programming is a new programming para-digm that defines a graph-oriented way to build enterprise software, using directed attributed graph databases as backend. Graph-oriented programming is inspired by object-oriented programming, functional programming, design by contract, rule-based programming and the semantic web. It integrates all those programming paradigms consistently. Graph-oriented programming enables software developers to build enterprise software that does not generate technical debt. Its use is particularly adapted to enterprise software managing very complex data structures, evolving regulations and/or high numbers of business rules.

Couplings in Enterprise Software

The way the software industry currently builds enterprise software generates a lot of "structural and temporal couplings". Structural coupling occurs when software and, in particular, data structures, are implemented such that artificial dependencies are generated. A dependency is artificial if it occurs in the implementation but not in the underlying semantic concepts. Temporal couplings are artificial dependencies generated by holding several versions of business rules in the same program, those rules being applicable to data that are stored in the last version of the data structures.

Those couplings are at the very core of what is commonly called "technical debt". This debt generates over-costs each time a software evolves. Generally, the requirements change, the software is partially redesigned to accommodate the modification, the data structures evolve, the existing data must be migrated, and all programs must be non-regressed. In order to implement a small modification in an enterprise software, a change in regulation for instance, overcoming the technical debt may represent up to 90–95% of the total workload [5, 6].

The software industry has, for a long time, identified the costs associated to technical debts, and in particular those costs seem to grow exponentially with time [5]. That means that the productivity of any maintenance team of fixed size will constantly decrease throughout the evolution process. In order to address this core issue of enterprise software, a lot of engineering-oriented work-arounds can be found: design patterns that are supposed to enhance software extensibility [1], software architecture practices that define modules and layers inside an enterprise software [2, 4], or best practices for software refactoring to reduce the costs of the refactoring phase itself [3]. However, every software vendor knows that the core problem of the technical debt has not been solved.

Graph-Oriented Programming

Graph-oriented programming is meant as an alternative programming paradigm not collecting technical debts. This paradigm is based on three concepts: (1) Using directed attributed graph databases to store the business entities without storing their relationships in the entities themselves, i.e. there are no foreign keys; (2) Designing programs so that the knowledge about relationships between entities (business nodes) is captured in functional code located "outside" of the nodes, encapsulated in graph transformation rules; (3) Using best practices in graph transformation design to guarantee a minimal or even no generation of technical debt. This programming paradigm can be applied using an object-oriented or functional programming language.

The expected advantages of using graph-oriented programming are multiple: reusability of software is increased due to less software dependencies; multiple views of the same data can be implemented in the same application; multiple versions of data structures and business rules can cohabit, meaning that the software and the data can be timelined; software maintenance can be done by adding new software rather than by modifying existing software.

At last, graph-oriented programming enables to build a different kind of enterprise software that proposes, through the use of a graph-oriented navigation, a new user experience, closer to our mental way of representing things.

The Approach Taken at GraphApps

At GraphApps, we developed a graph-oriented designer in Eclipse whose purpose is to model node and relationship types, as they occur in business applications, and to group them in semantic domains. Code generators, coupled to the designer, generate parameterized web pages proposing a default view of defined types of business entities. For each semantic domain, an independent `jar` file is generated. In addition, we developed a graph-oriented web framework, which loads the `jar` files and enables us to integrate them in the graphical web framework. All domains can be integrated without introducing any new code dependency. Each domain may include custom code, in order to implement graph transformations, web page modifications, or new pages. Moreover, the framework proposes reusable components that offer generic reusable mechanisms such as business node classification (every business node can be referenced in a tree of shared folders), business node timelines, navigation history, personalized links between business nodes, or alternate navigation.

Those tools support a quick prototyping of large and complex applications, the implementation of time-based business rules, and the cooperative work of several teams collaborating to the same core model. The way the code is organized enables us to modify the behavior of the core system, without having to modify existing code, migrating data, or performing non-regressing testing. We have used this set of tools for many business prototypes and we are using it currently to build a complete innovative aerospace maintenance information system (composed by many semantic domains) from scratch.

Conclusion

The paradigm of graph-oriented programming enables us to build a new generation of enterprise software that will be much easier to maintain and that can address the high complexity of business entity structures and their life cycles, as well as time-sensitive business rules. This paradigm may be used to rewrite a huge number of enterprise software in the coming decades in order to decrease drastically the maintenance costs, to enhance the capability of personalization of the software and to create new user experiences by proposing more intuitive ways to navigate within the software.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object Oriented Software. Addison-Wesley (1994)
2. Buschmann, F.: *POSA* Volume 1 - A System of Patterns. Wiley (1996)
3. Fowler, M.: Refactoring. Addison-Wesley (1999)
4. Alur, D.: Core J2EE Patterns, 2nd edn. Prentice-Hall (2003)
5. Nugroho, A., Joost, V., Tobias, K.: An empirical model of technical debt and interest. In: Proceedings of the 2nd Workshop on Managing Technical Debt. ACM (2011)
6. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. Syst. Softw. **101**, 193–220 (2015)

Contents

Graph Languages

Splicing/Fusion Grammars and Their Relation to Hypergraph Grammars

Hans-Jörg Kreowski(✉), Sabine Kuske, and Aaron Lye

Department of Computer Science and Mathematics,
University of Bremen, P.O. Box 33 04 40, 28334 Bremen, Germany
{kreo,kuske,lye}@informatik.uni-bremen.de

Abstract. In this paper, we introduce splicing/fusion grammars as a device for generating hypergraph languages. They generalize the formerly introduced notion of fusion grammars by adding splicing rules that split node sets into two node sets and equip them with complementary hyperedges. As a result a derivation step in such a grammar is either a fusion of complementary hyperedges, a multiplication of a connected component or a splicing. We prove two main results demonstrating the generative power of splicing/fusion grammars. First, Chomsky grammars are transformed into splicing/fusion grammars where the transformation mimics a corresponding transformation of Chomsky grammars into splicing systems as studied in the context of DNA computing. Second, hypergraph grammars are transformed into splicing/fusion grammars.

1 Introduction

In various scientific fields like DNA computing, chemistry, tiling, fractal geometry, visual modeling and others, one encounters various fusion processes. In [1], this principle is captured in the formal framework of fusion grammars which are generative devices on hypergraphs.

A fusion grammar provides fusion rules the application of which fuse the attachment nodes of complementary hyperedges. In addition, within a derivation, the multiplication of connected components of processed hypergraphs is allowed. Fusion grammars are more powerful than hyperedge replacement grammars.

In this paper, we generalize fusion grammars to splicing/fusion grammars by adding splicing rules to fusion grammars. A splicing rule is reverse to a fusion rule and allows to cut a connected hypergraph in two parts. The new type of grammar is very powerful as two main results prove: On one hand, Chomsky grammars, on the other hand, hypergraph grammars (satisfying a certain connnectivity condition) are transformed into splicing/fusion grammars. This shows, in particular, that the generative power of splicing/fusion grammars is much greater than the generative power of fusion grammars because the membership problem of fusion grammars is decidable, whereas it is undecidable for splicing/fusion grammars simulating Chomsky grammars of type 0.

L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 3–19, 2018.
https://doi.org/10.1007/978-3-319-92991-0_1

The paper is organized as follows. Section 2 provides preliminaries for graph-transformation. In Sect. 3, the notion of fusion grammars is recalled. In Sect. 4 splicing/fusion grammars are introduced. Section 5 contains the transformation of Chomsky grammars into splicing/fusion grammars, and Sect. 6 the transformation of hypergraph grammars into splicing/fusion grammars. Section 7 concludes the paper.

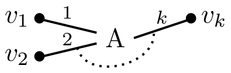
2 Preliminaries

In this section, basic notions and notations of hypergraphs and hypergraph transformation are recalled (see, e.g., [2]).

2.1 Hypergraphs, Morphisms, Basic Constructions

A *hypergraph* over Σ is a system $H = (V, E, att, lab)$ where V is a finite set of *nodes*, E is a finite set of *hyperedges*, $att\colon E \to V^*$ is a function, called *attachment* (assigning a string of attachment nodes to each edge), and $lab\colon E \to \Sigma$ is a function, called *labeling*.

The length of the attachment $att(e)$ for $e \in E$ is called *type* of e, and e is called A-hyperedge if A is its label. The components of $H = (V, E, att, lab)$ may also be denoted by V_H, E_H, att_H, and lab_H respectively. The class of all hypergraphs over Σ is denoted by $\mathcal{H}_\Sigma$.

In drawings, a A-hyperedge e with attachment $att(e) = v_1 \cdots v_k$ is depicted by 

. Moreover, a hyperedge of type 2 is depicted as an edge by $\bullet\xrightarrow{A}\bullet$ instead of $\bullet\overset{1}{—}A\overset{2}{—}\bullet$ and a hyperedge of type 1 by A instead of $\bullet\overset{1}{—}A$. If there are two edges with the same label, but in opposite directions, we may draw them as an undirected edge. We assume the existence of a special label $* \in \Sigma$ that is omitted in drawings.

Given $H, H' \in \mathcal{H}_\Sigma$, H is a *subhypergraph* of H' if $V_H \subseteq V_{H'}$, $E_H \subseteq E_{H'}$, $att_H(e) = att_{H'}(e)$, and $lab_H(e) = lab_{H'}(e)$ for all $e \in E_H$. This is denoted by $H \subseteq H'$.

Given $H, H' \in \mathcal{H}_\Sigma$, a (*hypergraph*) *morphism* $g\colon H \to H'$ consists of two mappings $g_V\colon V_H \to V_{H'}$ and $g_E\colon E_H \to E_{H'}$ such that $att_{H'}(g_E(e)) = g_V^*(att_H(e))$ and $lab_{H'}(g_E(e)) = lab_H(e)$ for all $e \in E_H$, where $g_V^*\colon V_H^* \to V_{H'}^*$ 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 $v_1 \cdots v_n \in V_H^*$. H and H' are *isomorphic*, denoted by $H \cong H'$, if there is an isomorphism $g\colon H \to H'$, i.e., a morphism with bijective mappings. Clearly, $H \subseteq H'$ implies that the two inclusions $V_H \subseteq V_{H'}$ and $E_H \subseteq E_{H'}$ define a morphism from $H \to H'$. Given a morphism $g\colon H \to H'$, the image of H in H' under g defines the subgraph $g(H) \subseteq H'$.

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 $H = H' - (V, E) = (V_{H'} - V, E_{H'} - E, att_H, lab_H)$ with $att_H(e) = att_{H'}(e)$ and $lab_H(e) = lab_{H'}(e)$ for all $e \in E_{H'} - E$ defines a subgraph

$H \subseteq H'$ if $att_{H'}(e) \in (V_{H'} - V)^*$ for all $e \in E_{H'} - E$, i.e., no remaining hyperedge is attached to a removed node. This condition is called *dangling condition.*

We use frequently the discrete hypergraph $[k]$ with the nodes $1, \ldots, k$ for some $k \in \mathbb{N}$ and the handle $A^\bullet$ consisting of the nodes $1, \ldots, k$ for some $k \in \mathbb{N}$ and a single hyperedge labeled by $A \in \Sigma$ of type k attached to $1 \ldots k$. Hence, in particular, $[k] \subseteq A^\bullet$. Given $H, H' \in \mathcal{H}_\Sigma$, the *disjoint union* of H and H' is denoted by $H + H'$. A special case is the disjoint union of H with itself k times, denoted by $k \cdot H$. The disjoint union is unique up to isomorphism. It is easy to see that the disjoint union is commutative and associative. Moreover, there are injections (injective morphisms) $in_H \colon H \to H + H'$ and $in_{H'} \colon H' \to H + H'$ such that $in_H(H) \cup in_{H'}(H') = H + H'$ and $in_H(H) \cap in_{H'}(H') = \emptyset$. Each two morphisms $g_H \colon H \to Y$ and $g_{H'} \colon H' \to Y$ define a unique morphism $\langle g_H, g_{H'} \rangle \colon H + H' \to Y$ with $\langle g_H, g_{H'} \rangle \circ in_H = g_H$ and $\langle g_H, g_{H'} \rangle \circ in_{H'} = g_{H'}$. In particular, one gets $g = \langle g \circ in_H, g \circ in_{H'} \rangle$ for all morphisms $g \colon H + H' \to Y$ and $g + g' = \langle in_Y \circ g, in_{Y'} \circ g' \rangle \colon H + H' \to Y + Y'$ for morphisms $g \colon H \to Y$ and $g' \colon H' \to Y'$. The disjoint union is the coproduct in the category of hypergraphs.

The fusion of nodes is defined as a quotient by means of an equivalence relation $\equiv$ on the set of nodes V_H of H as follows: $H/{\equiv} = (V_H/{\equiv}, E_H, att_{H/\equiv}, lab_H)$ with $att_{H/\equiv}(e) = [v_1] \cdots [v_k]$ for $e \in E_H$, $att_H(e) = v_1 \cdots v_k$ where $[v]$ denotes the equivalence class of $v \in V_H$ and $V_H/{\equiv}$ is the set of equivalence classes. It is easy to see that $f \colon H \to H/{\equiv}$ given by $f_V(v) = [v]$ for all $v \in V_H$ and $f_E(e) = e$ for all $e \in E_H$ defines a *quotient morphism.*

Let $H \in \mathcal{H}_\Sigma$. Then a sequence of triples $(i_1, e_1, o_1) \ldots (i_n, e_n, o_n) \in (\mathbb{N} \times E_H \times \mathbb{N})^*$ is a *path* from $v \in V_H$ to $v' \in V_H$ if $v = att_H(e_1)_{i_1}, v' = att_H(e_n)_{o_n}$ and $att_H(e_j)_{o_j} = att_H(e_{j+1})_{i_{j+1}}$ for $j = 1, \ldots, n-1$ where, for each $e \in E_H$, $att_H(e)_i = v_i$ for $att_H(e) = v_1 \cdots v_k$ and $i = 1, \ldots, k$. H is *connected* if each two nodes are connected by a path. A subgraph C of H is a *connected component* of H if $C \subseteq C' \subseteq H$ for a connected C' implies $C = C'$. The set of connected components of H is denoted by $\mathcal{C}(H)$.

We use the *multiplication* of H defined by means of $\mathcal{C}(H)$ as follows. Let $m \colon \mathcal{C}(H) \to \mathbb{N}_{>0}$ be a mapping, called *multiplicity*, then $m \cdot H = \sum\limits_{C \in \mathcal{C}(H)} m(C) \cdot C$.

2.2 Pushout and Pushout Complement

Let C be a category and $a \colon K \to L, g \colon L \to H, d \colon K \to I$, and $m \colon I \to H$ be morphisms with $g \circ a = m \circ d$. Then H together with g and m is a pushout of a and d if, for each pair of morphisms $g' \colon L \to X$ and $m' \colon I \to X$ with $g' \circ a = m' \circ d$, there is a unique morphism $x \colon H \to X$ with $x \circ g = g'$ and $x \circ m = m'$.

Given this pushout, the object I with the morphisms $K \xrightarrow{d} I \xrightarrow{m} H$ is called *pushout complement* of $K \xrightarrow{a} L \xrightarrow{g} H$.

In the category of hypergraphs, pushouts exist and can be constructed (up to isomorphism) as the quotient $(L + I)/(a = d)$, where $a = d$ is the equivalence relation induced by the relation $\{(a(x), d(x)) \mid x \in K\}$ (for nodes and hyperedges separately).

A pushout complement of hypergraph morphisms $K \xrightarrow{a} L \xrightarrow{g} H$ exists if and only if g satisfies the gluing condition, i.e., $H - (g(L) - g(a(K)))$ satisfies the dangling condition and g satisfies the identification condition which requires that for all $x, y \in L$ with $g(x) = g(y)$, $x = y$ or there are $\overline{x}, \overline{y} \in K$ with $a(\overline{x}) = x$ and $a(\overline{y}) = y$ (for nodes and hyperedges separately). If g satisfies the gluing condition, a particular pushout complement can be constructed by $I = H - (g(L) - g(a(K))$ with the inclusion of I into H as m and the restriction of $g \circ a$ to K and I as d. If a is injective, then this pushout complement is unique up to isomorphism.

2.3 Rule Application and Hypergraph Grammars

A (hypergraph grammar) *rule* $r = (L \xleftarrow{a} K \xrightarrow{b} R)$ consists of $L, K, R \in \mathcal{H}_\Sigma$ and two morphisms a, b where a is injective.

The application of a rule r to a hypergraph $H \in \mathcal{H}_\Sigma$ is denoted by $H \underset{r}{\Longrightarrow} H'$ and called a *direct derivation*. A direct derivation consists of a double pushout (see, e.g., [3])

$$\begin{tikzcd}
L \arrow[d, "g"'] & K \arrow[l, "a"'] \arrow[r, "b"] \arrow[d, "d"] & R \arrow[d, "h"] \\
H & I \arrow[l, "m"] \arrow[r, "m'"] & H'
\end{tikzcd}$$

The two pushouts can be constructed in the following way. To apply r to H, one needs a *matching* morphism $g \colon L \to H$ subject to the *gluing condition.* Then one removes $g(L) - g(K)$ from H yielding the intermediate hypergraph I with $I \subseteq H$ (i.e., choosing m as inclusion) and a morphism $d \colon K \to I$ restricting g to K and I. Finally, I and R are merged along the morphisms d and b meaning that the resulting hypergraph is $H' = (I + R)/(d = b)$ being the quotient of $I + R$ through the equivalence relation $d = b$ induced by the relation $\{(d_V(v), b_V(v)) \mid v \in V_K\}$. The inclusions in_I and in_R into $I + R$ composed with the quotient morphism define morphisms $m' \colon I \to H'$ and $h \colon R \to H'$ such that $m' \circ d = h \circ b$ according to the definition of the equivalence $d = b$.

A *derivation* from H to H' is the sequential composition of direct derivations, i.e., $der = (H = H_0 \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} H_n = H')$ for some $n \in \mathbb{N}$. If $r_1, \ldots, r_n \in P$ (for some set P of rules), then der can be denoted as $H \overset{n}{\underset{P}{\Longrightarrow}} H'$ or as $H \overset{*}{\underset{P}{\Longrightarrow}} H'$.

Let $N \subseteq \Sigma$ be a set of nonterminals where each $A \in N$ has a *type* $k(A) \in \mathbb{N}$. A *hypergraph grammar* is a system $HGG = (N, T, P, S)$ where $N \subseteq \Sigma$ is a set of typed *nonterminals*, $T \subseteq \Sigma$ is a set of *terminals* with $T \cap N = \emptyset$, P is a set of rules and $S \in N$ is the *start symbol.*

The language $L(HG)$ is defined as $\{X \in \mathcal{H}_T \mid S^\bullet \overset{*}{\underset{P}{\Longrightarrow}} X\}$.

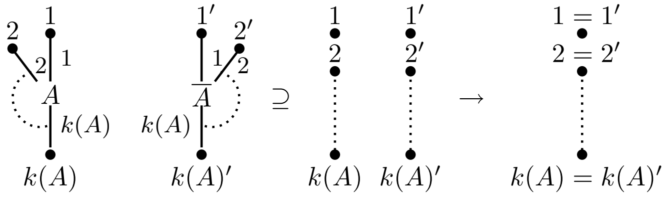
Without loss of generality, one can assume that, for each rule $L \xleftarrow{a} K \xrightarrow{b} R$ the gluing graph K is discrete such that it can be chosen as $[k]$ for some $k \in \mathbb{N}$ and a is an inclusion.

3 Fusion Grammars

In this section, we recall the notion of fusion grammars as defined in [1].

A fusion grammar provides a set of fusion rules. The application of a fusion rule merges the corresponding attachment nodes of complementary hyperedges. Complementarity is defined on a typed set F of fusion labels. Given a hypergraph, the set of all possible fusions is finite as fusion rules never create anything and each fusion decreases the number of hyperedges by two. To overcome this limitation, we allow arbitrary multiplications of connected components within derivations in addition to fusion. The start hypergraph and the derived hypergraphs consist usually of several connected components which are the building blocks of fusion and the objects of interest. Some of them may are purely auxiliary, some may never terminate. Therefore, the generated language does not contain derived hypergraphs, but only their marked and terminal connected components where markers are used as an additional feature to distinguish between wanted and unwanted connected components.

Definition 1. Let $F \subseteq \Sigma$, called *fusion alphabet*, and $k\colon F \to \mathbb{N}$ be a type function. Let $\overline{A} \notin F$ be the complementary fusion label for each $A \in F$ such that $\overline{A} \neq \overline{B}$ for all $A \neq B$. The typing is extended to complementary labels by $k(\overline{A}) = k(A)$ for all $A \in F$. Then $A \in F$ specifies the following *fusion rule*, denoted by $fr(A)$



i.e., formally, $fr(A) = (A^\bullet + \overline{A}^\bullet \xleftarrow{in+\overline{in}} [k(A)] + [k(A)] \xrightarrow{\langle 1_{[k(A)]}, 1_{[k(A)]}\rangle} [k(A)])$ where $in, \overline{in}$ are the inclusions of $[k(A)]$ into $A^\bullet$ and $\overline{A}^\bullet$, respectively, and $1_{[k(A)]}$ denotes the identity on $[k(A)]$.

The numbers at the nodes identify them so that the left-hand side inclusion and the right-hand side morphism are made visible. The morphism maps each attachment node and its primed counterpart to the same right-hand side node. In the formal version of the rule, the priming is not needed because the disjoint union takes care that the components are properly separated from each other.

Definition 2. 1. A *fusion grammar* is a system $FG = (Z, F, M, T)$ where Z is a *start hypergraph*, $F \subseteq \Sigma$ is a fusion alphabet, $M \subseteq \Sigma$ with $M \cap (F \cup \overline{F}) = \emptyset$ is a set of *markers*, and $T \subseteq \Sigma$ with $T \cap (F \cup \overline{F}) = \emptyset = T \cap M$ is a set of *terminal labels*.

2. A *derivation step* $H \Longrightarrow H'$ for some $H, H' \in \mathcal{H}_\Sigma$ is either a rule application $H \underset{r}{\Longrightarrow} H'$ for some rule in $fr(F) = \{fr(A) \mid A \in F\}$ or a multiplication $H \underset{m}{\Longrightarrow} m \cdot H$ for some multiplicity m.

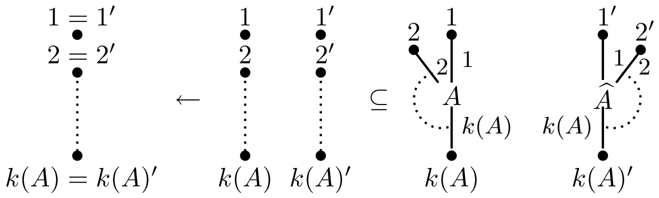
3. A *derivation* $H \overset{n}{\Longrightarrow} H'$ of length n is a sequence $H_0 \Longrightarrow H_1 \Longrightarrow \ldots \Longrightarrow H_n$ with $H = H_0$ and $H' = H_n$ including the case $n = 0$ with $H_0 = H = H' = H_n$. One may write $H \overset{*}{\Longrightarrow} H'$.
4. $L(FG) = \{rem_M(Y) \mid Z \overset{*}{\Longrightarrow} H, Y \in \mathcal{C}(H) \cap (\mathcal{H}_{T \cup M} - \mathcal{H}_T)\}$ is the *generated language* of FG where $rem_M(Y)$ is the hypergraph obtained when removing all hyperedges with labels in M from Y.

Remark 1. If all components of the start hypergraph have hyperedges with markers, then all connected components of derived hypergraphs have hyperedges with markers so that markers have no selective effect. This defines the special case of fusion grammars without markers. Their generated language is defined by $L(FG) = \{Y \mid Z \overset{*}{\Longrightarrow} H, Y \in \mathcal{C}(H) \cap \mathcal{H}_T\}$.

4 Splicing/Fusion Grammars

In this section, we introduce the splicing of hypergraphs by means of splicing rules that are reverse to fusion rules. This is a prerequisite for the definition of splicing/fusion grammars in the next section. A splicing rule splits a number of nodes in two parts each and attaches a hyperedge to the nodes of the first part and a complementary hyperedge to the nodes of the second part. As a splicing rule can be applied to every collection of k nodes, splicing is very variable and unrestricted, but too much sometimes. Therefore, we allow its regulation by a context condition.

Definition 3. Let $A \in F$ with $type(A) = k$ and $\widehat{A} \in \Sigma$, called *complementary splicing label* of A with $type(A) = type(\widehat{A})$. Then the *splicing rule* $sr(A)$ of A has the form



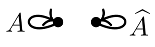
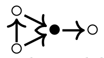
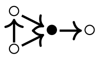
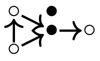
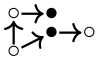
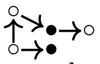
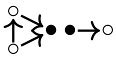
i.e., formally, $K \xleftarrow{\langle 1_K, 1_K\rangle} K + K \xrightarrow{in+\widehat{in}} A^\bullet + \widehat{A}^\bullet$, where $K = [k(A)]$, in is the inclusion of K into $A^\bullet$ and $\widehat{in}$ is the respective inclusion of K into $\widehat{A}^\bullet$.

Note that a splicing rule is not a hypergraph grammar rule because in this case gluing nodes are not embedded injective in the left-hand side.

An application of a splicing rule $sr(A)$ to a hypergraph H, denoted by $H \underset{sr(A)}{\Longrightarrow} H'$, is defined by a double pushout

$$\begin{tikzcd}
K \arrow[d, "f"'] & K + K \arrow[l, "\langle 1_K, 1_K\rangle"'] \arrow[r, "in + \widehat{in}"] \arrow[d, "c"] & A^\bullet + \widehat{A}^\bullet \arrow[d, "f'"] \\
H & C \arrow[l, "e"] \arrow[r, "e'"'] & H'
\end{tikzcd}$$

Obviously, it does not make much sense to require an injective morphism $e\colon C \to H$ because e would be an isomorphism and nothing would be spliced. Therefore, we allow arbitrary pushout complements. But there are many choices of the set of nodes of C as well as of the attachment of hypergraphs. By construction, e must be a bijection between $C - c(K+K)$ and $H - f(K)$. The mapping of $K+K$ to $c(K+K)$ can be chosen such that $e(c(K+K)) = f(K)$. Further, if a hyperedge of H is attached to $f(k)$ for some k, then its counterpart in C is attached to $c(in(k))$ or $c(\widehat{in}(k))$ which may be different.

Example 1. Consider the splicing rule sr(A) = ($\bullet \leftarrow \bullet \quad \bullet \rightarrow$ ). Applying $sr(A)$ to the graph  matching the node of its left-hand side with the middle node of the graph yields the following five pushout complements (up to isomorphism):     . The last four cases show proper node splittings that differ from each other by the distribution of the three edges attached to the middle node. Particularly interesting is the last case as the graph is separated into two proper subgraphs.

To cut down the number of choices, one may restrict the application of splicing rules by some condition. There are various choices like, for example, the frequently used positive and negative context conditions. We apply another context condition which does not only restrict the matching, but also the pushout complement.

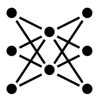
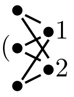
Definition 4. A splicing rule *with fixed disjoint context* $srfdc(A,a)$ consists of a splicing rule $sr(A)$ and a morphism $a\colon K \to X$ for some context X. It is denoted by $srfdc(A,a) = (X \xleftarrow{a} K \xleftarrow{\langle 1_K, 1_K\rangle} K + K \xrightarrow{in+\widehat{in}} A^\bullet + \widehat{A}^\bullet)$.

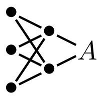
The rule is applicable to a hypergraph H if the pushout complement can be chosen in the following way

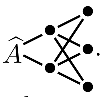
$$\begin{tikzcd}
K \arrow[d, "f"'] & K + K \arrow[l, "\langle 1_K, 1_K\rangle"'] \arrow[d, "y+a"] \\
H & Y + X \arrow[l, "\langle m, b\rangle"]
\end{tikzcd}$$

where $m\colon Y \to H$ is injective and the complement consists of two disjoint parts one of which is the pregiven context X.

Such a pushout complement does not always exist as there may be no morphism $b\colon X \to H$ or H may not be separable in the required way. But if the pushout complement exists, then $Y, y\colon K \to Y$ and $m\colon Y \to H$ are unique up to isomorphism. Without loss of generality, Y can be chosen as the subgraph $H - (b(X) - b(a(K)))$ meaning that all hyperedges of $b(X)$ are removed as well as all nodes that are not gluing nodes.

Example 2. Consider the graph $fence_0 =$  and the splicing rule with fixed disjoint context $cut = ($ 

$\supseteq [2] \leftarrow [2] + [2] \subseteq A^\bullet + \widehat{A}^\bullet)$. cut can be applied to $fence_0$ matching the nodes in the middle yielding $left =$ 

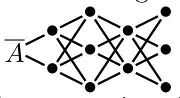
and $right =$ 

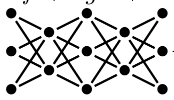
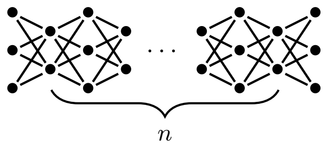
.

Note that we omit the numbering of the tentacles because it is irrelevant in this example.

If one equips a fusion grammar with an additional splicing mechanism, then one gets a splicing/fusion grammar.

Definition 5. 1. A *splicing/fusion grammar* is a system $SFG = (Z, F, M, T, SR)$ where (Z, F, M, T) is a fusion grammar and SR is a set of splicing rules where some may have fixed disjoint context. The complementary splicing labels of $A \in F$ in splicing rules, denoted by $\widehat{A}$, may be different from the complementary fusion labels $\overline{A}$.
2. A derivation step in SFG may be a multiplication, an application of a fusion rule or an application of a splicing rule.
3. The generated language $L(SFG)$ is defined as the fusion grammar by using the more general derivations.

Example 3. To illustrate how splicing/fusion grammars work, we continue the Example 2. Consider the grammar $FENCE = (fence_0 + comp, \{A\}, \emptyset, \{*\}, \{cut\})$ with $comp =$ 

. No fusion is possible in the start hypergraph, but one can produce n copies of $comp$ and cut can be applied to $fence_0$ yielding $left + right + n \cdot comp$. Now the fusion of $left$ and $comp$ is possible yielding $fence_1 =$  $+ right + (n-1) \cdot comp$. This splicing of a fence followed by a fusion can be iterated generating $fence_n =$ 

$+ n \cdot right$.

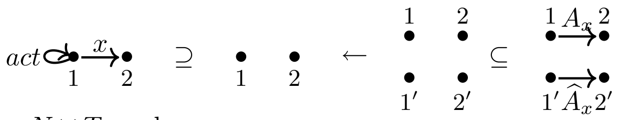
5 Transformation of Chomsky Grammars into Splicing/Fusion Grammars

In this section, we translate Chomsky grammars into splicing/fusion grammars where strings are modeled by cycles. The transformation mimics the translation of Chomsky grammars into splicing systems as it can be found in Păun, Rozenberg and Salomaa [4] in various variants.

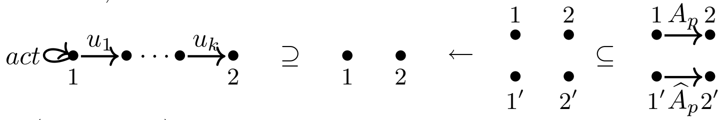
Let $CG = (N, T, P, S)$ be a Chomsky grammar with $N \cap T = \emptyset, S \in N$ and $P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$. The transformation yields the splicing/fusion grammar

$$SFG(CG) = (Z(CG), \{A_y \mid y \in N \cup T \cup P\} \cup \{act\}, \emptyset, T \cup \{be\}, SR(CG)).$$

The fusion symbol *act* is of type 1, all the other fusion symbols are of type 2. Moreover, we assume that the fusion alphabet, the alphabets of the complementary fusion and splicing labels and the terminal alphabet are pairwise disjoint. $Z(CG)$ consists of the following connected components: $S \circlearrowleft \overset{be}{\bullet} \circlearrowright act$, $\bullet \circlearrowright \overline{act}$, $c_x = \bullet \underset{\overline{A}_x}{\overset{x}{\rightleftarrows}} \bullet \circlearrowright act$ for each $x \in N \cup T$, and $c_p = \bullet \xrightarrow{v_1} \bullet \cdots \bullet \xrightarrow{v_l} \bullet \circlearrowright act$ with $\overline{A}_p$ from last to first node, for each $p = (u, v_1 \dots v_l) \in P$ with $v_j \in N \cup T$ for $j = 1, \dots, l$. $SR(CG)$ consists of the following splicing rules with fixed disjoint context:

– $spl(x) = ($  $)$

for each $x \in N \cup T$, and

– $spl(p) = ($  $)$

for each $p = (u_1 \dots u_k, v) \in P$.

To explain how this splicing/fusion grammar works, one may consider the following cycle representation of strings: $cyc(x_1 \dots x_n)$ for $x_1 \dots x_n \in N \cup T, n \geq 1$, which is depicted in Fig. 1a. It can be defined more formally as $([n], \{(i, i+1, x_i) \mid i = 1, \dots, n-1\} \cup \{(n, 1, x_n\} \cup \{(1, 1, be)\}, pr_1, pr_2, pr_3)$ where sources, targets and labels of the edges are given by the projections. Moreover, define $cyc(\lambda) = be \circlearrowleft \bullet$.

Consider now $cyc(x_1 \dots x_n)^{act,i}$ which is $cyc(x_1 \dots x_n)$ with an additional *act*-loop at node i. Then the rule $spl(x_i)$ can be applied yielding two disjoint components. One component is obtained by removing the *act*-loop from the cycle and replacing the x_i-edge by an A_{x_i}-edge yielding $cyc(x_1 \dots x_{i-1} A_{x_i} x_{i+1} \dots x_n)$. The other component is the context of the rule with an additional $\widehat{A}_{x_i}$-edge parallel to the x_i-edge. Therefore, the two components look as the ones depicted in Fig. 1b. Afterwards, the cycle can be fused with the component c_{x_i} of $Z(CG)$ yielding Fig. 1c, i.e., $cyc(x_1 \dots x_n)^{act,i+1}$. In other words, an *act*-loop on $cyc(x_1 \dots x_n)$ can be moved around the cycle to each node by repetition of these two rule applications.

Consider now $cyc(x_1 \dots x_n)^{act,i}$ with $x_i \dots x_{i+k-1} = u_1 \dots u_k$ for some $p = (u_1 \dots u_k, v_1 \dots v_l) \in P$. Then the splicing rule $spl(p)$ can be applied

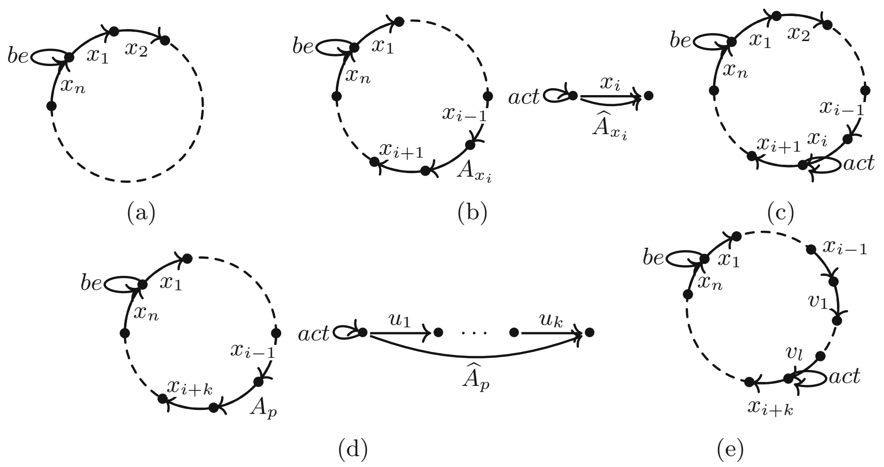


Fig. 1. Components of the derivation

yielding Fig. 1d. Afterwards, $cyc(x_1 \dots x_{i-1} A_p x_{i+1} \dots x_n)$ can be fused with the component c_p yielding Fig. 1e. This means that a direct derivation $w = x_1 \dots x_{i-1} u_1 \dots u_k x_{i+k} \dots x_n \quad \rightarrow \quad x_1 \dots x_{i-1} v_1 \dots v_l x_{i+k} \dots x_n = w'$ in CG can be simulated by a derivation $H \overset{*}{\Longrightarrow} H'$ in $SFG(CG)$ in such a way that H' contains the connected component $cyc(w')^{act,i+k}$ provided that H contains $cyc(w)^{act,j}$ for some $j \in \{1, \dots, n\}$, a copy of c_p, and enough copies of the components c_x for $x \in N \cup T$. The derivation may start with moving the *act*-loop from j to i and finish with applying the rule $spl(p)$ followed by the application of $fr(A_p)$. Consequently, a derivation $S \xrightarrow[P]{*} \overline{w}$ in CG induces a derivation $Z(CG) \overset{*}{\Longrightarrow} H$ such that $cyc(\overline{w})^{act,m}$ for some m is a connected component of H, because it can be done step by step and $Z(CG)$ contains $cyc(S)^{act,1}$ and the components c_x for $x \in N \cup T$ and c_p for $p \in P$. The latter ones can be multiplied as much as needed, and the start cycle is a special case of $cyc(w)^{act,j}$. Finally, the *act*-loop can be fused with the component $\bullet\!\!\circlearrowleft\overline{act}$ removing the *act*-loop from $cyc(\overline{w})^{act,j}$ yielding $cyc(\overline{w})$. If $\overline{w} \in T^*$, i.e., $\overline{w} \in L(CG)$, then $cyc(\overline{w})$ is terminal in $SFG(CG)$ such that $cyc(\overline{w}) \in L(SFG(CG))$. This proves

$$cyc(L(CG)) = \{cyc(\overline{w}) \mid \overline{w} \in L(CG)\} \subseteq L(SFG(CG)).$$

A closer look reveals further properties of the derivation process in $SFG(CG)$ starting from $Z(CG)$. Each derived graph has exactly one cycle as a connected component either of the form $cyc(w)^{act,j}$ for some $w \in (N \cup T)^*$ or $cyc(uA_yv)$ for some $u, v \in (N \cup T)^*$ and for a fusion symbol A_y with $y \in N \cup T \cup P$ as long as the cycle is not multiplied. The components c_x for $x \in N \cup T$ and c_p for $p \in P$ cannot be spliced because the *act*-loops are attached to nodes without outgoing edges. The second component that results from splicing besides the cycles contains an edge labeled with a complementary splicing label that is neither a fusion symbol nor a complementary fusion symbol nor a terminal symbol. Hence, one cannot get rid of these edges so that these components never contribute to the generated language. The components c_x and c_p can contribute to the language, but only by fusions with the cycles. Finally, the cycle $cyc(\overline{w})^{act,j}$ for some $\overline{w} \in T^*$ can be terminated by fusing the two complementary *act*-hyperedges. This means, without loss of generality, that only derivations in $SFG(CG)$ may be considered which start with $Z(CG)$, multiply the component c_x and c_p as needed, apply the splicing and the corresponding fusion rule alternatively to the single present cycle until all labels except *act* are terminal, and end with the *act*-$\overline{act}$-fusion. The moving of the *act*-loop around a cycle $cyc(w)^{act,j}$ does not change the underlying string w. The substitution of $\bullet\xrightarrow{u_1}\bullet\cdots\bullet\xrightarrow{u_k}\bullet$ by $\bullet\xrightarrow{v_1}\bullet\cdots\bullet\xrightarrow{v_l}\bullet$ for some $p = (u_1 \dots u_k, v_1 \dots v_l) \in P$ by the respective pair of a splicing and a fusion corresponds to a direct derivation $w = xu_1 \dots u_k y \ \rightarrow \ xv_1 \dots v_l y = w'$ in CG provided that it does not match with the initial and the final section simultaneously, i.e., $w = u_i \dots u_k z u_1 \dots u_{i-1}$ for some $1 < i < k$. But this cannot happen because one of the middle nodes of the match has the *be*-loop so that splicing with the fixed disjoint context of $spl(p)$ is not possible. Summarizing, each string $\overline{w}$ underlying a cycle $cyc(\overline{w}) \in L(SFG(CG))$ is derivable from S in CG such that $\overline{w} \in L(CG)$.

Altogether, this proves the following result.

Theorem 1. *Let $CG = (N, T, P, S)$ be a Chomsky grammar and $SFG(CG)$ the corresponding splicing/fusion grammar. Then $cyc(L(CG)) = L(SFG(CG))$.*

6 Transformation of Hypergraph Grammars into Splicing/Fusion Grammars

In this section, hypergraph grammars are transformed into splicing/fusion grammars in such a way that the language generated by each source grammar coincides with the language generated by the target grammar. As splicing/fusion grammars generate only connected hypergraphs due to the employed selection mechanism, we restrict hypergraph grammars in such a way that the derivation process preserves connectedness.

Definition 6. A hypergraph grammar $HGG = (N, T, P, S)$ is called *connective* if the following holds: For each rule $r = (L \supseteq [k] \xrightarrow{b} R)$ for some $k \geq 1$, each connected component of L and R contains some gluing node, and if two gluing nodes i, j are connected by a path of L, then the nodes $b(i)$ and $b(j)$ are connected by a path in R.

It is easy to see that the application of such rules preserves connectedness. Therefore, the generated language $L(HGG)$ contains only connected hypergraphs.

The corresponding splicing/fusion grammar

$$SFG(HGG) = (Z(HGG), F(HGG), M(HGG), T(HGG), SR(HGG))$$

is constructed as follows:

- The alphabets are $F(HGG) = \{A_r \mid r \in P\}, M(HGG) = \{\mu\}, T(HGG) = T$, where the type of μ is 0 and the type of A_r for $r \in P$ is the number of gluing nodes of r, the complementary fusion label is denoted by $\overline{A}_r$, the complementary splicing label by $\widehat{A}_r$. Moreover, $F(HGG), \overline{F}(HGG) = \{\overline{A}_r \mid r \in P\}, \widehat{F}(HGG) = \{\widehat{A}_r \mid r \in P\}, M(HGG), T$ are pairwise disjoint.
- The start graph $Z(HGG)$ has the connected components $S^\bullet_\mu$ and $C(r)$ for each $r \in P$ where $S^\bullet_\mu$ is the hypergraph $S^\bullet$ together with an additional hyperedge labeled by μ and $C(r)$ for $r = (L \supseteq [k] \xrightarrow{b} R)$ is R together with an additional hyperedge y_r with label $\overline{A}_r$ and attachment $b(1)\cdots b(k)$.
- The set $SR(HGG)$ consists of the splicing rules with fixed disjoint context $spl(r) = (L \supseteq [k(A_r)] \xleftarrow{\langle 1_{[k(A_r)]}, 1_{[k(A_r)]}\rangle} [k(A_r)] + [k(A_r)] \xrightarrow{in+\widehat{in}} A^\bullet_r + \widehat{A}^\bullet_r$ for each $r = (L \supseteq [k] \xrightarrow{b} R) \in P$.

Note that the addition of a hyperedge of type 0 to a connected hypergraph yields a connected hypergraph as both hypergraphs have the same set of nodes and the same set of paths so that connectedness is preserved.

We want to prove now the following correctness result.

Theorem 2. *Let $HGG = (N, T, P, S)$ be a connective hypergraph grammar and $SFG(HGG)$ its corresponding splicing/fusion grammar. Then $L(HGG) = L(SFG(HGG))$.*

The proof of the theorem relies on the following pushout property.

Lemma 1. Let C be a category with coproduct (denoted by $+$) and pushouts. Consider the following diagrams:

$$\begin{tikzcd} L \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "a"'] \arrow[d, "d"] \\ H & I \arrow[l, "m"] \end{tikzcd} \qquad \begin{tikzcd} K \arrow[d, "g\circ a"'] \arrow[dr, phantom, "(2)"] & K+K \arrow[l, "\langle 1_K, 1_K\rangle"'] \arrow[d, "d+a"] \\ H & I+L \arrow[l, "\langle m, g\rangle"] \end{tikzcd} \tag{1, 2}$$

Then the left diagram is a pushout if and only if the right diagram is a pushout.

Proof. Using the coproduct property of $K+K$, (2) commutes iff its compositions with the two injections $in_i\colon K \to K+K$ for $i = 1, 2$ commute, i.e., together with the properties of induced morphisms (3) and (4):

$$g \circ a = g \circ a \circ \langle 1_K, 1_K\rangle \circ in_1 \underset{(2)}{=} \langle m, g\rangle \circ (d+a) \circ in_1 = \langle m, g\rangle \circ in_I \circ d = m \circ d, \tag{3}$$

$$g \circ a = g \circ a \circ \langle 1_K, 1_K\rangle \circ in_2 \underset{(2)}{=} \langle m, g\rangle \circ (d+a) \circ in_2 = \langle m, g\rangle \circ in_L \circ a = g \circ a \tag{4}$$

where $in_I\colon I \to I+L$ and $in_L\colon L \to I+L$ are the injections of the coproduct $I+L$. While (4) holds always, (3) holds iff (1) is commutative.

Using the same arguments, the universal pushout property of (1) turns out to be equivalent to the universal pushout property of (2).

Let $i\colon I \to X$ and $l\colon L \to X$ for some X be two morphisms with $l \circ a = i \circ d$. If (1) is a pushout, then there is a unique $x\colon H \to X$ with

$$x \circ m = i \text{ and } x \circ g = l. \tag{5}$$

Let now $k\colon K \to X$ and $f\colon I + L \to X$ be two morphisms with

$$k \circ \langle 1_K, 1_K \rangle = f \circ (d + a). \tag{6}$$

(6) is equivalent to (7) and (8):

$$k = \; k \circ \langle 1_K, 1_K \rangle \circ in_1 = f \circ (d + a) \circ in_1 = f \circ in_I \circ d, \tag{7}$$
$$k = \; k \circ \langle 1_K, 1_K \rangle \circ in_2 = f \circ (d + a) \circ in_1 = f \circ in_L \circ a. \tag{8}$$

Choosing $i = f \circ in_I$ and $l = f \circ in_L$, we get a unique $x\colon H \to X$ with $x \circ m = f \circ in_I$ and $x \circ g = f \circ in_L$ because (1) is a pushout. This implies:

$$x \circ g \circ a = f \circ in_L \circ a = k, \tag{9}$$
$$x \circ \langle m, g \rangle \circ in_I = x \circ m = f \circ in_I, , \tag{10}$$
$$x \circ \langle m, g \rangle \circ in_L = x \circ g = f \circ in_L. \tag{11}$$

Therefore, using the coproduct property of $I + L$, one gets $x \circ \langle m, g \rangle = f$, which shows together with (9), that (2) is a pushout if (1) is a pushout.

Conversely, let (2) be a pushout. Let i and l be morphisms that satisfy $l \circ a = i \circ d$. This implies:

$$\langle i, l \rangle \circ (d + a) \circ in_1 = \langle i, l \rangle \circ in_I \circ d = i \circ d = l \circ a, \tag{12}$$
$$\langle i, l \rangle \circ (d + a) \circ in_2 = \langle i, l \rangle \circ in_L \circ a = l \circ a. \tag{13}$$

Using the coproduct property of $K + K$, (12) and (13) induce (14):

$$\langle i, l \rangle \circ (d + a) = l \circ a \circ \langle 1_K, 1_K \rangle. \tag{14}$$

As (2) is pushout, (14) implies a unique morphism $x\colon H \to X$ with $x \circ g \circ a = l \circ a$ and $x \circ \langle m, g \rangle = \langle i, l \rangle$ so that $x \circ m = x \circ \langle m, g \rangle \circ in_I = \langle i, l \rangle \circ in_I = i$ and $x \circ g = x \circ \langle m, g \rangle \circ in_L = \langle i, l \rangle \circ in_L = l$. In other words, (2) is pushout, which completes the proof.

Proof (Theorem 2). We start by showing that a direct derivation $H \underset{r}{\Longrightarrow} H'$ for $r = (L \supseteq [k] \xrightarrow{b} R) \in P$ can be simulated in $SFG(HGG)$ if H is connected. The direct derivation is defined by the double pushout (15), (16) where a is the inclusion of $[k]$ into L.

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "(15)"] & {[k]} \arrow[l, "a"'] \arrow[r, "b"] \arrow[d, "d"] \arrow[dr, phantom, "(16)"] & R \arrow[d, "h"] \\
H & I \arrow[l, "m"] \arrow[r, "m'"'] & H'
\end{tikzcd} \tag{15, 16}$$

Due to the Lemma 1, the pushout (15) induces the pushout (17) so that the application of the splicing rule $spl(r)$ is defined and completed by the pushout (18)

$$\begin{tikzcd}
K \arrow[d, "{m \circ d = g \circ a}"'] \arrow[dr, phantom, "(17)"] & K+K \arrow[l, "{\langle 1_K, 1_K \rangle}"'] \arrow[d, "d+a"] & K+K \arrow[r, "in+\widehat{in}"] \arrow[d, "d+a"'] \arrow[dr, phantom, "(18)"] & A_r^{\bullet}+\widehat{A}_r^{\bullet} \arrow[d, "d'+a'"] \\
H & I+L \arrow[l, "{\langle m, g \rangle}"] & I+L \arrow[r, "in'+\widehat{in'}"'] & I^{A_r,d}+L^{\widehat{A}_r,a}
\end{tikzcd} \tag{17, 18}$$

where $K = [k(A)]$ and $H^{A,f}$ for $H \in \mathcal{H}_\Sigma$, $A \in \Sigma$ with type k and $f\colon [k] \to H$ denotes the hypergraph H with an additional hyperedge which has the attachments $f(1) \dots f(k)$ and the label A which can be constructed for the components of a coproduct separately.

If H is connected, then $I^{A_r,d}$ is connected because the new hyperedge connects all nodes that are connected through $g(L)$ in H. Further, the fusion of $I^{A_r,d}$ and $C(r) = R^{\overline{A}_r, in_{\overline{A}_r}}$ is defined by the double pushout

$$\begin{tikzcd}
A_r^{\bullet}+\overline{A}_r^{\bullet} \arrow[d, "d'+b'"'] \arrow[dr, phantom, "(19)"] & K+K \arrow[l, "in_1+in_2"'] \arrow[r, "{\langle 1_K, 1_K \rangle}"] \arrow[d, "d+b"] \arrow[dr, phantom, "(20)"] & K \arrow[d, "h"] \\
I^{A_r,d}+R^{\overline{A}_r, in_{\overline{A}_r}} & I+R \arrow[l, "\supseteq"] \arrow[r, "m''"'] & H''
\end{tikzcd} \tag{19, 20}$$

where b' maps the $\overline{A}_r$-hyperedge to the corresponding hyperedge attached to R. Due to the Lemma 1, the pushout (20) induces the pushout

$$\begin{tikzcd}
{[k]} \arrow[r, "b"] \arrow[d, "d"'] \arrow[dr, phantom, "(21)"] & R \arrow[d, "m'' \circ in_R"] \\
I \arrow[r, "m'' \circ in_I"'] & H''
\end{tikzcd} \tag{21}$$

As both pushouts (16) and (21) are pushouts of b and d, H' and H'' are isomorphic.

According to the extension theorem in [5] the splicing and the fusion can be done on every hypergraph J that has H and $C(r)$ as connected components yielding $J \underset{spl(r)}{\Longrightarrow} J - H + I^{A_r,d} + L^{\widehat{A}_r,a} \underset{fr(A_r)}{\Longrightarrow} J - (H + C(r) + I^{A_r,d}) + L^{\widehat{A}_r,a} + H'$.

If H is marked by a μ-hyperedge, then $I^{A_r,d}$ is also marked by a μ-hyperedge because L is not marked, but L and I together cover H. Therefore, $I^{A_r,d}$ is marked by a μ-hyperedge from which $H' = H''$ inherits the marker.

As this reasoning applies to every derivation step of a derivation $Z^\bullet \underset{r_1}{\Longrightarrow} \dots \underset{r_n}{\Longrightarrow} \overline{H}$ in HGG, there is a derivation $Z(HGG) \overset{*}{\Longrightarrow} J$ in $SFG(HGG)$ such that J has a connected component $\overline{H}_\mu$, i.e., $\overline{H}$ with an additional μ-hyperedge. The derivation can start with the multiplication of the $C(r)$ for $r \in P$ as much as needed followed by the application of $spl(r_i)$ and $fr(A_{r_i})$ for $i = 1, \dots, n$ to the connected component with the μ-hyperedge. If $\overline{H}$ is terminal, then $\overline{H}$ is in $L(HGG)$ as well as in $L(SFG(HGG))$ as $\overline{H}$ results from $\overline{H}_\mu$. In other words, $L(HGG) \subseteq L(SFG(HGG))$.

The converse inclusion can be proved as follows. Consider $\overline{H} \in L(SFG(HGG))$. Then there is a derivation $Z(HGG) \overset{*}{\Longrightarrow} J$ in $SFG(HGG)$ where J has $\overline{H}_\mu$ as connected component. The derivation process has the following property in addition to the splicing and fusion properties of μ-components (i.e., μ-marked connected components) as considered above: Each application of a splicing rule produces one connected component with an $\widehat{F}$-hyperedge, i.e., a hyperedge labeled with a complementary splicing label. $\widehat{F}$-hyperedges cannot be removed. Therefore, one can assume that the given derivation does not process these components any further. The connected component $C(r)$ for $r \in P$ in $Z(HGG)$ has a single $\overline{F}$-hyperedge, i.e., a hyperedge labeled by a complementary fusion label. If a splicing rule is applied to such a connected hypergraph, then the $\overline{F}$-hyperedge is transmitted to one of the resulting components.

The application of a fusion rule to two such connected components preserves one $\overline{F}$-hyperedge. To get rid of it, one may apply a fusion rule to a single component with an $\overline{F}$-hyperedge or to such a component and a μ-component. In the first case, one gets components that cannot be fused with a μ-component. Therefore, one can assume that the given derivation has no derivation step of this kind. In the second case, the resulting μ-component has no $\overline{F}$-hyperedge and no $\widehat{F}$-hyperedge if the fused μ-component has none such hyperedges. In particular, μ-components cannot be fused with each other. Therefore, one can assume that neither the initial component $S^\bullet_\mu$ nor any derived μ-component is multiplied in the given derivation. Moreover, it is not difficult to see that each two derivation steps can be interchanged with each other with the exception of applying a splicing rule $spl(r)$ followed by the application of the fusion rule $fr(A_r)$ with respect to the A_r-hyperedge created by the splicing. Summarizing, each derivation $Z(HGG) \overset{*}{\Longrightarrow} J$ can be assumed to be in the normal form where, besides multiplication of the $C(r)$ for $r \in P$, the derivation steps are an alternation between an application of a splicing rule $spl(r)$ for some $r \in P$ and an application of the fusion rule $fr(A_r)$, both involving the only μ-component. As shown above, such a derivation corresponds to a derivation $S^\bullet \underset{P}{\overset{*}{\Longrightarrow}} \overline{H}$ in HGG if $\overline{H}_\mu$ is the μ-component of J. Provided that $\overline{H}$ is terminal, this proves $L(SFG(HGG)) \subseteq L(HGG)$ completing the proof.

7 Conclusion

In this paper, we have introduced splicing/fusion grammars as a device for generating hypergraph languages. Derivations operate on connected components of hypergraphs: They can be multiplied, fixed parts can be cut off by splicing and the attachments of complementary hyperedges can be fused. We proved that splicing/fusion grammars can simulate Chomsky grammars on one hand and connective hypergraph grammars on the other hand. Both results show the significant generative power of these grammars. In particular, it is much greater than the generative power of fusion grammars (i.e., splicing/fusion grammars with an empty set of splicing rules). This follows from the fact that the membership problem of fusion grammars is decidable.

Further research on splicing/fusion grammars may include the following topics.

The basic operations on hypergraphs, splicing, fusion, and multiplication, that are employed in the derivation process of splicing/fusion grammars are inspired and motivated by the corresponding operations in DNA computing. An interesting question in this connection is how language-generating devices based on DNA computing like sticker systems, insertion/deletion systems and splicing system (see, e.g., [4]) are related to splicing/fusion grammars.

The idea behind DNA computing is to interpret tubes of DNA molecules as data structures and chemical reactions among the molecules as computational steps. The result of a computation depends on the molecules one finds in the tube at the end of the process. From a mathematical point of view, a tube of molecules is a multiset of molecules, as many structurally identical molecules may be present. In splicing/fusion grammars, molecules are replaced by connected hypergraphs and multisets of molecules are replaced by hypergraphs (where a hypergraph represents a multiset of connected hypergraphs by counting the number of isomorphic connected components). This is the reason why the language generated by a splicing/fusion grammar consists of connected components of derived hypergraphs. On the other hand, the restriction to connected hypergraphs can be inconvenient and undesirable. Therefore, it would be helpful to find a way to overcome the connectedness requirement.

In this paper, we use a special case of splicing where some fixed part of a hypergraph is cut off. Moreover, in the main results of Sects. 5 and 6, the cut-off is "junk", as it cannot contribute to members of the generated language. But there are various other meaningful possibilities. For example, splicing can cut a hypergraph into two pieces that both may be useful in the further processing. Or other positive or negative context conditions may be required. We are convinced that interesting phenomena can be modeled and analyzed in this way.

Acknowledgment. We are greatful to the anonymous reviewers for their valuable comments.

References

1. Kreowski, H.-J., Kuske, S., Lye, A.: Fusion grammars: a novel approach to the generation of graph languages. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 90–105. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_6
2. Kreowski, H.-J., Klempien-Hinrichs, R., Kuske, S.: Some essentials of graph transformation. In: Ésik, Z., Martín-Vide, C., Mitrana, V. (eds.) Recent Advances in Formal Languages and Applications. Studies in Computational Intelligence, vol. 25, pp. 229–254. Springer, Berlin (2006). https://doi.org/10.1007/978-3-540-33461-3_9
3. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1, pp. 163–245. World Scientific, Singapore (1997)
4. Păun, G., Rozenberg, G., Salomaa, A.: DNA Computing — New Computing Paradigms. Springer, Heidelberg (1998). https://doi.org/10.1007/978-3-662-03563-4
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. (eds.): Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2

Synchronous Hyperedge Replacement Graph Grammars

Corey Pennycuff, Satyaki Sikdar, Catalina Vajiac, David Chiang, and Tim Weninger(✉)

University of Notre Dame, Notre Dame, IN 46556, USA
{cpennycu,ssikdar,cvajiac,dchiang,tweninge}@nd.edu

Abstract. Discovering the underlying structures present in large real world graphs is a fundamental scientific problem. Recent work at the intersection of formal language theory and graph theory has found that a Probabilistic Hyperedge Replacement Grammar (PHRG) can be extracted from a tree decomposition of any graph. However, because the extracted PHRG is directly dependent on the shape and contents of the *tree decomposition*, rather than from the dynamics of the graph, it is unlikely that informative graph-processes are actually being captured with the PHRG extraction algorithm. To address this problem, the current work adapts a related formalism called Probabilistic Synchronous HRG (PSHRG) that learns synchronous graph production rules from temporal graphs. We introduce the PSHRG model and describe a method to extract growth rules from the graph. We find that SHRG rules capture growth patterns found in temporal graphs and can be used to predict the future evolution of a temporal graph. We perform a brief evaluation on small synthetic networks that demonstrate the prediction accuracy of PSHRG versus baseline and state of the art models. Ultimately, we find that PSHRGs seem to be very good at modelling dynamics of a temporal graph; however, our prediction algorithm, which is based on string parsing and generation algorithms, does not scale to practically useful graph sizes.

Keywords: Graph generation · Hyperedge replacement
Temporal graphs

1 Introduction

The discovery and analysis of network patterns is central to the scientific enterprise. Thus, extracting the useful and interesting building blocks of a network is critical to the advancement of many scientific fields. Indeed the most pivotal moments in the development of a scientific field are centered on discoveries about the structure of some phenomena [14]. For example, chemists have found that many chemical interactions are the result of the underlying structural properties of interactions between elements [7]. Biologists have agreed that tree structures are useful when organizing the evolutionary history of life [8], sociologists find

L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 20–36, 2018.
https://doi.org/10.1007/978-3-319-92991-0_2

that triadic closure underlies community development [11], and neuroscientists have found “small world” dynamics within the brain [4]. Unfortunately, current graph research deals with small pre-defined patterns [13] or frequently reoccurring patterns [15], even though interesting and useful information may be hidden in unknown and non-frequent patterns.

Pertinent to this task are algorithms that learn the LEGO-like building blocks of real world networks in order to gain insights into the mechanisms that underlie network growth and evolution. In pursuit of these goals, Aguinaga *et al.* recently expanded on the relationship between graph theory and formal language theory that allows for a *Hyperedge Replacement Grammar* (HRG) to be extracted from the *tree decomposition* of a graph [2]. In this perspective, the extracted HRG contains the precise building blocks of the graph as well as the instructions by which these building blocks ought to be pieced together [9,12]. In addition, this framework is able to extract patterns of the network's structure from small samples of the graph probabilistically via a Probabilistic HRG (PHRG) in order to generate networks that have properties that match those of the original graph [1].

In their typical use-case, context free grammars (CFGs) and their probabilistic counterpart PCFGs are used to represent and generate patterns of strings through rewriting rules. A natural language parser, for example, learns how sentences are recursively built from smaller phrases and individual words. In this case, it is important to note that the CFG production rules used by natural language parsers encode the way in which sentences are logically constructed, that is, the CFG contains descriptive information about how nouns and verbs work together to build coherent sentences. CFGs can therefore generate new sentences that are at least grammatically correct. This is not the case with HRGs.

On the contrary, the PHRG is completely dependent on the graph's tree decomposition. Unfortunately, there are many ways to perform a tree decomposition on a given graph, and even the optimal, *i.e.*, minimal-width, tree decomposition is not unique. As a result, the production rules in a standard HRG are unlikely to represent the temporal processes that generated the graph.

In the present work we address this problem by learning rules from a temporal graph, in which edges are added and removed at various timesteps, using a related formalism called Synchronous CFGs. SCFGs are a lot like regular CFGs, except that their production rules have two right hand sides, a *source* and a *target*. SCFGs are typically used to perform natural language translation by producing related sentences from a source language into a target language. We reproduce an example synchronous grammar from Chiang [5] for illustration purposes here:

$$\begin{array}{rlcc}
\mathsf{S} \rightarrow & \mathsf{NP}_1\ \mathsf{VP}_2 & : & \mathsf{NP}_1\ \mathsf{VP}_2 \\
\mathsf{VP} \rightarrow & \mathsf{V}_1\ \mathsf{NP}_2 & : & \mathsf{NP}_2\ \mathsf{V}_1 \\
\mathsf{NP} \rightarrow & \texttt{i} & : & \texttt{watashi ha} \\
\mathsf{NP} \rightarrow & \texttt{the box} & : & \texttt{hako wo} \\
\mathsf{V} \rightarrow & \texttt{open} & : & \texttt{akemasu}
\end{array}$$

where the subscript numbers represent links that synchronize the nonterminals between the source and target RHSs. This grammar can then be used to generate sentences in both English and Japanese. Starting from the synchronous starting nonterminal S_1, we apply these rules to generate the following sentences synchronously (Table 1).

Table 1. Derivation of two sentences from a synchronous CFG.

Rule	English	Japanese
	S_1	S_1
1	NP_2 VP_3	NP_2 VP_3
2	NP_2 V_4 NP_5	NP_2 NP_5 V_4
3	`i` V_4 NP_5	`watashi ha` NP_5 V_4
4	`i` V_4 `the box`	`watashi ha hako wo` V_4
5	`i open the box`	`watashi ha hako wo akemasu`

From the synchronous grammar formalism we see that the production rules encode precise translations between the source and target language. Apart from word-to-word translations, an analysis of these rules would also show the differences in how sentences are constructed in each language. A machine translation system would translate a sentence by parsing the given sentence using the LHS and source-RHS rules. An application of the production rules from the resulting parse tree could regenerate the original source sentence if the source-RHS was applied, or it would generate a translation into the target language if the target-RHSs were applied instead.

Synchronous HRGs (SHRGs, pronounced "shergs") are to HRGs as synchronous CFGs are to CFGs. SHRGs have been proposed for performing natural language understanding and generation by mapping between strings and abstract meaning representations (AMRs), which are graphical representations of natural language meaning.

The present work expands upon work in SHRGs with the observation that the temporal dynamics of a graph, *i.e.*, the changes from one timestep to another, can be represented like a translation from a source language to a target language. *Towards this goal, this paper presents a SHRG extraction algorithm for connected, temporal hypergraphs.* We find that these SHRGs encode interesting information about the temporal dynamics of the graph, and we show that a parse of a graph can be used to predict its future growth.

2 Definitions

Before we describe the SHRG extraction method, some background definitions are needed.

A *hypergraph* is a tuple (V, E), where V is a finite set of vertices and $E \subseteq V^*$ is a set of hyperedges, each of which is a set of zero or more vertices. A (common) *graph* is a hypergraph in which every edge connects exactly two vertices.

2.1 Synchronous Hyperedge Replacement Grammars

A HRG generates hypergraphs in a way analogous to the way a context-free grammar generates strings. It has rules of the form $A \rightarrow R$, which means that an edge labeled A can be rewritten with a hypergraph fragment R. A HRG derivation starts with the start nonterminal S and applies a rule $S \rightarrow R$ to it. The replacement R can itself have other nonterminal hyperedges, so this process is repeated until there are no more nonterminals in the graph.

A SHRG is similar, but each production has two right-hand sides, which we call the source RHS, R_S and the target RHS, R_T. A SHRG generates two graphs by repeatedly choosing a nonterminal A and rewriting it using a production rule $A \rightarrow R_S : R_T$. We define SHRGs more formally as follows.

Definition 1. *A* synchronous hyperedge replacement grammar *(SHRG) is a tuple $G = \langle N, T, S, \mathcal{P} \rangle$, where*

1. *N is a finite set of nonterminal symbols. Each nonterminal A has a nonnegative integer* rank*, which we write $|A|$.*
2. *T is a finite set of terminal symbols.*
3. *$S \in N$ is a distinguished starting nonterminal, and $|S| = 0$.*
4. *$\mathcal{P}$ is a finite set of production rules $A \rightarrow R_S : R_T$, where*
   - *A, the left hand side (LHS), is a nonterminal symbol.*
   - *R_S and R_T comprise the right hand side (RHS) and are hypergraphs whose edges are labeled by symbols from $T \cup N$. If an edge e is labeled by a nonterminal B, we must have $|e| = |B|$.*
   - *Exactly $|A|$ vertices of R_S are designated* external vertices*. The other vertices in R_S and R_T are called* internal *vertices.*
   - *There is a partial graph isomorphism, which we call the* linking *relation, between R_S and R_T. At minimum, the linking relation includes all the nonterminal edges of both R_S and R_T. It should respect edge labels as well as the distinction between internal and external vertices.*

When illustrating SHRG rules, we draw the LHS A as a hyperedge labeled A with arity $|A|$. We draw the RHS-pair as two hypergraphs separated by a dashed vertical bar, with external vertices drawn as solid black circles and the internal vertices as open white circles. Nonterminal hyperedges are drawn with boxes labeled with a nonterminal symbol and a subscript index that is *not* part of the nonterminal label; nonterminals with the same index are linked.

2.2 Tree Decompositions

Definition 2. *A* tree decomposition *of a hypergraph $H = (V, E)$ is a rooted tree T whose nodes are called* bags*. Each bag η is labeled with a $V_\eta \subseteq V$ and $E_\eta \subseteq E$, such that the following properties hold:*

1. *Vertex Cover: For each $v \in V$, there is a vertex $\eta \in T$ such that $v \in V_\eta$.*
2. *Edge Cover: For each hyperedge $e_i = \{v_1, \dots, v_k\} \in E$ there is exactly one node $\eta \in T$ such that $e \in E_\eta$. Moreover, $v_1, \dots, v_k \in V_\eta$.*
3. *Running Intersection: For each $v \in V$, the set $\{\eta \in T \mid v \in V_\eta\}$ is connected.*

Definition 3. *The* width *of a tree decomposition is* $\max(|V_\eta - 1|)$*, and the* treewidth *of a graph H is the minimal width of any tree decomposition of H.*

3 Related Work

Tree decompositions and HRGs have been studied separately for some time in discrete mathematics and graph theory literature. HRGs are conventionally used to generate graphs with very specific structures, *e.g.*, rings, trees, stars. A drawback of many current applications of HRGs is that their production rules must be manually defined. For example, the production rules that generate a ring-graph are distinct from those that generate a tree, and defining even simple grammars by hand is difficult or impossible. Very recently, Kemp and Tenenbaum developed an inference algorithm that learned probabilities of the HRG's production rules from real world graphs, but they still relied on a handful of rather basic hand-drawn production rules (of a related formalism called vertex replacement grammar) to which probabilities were learned [13]. Kukluk et al. Holder and Cook were able to define a grammar from frequent subgraphs [15], but their methods have a coarse resolution because *frequent* subgraphs only account for a small portion of the overall graph topology.

In earlier work we showed that an HRG can be extracted from a static graph and used to generate new graphs that maintained the same global and local properties as the original graph. We also proved under certain (impractical) circumstances that the HRG can be used to generate an isomorphic copy of the original graph [2].

Prior work in HRGs extract their production rules directly from a tree decomposition, but because the tree decomposition can vary significantly, the production rules may also vary significantly. Although good at generating new synthetic graphs that are similar to the original graph, the production rules of HRGs do not describe the growth process that created the graph.

4 Extracting SHRGs from Temporal Graphs

In this section, we show how to extract growth rules from *temporal graphs* into a SHRG. A temporal graph represents a set of entities whose identities persist over time, and an edge in graph $H^{(i)}$ represents the interaction of a set of entities at time i. More formally,

Definition 4. *A* temporal graph *is a finite set of vertices V together with a sequence of graphs $H^{(1)}, \dots, H^{(n)}$, where for each graph $H^{(i)} = (V^{(i)}, E^{(i)})$, $V^{(i)} \subseteq V$.*

Furthermore, in this paper, we assume that each $H^{(i)}$ is connected, so that a vertex only appears when it becomes connected to the graph.

4.1 Method

Figure 1 illustrates an example graph with 4 timesteps $H^{(1)}, \ldots, H^{(4)}$. Each timestep adds and/or deletes one or more edges as shown by bold or dashed lines, respectively. Vertices are marked with lowercase Latin letters for illustration purposes only. Although the example graph is directed and simple, and each edge only connects two vertices, our only requirement is that the graph must be a single connected component.

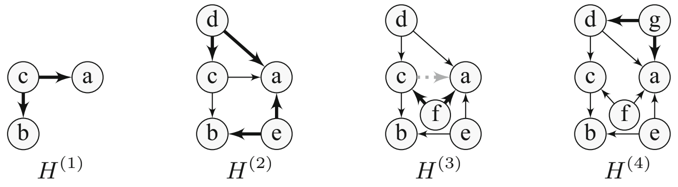


Fig. 1. Example of a temporal, directed, connected graph with 4 timesteps. Bold edges show additions within a timestep; dashed edges show deleted edges within a timestep.

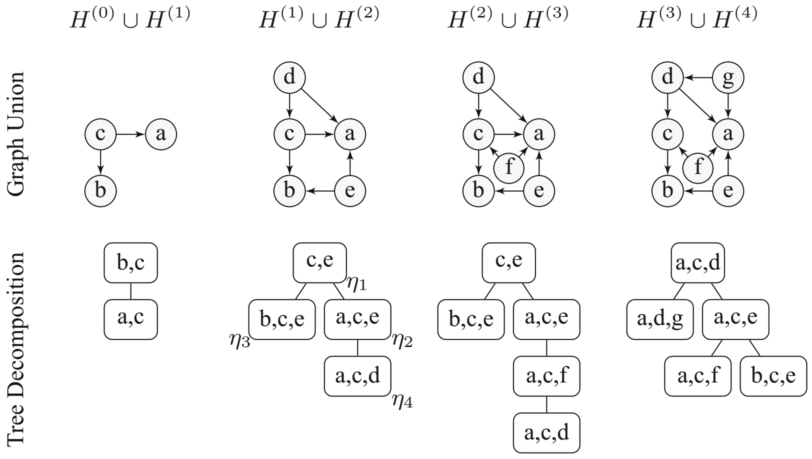


Fig. 2. Graph unions from the running example established in Fig. 1. $\eta_1 \ldots \eta_4$ correspond to rules that will be extracted in Fig. 3.

In previous work, we showed that a (non-synchronous) HRG can be extracted from a graph using its tree decomposition [2]. In the present work, we use the same idea to extract a SHRG from a temporal graph; the primary difference is that our goal is now to encode differences from one timestep to another in the production rules. Our key insight is to treat two adjacent timesteps of the graph, $H^{(i)}$ and $H^{(i+1)}$, as coming from two "languages." Then, the evolution of the graph can be viewed as "translation" from one language to the other. The challenge that remains is how to extract a SHRG from two timesteps of the temporal graph.

For this task, we first note that the extracted SHRG should:

1. describe the temporal dynamics of the graph,
2. generate both graphs, and
3. predict the future growth of a graph.

Our non-synchronous method uses a tree decomposition of the graph as a guide for extracting HRG rules. In the synchronous case, we form the union of two timesteps, $H^{(i)} \cup H^{(i+1)}$, and find a tree decomposition of the union.

The computational complexity of some downstream applications can be improved if the extracted grammar has at most two nonterminals in each RHS, and a rule with only one nonterminal should have at least one internal vertex [1]. As we will see, we can ensure this by a simple normalization step on the tree decomposition. First, if a bag η has $r > 2$ children, make a copy η'. The parent of η becomes the parent of η', and η' becomes the parent of η, and one of the children of η becomes a child of η'. Second, if a bag η contains the same vertices as its only child, merge η with its child. Figure 2 shows four tree decompositions for the four graph unions corresponding to the four timesteps of Fig. 1 (where $H^{(0)}$ is understood to be the empty graph). These tree decompositions have been binarized and pruned.

We extract a SHRG from the tree decomposition of the graph union in the following way. The nonterminal alphabet is $\{\mathrm{S}, \mathrm{N}^1, \mathrm{N}^2, \ldots\}$, where $|\mathrm{N}^i| = i$. For brevity, we omit the superscript and simply write all the N^i as N. Let η be a bag of the tree decomposition of $H^{(i)} \cup H^{(i+1)}$, let η' be its parent, and let $\eta_1, \ldots, \eta_m$ be its children. Bag η corresponds to a SHRG production $\mathrm{A}^{|\eta' \cap \eta|} \rightarrow R_S : R_T$ as follows. Then, R_S and R_T are formed as follows.

– Let R_S and R_T be isomorphic copies of the induced subgraphs $H^{(i)}[V_\eta]$ and $H^{(i+1)}[V_\eta]$, respectively. Vertices copied from the same vertex are linked, and edges copied from the same edge are linked.
– In both R_S and R_T, mark the (copies of) vertices in $\eta' \cap \eta$ as external vertices.
– In both R_S and R_T, remove all edges between external nodes.
– For each η_i, create in both R_S and R_T a hyperedge labeled $\mathrm{A}^{|\eta \cap \eta_i|}$ connecting the (copies of) vertices in $\eta \cap \eta_i$. These two hyperedges are linked.

Figure 3 illustrates part of the grammar extracted from the graph originally presented in Fig. 1. The rules shown are extracted from $H^{(1)} \cup H^{(2)}$; the rule labeled η_1 corresponds to bag η_1 in the decomposition of $H^{(1)} \cup H^{(2)}$. We note again that the node labels a–e are shown for illustration purposes only; italicized labels x and y, which are drawn above the vertices, indicate the linking relation.

The root bag η_1 in the tree decomposition has no parent. So the rule extracted from η_1 has the start nonterminal, S, as its LHS. The RHSs in η_1 are isomorphic because the subgraph induced by nodes c and e are the same in both $H^{(1)}$ and $H^{(2)}$. This rule has two nonterminal hyperedges. The first is drawn between c and e because the intersection of η_1 and its left-child η_3 is c and e. The second nonterminal hyperedge is also drawn between c and e because the intersection between η_1 and its right-child η_2 is also c and e. Next we see that η_2 has a LHS

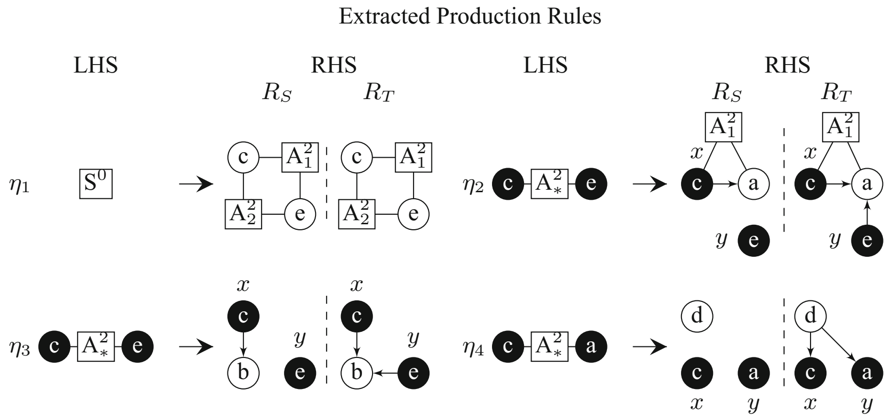


Fig. 3. Rules extracted from $H^{(1)} \cup H^{(2)}$ from Fig. 2. RHSs include internal and external vertices from the union-graph, nonterminal hyperedges corresponding to each tree decomposition node's children, and terminal edges induced from the source and target graphs respectively. Vertices are labeled a, b, c, d, e for illustration purposes. The labels x and y indicate the linking relation.

of size-2, because η_2 and its parent η_1 have two vertices, c and e, in common; c and e are also marked as external vertices on the RHSs. The subgraphs induced by a, c, and e in $H^{(1)}$ and $H^{(2)}$ are drawn as R_S and R_T respectively, and a nonterminal edge is drawn between a and c on both R_S and R_T because η_2 and its only-child η_4 share vertices a and c. Finally, rules corresponding to η_3 and η_4 are drawn in a similar fashion, except that they do not contain any nonterminal hyperedges on their RHSs because η_3 and η_4 do not have any children, *i.e.*, leaf-nodes in the tree decomposition produce terminal rules in the grammar.

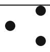
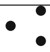
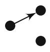
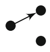
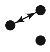
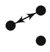
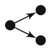
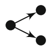
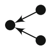
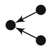
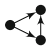
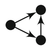
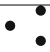
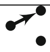
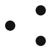
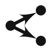
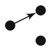
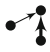
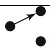
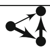
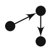
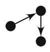
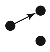
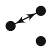
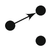
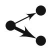
Note that this example only shows the rules that are extracted from one time step. Rules will also be extracted from the other three time steps to complete the SHRG grammar. In the frequent case that two extracted rules are identical, we do not store the same rule twice. Instead, we generate a probabilistic SHRG, called a PSHRG, by keeping a count of the number of times we see a rule and then normalizing the rule counts for each unique LHS (analogous to how a PHRG adds probabilities to an HRG). For example, if there existed an η_5 that produced a rule that was identical to η_2, we would note that we saw η_2 twice.

4.2 Exploring the Grammar

Using the PSHRG extraction methodology described above, our first task is to see if the PSHRG grammar is able to capture the dynamic processes involved within the evolution of a graph. A simple test in this task is to see if PSHRG can model the dynamics of a known generative process like the preferential attachment dynamics found in the Barabasi-Albert (BA) generative model [3].

Except for the edge deletion in $H^{(3)}$, the running example introduced in Fig. 1 and used throughout this paper represents the BA generative model. This

Table 2. RHSs of rules extracted from a graph grown with using the Barabasi-Albert (BA) preferential attachment process and compared with three versions of the Erdos-Renyi (ER) random graph process. The addition of outgoing wedges in line 8 and the attachment of an edge where one already exist in line 9 indicates that the extracted PSHRG grammar contains the dynamic processes used to generate BA graphs.

		R_S		R_T	BA	ER_1	ER_2	ER_3
Static Rules	1		:		0.051	0.157	0.171	0.153
	2		:		0.133	0.545	0.542	0.412
	3		:		0.000	0.006	0.017	0.007
	4		:		0.260	0.047	0.034	0.029
	5		:		0.032	0.028	0.030	0.017
	6		:		0.025	0.009	0.000	0.002
BA Rules	7		:		0.243	0.164	0.166	0.325
	8		:		0.097	0.000	0.000	0.001
	9		:		0.155	0.012	0.004	0.014
Other Rules	10		:		0.001	0.000	0.000	0.002
	11		:		0.000	0.021	0.031	0.028
	12		:		0.000	0.006	0.006	0.008
	13		:		0.000	0.006	0.005	0.007

process grows graphs by creating n-nodes and wiring them together in a particular way. Specifically, in each timestep, a new vertex is connected to the graph by connecting m directed edges outward from the new vertex to m existing vertices. The connecting vertices are chosen from within the larger graph through a stochastic process that assigns a connecting probability proportional to the number of in-links that each vertex already contains. Except for the deleted edge, the graph from Fig. 1 is created with $n = 7$ and $m = 2$, where each new vertex is connected to 2 existing vertices with a preference to attach to already well-connected vertices. Graphs generated by the BA process have well known properties. Most notable, for the purposes of the present work, is that this generative process is easy to visually identify.

What does the PSHRG grammar of the BA graph look like? Does the PSHRG capture the generative process of the BA model? To answer these questions, we generated 1000 graphs using the BA generative method with $n = 10$ and $m = 2$ over $n - m$ timesteps. We extracted a PSHRG from these 1000 synthetic temporal graphs. Ignoring the LHSs, we counted the graph-patterns found in R_S and R_T, and illustrate their occurrence rates in Table 2.

We compare these rules against graphs generated by 3 variations of the Erdos-Renyi (ER) random graph generation process [10]. The ER_1 random model creates a random temporal graph by generating n vertices in the first timestep, and then draws edges between two vertices with probability p at each timestep. ER_1 therefore creates a size n graph over $n(n-1)$ timesteps, where $n(1-p)$ timesteps are expected to contain no changes; when changes do occur, directed edges are drawn between two random vertices. The second variation ER_2 only counts a timestamp if an edge is created; the random graph in this case is expected to contain $p \cdot n(n-1)$ timesteps and directed edges. The third variation ER_3 creates 2 directed edges per timestep, *i.e.*, $p \cdot n(n-1)$ edges over $p \cdot n(n-1)/2$ timesteps.

We ignore nonterminal (hyper-)edges, remove unconnected nodes, and merge isomorphic, RHSs to create the illustration shown in Table 2. Normalized counts accompany R_S and R_T for the BA model and the three variants of the ER model. Rules 1–6 contain production rules that represent patterns that do not change from one timestep to the next; we find that the ER variants have similar counts with an unchanged edge representing more than half of all rules. The BA model produces some unchanged edges, but we also find that about a quarter of the rules are unchanged wedges (rule 4). The large number of outgoing wedges (rule 4) is indicative of the types of patterns that we expect to find in graphs generated by the BA model. Rule 7 shows that edges can be created between two vertices; there is only a slight difference between the BA model and ER variants. The creation of outgoing wedges (rule 8) represents nearly 10% of the patterns found in the BA model, but is never found in the ER variants. Rule 9 represents an interesting pattern of growth where a new vertex is attached to an existing vertex that already has an in-edge; these patterns are found in over 15% of BA rules, but only 1% of the ER variants. Rules 8 and 9 especially represent what we would expect to find in a representation of the BA model. Rule 8 has an obvious relationship with the BA model, and rule 9 represents the preferential attachment process that is encoded into the edge wiring process of the BA model.

In contrast, the ER variants have a large variation in their growth patterns. Rules 10–13 show only some of the most frequent patterns. Note that the BA model does not generate any of these rules; in fact, because loops are not possible in the BA model, rules 3 and 12 will never be extracted from a graph generated by the BA model.

In summary, these results show promising signs that the PSHRG does encode information about the growth properties of the underlying model. However, the present work only investigates the BA model in comparison to a random model. Further work is needed to make strong claims about the representative power of PSHRGs.

5 Predicting the Next Timestep

Here we show that PSHRGs extracted from the original temporal graph H can be used to generate new graphs.

Previous work in this area has shown that HRGs (not SHRGs) are able to generate isomorphic copies of the original graph, which is theoretically interesting but not practically useful. Other algorithms are able to generate fixed-size graphs of any size such that the new graphs have properties that are similar to the original graph [1].

5.1 Method

To generate new graphs from a probabilistic HRG (PHRG), we start with the special starting nonterminal $H' = S$. From this point, H^* can be generated as follows: (1) Pick any nonterminal A in H'; (2) Find the set of rules $(A \rightarrow R)$ associated with LHS A; (3) Randomly choose one of these rules with probability proportional to its count; (4) Choose an ordering of its external vertices with uniform probability; (5) Replace A in H' with R to create H^*; (6) Replace H' with H^* and repeat until there are no more nonterminal edges.

Fixed-size generation uses algorithms built for PHRGs, which combines $A \rightarrow R$ production rules that are identical and considers the normalized counts as a rule's firing probability. Prior work in fixed-size string generation has been adapted to select a rule firing order π over the rules in the PHRG so that the generated graph is guaranteed to be the requested size [1].

The present work extends this generation algorithm to PSHRGs in order to generate the next timestep of a temporal graph. This process is straightforward but surprisingly effective. Recall that PSHRG rules are of the form $A \rightarrow R_S : R_T$. We can break these rules into their respective parts to create a source-PHRG $A \rightarrow R_S$ and a target-PHRG $A \rightarrow R_T$. These two PHRGs can be later reconciled by attaching an id to each unique rule.

Next we employ the hypergraph parsing algorithm by Chiang et al. to generate a parse tree using the rules present in the source-PHRG [6]. A rule ordering π is constructed from Chiang et al.'s algorithm containing the rule ids from the source-PHRG. Finally, to generate the next graph timestep, we apply the rules from the target-PHRG according to π. Recall that the source and target sides of the PSHRG, and thus the rules from the source-PHRG and target-PHRG, are *synchronized*, *i.e.*, they contain the same LHS and same number of vertices and nonterminal hyperedges on the RHSs; therefore π applied to the target-PHRG rules is guaranteed to "fire" fully and create a prediction of the next timestep.

An example parse-and-generate process is illustrated in Fig. 4 using the rules extracted from $H^{(1)} \cup H^{(2)}$ from Fig. 2 and illustrated in Fig. 3. In this example, we parse $H^{(1)}$ with the source-PHRG. Among the (very) many possibilities, Chiang et al.'s parsing algorithm finds a π of Rules 1, 2, 3, and 4. Therefore, we are guaranteed that executing the source-PHRG over π will yield $H^{(1)}$.

Next, the bottom row of Fig. 4 illustrates the generation of a target graph by executing the target-PHRG over π. This carefully constructed example shows that the target graph is exactly the graph we were expecting to find. However there are several practical considerations that are not shown in this example:

1. The PSHRG extracted rules from the target graph, which encodes test data into the training model.

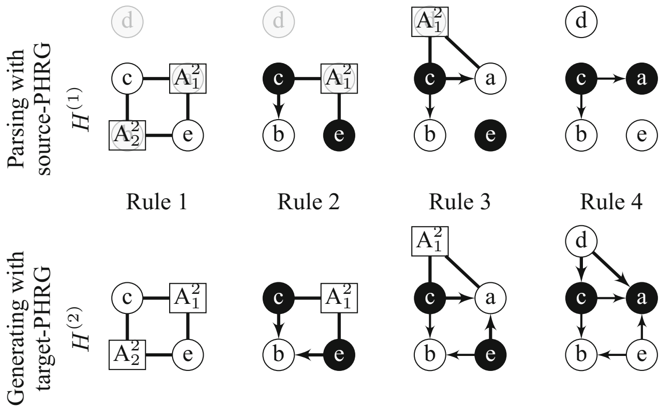


Fig. 4. Parsing graph $H^{(1)}$ with source-PHRG (top). Generating graph $H^{(2)}$ using the target-PHRG (bottom) using the PSHRG from Fig. 3.

2. Given a source-PHRG (from a PSHRG), there are many different ways to parse the same graph. A different π output from the graph parser may generate graphs that are wildly different than the target-graph.

To address practical consideration #1, a proper experimental setup should not extract a PSHRG from the target-graph. Consider once again the running example with $H^{(0)}, \ldots, H^{(4)}$. Our task is to predict $H^{(4)}$ by learning a PSHRG from $H^{(0)}, \ldots, H^{(3)}$ (without $H^{(3)} \cup H^{(4)}$); then, $H^{(3)}$ is parsed with the source-PHRG to generate π. However, it is important to note that *only* the target-PHRG is guaranteed to contain the information from $H^{(3)}$. Therefore, it is possible that the source-PHRG does not contain enough information to parse $H^{(3)}$. If the source-PHRG fails to parse the graph, then we cannot predict a target graph.

Practical consideration #2 has two parts. First, many different parses of the source-graph may be possible given a source-PHRG; therefore many different πs could be created for a single source graph. Chiang et al.'s graph parser will generate an optimal parse (*i.e.*, the parse with the highest probability), but there may be many equally-optimal parses. Second, unlike string parsing, which contains many different terminal and nonterminal labels, *i.e.*, distinct terms and specific grammatical structures, graphs are only labeled with 'vertex' and 'edge'; although nonterminal edges do contain an arity label (latin characters in example figures are for clarity only). The ambiguity contained in the unlabeled vertices and edges creates a large search space for the graph parser. The graph parser, as a result, is unable to efficiently parse large graphs with large PSHRGs.

These practical considerations result in two primary outcomes: (1) the PSHRG extraction method may not parse some graphs, and therefore may not work in some cases; (2) given a successful parse of the graph, a graph generated by firing the target-PHRG over π will only generate an estimate of the target-graph. The next section presents a small study that explores these practical considerations across a variety of temporal graphs.

5.2 Graph Prediction Experiments

Here we test the ability of PSHRG to predict the future evolution of a temporal graph. The methodology is straightforward. Given a temporal graph H with n timesteps, we will extract a PHSRG grammar from $H^{(1)} \dots H^{(n-1)}$. Then we will use the source-PHRG from the extracted PSHRG to parse $H^{(n-1)}$. This will create a rule-firing order π guaranteed to generate $H^{(n-1)}$ from the source-PHRG. Because the source-PHRG and target-PHRG are synchronized in the PSHRG, we execute the synchronized rules in the target-PHRG via π. This will generate an estimate of $H^{(n)}$ we denote as H^*. Our goal is to generate H^* that is similar to the ground-truth graph $H^{(n)}$.

There are several ways to measure the similarity between $H^{(n)}$ and H^*. First we compare the distributions of each graph's in-degree, out-degree, and PageRank scores. Secondly, we employ a relatively new metric called the Graphlet Correlation Distance (GCD) [16]. The GCD measures the frequency of the various graphlets present in each graph, *i.e.*, the number of edges, wedges, triangles, squares, 4-cliques, etc., and compares the graphlet frequencies of each node across two graphs. In all cases, the evaluation metrics are distances; lower values are better.

Existing graph parsing algorithms were built to process small, sentence-sized graphs (in the domain of computational linguistics). Chiang et al.'s graph parsing algorithm operates in polynomial time given a graph of bounded degree [6]; unfortunately, the general (hyper-)graphs that we consider do not have bounded degree, thus the computational complexity of the graph parser on general graphs results in impractical running times. So our test graphs must remain small for now. Specifically, we test with graphs with 5–12 nodes. We repeat each experiment 50 times and plot the mean.

For evaluation we create synthetic graphs using algorithms that generate graphs with well known properties. In addition to the Barabasi-Albert (BA) process discussed in Sect. 3, we also use the Powerlaw-Cluster graph (PLC) and the GN, GNR, and GNC (*i.e.*, "growing graph") processes to generate temporal graphs. Each graph process contains various parameters that can be tuned; we use $k = 2$ and $k = 3$ for BA and include $p = 0.25$ and $p = 0.5$ for the PLC variant, and $p = 0.5$ for GNR; we used the default parameters found in the NetworkX package for the GN and GNC processes.

We compare the results of the PSHRG prediction to a random graph (ER) and to a state-of-the-art temporal model called the Separable Temporal Exponential Random Graph Model (STERGM). As a baseline, the ER "prediction" simply generates a graph with the proper number of nodes and edges. STERGM is a temporal-graph extension of static ERG model, which creates maximum-likelihood estimates for various parameters like the number edges, wedges, triangles, etc. within a given graph. Similar to how we extract PSHRG rules, we train STERGM on the first $n - 1$ timesteps of each graph and simulate the n^{th} timestep.

We begin with a simple test: first we compare the number of edges found in graphs generated by STERGM, ER, and PSHRG models to the held-out graph

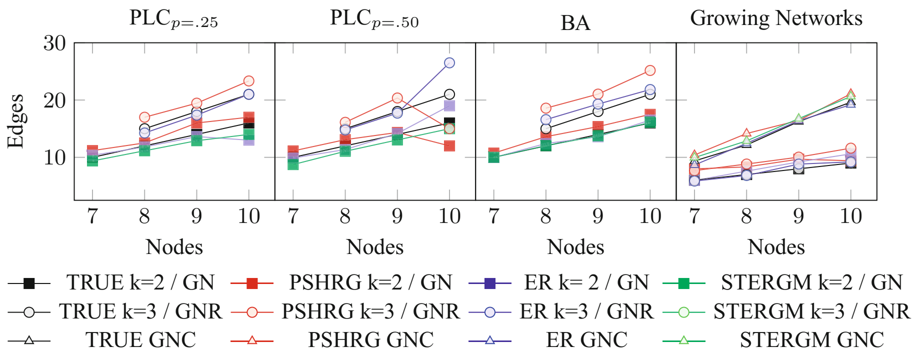


Fig. 5. Number of edges generated by ER, STERGM, and PSHRG graph generators for each of the PLC, BA, and Growing Networks (GN, GNR, GNC) graph processes.

$H^{(n)}$. Figure 5 shows that BA, ER, and the GN-variants perform well compared to the original graph, which is often occluded in Fig. 5 because of the close overlap.

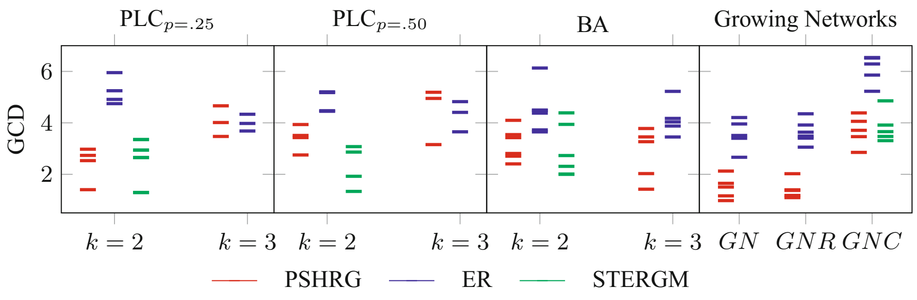


Fig. 6. Graphlet Correlation Distance (GCD). Dashes represent mean GCD scores for various graph sizes (bottom-to-top almost always represents smaller-to-larger graphs), parameters, and models. Lower is better. PSHRG is usually the best. (Color figure online)

Next we use the GCD to measure the correlation between the various graphlets found in the graphs. The GCD can range from $[0, +\infty]$, where the GCD is 0 if the two graphs are isomorphic. Colored dashes in Fig. 6 represent the mean GCD scores comparing the graphs generated by ER, STERGM, and PSHRG against the temporal graphs for various graph sizes. From bottom-to-top, dashes almost always represent increasing number of nodes, *i.e.*, the bottom dash almost always represents a comparison with $n = 5$ and the top dash almost always represents a comparison of $n = 12$. No dashes are missing, but may be occluded due to overlap. However, STERGM results are missing for PLC$_{p=.50}$ with $k = 3$ and BA with $k = 3$ because STERGM failed to produce any graphs either through model degeneracy or other error. The model degeneracy problems

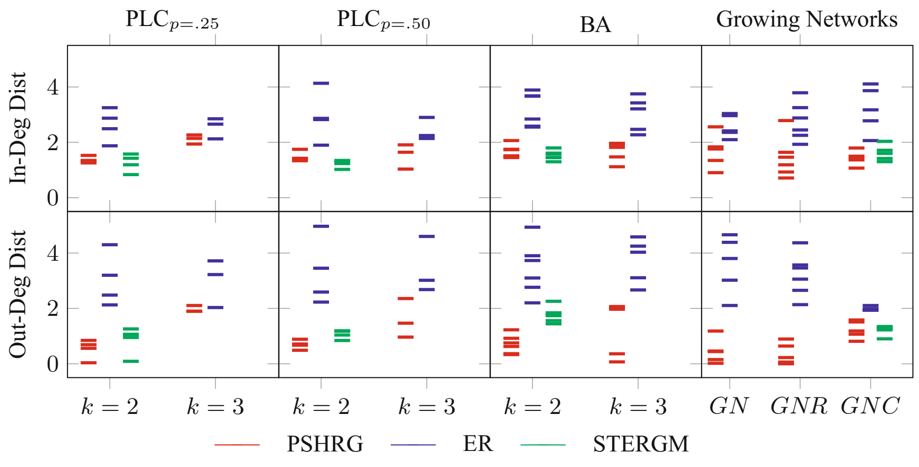


Fig. 7. CVM-test statistics of in-degree (top) and out-degree (bottom) distributions for various graph sizes (bottom-to-top almost always represents smaller-to-larger graphs), parameters, and models. Lower is better. PSHRG and STERGM (when available) results are competitive.

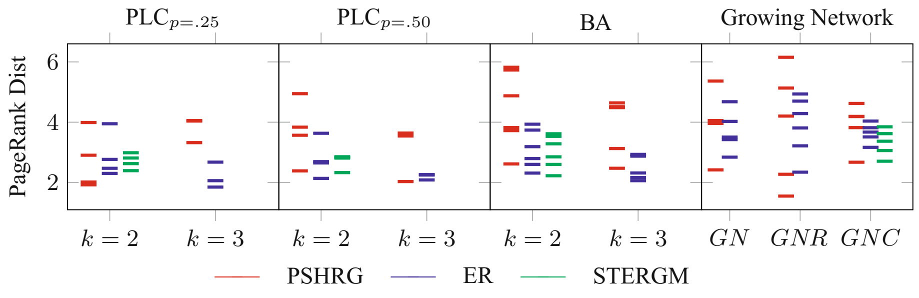


Fig. 8. CVM-test statistics of PageRank distributions for various graph sizes (bottom-to-top almost always represents smaller-to-larger graphs), parameters, and models. Lower is better. STERGM gives more consistently-lower results.

are a frequent problem in ERGM and STERGM; in these cases no prediction is output by the program. In most cases we find the PSHRG results in the lowest GCD scores, and therefore best matches the output created by the graph processes.

Next we compare the degree and PageRank distributions of the ground-truth graph against STERGM, ER, and PSHRG models. We calculate the distance between distributions using the Cramer-von Mises statistic (CVM), which is roughly defined as the absolute distance between two CDFs. In Fig. 7 (top and bottom) dashes represent the mean CVM-test statistic for the indegree distribution and outdegree distribution respectively. These results show that PSHRG is competitive with or outperforms STERGM in predicting the output of the temporal graph process.

The PageRank distances illustrated in Fig. 8 shows that PSHRG results vary wildly. These results are peculiar because of the known correlation between PageRank and indegree, but may also be due to assumptions made in the design of the CVM-test.

6 Conclusions

The present work presents a method to extract synchronous grammar rules from a temporal graph. We find that the synchronous probabilistic hyperedge replacement grammar (PSHRG), with RHSs containing "synchronized" source- and target-PHRGs, is able to clearly and succinctly represent the graph dynamics found in the graph process. We also find that the source-PHRG grammar extracted from the graph can be used to parse a graph. The parse creates a rule-firing ordering that, when applied to the target-PHRG, can generate graphs that are predictive of the future growth of the graph.

The PSHRG model is currently limited in its applicability due to the computational complexity of the graph parser. We are confident that future work in graph parsers will enable PSHRGs to model much larger temporal graphs. The limitation of graph parsing however does not affect the ability of PSHRGs to extract and encode the dynamic properties of the graph. As a result, we expect that PSHRGs may be used to discover previously unknown graph dynamics from large real world networks.

Acknowledgments. We thank Chiemi Matsumoto and Peter Bui for their help with this project. This work is sponsored by grant from the US NSF IIS (16-52492).

References

1. Aguinaga, S., Chiang, D., Weninger, T.: Learning hyperedge replacement graph grammars for graph generation. IEEE Trans. Pattern Anal. Mach. Intell. (2018). https://doi.org/10.1109/TPAMI.2018.2810877
2. Aguinaga, S., Palacios, R., Chiang, D., Weninger, T.: Growing graphs from hyperedge replacement grammars. In: CIKM. ACM (2016)
3. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. science **286**(5439), 509–512 (1999)
4. Bullmore, E., Sporns, O.: Complex brain networks: graph theoretical analysis of structural and functional systems. Nat. Rev. Neurosci. **10**(3), 186–198 (2009)
5. Chiang, D.: An introduction to synchronous grammars. https://www.nd.edu/~dchiang/papers/synchtut.pdf
6. Chiang, D., Andreas, J., Bauer, D., Hermann, K.M., Jones, B., Knight, K.: Parsing graphs with hyperedge replacement grammars. In: Proceedings of ACL, pp. 924–932 (2013)
7. Clarke, B.L.: Theorems on chemical network stability. J. Chem. Phys. **62**(3), 773–775 (1975)

8. Doolittle, W.F., Bapteste, E.: Pattern pluralism and the tree of life hypothesis. Proc. Nat. Acad. Sci. **104**(7), 2043–2049 (2007)
9. Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement graph grammars. Handb. Graph Grammars **1**, 95–162 (1997)
10. Erdos, P., Rényi, A.: On the evolution of random graphs. Bull. Inst. Internat. Stat. **38**(4), 343–347 (1961)
11. Granovetter, M.S.: The strength of weak ties. Am. J. Sociol. **78**(6), 1360–1380 (1973)
12. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0013875
13. Kemp, C., Tenenbaum, J.B.: The discovery of structural form. Proc. Nat. Acad. Sci. **105**(31), 10687–10692 (2008)
14. Kuhn, T.S.: The Structure of Scientific Revolutions. University of Chicago Press, Chicago (2012)
15. Kukluk, J., Holder, L., Cook, D.: Inferring graph grammars by detecting overlap in frequent subgraphs. Int. J. Appl. Math. Comput. Sci. **18**(2), 241–250 (2008)
16. Yaveroğlu, Ö.N., Milenković, T., Pržulj, N.: Proper evaluation of alignment-free network comparison methods. Bioinformatics **31**(16), 2697–2704 (2015)

CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers

Barbara König, Maxime Nederkorn, and Dennis Nolte(✉)

Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Duisburg, Germany
{barbara_koenig,dennis.nolte}@uni-due.de,
maxime.nederkorn@stud.uni-due.de

Abstract. When specifying graph languages via type graphs, cores are a convenient way to minimize the type graph without changing the graph language, i.e. the set of graphs typed over the type graph. However, given a type graph, the problem of finding its core is NP-hard. Using the Tool *CoReS*, we automatically encode all required properties into SAT- and SMT-formulas to iteratively compute cores by employing the corresponding solvers. We obtain runtime results to evaluate and compare the two encoding approaches.

1 Introduction

Type graphs provide a convenient typing mechanism for graphs that has been used in many papers on graph transformation (cf. [5,7]). Type graphs specify legal instance graphs and they can also be seen as describing a graph language, i.e., all graphs that can be mapped into a given type graph via a graph morphism. This view has been worked out in [4], including the study of closure properties and decidability.

As for finite automata the question of minimizing type graphs is relevant: given a type graph, what is the smallest graph that specifies the same language? Such minimal graphs have been studied in graph theory under the name of cores [10]. In [4] we have shown how cores can be exploited for invariant checking based on type graphs. Since the computation of cores is an NP-hard problem, we have used SAT- and SMT-solvers in order to address this problem and found that they perform efficiently and that the SAT-solver outperforms the SMT-solver.

For instance tools that need to check for the existence of morphisms (between several source graphs and a fixed target graph) can profit from the pre-computation of a core graph. By minimizing the target graph, without changing the set of graphs that can be mapped into it, computation times can be reduced to improve the overall performance of the analysis.

For this purpose we wrote the tool *CoReS*, which can be embedded into other graph transformation based tools to efficiently compute core graphs.

2 Preliminaries

We first introduce graphs, graph morphisms and cores. In the context of this paper we use edge-labelled directed graphs. Please note that the results can easily be extended to graphs with additional node labels or even hypergraphs.

Definition 1 (Graph). *Let Λ be a fixed set of edge labels. A Λ-*labeled graph *is a tuple $G = \langle V, E, src, tgt, lab \rangle$, where V is a finite set of nodes, E is a finite set of edges, $src, tgt \colon E \to V$ assign to each edge a source and a target node, and $lab \colon E \to \Lambda$ is a labeling function.*

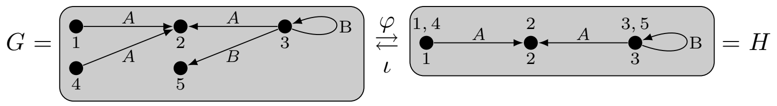
Even though the above definition allows for multi-edges, in the encodings we will not consider two parallel edges with the same label. Please note that a type graph containing two such edges can not be minimal in the sense defined in the following.

Definition 2 (Graph morphism). *Let G, G' be two Λ-labeled graphs. A* graph morphism *$\varphi \colon G \to G'$ consists of two functions $\varphi_V \colon V_G \to V_{G'}$ and $\varphi_E \colon E_G \to E_{G'}$, such that for each edge $e \in E_G$ it holds that $src_{G'}(\varphi_E(e)) = \varphi_V(src_G(e))$, $tgt_{G'}(\varphi_E(e)) = \varphi_V(tgt_G(e))$ and $lab_{G'}(\varphi_E(e)) = lab_G(e)$. If φ is bijective it is called an isomorphism.*

We will often drop the subscripts V, E and write φ instead of φ_V, φ_E. Last, we will revisit the definition of *retracts* and *cores* from [10]. *Cores* are a convenient way to minimize type graphs, as all type graphs specifying equivalent graph languages, share a unique core up to isomorphism [10].

Definition 3 (Retract and core). *A graph H is called a* retract *of a graph G if H is a subgraph of G and in addition there exists a morphism $\varphi \colon G \to H$, which is the identity when restricted to H, i.e., $\varphi_{|H} = id_H$. A graph H is called a* core *of G, written $H = core(G)$, if it is a retract of G and has no proper retracts.*

Example 1. The graph H is a proper retract of G, where the inclusion ι is indicated by the numbers under the nodes, while morphism φ is indicated by the numbers over the nodes:



Since the resulting graph H does have a proper retract, it is not yet a core of G.

Given a graph G and its core H, the set of graphs that can be typed over G respectively H (i.e., the graphs for which there exists a morphism into G respectively H) coincide.

The fact that the computation of the core is NP-hard can be easily seen from a reduction from 3-colourability: Let T be a "triangle" graph with three nodes and edges connecting all pairs of distinct nodes. A graph G is 3-colourable if and only if the core of $G \uplus T$ (the disjoint union of G and T) is T.

3 Encoding Retract Properties

We encode the problem of finding a core into an SMT formula which yields a model in form of the graph morphism $\varphi\colon G \to H$ if a given graph G has a proper retract H. Once we find a retract of G, we iterate this procedure until we arrive at the core.

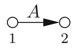
Following the definitions from the previous section, the morphism into the proper retract needs to satisfy the following three conditions:

– *Graph morphism property*: The morphism φ needs to preserve the structure.
– *Subgraph property*: The morphism φ needs to be a non-surjective endomorphism, i.e. it maps the graph to a proper subgraph structure of itself.
– *Retract property*: φ restricted to its image is an identity morphism.

These properties can easily be encoded into an SMT-formula, but are more tedious to specify in a SAT-formula as we will see later. We are using the SMT-LIB2 format [3] where every operator is used in prefix notation.

SMT-LIB2 Encoding: The encoding starts with a specification of the input graph. Following Definition 1 we need to specify the three sets V, E and Λ. Since φ is assumed to be a (non-surjective) endomorphism, we only have to specify one graph G:

`(declare-datatypes () ((V v1 ... vN)))`	$\mid (V = \{v_1, \dots, v_n\})$
`(declare-datatypes () ((E e1 ... eM)))`	$\mid (E = \{e_1, \dots, e_m\})$
`(declare-datatypes () ((L A ...)))`	$\mid (\Lambda = \{A, \dots\})$

Afterwards we encode the *src*, *tgt* and *lab* functions. For instance the graph 

can be encoded in the following way:

`(declare-fun src (E) V)`	$\mid src\colon E \to V$	`(assert (= (src e1) v1))`	$\mid src(e_1) = v_1$
`(declare-fun tgt (E) V)`	$\mid tgt\colon E \to V$	`(assert (= (tgt e1) v2))`	$\mid tgt(e_1) = v_2$
`(declare-fun lab (E) L)`	$\mid lab\colon E \to \lambda$	`(assert (= (lab e1) A))`	$\mid lab(e_1) = A$

Next, we specify the constraints for the searched retract morphism $\varphi\colon G \to H$. The retract morphism $\varphi\colon G \to H$ actually is an endomorphism $\varphi'\colon G \to G$ with $img(\varphi') = H$ and $H \subseteq G$. Therefore, we can use $\varphi_V\colon V \to V$ and $\varphi_E\colon E \to E$.

`(declare-fun vphi (V) V)`	$\mid \varphi_V\colon V \to V$
`(declare-fun ephi (E) E)`	$\mid \varphi_E\colon E \to E$
`(assert (forall ((e E)) (= (src (ephi e)) (vphi (src e)))))`	$\mid src(\varphi_E(e)) = \varphi_V(src(e))$
`(assert (forall ((e E)) (= (tgt (ephi e)) (vphi (tgt e)))))`	$\mid tgt(\varphi_E(e)) = \varphi_V(tgt(e))$
`(assert (forall ((e E)) (= (lab (ephi e)) (lab e))))`	$\mid lab(\varphi_E(e)) = lab(e)$

To achieve a non-surjective morphism, we demand the existence of a node that is not in the image of φ:

```
(assert (exists ((v1 V)) not(exists ((v2 V)) (= v1 (vphi v2)))))
```

Last, we need to specify that for $\varphi\colon G \to H$ the retract property $\varphi_{|H} = id_H$ holds. We rephrase this requirement in the following way:

$$\forall x \in G\Big(\big(\exists y \in G\ (\varphi(y) = x)\big) \implies \varphi(x) = x\Big) \tag{1}$$

Every element in the image of φ is part of the retract and therefore always has to be mapped to itself. This yields the last requirements in our SMT-encoding:

```
(assert (forall ((v1 V)) (=> (exists ((v2 V)) (= v1 (vphi v2))) (= v1 (vphi v1)))))
(assert (forall ((e1 E)) (=> (exists ((e2 E)) (= e1 (ephi e2))) (= e1 (ephi e1)))))
```

This completes the SMT-encoding. We will now encode the same properties using propositional logic only. Therefore we need to additionally specify constraints in order to be able to describe functions.

SAT Encoding: Since in this paper we are working with directed edge-labelled graphs without parallel edges, an edge mapping of a morphism can always be deduced from its corresponding source and target node mappings and vice versa. Therefore, for every possible edge mapping, we encode solely the option to map its related source and target nodes in the corresponding way. Our set of atomic propositions $\mathcal{A}$ has size $|\mathcal{A}| = |V \times V|$. For a pair of nodes $(x, y) \in V \times V$ the atomic proposition $\mathcal{A} \ni \mathtt{A}x\text{-}y \equiv \mathtt{true}$ iff $\varphi_V(x) = y$ holds.

First, we need to specify the function property of the retract morphism, i.e. every node is mapped to exactly one node:

$$\bigwedge_{x \in V} \bigvee_{y \in V} \Big(\mathtt{A}x\text{-}y \wedge \big(\bigwedge_{z \in V \setminus \{y\}} \neg \mathtt{A}x\text{-}z\big)\Big) \qquad \mid \forall x \exists! y\ \varphi_V(x) = y$$

Next, the input graph structure is used to specify valid source and target node mappings. For any edge label $\lambda \in \Lambda$ let $E_\lambda = \{e \in E \mid lab(e) = \lambda\}$ be the equivalence class of edges sharing the same label. For two edges $e, e' \in E_{lab(e)}$ sharing the same label, a label preserving edge mapping $\varphi_E(e) = e'$ directly induces $(\varphi_V(src(e)) = src(e')) \wedge (\varphi_V(tgt(e)) = tgt(e'))$. Therefore, we get:

$$\bigwedge_{e \in E} \bigvee_{e' \in E_{lab(e)}} \Big(\big(\mathtt{A}src(e)\text{-}src(e')\big) \wedge \big(\mathtt{A}tgt(e)\text{-}tgt(e')\big)\Big)$$

To encode that we are searching for a proper subgraph, we specify that there exists one node which has no pre-image in φ and to encode the retract property from Definition 3, we adjust the predicate logic formalization above (1):

$$\bigvee_{x \in V} \Big(\bigwedge_{y \in V} \neg \mathtt{A}y\text{-}x\Big) \quad \mid \exists x \forall y\ \varphi(y) \neq x \quad \Bigg| \quad \bigwedge_{x \in V} \Big(\big(\bigvee_{y \in V} \mathtt{A}y\text{-}x\big) \Rightarrow \mathtt{A}x\text{-}x\Big) \quad \mid \varphi_{|H} = id_H$$

While the SMT encoding is less ad-hoc and easier to read and understand for a human, the slightly larger SAT encoding yields better runtime results when searching for a core graph.

4 Benchmarks and Conclusion

To evaluate the encodings, a tool named *CoReS* (Computation of Retracts encoded as SMT/SAT) has been implemented in Python. The tool, a short user manual and some examples are available on GitHub [1]. *CoReS* provides both a graphical user interface and a command-line interface. The tool generates SMT/SAT formulas from graphs which are passed to the corresponding solvers. *CoReS* can use any SMT-solver which supports the SMT-LIB2 format. For our set of examples we employed the SMT-solver Z3 [6] and the SAT-solver Limboole [2]. The results are parsed as soon as the SAT/SMT-solver terminates and produces a model for the formula. If there is a model, it yields a valid graph morphism into a retract and therefore *CoReS* computes the retract and restarts the encoding/solving process on the smaller graph until the solvers can not find a model anymore. In that case, the last input graph is the searched core.

We ran *CoReS* on randomly generated graphs which were generated using the model introduced in [9] which is closely related to the well known Erdős-Rényi model [8]. We used a Windows workstation with 4.5 GHz, i7-7700k CPU and 16GB RAM to compute average runtime results in seconds. We generated 125 random graphs each for a fixed number of nodes $|V|$, a fixed number of edge labels $|\Lambda|$ and a fixed probability ρ for the existence of an edge and report average runtimes. Our runtime results are the following:

		$\rho \cdot \lvert V\rvert \cdot \lvert\Lambda\rvert$									
		0.5		0.8		1.0		1.2		1.5	
|V|	$\lvert\Lambda\rvert$	SAT	SMT	SAT	SMT	SAT	SMT	SAT	SMT	SAT	SMT
16	1	.075	.116	.078	.344	.078	.733	.071	1.17	.070	3.01
	2	.067	.155	.096	.463	.080	1.12	.079	2.11	.078	4.21
	3	.063	.172	.100	.548	.074	1.14	.071	2.02	.073	4.09
32	1	.301	.620	.306	4.58	.396	12.4	.424	27.4	.500	67.5
	2	.389	1.08	.407	7.27	.415	14.9	.447	37.6	.450	121
	3	.322	1.52	.383	5.27	.365	19.3	.391	40.3	.382	110

Additional tests have shown that the SAT-based approach can be used on graphs consisting of 200 nodes, to compute cores with an average runtime of 556s. Due to the preprocessing of *CoReS*, the runtime of the SAT encoding becomes slightly better with more edge labels. For larger graphs consisting of more than 200 nodes, Limboole often crashed without providing any model.

Overall our SAT-based approach outperforms our straightforward SMT-based encoding for the computation of core graphs. It has to be taken into account that for the SAT-based approach we used preprocessing to reduce the number of atomic propositions. This preprocessing step, which only adds atomic propositions which conform to the structure preservation of a morphism, in combination with the already smaller exploration space of a SAT formula compared to an SMT formula, can explain the runtime results.

References

1. CoReS: https://github.com/mnederkorn/CoReS. Accessed 22 Feb 2018
2. Limboole: http://fmv.jku.at/limboole/index.html. Accessed 19 Feb 2018
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard - Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT 2010), Edinburgh, Scotland, July 2010
4. Corradini, A., König, B., Nolte, D.: Specifying graph languages with type graphs. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 73–89. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_5
5. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae **26**(3/4), 241–265 (1996)
6. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
8. Erdős, P., Rényi, A.: On random graphs I. Publicationes Mathematicae (Debrecen) **6**, 290–297 (1959)
9. Gilbert, E.N.: Random graphs. Ann. Math. Statist. **30**(4), 1141–1144 (1959)
10. Nešetřil, J., Tardif, C.: Duality theorems for finite structures (characterising gaps and good characterisations). J. Comb. Theor. Ser. B **80**, 80–97 (2000)

Graph Transformation Formalisms

Graph Surfing by Reaction Systems

Hans-Jörg Kreowski[1(✉)] and Grzegorz Rozenberg[2,3]

[1] Department of Computer Science, University of Bremen, Bibliothekstr. 5, 28359 Bremen, Germany
kreo@informatik.uni-bremen.de

[2] Leiden Institute of Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
g.rozenberg@liacs.leidenuniv.nl

[3] Department of Computer Science, University of Colorado, Boulder, CO 80309-0347, USA

Abstract. In this paper, we introduce graph-based reaction systems as a generalization of set-based reaction systems, a novel and well-investigated model of interactive computation. Graph-based reaction systems allow us to introduce a novel methodology for graph transformation, which is not based on the traditional "cut, add, and paste" approach, but rather on moving within a "universe" graph B (surfing on B) from a subgraph of B to a subgraph of B, creating subgraph trajectories within B. We illustrate this approach by small case studies: simulating finite state automata, implementing a shortest paths algorithm, and simulating cellular automata.

1 Introduction

The goal of this paper is to introduce a novel framework for graph transformation. It results from extending set-based reaction systems to graph-based reaction systems. We introduce the main notions and illustrate the framework by considering three case studies: a simulation of finite state automata, a parallel shortest-path algorithm, and a simulation of cellular automata.

The concept of reaction systems was introduced about ten years ago (see [9]) and has been intensely studied since then (see, e.g., [3,5–8,10,11,14,21,22]). It was inspired by the functioning of living cells and the original motivation was to provide a formal framework for the modeling of biochemical processes taking place in the living cell. It turned out to be a novel and actively investigated paradigm of interactive computation interesting also for modeling of information processing beyond biochemistry. The original notion of reaction systems and their interactive processes is purely set-theoretic. In this paper, we enhance the framework by a graph-based level of description so that graph-related problems and models can be handled in a natural way.

A graph-based reaction system consists of a finite background graph and a set of reactions. A reaction has three components: a reactant graph, an inhibitor which is a pair consisting of a set of nodes and a set of edges, and a product

L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 45–62, 2018.
https://doi.org/10.1007/978-3-319-92991-0_4

graph. Reactant and product are subgraphs of the background graph while the inhibitor consists of a subset of the set of nodes together with a subset of the set of edges of the background graph. Reactions specify basic transformations of states which are subgraphs of the background graph. A reaction is enabled by a state if the reactant is a subgraph of the state and no nodes or edges of the inhibitor are present in the state. The dynamics of a reaction system is defined by discrete interactive processes. In one step of a process, all enabled reactions are applied simultaneously and the union of their product graphs forms the successor state. This means that the processing is deterministic and parallel. Since the successor state T' of a state T consists of the product graphs of all reactions enabled on T, a node or an edge is sustained, i.e., it is also present in T', only if it is produced by one of the enabled reactions. In this sense, each consecutive graph produced by an interactive process is a "new" graph.

Most approaches to rule-based graph transformation consider abstract graphs meaning that the graph resulting from a rule application is uniquely constructed up to isomorphism. In our framework, reactions are applied to subgraphs of a concrete background graph yielding subgraphs. Therefore, the processing of graphs in graph-based reaction systems may be seen as a kind of surfing in analogy to the surfing on the Internet by following links, where our "websites" are subgraphs and our "links" are reactions.

The paper is organized as follows. After the preliminaries recalling the basic notions and notations of graphs in Sect. 2, Sect. 3 presents the notion of graph-based reaction systems and their interactive processes. In Sect. 4 finite-state automata are simulated by reaction systems operating on the state graphs of the automata. The computational potential of graph-based reaction systems is illustrated in Sect. 5 where a parallel shortest-path algorithm is considered. In Sect. 6, the well-known computational model of cellular automata is simulated by graph-based reaction systems. Discussion in Sect. 7 concludes the paper.

2 Preliminaries

In this section, the basic notions and notations to be used in this paper concerning graphs are recalled.

A (simple, directed, and edge-labeled) *graph* is a system $G = (V, \Sigma, E)$ where V is a finite set of *nodes*, Σ is a finite set of *edge labels*, and $E\ V \times V \times \Sigma$ is a set of *edges*.

For an edge $e = (v, v', x)$, v is called the *source* of e, v' the *target* of e, and x the *label* of e. An edge e with label x is an *x-edge*, and if source and target are equal, then it is also called an *x-loop* or a *loop*. The components V, Σ, and E of G are also denoted by V_G, Σ_G, and E_G respectively. The set of all graphs is denoted by $\mathcal{G}$. We reserve a special label $*$ which is always included in each label alphabet Σ. An edge labeled by $*$ can be considered as unlabeled. The label $*$ is used only for this purpose and to simplify the notation, it is omitted in drawings when used.

For graphs G and H such that $\Sigma_H = \Sigma_G$, H is a *subgraph* of G, denoted by H *sub* G, if $V_H \subseteq V_G$, and $E_H \subseteq E_G$. The inclusion, the union, and the intersection of subgraphs are defined componentwise. Thus, for subgraphs H and H' of a graph G with $\Sigma = \Sigma_H = \Sigma_{H'}$, $H \cup H' = (V_H \cup V_{H'}, \Sigma, E_H \cup E_{H'})$ and $H \cap H' = (V_H \cap V_{H'}, \Sigma, E_H \cap E_{H'})$. Obviously, the union and the intersection of subgraphs of G are subgraphs of G.

For a graph $G = (V, \Sigma, E)$, an ordered pair (X, Y) such that $X \subseteq V$ and $Y \subseteq E$ is called a *selector* of G. Note that X and Y are "independent" of each other: X may contain nodes that are neither sources nor targets of edges in Y and Y may contain edges such that their sources and targets are not in X.

The inclusion, union, and intersection of selectors are defined componentwise, i.e., for selectors $P = (X, Y)$ and $P' = (X', Y')$, $P \subseteq P'$ if $X \subseteq X'$ and $Y \subseteq Y'$, $P \cup P' = (X \cup X', Y \cup Y')$, and $P \cap P' = (X \cap X', Y \cap Y')$ respectively.

Subgraphs and selectors are closely related to each other. Given a selector $P = (X, Y)$ of a graph G, P induces a subgraph of G, denoted by $ind(P)$, by adding all sources and targets of edges of Y to X, i.e., $ind(P) = (X \cup \{v, v' \mid (v, v', x) \in Y\}, \Sigma, Y)$. This subgraph induction preserves inclusion, union, and intersection, i.e., for all selectors P and P' of G, $P \subseteq P'$ implies $ind(P)\ sub\ ind(P')$, $ind(P \cup P') = ind(P) \cup ind(P')$, and $ind(P \cap P') = ind(P) \cap ind(P')$.

On the other hand, given a subgraph H of a graph G, its selector is $U(H) = (V_H, E_H)$. The operation U is called *extraction.* Obviously, extraction preserves inclusion, union and intersection, i.e., for all subgraphs H and H' of G, $H\ sub\ H'$ implies $U(H) \subseteq U(H')$, $U(H \cup H') = U(H) \cup U(H')$, and $U(H \cap H') = U(H) \cap U(H')$. It is easy to see that the induced subgraph of the extraction of a subgraph yields the subgraph back, i.e., for all subgraphs H of G, $ind(U(H)) = H$ The other way around, the selector of a subgraph induced by some selector P contains P, i.e. $P \subseteq U(ind(P))$. As a consequence, we get for each selector P of a graph G and for each subgraph H of G that $P \subseteq U(H)$ if and only if $ind(P)\ sub\ H$.

Some special kinds of subgraphs are used in this paper. Given a graph G with $\Sigma = \Sigma_G$, a node $v \in V_G$ induces a subgraph $Out(v) = (\{v\} \cup \{v' \mid (v, v', x) \in E_G, v \neq v', x \in \Sigma\}, \Sigma, \{(v, v', x) \in E_G \mid v' \in V_G, v \neq v', x \in \Sigma\}$, called the *out-neighborhood* of v. Its set of nodes consists of v and its out-neighbors and its set of edges consists of the outgoing edges of v. Moreover, a subgraph consisting of an x-loop and the attached node v is denoted by $loop(v, x)$. The *empty graph* $(\emptyset, \Sigma, \emptyset)$ is denoted by $\emptyset$.

Finally, we use $\mathbb{N}$ to denote the set of natural numbers (including 0) and $\mathbb{N}^+$ to denote the set of positive integers.

3 Reaction Systems on Graphs

In this section, the basic notions and notations of reaction systems on graphs are introduced. While the original notion of reaction systems is purely set-theoretic, the concept is carried over to graphs and subgraphs instead of sets and subsets. Let us first recall the original notions concerning reaction systems.

A *set-based reaction system* $\mathcal{A}$ consists of a finite *background set* S and a finite set of *reactions* A each of which is of the form $b = (X, Y, Z)$, where X, Y, Z

are non-empty subsets of S such that $X \cap Y = \emptyset$. The components X, Y, Z are called the sets of *reactants*, *inhibitors*, and *products* of b, respectively. *States* are subsets of the background set S. A reaction $b = (X, Y, Z)$ is *enabled* by a state T if $X \subseteq T$ and $Y \cap T = \emptyset$. The application of b to T yields the *successor state* which is the union of the products of all enabled reactions.

In graph-based reaction systems, the background set is replaced by a background graph, and the states and the components of reactions are replaced by subgraphs with one exception. We use both nodes and edges as inhibitors, where we may use edges as inhibitors without necessarily forbidding their sources and targets. Therefore, we define inhibitors as selectors which leads to the following notion of a (graph-based) reaction.

Definition 1 (reaction). Let B be a graph. A *reaction* over B is a triple $b = (R, I, P)$ where R and P are non-empty subgraphs of B and I is a selector of B such that $I \cap U(R) = \emptyset$. R is called *reactant graph*, I is called *inhibitor*, and P is called *product graph.* The reaction b is called *uninhibited* if the inhibitor is empty, i.e. $I = (\emptyset, \emptyset)$.

We use the notations R_b, I_b and P_b to denote R, I and P, respectively.

Next we formalize the application of a reaction and of a set of reactions to a graph (which is a subgraph of the "universe" graph B).

Definition 2 (state, enabled reaction, result). Let B be a graph.

1. A *state* of B is a subgraph of B.
2. A reaction $b = (R, I, P)$ over B is *enabled* by a state T, denoted by $en_b(T)$, if $R \mathit{\ sub\ } T$ and $I \cap U(T) = \emptyset$.
3. The *result* of a reaction b on the state T is $res_b(T) = P_b$ if $en_b(T)$ and $res_b(T) = \emptyset$ otherwise.
4. The *result* of a set of reactions A over B on a state T is $res_A(T) = \bigcup_{b \in A} res_b(T)$.

It is important to notice (since the union of subgraphs of B is a subgraph of B) the result of A on T is a subgraph of B.

We are ready now to define the notion of a graph-based reaction system.

Definition 3 (graph-based reaction system). A *graph-based reaction system* is a pair $\mathcal{A} = (B, A)$ where B is a graph, called the *background graph* of $\mathcal{A}$, and A is a set of reactions over B. For a state T of B, also called a *state* of $\mathcal{A}$, the *result* of $\mathcal{A}$ on T is the result of A on T, i.e., $res_{\mathcal{A}}(T) = res_A(T)$.

Thus a graph-based reaction system $\mathcal{A} = (B, A)$ is basically a set of reactions A. In specifying $\mathcal{A}$, we also specify its background graph B which is a sort of a "universe" of $\mathcal{A}$, as for each reaction $b \in A$ both R_b and P_b are subgraphs of B, and $I_b = (X, Y)$ is such that X is a subset of the set of nodes of B and Y is a subset of the set of edges of B.

We are interested in dynamic processes associated with $\mathcal{A}$ which determine graph transformations specified by $\mathcal{A}$. Such processes are defined as follows.

Definition 4 (interactive process). Let $\mathcal{A} = (B, A)$ be a graph-based reaction system.

1. An *interactive process* in $\mathcal{A}$ is an ordered pair $\pi = (\gamma, \delta)$ such that γ, δ are sequences of subgraphs of B, $\gamma = C_0, \ldots, C_n$ and $\delta = D_0, \ldots, D_n$ for some $n \in I\!N^+$ where $D_i = res_{\mathcal{A}}(C_{i-1} \cup D_{i-1})$ for $i = 1, \ldots, n$. The sequence γ is the *context sequence*, the sequence δ is the *result sequence*, and the sequence $\tau = T_0, \ldots, T_n$ with $T_i = C_i \cup D_i$ for $i = 0, \ldots, n$ is the *state sequence.*
2. If the context sequence γ is such that $C_i = \emptyset$ for $i = 0, \ldots, n$, then π is *context-independent.*

Since the successor state T' of a state T is the union of product graphs of all reactions enabled on T, a node or an edge of T is sustained, i.e., it is also present in T', only if it is produced by one of the enabled reactions. In this sense, each consecutive graph generated by an interactive process is a "new" graph.

When π is context-independent, then $T_i = D_i$ for $i = 0, \ldots, n$ meaning that the result sequence and state sequence coincide and that the state sequence describes the whole process determined by its initial state $T_0 = D_0$. Therefore, whenever context-independent processes are considered, we will just focus on their state sequences.

The notion of graph-based reaction systems is chosen in such a way that set-based reaction systems (where also uninhibited reactions are allowed) can be seen as the special case where the background graphs are discrete (one can forget then the empty set of edges and the set of edge labels, and the remaining set of nodes forms the background set of a set-based reaction system).

The choice of selectors (rather than subgraphs) as inhibitors is significant and desirable. Having a subgraph as inhibitor would require that forbidding an edge would imply also forbidding its source and target nodes, which is too restrictive. On the other hand, choosing reactants and products as subgraphs seems to be a good choice. As pointed out in the preliminaries, a selector is included in the selector extracted from a subgraph if and only if its induced subgraph is included in the given subgraph. Therefore, it does not make any difference whether reactants are chosen as subgraphs or selectors. However, if one would define products as selectors, then the union of the products of all enabled reactions would not always be a subgraph so that one would have to define a successor state as the induced subgraph of this union. But this is equal to the union of the induced subgraphs of the involved selectors. Therefore, by using subgraphs to define products we avoid an additional subgraph induction.

For an interactive process π of a graph-based reaction system $\mathcal{A}$ over B, its state sequence $\tau = T_0, \ldots, T_n$ is a sequence of subgraphs of B. Hence, the consecutive steps of π define a trajectory $T_0, \ldots, T_n$ of subgraphs of B. In other words, following π through its state sequence, we *surf* on B moving from a subgraph of B to a subgraph of B. Therefore, π can be seen as a process of consecutive graph transformations beginning with T_0 and leading to T_n. In this way, we deal here with graph transformations determined by graph surfing.

4 Simulating Finite State Automata

To demonstrate how interactive processes work that are not context-independent, finite state automata are transformed into graph-based reaction systems such that the recognition of strings is modeled by certain interactive processes that run on the state graphs of the automata.

Let $\mathcal{F} = (Q, \Sigma, \phi, s_0, F)$ be a *finite state automaton* with the set of *states* Q, the set of *input symbols* Σ, the *state transition function* $\phi : Q \times \Sigma \to Q$, the *initial state* $s_0 \in Q$, and the set of *final states* $F \subseteq Q$. Then the corresponding graph-based reaction system $\mathcal{A}(\mathcal{F}) = (B(\mathcal{F}), A(\mathcal{F}))$ is constructed as follows.

The background graph extends the state graph of $\mathcal{F}$ by a *run*-loop at each node and an extra node with a loop for each input symbol. Formally, $B(\mathcal{F}) = (Q \cup \{input\}, \Sigma \cup \{init, fin, run, *\}, E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5)$ with $E_1 = \{(s, \phi(s,x), x) \mid s \in Q, x \in \Sigma\}$, $E_2 = \{(s_0, s_0, init)\}$, $E_3 = \{(s'', s'', fin) \mid s'' \in F\}$, $E_4 = \{(s, s, run) \mid s \in Q\}$, and $E_5 = \{(input, input, x) \mid x \in \Sigma\}$. The subgraph resulting from removing the *run*-loops and the node *input* and its loops is the *state graph* $gr(\mathcal{F})$ of $\mathcal{F}$, while $gr(\mathcal{F})^-$ denotes the state graph without the *init*-loop. The node *input* with its loops represents the input alphabet. The *run*-loops are used in the interactive processes.

The set of reactions $A(\mathcal{F})$ consists of seven types of uninhibited reactions given by drawings below.

1. replacing the *init*-loop by the *run*-loop: $\Big(\begin{tikzcd} s0 \arrow[loop left, "init"] \end{tikzcd}, (\emptyset,\emptyset), \begin{tikzcd} s0 \arrow[loop left, "run"] \end{tikzcd}\Big)$,

2. sustaining nodes of the state graph: $\big(s, (\emptyset,\emptyset), s\big)$ for $s \in Q$,

3. sustaining transition edges: $\Big(\begin{tikzcd} s \arrow[r, "x"] & \phi(s,x) \end{tikzcd}, (\emptyset,\emptyset), \begin{tikzcd} s \arrow[r, "x"] & \phi(s,x) \end{tikzcd}\Big)$ for $s \in Q, x \in \Sigma$ with $s \neq \phi(s,x)$,

4. sustaining transition loops: $\Big(\begin{tikzcd} s \arrow[loop left, "x"] \end{tikzcd}, (\emptyset,\emptyset), \begin{tikzcd} s \arrow[loop left, "x"] \end{tikzcd}\Big)$ for $s \in Q, x \in \Sigma$ with $s = \phi(s,x)$,

5. sustaining final-state loops: $\Big(\begin{tikzcd} s'' \arrow[loop left, "fin"] \end{tikzcd}, (\emptyset,\emptyset), \begin{tikzcd} s'' \arrow[loop left, "fin"] \end{tikzcd}\Big)$ for $s'' \in F$,

6. moving along transition edges due to input symbol:

$$\Big(\begin{tikzcd} s \arrow[loop left, "run"] \arrow[r, "x"] & \phi(s,x) & input \arrow[loop left, "x"] \end{tikzcd}, (\emptyset,\emptyset), \begin{tikzcd} \phi(s,x) \arrow[loop left, "run"] \end{tikzcd}\Big)$$

for $s \in Q, x \in \Sigma$ with $s \neq \phi(s,x)$,

7. staying at transition loops due to input symbol:

$$\Big(\begin{tikzcd} s \arrow[loop left, "run"] \arrow[loop right, "x"] & input \arrow[loop left, "x"] \end{tikzcd}, (\emptyset,\emptyset), \begin{tikzcd} s \arrow[loop left, "run"] \end{tikzcd}\Big)$$

for $s \in Q, x \in \Sigma$ with $s = \phi(s,x)$.

Let us consider now interactive processes given by context sequences of the form $cs(x_1 \cdots x_n) = \emptyset, loop(input, x_1), \ldots, loop(input, x_n), \emptyset$ for $n \in I\!N^+$ and $x_i \in \Sigma$ for $i = 1, \ldots, n$. Moreover, $cs(\lambda)$ for the empty string $\lambda \in \Sigma^*$ denotes the context sequence $\emptyset, \emptyset$. Obviously, this establishes a one-to-one correspondence between Σ^* and $cs(\Sigma^*) = \{cs(w) \mid w \in \Sigma^*\}$.

Let $\pi(x_1 \cdots x_n)$ denote the interactive process that has $cs(x_1 \cdots x_n)$, for $n \in \mathbb{N}^+$ and $x_i \in \Sigma$ for $i = 1, \ldots, n$, as its context sequence and $D_0, \ldots, D_{n+1}$ with $D_0 = gr(\mathcal{F})$ as its result sequence. In all reaction steps, the state graph without the *init*-loop $gr(\mathcal{F})^-$ is sustained due to the sustaining reactions of type 2 to 5, i.e., $gr(\mathcal{F})^-$ is a subgraph of D_i for $i = 1, \ldots, n+1$.

In the first step, the first reaction is enabled replacing the *init*-loop by a *run*-loop. From then on this reaction is not enabled ever again because the *init*-loop is never recreated. In each consecutive reaction step $i+1$ for $i = 1, \ldots, n$, the result graph D_{i+1} has a single *run*-loop at some node s_i and is accompanied by the context graph $loop(input, x_i)$ so that exactly one of the *run*-reactions is enabled putting the *run*-loop at node $\phi(s_i, x_i)$.

In the case of the interactive process $\pi(\lambda)$ that has $cs(\lambda) = \emptyset, \emptyset$ as its context sequence and D_0, D_1 as its result sequence with $D_0 = gr(\mathcal{F})$, one gets $D_1 = gr(\mathcal{F})^- \cup loop(s_0, run)$, as only the first reaction is enabled by $gr(\mathcal{F})$ and its application replaces the *init*-loop by the *run*-loop.

To illustrate how these interactive processes look like, let us consider the automaton $\mathcal{F}_{xy^*x}$ with the state graph given by the upper right graph D_0 in Fig. 1 and the context sequence

$$cs(xyyx) = \emptyset, loop(input, x), loop(input, y), loop(input, y), loop(input, x), \emptyset.$$

The resulting interactive process is given in Fig. 1.

We will demonstrate now that recognition processes in $\mathcal{F}$ correspond to specific processes in $\mathcal{A}(\mathcal{F})$. This is stated in the following lemma.

Lemma 1. Let $\mathcal{F} = (Q, \Sigma, \phi, s_0, F)$ be a finite state automaton. Let $cs(w)$ be the context sequence of $w \in \Sigma^*$. Let $w = uv$ for some $u, v \in \Sigma^*$ with $length(u) = i$. Let $D(u)$ be the $(i+1)$-th result graph of the interactive process $\pi(w)$. Then $D(u) = gr(\mathcal{F})^- \cup loop(\phi^*(s_0, u), run)$.

Proof. The proof is by induction on i.

Induction base: Let $i = 0$, i.e. $u = \lambda$. Then $D(\lambda)$ is the result of the first reaction step. Therefore, $D(\lambda) = gr(\mathcal{F})^- \cup loop(s_0, run)$ as pointed out above. This proves the statement for $i = 0$ because $s_0 = \phi^*(s_0, \lambda)$ by definition of ϕ^*.

Induction step: Consider $w = uxv$ with $length(u) = i$ and $length(ux) = i+1$. By the induction hypothesis, one gets $D(u) = gr(\mathcal{F})^- \cup loop(\phi^*(s_0, u), run)$. And the successor result graph is $D(ux) = gr(\mathcal{F})^- \cup loop(\phi(\phi^*(s_0, u), x), run)$ as described above. This proves the statement for $i+1$, because $\phi^*(s_0, ux) = \phi(\phi^*(s_0, u), x)$ by definition of ϕ^*. This completes the proof.

The lemma allows one to relate the recognition of strings by finite state automata to certain interactive processes. Let $\mathcal{F} = (Q, \Sigma, \phi, s_0, F)$ be a finite state automaton. An interactive process given by the context sequence $cs(w)$ can be considered as accepting $w \in \Sigma^*$ if the final result graph $D(w)$ contains the subgraph $loop(s'', run) \cup loop(s'', fin)$ for some $s'' \in F$, meaning that the *run*-loop ends up in a final state. Accordingly, the *accepted language* of $\mathcal{A}(\mathcal{F})$ can be

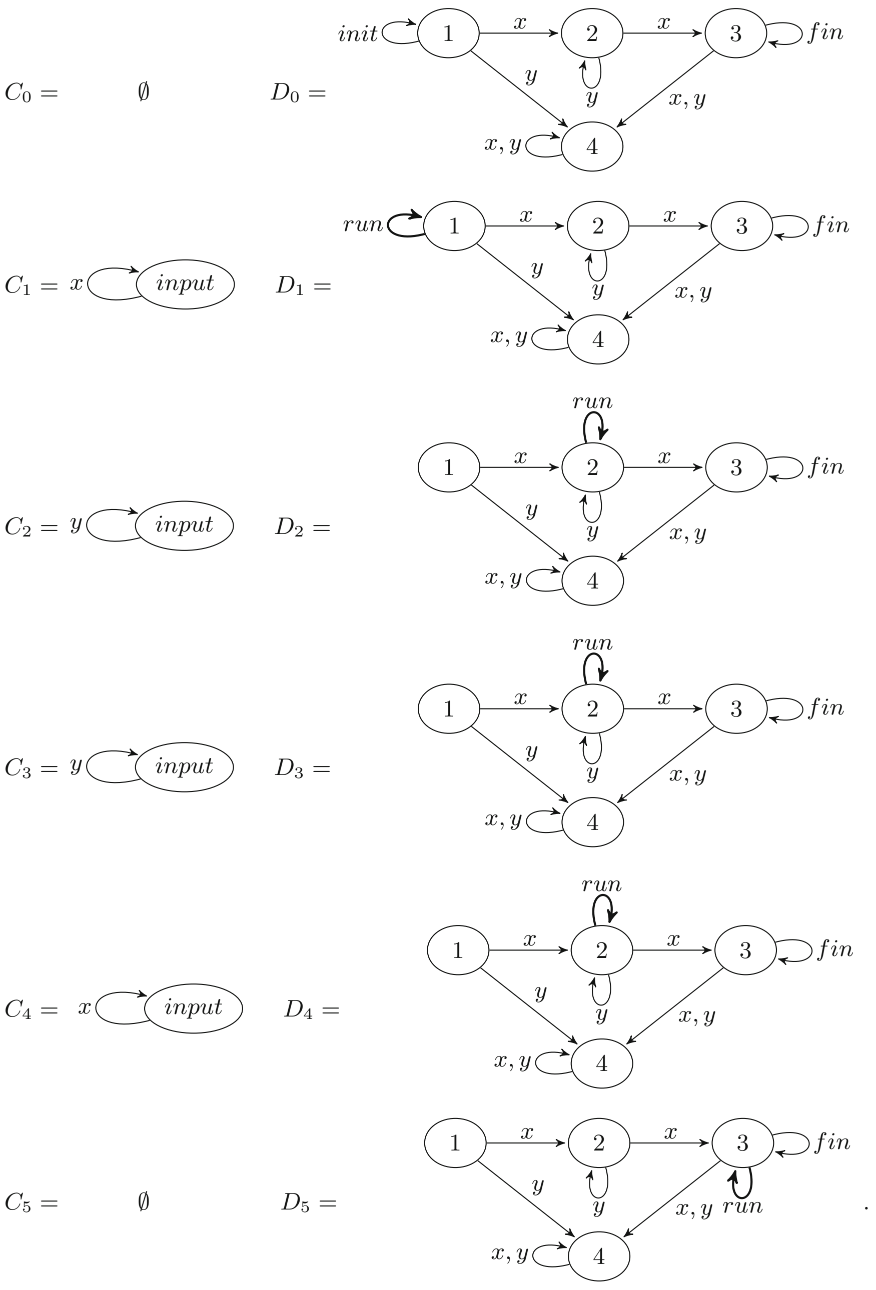


Fig. 1. An interactive-process sample

defined as $L(\mathcal{A}(\mathcal{F})) = \{w \in \Sigma^* \mid loop(s'', run) \cup loop(s'', fin)\ sub\ D(w), s'' \in F\}$.

It follows then by Lemma 1 that the recognized language of a finite state automaton and the accepted language of the corresponding graph-based reaction system coincide.

Theorem 1. Let $\mathcal{F}$ be a finite state automaton and $\mathcal{A}(\mathcal{F})$ the corresponding graph-based reaction system. Then $L(\mathcal{F}) = L(\mathcal{A}(\mathcal{F}))$.

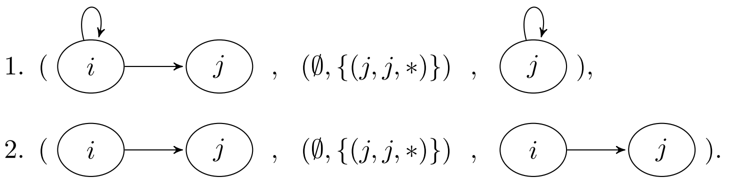
Proof. By definition, $w \in L(\mathcal{F})$ implies $\phi^*(s_0, w) \in F$. By Lemma 1, $D(w)$ contains the subgraph $loop(\phi^*(s_0, w), run)$ Therefore, $w \in L(\mathcal{A}(\mathcal{F}))$ if and only if the node $\phi^*(s_0.w)$ carries a fin-loop. This implies that $\phi^*(s_0, w) \in F$, which completes the proof.

5 Computing Shortest Paths

Graph-based reaction systems can be used to check graph properties and to compute functions on graphs. This is demonstrated by the family of graph-based reaction systems $SHORT(n) = (B(n), A(n))$ for $n \in \mathbb{N}^+$ that compute the lengths of shortest paths from an initial node.

The background graph is the complete unlabeled directed graph with n nodes $B(n) = ([n], \{*\}, [n] \times [n] \times \{*\})$ with the set of nodes $[n] = \{1, \ldots, n\}$ and all pairs of nodes as unlabeled edges, i.e. $*$ is the label of all edges.

The set $A(n)$ of reactions consists of two reactions for each pair $(i, j) \in [n] \times [n]$ of nodes with $i \neq j$:



The first reaction moves a loop from the source of an edge to its target provided that there is no loop on the target node. The second reaction sustains an edge together with its source and its target provided that the target carries no loop. Thus edges incoming to nodes with loops are not sustained. Loops are never sustained. A node is sustained only if it is the source or the target of a sustained edge. These reactions are a good illustration of the usefulness of using selectors (rather than subgraphs) as inhibitors.

To see what happens in context-independent interactive processes of this reaction system, consider the example in Fig. 2 where the process is represented by its state sequence. It starts with a state that is equipped with a single loop attached to node 2. In the resulting state of the first reaction step, the nodes 3, 4 and 5 have loops, after the second step, the nodes 6 and 7 have loops, after the third step, the nodes 8 and 9 have loops, and after the forth step, the node

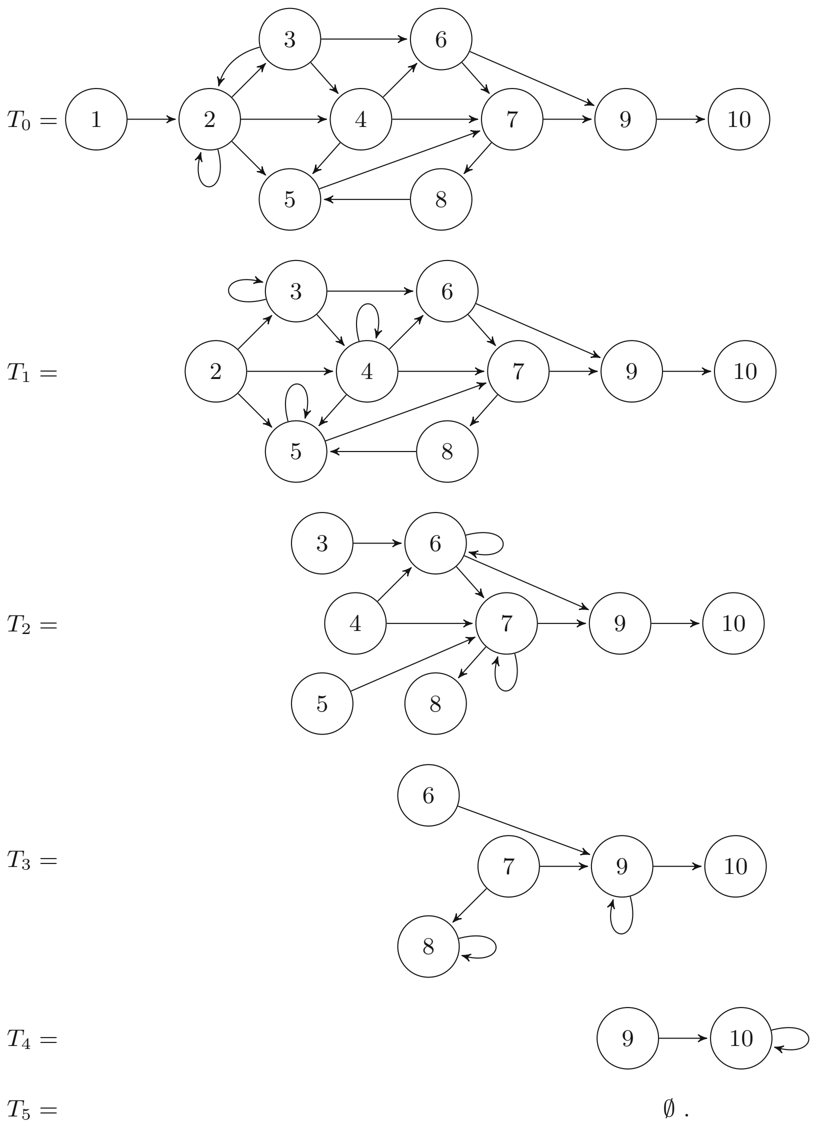


Fig. 2. The state sequence of a context-independent interactive process of *SHORT(10)*

10 has a loop. In other words, the number of steps in which a node gets a loop corresponds to the length of the shortest path from node 2 to this node.

In general, one can prove the following result.

Theorem 2. Let $\tau = T_0, \ldots, T_m$ for some $m \in I\!N^+$ be the state sequence of a context-independent interactive process in the reaction system $SHORT(n)$ for some $n \in I\!N^+$, where T_0 is a state with a single loop attached to node i_0. Then the following holds: A node i has a loop in state T_k for some k if and only if the length of a shortest path from i_0 to i has length k.

Proof. By induction on k.

Induction base for $k = 0$: The node i_0 is the only node with a loop in T_0, and the empty path (the path of length 0), from i_0 to i_0 is the shortest path of length 0.

Induction hypothesis: The statement holds for all $l \leq k - 1$ for some $k \geq 1$.

Induction step: Let i be a node of T_k with a loop. Then it got its loop from a node j in T_{k-1} with a loop by a reaction of type 1 that moves the loop from j to i along the edge (j, i). By induction hypothesis, a shortest path from i_0 to j has length $k - 1$. Extending this path by the edge (j, i), one gets a path from i_0 to i of length k. This must be a shortest path because otherwise i would acquire a loop earlier in the process (by induction hypothesis). But, due to the inhibitors of the reactions, no node gets a loop twice because all incoming edges of a node with loop are not present in the successor state and no edges (which are not loops) are created. This completes the proof.

6 Simulating Cellular Automata

The framework of cellular automata provides a quite old paradigm of computation with massive parallelism. The notion was introduced by von Neumann (see [18]) and has been intensively studied since then (see, e.g., [17]). In this section, we show that computations in cellular automata can be simulated by interactive processes of graph-based reaction systems. The major obstacle is that cellular automata may run on infinite networks of cells whereas reaction systems are strictly finite.

After recalling the notion of cellular automata and their computations that run on configurations, related graph-based reaction systems are constructed. We demonstrate then that computations in cellular automata correspond to interactive processes in a reasonable way.

6.1 Cellular Automata

A *cellular automaton* is a system $\mathcal{C} = (COL, w, k, \phi, CELL, \mathcal{N})$ where COL is a finite set of *colors*, $w \in COL$ is a special *default* color, $k \in I\!N^+$ is the *neighborhood size*, $\phi : COL^{k+1} \to COL$ is a *transition function* subject to the condition $\phi(w, \ldots, w) = w$ for the default color w, $CELL$ is a set of *cells*, and $\mathcal{N} = \{(N_i \mid i = 1, \ldots, n\}$ is the *neighborhood specification*, where each $N_i : CELL \to CELL$ for $i = 1, \ldots, k$ is the i-th *neighborhood function*. For each $v \in CELL$, $(N_1(v), \ldots, N_k(v))$ is the *neighborhood* of v subject to the condition that $N_i(v) \neq N_j(v)$ for all $i, j \in \{1, \ldots, k\}$ such that $i \neq j$.

The triple (COL, COL^k, ϕ) may be interpreted as a finite transition system with the state set COL, the input alphabet COL^k, and the deterministic transition function ϕ. We speak about colors rather than states to avoid any confusion with the states of reaction systems. The neighborhood specification $\mathcal{N}$ determines, for each cell, its k neighbors ordered from 1 to k.

A *configuration* of $\mathcal{C}$ is a function $\alpha : CELL \to COL$ such that the set of *active cells* $act(\alpha) = \{v \in CELL \mid \alpha(v) \neq w\}$ is finite. Although $CELL$ may be infinite, each configuration is finitely specified by its set of active cells.

Given a configuration α, one gets a uniquely determined *successor configuration* $\alpha' : CELL \to COL$ defined by

$$\alpha'(v) = \phi(\alpha(v), \alpha(N_1(v)), \ldots, \alpha(N_k(v)))$$

for each $v \in CELL$.

To get the new color of a cell, the transition function is applied to the current color of the cell and the colors of the neighbors as input. Note that, because a w-cell can only become active if some of its neighbors are active, for each configuration α, $act(\alpha')$ is finite so that α' is a configuration. Such a transition from α to its successor α' is denoted by $\alpha \to \alpha'$.

Given configurations α and β, a *computation* (in $\mathcal{C}$) from α to β of length $n \in \mathbb{N}^+$ is a sequence of configurations $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that $\alpha = \alpha_0$, $\beta = \alpha_n$, and $\alpha_0 \to \cdots \to \alpha_n$. We write $\alpha \to^+ \beta$ if there is a computation from α to β of length n for some $n \in \mathbb{N}^+$.

6.2 Related Graph-Based Reaction Systems

Cellular automata give rise to a special type of graph-based reaction systems that simulate their computational behavior of cellular automata. This requires to choose the proper background graphs and reactions as well as to resolve the potential infinity of the sets of cells.

Let $\mathcal{C} = (COL, w, k, \phi, CELL, \mathcal{N})$ be a cellular automaton. Then, for each finite subset $Z \subseteq CELL$, the *cells of interest*, a graph-based reaction system $\mathcal{A}(\mathcal{C}, Z) = (B(\mathcal{C}, Z), A(\mathcal{C}, Z))$ is defined as follows. The background graph $B(\mathcal{C}, Z) = (V, \Sigma, E)$ is defined by:

$V = Z \cup \{N_i(v) \mid v \in Z, i = 1, \ldots, k\}$,
$\Sigma = COL \cup \{1, \ldots, k\} \cup \{*\}$, and
$E = \{(v, N_i(v), i) \mid v \in Z, i = 1, \ldots, k\} \cup \{(v, v, c) \mid v \in Z, c \in COL\} \cup \{(v, v, w) \mid v \in V \setminus Z\}$.

The set of reactions $\mathcal{A}(\mathcal{C}, Z)$ consists of the following uninhibited reactions (recall that $Out(v)$ denotes the out-neighborhood of v, see the formal definition at the end of Sect. 2):

$(Out(v) \cup loop(v, c) \cup \bigcup_{i=1}^{k} loop(N_i(v), c_i),\ \ (\emptyset, \emptyset),\ \ loop(v, \phi(c, c_1, \ldots, c_k)))$
for $v \in Z$ and $c, c_i \in COL$, $i = 1, \ldots, k$,
$(Out(v),\ \ (\emptyset, \emptyset),\ \ Out(v))$ for $v \in Z$, and
$(loop(v, w),\ \ (\emptyset, \emptyset),\ \ loop(v, w))$ for $v \in V \setminus Z$.

Here is the intuition behind the formal definition above.

The set of nodes of the background graph consists of all cells of interest and all their neighbors. The edges that are not loops connect the cells of interest with their neighbors (directed from each cell of interest to its neighbors), where the labels reflect the order in the neighborhood. Moreover, there are loops at each cell of interest, one for each color, and the cells that are not cells of interest carry a w-loop each.

There are three kinds of reactions, all uninhibited.

1. For each cell of interest $v \in Z$ and each combination $(c_0, c_1, \ldots, c_k)$ of labels of loops on v and all its neighbors (where the label of each cell not in Z is w), there is exactly one uninhibited reaction such that its reactant graph is of the form to the left and its product graph is of the form to the right:

$$\begin{tikzcd}
& v \arrow[loop above, "c_0"] \arrow[dl, "1"'] \arrow[dr, "k"] & & & \\
vN_1(v) \arrow[loop below, "c_1"] & \cdots & N_k(v) \arrow[loop below, "c_k"] & & v \arrow[loop above, "\phi(c_0,\ldots,c_k)"]
\end{tikzcd}$$

2. For each cell of interest $v \in Z$, there is exactly one uninhibited reaction such that its reactant graph is of the form:

$$\begin{tikzcd}
& v \arrow[dl, "1"'] \arrow[dr, "k"] & \\
N_1(v) & \cdots & N_k(v)
\end{tikzcd}$$

   and its product is identical to the reactant graph.
3. For each cell $v \in V \setminus Z$, there is exactly one uninhibited reaction such that its reactant graph is of the form:

$$\begin{tikzcd}
v \arrow[loop above, "w"]
\end{tikzcd}$$

   and its product is identical to the reactant graph.

To see how the reactions work, we consider well-formed states. A state T *sub* $B(\mathcal{C}, Z)$ is *well-formed* if T contains all nodes and all (out-)neighborhood edges of $B(\mathcal{C}, Z)$, and also one loop per node. Such a state is determined by a function $lab : V_{B(\mathcal{C},Z))} \to COL$ fixing the labels of the loops. We use T_{lab} to denote this state.

A reaction step on T_{lab} yields a well-formed state $T_{lab'}$. Each node $v \in Z$ and all of its out-neighbors have single loops, so that the reaction of the first kind with the fitting loop labels applies to the out-neighborhood of $v \in Z$, sustains v and produces a loop at v. The second kind of reactions sustains the out-neighborhood of v. The third kind of reactions sustains the w-loops of nodes that are not cells of interest. Altogether, the resulting state is well-formed with respect to the labeling function $lab'\colon V_{B(\mathcal{C},Z))} \to COL$ with $lab'(v) = \phi(lab(v), lab(N_1(v), \ldots, lab(N_k(v))$ for $v \in Z$ and $lab'(v) = w$ otherwise.

This consideration allows to relate the context-independent interactive processes in $\mathcal{A}(\mathcal{C}, Z))$ on well-formed states with the computations in the modeled cellular automaton. The formulation of this result makes use of the fact that each labeling function $lab : V_{B(\mathcal{C},Z))} \to COL$ can be extended to a configuration $\alpha(lab) : CELL \to COL$ defined by $\alpha(lab)(v) = lab(v)$ for $v \in V_{B(\mathcal{C},Z))}$ and $\alpha(lab)(v) = w$ otherwise.

Theorem 3. Let $\mathcal{C} = (COL, k, \phi, w, CELL, \mathcal{N})$ be a cellular automaton. Let $\mathcal{A}(\mathcal{C}, Z)$ be the reaction system defined by some $Z \subsetneq CELL$. Let T_{lab_0} be a well-formed state for some labeling function $lab_0 : V_{B(\mathcal{C},Z)} \to COL$. Let, for some $n \in I\!N^+$, $\alpha(lab_0) = \alpha_0 \to \alpha_1 \to \cdots \to \alpha_n$ be a computation in $\mathcal{C}$ starting in the configuration that extends lab_0 subject to the condition $act(\alpha_i) \subsetneq Z$ for $i = 1, \ldots, n$. Let $T_{lab_0} \to \cdots \to T_{lab_n}$ be the state sequence of the corresponding interactive process of the same length. Then one gets $\alpha_i = \alpha(lab_i)$ for $i = 1, \ldots, n$.

Proof. The theorem is proven by induction on n.

We begin with the induction step. Consider $\alpha(lab_0) = \alpha_0 \to \alpha_1 \to \cdots \to \alpha_{n+1}$. Using the induction hypothesis for n, one gets $\alpha_i = \alpha(lab_i)$ for $i = 1, \ldots, n$, where the labeling functions are given by the state sequence $T_{lab_0} \to \cdots \to T_{lab_n}$ of some interactive process. Due to the definition of a reaction step, $T_{lab_{n+1}}$ is specified by $lab_{n+1} : V_{B(\mathcal{C},Z)} \to COL$ given by $lab_{n+1}(v) = \phi(lab_n(v), lab_n(N_1)(v), \ldots, lab_n(N_k)(v)))$ for $v \in Z$ and $lab_{n+1}(v) = w$ otherwise. Now one can show that $\alpha_{n+1} = \alpha(lab_{n+1})$ by considering two cases. For $v \in Z$, one obtains the following sequence of equalities using the definition of α_{n+1}, the induction hypothesis, the definition of $\alpha(lab)$ for some lab, the definition of lab_{n+1}, and the definition of $\alpha(lab_{n+1})$ in this order:

$$\begin{aligned}
\alpha_{n+1}(v) &= \phi(\alpha_n(v), \alpha_n(N_1(v)), \ldots, \alpha_n(N_k(v))) \\
&= \phi(\alpha(lab_n)(v), \alpha(lab_n)(N_1(v)), \ldots, \alpha(lab_n)(N_k(v))) \\
&= \phi(lab_n(v), lab_n(N_1(v)), \ldots, lab_n(N_k(v))) \\
&= lab_{n+1}(v) \\
&= \alpha(lab_{n+1})(v).
\end{aligned}$$

Finally, one can show the equality for $v \in CELL \setminus Z$. As $act(\alpha_{n+1}) \subsetneq Z$, we get $\alpha_{n+1}(v) = w$ for all $v \in CELL \setminus Z$. The same holds for $\alpha(lab_{n+1})$, because $\alpha(lab_{n+1})(v) = w$ for $v \in CELL \setminus V_{B(\mathcal{C},Z)}$ by definition and $\alpha(lab_{n+1})(v) = lab_{n+1}(v)$ for $v \in V_{B(\mathcal{C},Z)}$, but $lab_{n+1}(v) = w$ for $v \in V_{B(\mathcal{C},Z)} \setminus Z$ due to the definitions of $\alpha(lab_{n+1})$ and lab_{n+1}.

To prove the induction base for $n = 1$, the argumentation of the induction step can be repeated for α_1 and $\alpha(lab_1)$, as was done for α_{n+1} and $\alpha(lab_{n+1})$, using the fact that $\alpha_0 = \alpha(lab_0)$ by definition.

This completes the proof.

If a computation of a cellular automaton reaches a configuration in which a cell that is not of interest becomes active, then the related graph-based reaction system cannot produce the color of this cell because only cells of interest can be recolored. But if one considers an arbitrary computation of finite length, then the set of active cells along the computation is finite so that the graph-based reaction system with the set of active cells as the set of cells of interest or some larger set simulates the given computation.

7 Discussion

We have introduced graph-based reaction systems as a generalization of set-based reaction systems enhancing in this way the framework of reaction systems so that we can deal with processing (transformation) of graphs. This framework is illustrated by three small case studies: recognition of strings by finite state automata through processing their state graphs, modeling of graph algorithms through a parallel shortest-path algorithm, and modeling of parallel computation through a simulation of cellular automata.

Graph-based reaction systems provide a novel approach to graph transformation. Traditional approaches to graph transformation (see. e.g., the handbook volumes [13,20] and individual papers, such as [4,12,16]) follow the "cut, add, and paste" methodology. Given a graph T, one removes some parts of it and pastes some new parts into the remainder of T. The framework of reaction systems employs a novel, very different methodology: no "cut, add, and paste" takes place.

Instead, the whole process of graph transformation takes place within a given "graph universe" specified by the background graph B. Then, given a subgraph T of B, the reactions of a graph-based reaction system together with a possible context graph C (which is also a subgraph of B) determine the subgraph T_1 which is the successor of T. In this way, T is transformed to T_1 and consequently one moves in B from its subgraph T to its subgraph T_1. Iterating this procedure (for a given sequence γ of context graphs), one moves from T_1 to T_2, from T_2 to T_3, etc. creating in B a trajectory of subgraphs of B. Each such trajectory determines a specific way of surfing on B.

The idea of graph transformation by surfing on a given universe graph originates in [11]. The graph universe there is given by a (possibly infinite) well-founded partial-order graph $\mathcal{Z}$ (called zoom structure), and the surfing is directed by one of reaction systems given by a finite set $\mathcal{F}$ – each of reaction systems in $\mathcal{F}$ explores a specific segment of $\mathcal{Z}$. An inzoom of $\mathcal{Z}$ is a finite reverse walk in $\mathcal{Z}$ (i.e., a walk against the directions of edges in $\mathcal{Z}$). A state of $\mathcal{Z}$ is a finite set of inzooms of $\mathcal{Z}$, and a reaction system $\mathcal{A}$ over $\mathcal{Z}$ transforms states of $\mathcal{Z}$ into states of $\mathcal{Z}$ (the background set of $\mathcal{A}$ is a subset of the set of inzooms of $\mathcal{Z}$). From a graph-theoretic point of view, each reverse walk in $\mathcal{Z}$ represents a finite (one-)path graph and so each state T of $\mathcal{Z}$ represents a subgraph G_T of $\mathcal{Z}$ (which is the union of the path graphs of T). Therefore, moving in $\mathcal{A}$ from state T to state T' corresponds to moving in $\mathcal{Z}$ from its subgraph G_T to its subgraph $G_{T'}$.

From the point of view of graph transformation, graph-based reaction systems generalize the setup above in several ways:

(i) by allowing the background graph to be an arbitrary graph,
(ii) by allowing states as well as reactants and products to be arbitrary subgraphs of the background graph, and
(iii) by allowing inhibition by nodes and edges (rather than only by inzooms).

In this paper, we introduced a novel framework for graph transformation through (interactive processes of) graph-based reaction systems. There are several "natural" research directions to be pursued in order to develop this framework.

1. Further case studies utilizing this framework should be investigated. For example, graph-based variants of tilings or of fractals such as the approximation of the Sierpinski triangle and the evaluation of reversible circuits on their diagrammatic representations are good candidates (see. e.g., [1,15,19]).
2. Surfing on graphs through graph-based reaction systems brings sequences of graphs (rather than traditional sets of graphs) to the forefront of research. Investigation of sequences of sets "generated" by set-based reaction systems turned out to be very interesting and fruitful (see, e.g., [6,14,21,22]). Clearly, this line of research should be pursued for sequences of graphs defined by graph-based reaction systems.
3. A natural first step towards the understanding of the dynamics of graph-based reaction systems is the investigation of context-independent processes. For example, what sequences of graphs are generated by such processes?
4. An important research theme is "the structure of a system vs. its behavior". In the case of graph-based reaction systems it may be translated into the influence of the structure of reactions on interactive processes. For example, what kind of processes (sequences of graphs) result from restricting the reactions by requiring that each inhibitor $I = (X, Y)$ is such that $|Y| = 1$ (or dually that $|X| = 1$)? Graph-based reaction system satisfying such restriction(s) are kind of minimal systems.

Finally, we would like to mention that another kind of relationship between graphs and reaction systems is discussed in [2], where one considers networks of reaction systems. Such a network is a graph where reaction systems are placed in the nodes of the graph and then edges between nodes provide communication channels between reaction systems placed in the corresponding nodes.

Acknowledgement. We are grateful to the anonymous reviewers for their helpful comments.

References

1. Abdessaied, N., Drechsler, R.: Reversible and Quantum Circuits: Optimization and Complexity Analysis. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31937-7
2. Bottoni, P., Labella, A., Rozenberg, G.: Networks of reaction systems. Int. J. Found. Comput. Sci. (2018, to appear)
3. Brijder, R., Ehrenfeucht, A., Main, M.G., Rozenberg, G.: A tour of reaction systems. Int. J. Found. Comput. Sci. **22**(7), 1499–1517 (2011)
4. Drewes, F., Habel, A., Kreowski, H.-J.: Hyperedge replacement graph grammars (Chap. 2). In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation: Foundations, vol. 1, pp. 95–162. World Scientific Publishing Co., Singapore (1997)
5. Ehrenfeucht, A., Main, M.G., Rozenberg, G.: Combinatorics of life and death for reaction systems. Int. J. Found. Comput. Sci. **21**(3), 345–356 (2010)
6. Ehrenfeucht, A., Main, M.G., Rozenberg, G.: Functions defined by reaction systems. Int. J. Found. Comput. Sci. **22**(1), 167–178 (2011)
7. Ehrenfeucht, A., Petre, I., Rozenberg, G.: Reaction systems: a model of computation inspired by the functioning of the living cell. In: Konstantinidis, S., Moreira, N., Reis, R., Shallit, J. (eds.) The Role of Theory in Computing, pp. 11–32. World Scientific Publishing Co., Singapore (2017)
8. Ehrenfeucht, A., Rozenberg, G.: Events and modules in reaction systems. Theoret. Comput. Sci. **376**(1–2), 3–16 (2007)
9. Ehrenfeucht, A., Rozenberg, G.: Reaction systems. Fundamenta Informaticae **75**(1–4), 263–280 (2007)
10. Ehrenfeucht, A., Rozenberg, G.: Introducing time in reaction systems. Theoret. Comput. Sci. **410**(4–5), 310–322 (2009)
11. Ehrenfeucht, A., Rozenberg, G.: Zoom structures and reaction systems yield exploration systems. Int. J. Found. Comput. Sci. **25**(4–5), 275–305 (2014)
12. Ehrig, H.: Introduction to the algebraic theory of graph grammars (a survey). In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0025714
13. Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation: Concurrency, Parallelism, and Distribution, vol. 3. World Scientific Publishing Co., Singapore (1999)
14. Formenti, E., Manzoni, L., Porreca, A.E.: Fixed points and attractors of reaction systems. In: Beckmann, A., Csuhaj-Varjú, E., Meer, K. (eds.) CiE 2014. LNCS, vol. 8493, pp. 194–203. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08019-2_20
15. Grünbaum, B., Shephard, G.C.: Tilings and Patterns. W. H. Freeman, New York (1987)
16. Janssens, D., Rozenberg, G.: Graph grammars with neighbourhood-controlled embedding. Theoret. Comput. Sci. **21**, 55–74 (1982)
17. Kari, J.: Theory of cellular automata: a survey. Theoret. Comput. Sci. **334**(1–3), 3–33 (2005)
18. von Neumann, J.: The general and logical theory of automata. In: Jeffress, L. (ed.) Cerebral Mechanisms in Behavior - The Hixon Symposium. Wiley, New York (1951)
19. Peitgen, H., Jürgens, H., Saupe, D.: Chaos and Fractals: New Frontiers of Science. Springer, Berlin (2004). https://doi.org/10.1007/b97624

20. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Foundations, vol. 1. World Scientific Publishing Co., Singapore (1997)
21. Salomaa, A.: Functions and sequences generated by reaction systems. Int. J. Found. Comput. Sci. **466**(4–5), 87–96 (2012)
22. Salomaa, A.: On state sequences defined by reaction systems. In: Constable, R.L., Silva, A. (eds.) Logic and Program Semantics. LNCS, vol. 7230, pp. 271–282. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29485-3_17

Probabilistic Graph Programs for Randomised and Evolutionary Algorithms

Timothy Atkinson(✉), Detlef Plump, and Susan Stepney

Department of Computer Science, University of York, York, UK
{tja511,detlef.plump,susan.stepney}@york.ac.uk

Abstract. We extend the graph programming language GP 2 with probabilistic constructs: (1) choosing rules according to user-defined probabilities and (2) choosing rule matches uniformly at random. We demonstrate these features with graph programs for randomised and evolutionary algorithms. First, we implement Karger's minimum cut algorithm, which contracts randomly selected edges; the program finds a minimum cut with high probability. Second, we generate random graphs according to the $G(n, p)$ model. Third, we apply probabilistic graph programming to evolutionary algorithms working on graphs; we benchmark odd-parity digital circuit problems and show that our approach significantly outperforms the established approach of Cartesian Genetic Programming.

1 Introduction

GP 2 is a rule-based graph programming language which frees programmers from handling low-level data structures for graphs. The language comes with a concise formal semantics and aims to support formal reasoning on programs; see, for example, [11,19,22]. The semantics of GP 2 is nondeterministic in two respects: to execute a rule set $\{r_1, \ldots, r_n\}$ on a host graph G, any of the rules applicable to G can be picked and applied; and to apply a rule r, any of the valid matches of r's left-hand side in the host graph can be chosen. GP 2's compiler [4] has been designed by prioritising speed over completeness, thus it simply chooses the first applicable rule in textual order and the first match that is found.

For some algorithms, compiled GP 2 programs reach the performance of hand-crafted C programs. For example, [1] contains a 2-colouring program whose runtime on input graphs of bounded degree matches the runtime of Sedgewick's program in *Graph Algorithms in C*. Clearly, this implementation of GP 2 is not meant to produce different results for the same input or make random choices with pre-defined probabilities.

However, probabilistic choice is a powerful algorithmic concept which is essential to both *randomised* and *evolutionary* algorithms. Randomised algorithms take a source of random numbers in addition to input and make random choices

T. Atkinson—Supported by a Doctoral Training Grant from the Engineering and Physical Sciences Research Council (EPSRC) in the UK.

© Springer International Publishing AG, part of Springer Nature 2018
L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 63–78, 2018.
https://doi.org/10.1007/978-3-319-92991-0_5

during execution. There are many problems for which a randomised algorithm is simpler or faster than a conventional deterministic algorithm [18]. Evolutionary algorithms, on the other hand, can be seen as randomised heuristic search methods employing the generate-and-test principle. They drive the search process by variation and selection operators which involve random choices [6]. The existence and practicality of these probabilistic algorithms motivates the extension of graph programming languages to the probabilistic domain. Note that our motivation is different from existing simulation-driven extensions of graph transformation [10,14]: we propose high-level *programming* with probabilistic constructs rather than *specifying* probabilistic models.

To cover algorithms on graphs that make random choices, we define *Probabilistic GP 2* (P-GP 2) by extending GP 2 with two constructs: (1) choosing rules according to user-defined probabilities and (2) choosing rule matches uniformly at random. We build on our preliminary GP 2 extension [1], where all rule sets are executed by selecting rules and matches uniformly at random. In contrast, we propose here to extend GP 2 conservatively and allow programmers to use both probabilistic and conventional execution of rule sets. In addition, weighted rules can be used to define more complex probability distributions.

We present three case studies in which we apply P-GP 2 to randomised and evolutionary algorithms. The first example is Karger's randomised algorithm for finding a minimum cut in a graph [12]. Our implementation of the algorithm comes with a probabilistic analysis, which guarantees a high probability that the cut computed by the program is minimal. The second example is sampling from Gilbert's $G(n, p)$ random graph model [9]. The program generates random graphs with n vertices such that each possible edge occurs with probability p.

To our knowledge, these graph programs are the first implementations of the randomised algorithms using graph transformation. Our final case study is a novel approach to evolving graphs by graph programming [2]. We use graphs to represent individuals and graph programs as probabilistic mutation operators. Whereas our examples of randomised algorithms allow to analyse the probabilities of their results, performance guarantees for evolutionary algorithms are difficult to derive and we therefore turn to empirical evaluation. We use the well established approach of Cartesian Genetic Programming (CGP) as a benchmark for a set of digital circuit synthesis problems and show that our approach outperforms CGP significantly.

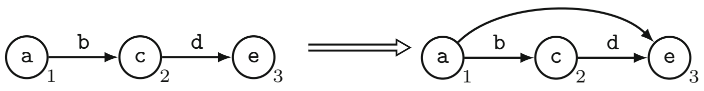
The rest of this paper is arranged as follows. Section 2 introduces the graph programming language GP 2, and Sect. 3 explains our probabilistic extension to GP 2. Sections 4 and 5 detail our applications of this extension to randomised and evolutionary algorithms, respectively. Section 6 summarises this work and proposes future topics of work.

2 Graph Programming with GP 2

This section briefly introduces the graph programming language GP 2; see [20] for a detailed account of the syntax and semantics of the language, and [4] for

```
Main := link!

link(a,b,c,d,e:list)
```



```
where not edge(1,3)
```

Fig. 1. A GP 2 program computing the transitive closure of a graph.

its implementation. A graph program consists of declarations of graph transformation rules and a main command sequence controlling the application of the rules. The rules operate on host graphs whose nodes and edges are labelled with integers, character strings or lists of integers and strings. Variables in rules are of type `int`, `char`, `string`, `atom` or `list`, where `atom` is the union of `int` and `string`. Atoms are considered as lists of length one, hence integers and strings are also lists. For example, in Fig. 1, the list variables `a`, `c` and `e` are used as edge labels while `b` and `d` serve as node labels. The small numbers attached to nodes are identifiers that specify the correspondence between the nodes in the left and the right graph of the rule.

Besides carrying list expressions, nodes and edges can be *marked*. For example, in the program of Fig. 4, the end points of a randomly selected edge are marked blue and red to redirect all edges incident to the blue node to the red node.

The principal programming constructs in GP 2 are conditional graph-transformation rules labelled with expressions. The program in Fig. 1 applies the single rule `link` *as long as possible* to a host graph. In general, any subprogram can be iterated with the postfix operator "`!`". Applying `link` amounts to nondeterministically selecting a subgraph of the host graph that matches `link`'s left graph, and adding to it an edge from node 1 to node 3 provided there is no such edge (with any label). The application condition `where not edge(1,3)` ensures that the program terminates and extends the host graph with a minimal number of edges. Rule matching is injective and involves instantiating variables with concrete values. Also, in general, any unevaluated expressions in the right-hand side of the rule are evaluated before the host graph is altered (this has no effect on the `link` rule because it does not contain operators).

Besides applying individual rules, a program may apply a rule set $\{r_1, \ldots, r_n\}$ to the host graph by nondeterministically selecting a rule r_i among the applicable rules and applying it. Further control constructs include the sequential composition $P; Q$ of programs P and Q, and the branching constructs `if` T `then` P `else` Q and `try` T `then` P `else` Q. To execute the `if`-statement, test T is executed on the host graph G and if this results in some graph, program P is executed on G. If T fails (because a rule or set of rules cannot be matched), program Q is executed on G. The `try`-statement behaves in the same way if T fails, but if T produces a graph H, then P is executed on H rather than on G.

Given any graph G, the program in Fig. 1 produces the smallest transitive graph that results from adding unlabelled edges to G. (A graph is *transitive* if for each directed path from a node v_1 to another node v_2, there is an edge from v_1 to v_2.) In general, the execution of a program on a host graph may result in different graphs, fail, or diverge. The *semantics* of a program P maps each host graph to the set of all possible outcomes. GP 2 is computationally complete in that every computable function on graphs can be programmed [20].

3 P-GP 2: A Probabilistic Extension of GP 2

We present a conservative extension to GP 2, called Probabilistic GP 2 (P-GP 2), where a rule set may be executed probabilistically by using additional syntax. Rules in the set will be picked according to probabilities specified by the programmer, while the match of a selected rule will be chosen uniformly at random. When the new syntax is not used, a rule set is treated as nondeterministic and executed as in GP 2's implementation [4]. This is preferable when executing confluent rule sets where the discovery of all possible matches is expensive and unnecessary.

3.1 Probabilistic Rule Sets

To formally describe probabilistic decisions in P-GP 2, we consider the application of a rule set $\mathcal{R} = \{r_1, \ldots, r_n\}$ to some host graph G. The set of all possible rule-match pairs from $\mathcal{R}$ in G is denoted by $G^{\mathcal{R}}$:

$$G^{\mathcal{R}} = \{(r_i, g) \mid r_i \in \mathcal{R} \text{ and } G \Rightarrow_{r_i,g} H \text{ for some graph } H\} \tag{1}$$

We make separate decisions for choosing a rule and a match. The first decision is to choose a rule, which is made over the subset of rules in $\mathcal{R}$ that have matches in G, denoted by $\mathcal{R}^G$:

$$\mathcal{R}^G = \{r_i \mid r_i \in \mathcal{R} \text{ and } G \Rightarrow_{r_i,g} H \text{ for some match } g \text{ and graph } H\} \tag{2}$$

Once a rule $r_i \in \mathcal{R}^G$ is chosen, the second decision is to choose a match with which to apply r_i. The set of possible matches of r_i is denoted by G^{r_i}:

$$G^{r_i} = \{g \mid G \Rightarrow_{r_i,g} H \text{ for some graph } H\} \tag{3}$$

We assign a probability distribution (defined below) to $G^{\mathcal{R}}$ which is used to decide particular rule executions. This distribution, denoted by $P_{G^{\mathcal{R}}}$, has to satisfy:

$$P_{G^{\mathcal{R}}} : G^{\mathcal{R}} \to [0,1] \quad \text{such that} \quad \sum_{(r_i,g)\in G^{\mathcal{R}}} P_{G^{\mathcal{R}}}(r_i, g) = 1 \tag{4}$$

where $[0, 1]$ denotes the real-valued (inclusive) interval between 0 and 1.

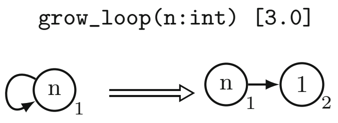


Fig. 2. A P-GP 2 declaration of a rule with associated weight 3.0. The weight is indicated in square brackets after the variable declaration.

P-GP 2 allows the programmer to specify $P_{G^\mathcal{R}}$ by rule declarations in which the rule can be associated with a real-valued positive weight. This weight is listed in square brackets after the rule's variable declarations, as shown in Fig. 2. This syntax is optional and if a rule's weight is omitted, the weight is 1.0 by default. In the following we use the notation $w(r)$ for the positive real value associated with any rule r in the program.

To indicate that the call of a rule set $\{r_1, \ldots, r_n\}$ should be executed probabilistically, the call is written with square brackets:

$$\texttt{[}r_1, \ldots, r_n\texttt{]} \tag{5}$$

This includes the case of a probabilistic call of a single rule r, written $\texttt{[}r\texttt{]}$, which ignores any weight associated with r and simply chooses a match for r uniformly at random. Given a probabilistic rule set call $\mathcal{R} = \texttt{[}r_1, \ldots, r_n\texttt{]}$, the probability distribution $P_{G^\mathcal{R}}$ is defined as follows. The summed weight of all rules with matches in G is $\sum_{r_x \in \mathcal{R}^G} w(r_x)$, and the weighted distribution over rules in $\mathcal{R}^G$ assigns to each rule $r_i \in \mathcal{R}^G$ the following probability:

$$\frac{w(r_i)}{\sum\limits_{r_x \in \mathcal{R}^G} w(r_x)} \tag{6}$$

The uniform distribution over the matches of each rule $r_i \in \mathcal{R}^G$ assigns the probability $1/\left|G^{r_i}\right|$ to each match $g \in G^{r_i}$. This yields the definition of $P_{G^\mathcal{R}}$ for all pairs $(r_i, g) \in G^\mathcal{R}$:

$$P_{G^\mathcal{R}}(r_i, g) = \frac{w(r_i)}{\sum\limits_{r_x \in \mathcal{R}^G} w(r_x)} \times \frac{1}{|G^{r_i}|} \tag{7}$$

In the implementation of P-GP 2, the probability distribution $P_{G^\mathcal{R}}$ decides the choice of rule and match for $\mathcal{R} = \texttt{[}r_1, \ldots, r_n\texttt{]}$ (based on a random-number generator). Note that this is correctly implemented by first choosing an applicable rule r_i according to the weights and then choosing a match for r_i uniformly at random. The set of all matches is computed at run-time using the existing search-plan method described in [3]. Note that this is an implementation decision that is not intrinsic to the design of P-GP 2.

If a rule set $\mathcal{R}$ is called using GP 2 curly-brackets syntax, execution follows the GP 2 implementation [4]. Hence our language extension is conservative; existing

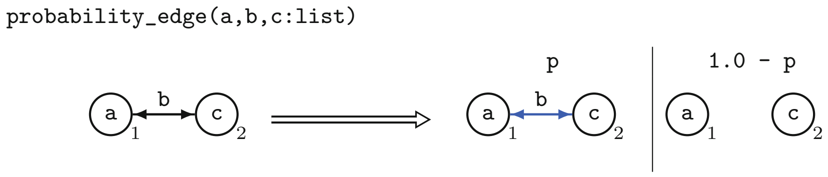


Fig. 3. A PGTS rule with multiple right-hand sides. The probability of each right-hand side is the value given above it.

GP 2 programs will execute exactly as before because probabilistic behaviour is invoked only by the new syntax. The implementation of P-GP 2 is available online[1].

3.2 Related Approaches

In this section we address three other approaches to graph transformation which incorporate probabilities. All three aim at modelling and analysing systems rather than implementing algorithms by graph programs, which is our intention. The port graph rewriting framework PORGY [8] allows to model complex systems by transforming port graphs according to strategies formulated in a dedicated language. Probability distributions similar to those in this paper can be expressed in PORGY using the `ppick` command which allows probabilistic program branching, possibly through external function calls.

Stochastic Graph Transformation Systems [10] (SGTS) are an approach to continuous-time graph transformation. Rule-match pairs are associated with continuous probability functions describing their probability of executing within a given time window. While the continuous time model is clearly distinct to our approach, the application rates associated with rules in SGTS describe similar biases in probabilistic rule choice as our approach.

Closest to our approach are Probabilistic Graph Transformation Systems (PGTS) [14]. This model assumes nondeterministic choice of rule and match as in conventional graph transformation, but executes rules probabilistically. In PGTS, rules have single left-hand-sides but possibly several right-hand sides equipped with probabilities. This mixture of nondeterminism and probabilistic execution gives rise to Markov decision processes. There are clear similarities between our approach and PGTS: both operate in discrete steps and both can express nondeterminism and probabilistic behaviour. However, PGTS are strict in their allocation of behaviour; rule and match choice is nondeterministic and rule execution is probabilistic. In our approach, a programmer may specify that a rule set is executed in either manner. It seems possible to simulate (unnested) PGTS in our approach by applying a nondeterministic rule set that chooses a rule and its match followed by a probabilistic rule set which executes one of the right-hand sides of this rule. For example, the first loop in the $G(n, p)$ program

[1] https://github.com/UoYCS-plasma/P-GP2.

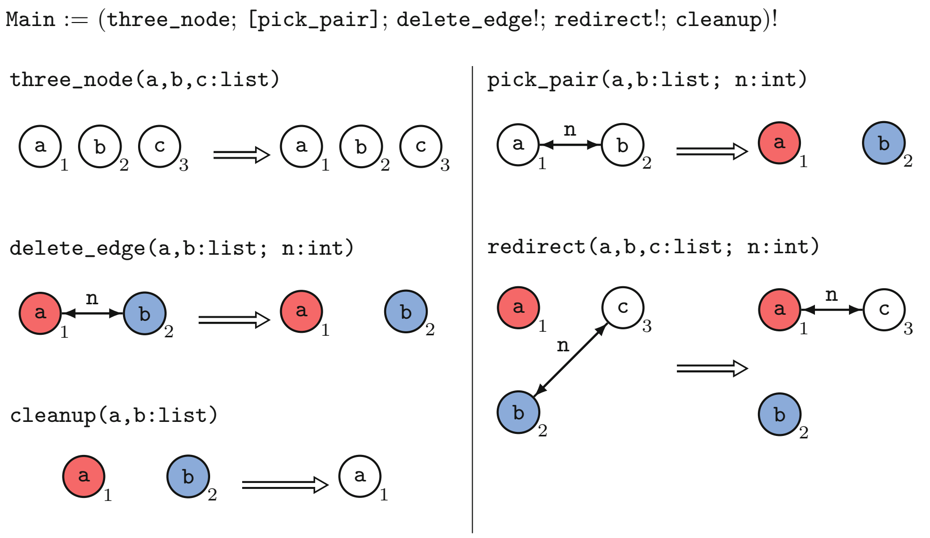

Main := (three_node; [pick_pair]; delete_edge!; redirect!; cleanup)!
three_node(a,b,c:list)
pick_pair(a,b:list; n:int)
delete_edge(a,b:list; n:int)
redirect(a,b,c:list; n:int)
cleanup(a,b:list)
a
b
c
n
1
2
3

Fig. 4. The contraction procedure of Karger's algorithm implemented in P-GP 2 (Color figure online)

in Fig. 6 simulates a single PGTS rule; `pick_edge` nondeterministically chooses a match, and `[keep_edge, delete_edge]` probabilistically executes some right-hand side on the chosen match. Figure 3 visualises this single PGTS rule.

4 Application to Randomised Algorithms

4.1 Karger's Minimum Cut Algorithm

Karger's contraction algorithm [12] is a randomised algorithm that attempts to find a minimum cut in a graph G, that is, the minimal set of edges to delete to produce two disconnected subgraphs of G. The contraction procedure repeatedly merges adjacent nodes at random until only two remain. As this algorithm is designed for multi-graphs (without loops or edge direction), we model an edge between two nodes as two directed edges, one in each direction. For visual simplicity, we draw this as a single edge with an arrow head on each end. We assume that input graphs are unmarked, contain only simulated directed edges, and are connected. We also assume that edges are labelled with unique integers, as this allows us to recover the cut from the returned solution.

Figure 4 shows a P-GP 2 implementation of this contraction procedure. This program repeatedly chooses an edge to contract at random using the `pick_pair` rule, which marks the surviving node `red` and the node that will be deleted `blue`. The nodes' common edges are deleted by `delete_edge` and all other edges connected to the `blue` node that will be deleted are redirected to connect to the `red` surviving node by `redirect`. In the final part of the loop, `cleanup`

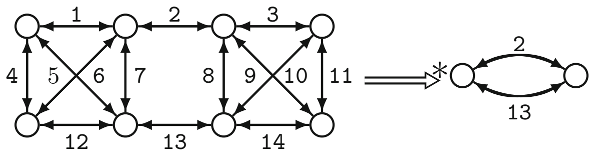


Fig. 5. Karger's contraction algorithm applied to a simple 8-node graph to produce a minimal 2-edge cut. The probability of producing this cut is at least $\frac{1}{28}$; our implementation generated this result after seven runs.

deletes the `blue` node and unmarks the `red` node. This sequence is applied as long as possible until the rule `three_node` is no longer applicable; this rule is an identity rule ensuring that a contraction will not be attempted when only 2 nodes remain. The final graph produced by this algorithm represents a cut, where the edges between the 2 surviving nodes are labelled with integers. The edges with corresponding integer labels in the input graph are removed to produce a cut.

Karger's analysis of this algorithm finds a lower bound for the probability of producing a minimum cut. Consider a minimum cut of c edges in a graph of n nodes and e edges. The minimum degree of the graph must be at least c, so $e \geq \frac{n.c}{2}$. If any of the edges of the minimum cut are contracted, that cut will not be produced. Therefore the probability of the cut being produced is the probability of not contracting any of its edges throughout the algorithm's execution. The probability of picking such an edge for contraction is:

$$\frac{c}{e} \leq \frac{c}{\frac{n.c}{2}} = \frac{2}{n} \tag{8}$$

Thus the probability p_n of never contracting any edge in c is:

$$p_n \geq \prod_{i=3}^{n} 1 - \frac{2}{i} = \frac{2}{n(n-1)} \tag{9}$$

For example, applying Karger's algorithm to the host graph G shown in Fig. 5 can produce one possible minimum cut (cutting 2 edges), which happens with probability greater or equal to than $\frac{1}{28}$. By using rooted nodes (see [4]) it is possible to design a P-GP 2 program that executes this algorithm on a graph with edges E in $O(|E|^2)$ time, with `pick_pair` being the limiting rule taking linear time to find all possible matches, applied $|E| - 2$ times.

4.2 $G(n, p)$ Model for Random Graphs

The $G(n, p)$ model [9] is a probability distribution over graphs of n vertices where each possible edge between vertices occurs with probability p. Here we describe an algorithm for sampling from this distribution for given parameters n and p. This model is designed for simple graphs and so we model an edge between two nodes, in a similar manner to that used in Karger's algorithm, as two directed edges, one in each direction.

```
Main := (pick_edge; [keep_edge, delete_edge])!; unmark_edge!
```

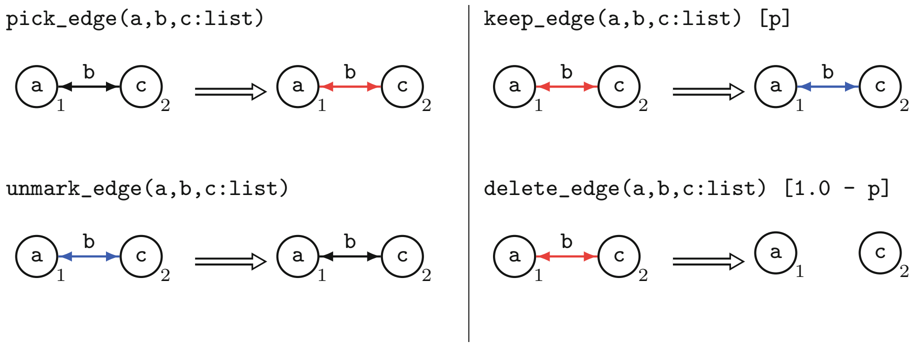


Fig. 6. P-GP 2 program for sampling from the $G(n, p)$ model for some probability p. The input is assumed to be a connected unmarked graph with n vertices. (Color figure online)

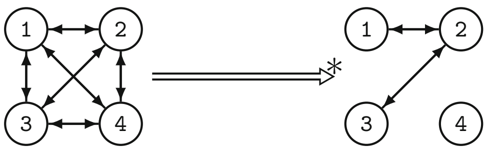


Fig. 7. The $G(n, p)$ program applied to a complete 4-node graph with $p = 0.4$. The probability of producing this result is 0.0207.

As we are concerned with a fixed number of vertices n, we assume an unmarked input graph with n vertices and for each pair of vertices v_1, v_2 exactly one edge with v_1 as its source and v_2 as its target – effectively a fully connected graph with two directed edges simulating a single undirected edge. Then $G(n, p)$ can be sampled by parameterising the GP 2 algorithm given in Fig. 6 by p. In this algorithm, every undirected edge in the host graph is chosen nondeterministically by `pick_edge`, marking it `red`. Then this edge is either kept and marked `blue` by `keep_edge` with probability p or it is deleted by `delete_edge` with probability $1 - p$. After all edges have either been deleted or marked `blue`, `unmark_edge` is used to remove the surviving edges' marks. By applying this algorithm, each possible edge is deleted with probability $1 - p$ and hence occurs with probability p, sampling from the $G(n, p)$ model.

Sampling from the $G(n, p)$ model yields a uniform distribution over graphs of n nodes and M edges and each such graph occurs with probability:

$$p^M(1-p)^{\binom{n}{2}-M} \tag{10}$$

Figure 7 shows a possible result when applying this algorithm to a simple 4-node input with $p = 0.4$.

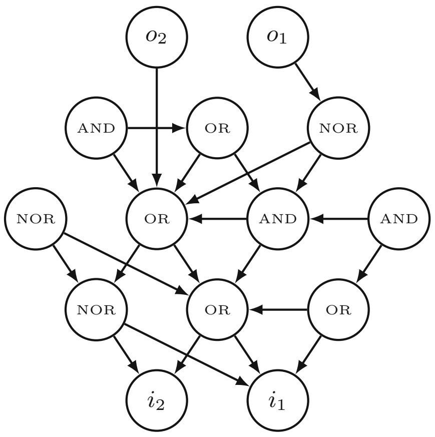


Fig. 8. An example EGGP Individual for a digital circuit problem. Outgoing edges of nodes represent the nodes that they use as inputs; for example $o_2 = (i_2 \downarrow i_1) \vee (i_2 \vee i_1)$.

5 Application to Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a class of meta-heuristic search and optimisation algorithms that are inspired by the principles of neo-Darwinian evolution. In its most general sense, an EA is an iterative process were a population of individual candidate solutions to a given problem are used to generate a new population using mutation and crossover operators. Individuals from the existing population are selected to reproduce according to a fitness function; a measure of how well they solve a given problem. Mutation operators make (typically small) changes to an individual solution, whereas crossover operators attempt to combine two individual solutions to generate a new solution that maintains some of the characteristics of both parents.

Graphs have been used extensively in EAs due to their inherent generalisation of various problems of interest; digital circuits, program syntax trees and neural networks are commonly studied examples. For example, there are extensions of Genetic Programming (a type of EA that evolves program syntax trees) that incorporate graph-like structures, such as Parallel Distributed GP [21] and MOIST [15]. Neural Evolution of Augmenting Topologies [23] evolves artificial neural networks treated as graph structures. Cartesian Genetic Programming (CGP) evolves strings of integers that encode acyclic graphs [17], and has been applied to various problems such as circuits and neural networks [24].

In [2], probabilistic graph programming (specifically, the P-GP 2 variant described in [1]) was proposed as a mechanism for specifying mutation operators. The approach, Evolving Graphs by Graph Programming (EGGP), was evaluated on a set of classic digital circuit benchmark problems and found to make statistically significant improvements in comparison to an existing implementation of CGP [25]. In the rest of this section, we explain the implementation of EGGP using probabilistic graph programming and present new benchmark results for a set of odd parity digital circuit synthesis problems.

5.1 Evolving Graphs by Graph Programming

Individuals in EGGP represent computational networks with fixed sets of inputs and outputs. They have a fixed set of nodes, each either representing a function, an input or an output. Output and function nodes have input connections, which are given by their outgoing edges. These edges are labelled with integers to indicate the ordering of the inputs of that node; this is an necessary feature for asymmetric functions such as division. Figure 8 shows an example EGGP individual for digital circuit synthesis with 2 inputs, 2 outputs and function set $\{\texttt{AND}, \texttt{OR}, \texttt{NAND}, \texttt{NOR}\}$.

Definition 1. *[EGGP Individual] An EGGP Individual over function set F is a directed graph $I = \{V, E, s, t, l, a, V_i, V_o\}$ where V is a finite set of nodes and E is a finite set of edges. $s\colon E \to V$ is a function associating each edge with its source. $t\colon E \to V$ is a function associating each edge with its target. $V_i \subseteq V$ is a set of input nodes. Each node in V_i has no outgoing edges and is not associated with a function. $V_o \subseteq V$ is a set of output nodes. Each node in V_o has one outgoing edge, no incoming edges and is not associated with a function. $l\colon V \to F$ labels every "function node" that is not in $V_i \cup V_o$ with a function in F. $a\colon E \to \mathbb{Z}$ labels every edge with a positive integer.*

In our implementation of EGGP in P-GP 2, function associations are encoded by labelling a node with a string representation of its function. For example, in Fig. 8, a node with function `AND` is labelled with the string "`AND`". Input and output nodes are encoded by labelling each of those nodes with a list of the form `a:b` where `b` is the string "`IN`" or "`OUT`" respectively, and `a` is a list uniquely identifying that output or input. The function a described in Definition 1 is used to order inputs, an important feature for avoiding ambiguity in asymmetric functions; as we deal with symmetric functions in this work these details are omitted from Fig. 8.

In [2], two types of atomic mutations are used. The first, function mutation, relabels a function node with a different function, where the new function is chosen with uniform probability. The second mutation, edge mutation, redirects an edge so that a function node or output node uses a different input. In [2] and here, digital circuits are of interest, therefore we require mutation operators that preserve acyclicity.

The edge mutation used, given in Fig. 9, selects an edge to mutate with uniform probability using `pick_edge`. The source of this edge is marked `blue` and its target is marked `red`. Then `mark_output` is applied as long as possible. This marks all nodes, for which there is a directed path to the source of the selected edge, `blue`. The remaining unmarked nodes have no such directed path to the source of the selected edge and so can be targeted by a new edge from that source without introducing a cycle. A new (unmarked) target is chosen with uniform

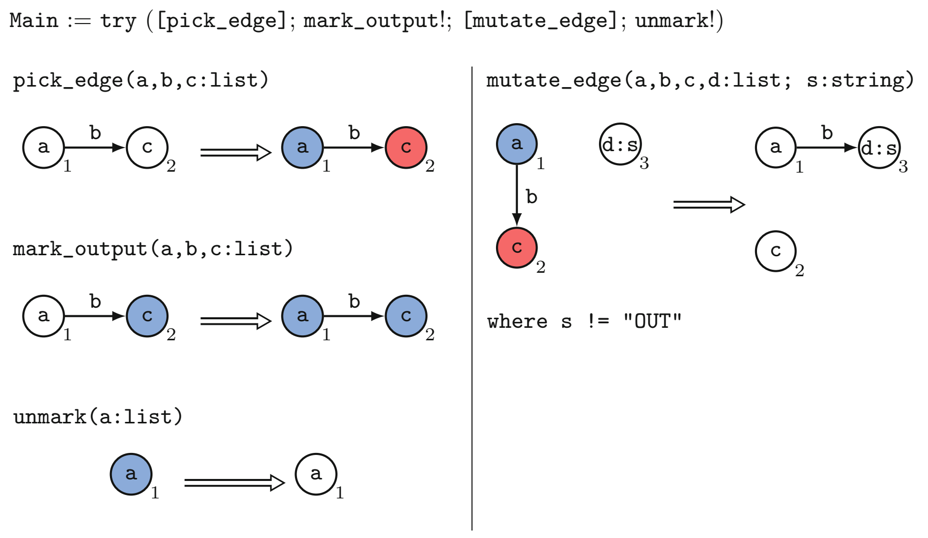

Main := try ([pick_edge]; mark_output!; [mutate_edge]; unmark!)
pick_edge(a,b,c:list)
mark_output(a,b,c:list)
unmark(a:list)
mutate_edge(a,b,c,d:list; s:string)
where s != "OUT"

Fig. 9. A P-GP 2 program for mutating an individual's edge while preserving acyclicity. (Color figure online)

probability using `mutate_edge`, executing the edge mutation. The condition of `mutate_edge` means that an output node cannot be targeted, thereby preserving the requirement that output nodes have no incoming edges. Finally, `unmark` unmarks all `blue` marked nodes, returning the now mutated EGGP individual to an unmarked state. The entire program is surrounded by a `try` statement to prevent errors in the case that an edge is chosen to mutate but there are no valid new targets.

In our implementation[2], edge and node mutations are written as P-GP 2 programs which are compiled to C code and integrated with raw C code performing the rest of the evolutionary algorithm. As a crossover operator has not yet been developed for EGGP, the $1 + \lambda$ evolutionary algorithm is used, where in each generation 1 individual survives and is used to generate λ new solutions. A more detailed explanation of EGGP and its parameters is available in [2].

5.2 Odd-Parity Benchmark Problems

Here we compare EGGP against the commonly used graph-based evolutionary algorithm Cartesian Genetic Programming (CGP) for a new set of benchmark odd-parity circuit synthesis problems. CGP is a standard approach in the literature that uses a graph-based representation of solutions, but uses linear encodings that do not exploit graph transformations during mutation. The problems studied are given in Table 1 and complement the even-parity problems examined in [2]. We use the publicly available CGP library [25] to produce CGP results.

[2] https://github.com/UoYCS-plasma/EGGP.

Table 1. Digital circuit benchmark problems.

Problem	Inputs	Outputs
5-bit odd parity (5-OP)	5	1
6-bit odd parity (6-OP)	6	1
7-bit odd parity (7-OP)	7	1
8-bit odd parity (8-OP)	8	1

Table 2. Results from Digital Circuit benchmarks for CGP and EGGP. The p value is from the two-tailed Mann-Whitney U test. Where $p < 0.05$, the effect size from the Vargha-Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**.

Problem	EGGP			CGP			p	A
	ME	MAD	IQR	ME	MAD	IQR		
5-OP	38,790	13728	29,490	96,372	41,555	91,647	10^{-18}	**0.86**
6-OP	68,032	22,672	52,868	502,335	274,132	600,291	10^{-31}	**0.97**
7-OP	158,852	69,477	142,267	1,722,377	934,945	2,058,077	10^{-33}	**0.99**
8-OP	315,810	128,922	280,527	7,617,310	4,221,075	9,830,470	10^{-34}	**0.99**

For both algorithms we use the following common parameters. We use the $1+\lambda$ evolutionary algorithm with $\lambda = 4$. 100 fixed nodes are used for each individual. Fitness is defined as the number of incorrect bits in the entire truth table of a given individual. CGP is applied with a mutation rate of 0.04, considered to be appropriate in [17], whereas EGGP has been observed to perform better at lower rates and is applied with a mutation rate of 0.01.

For both algorithms, we execute 100 runs until a solution is found and measure the number of evaluations required to find a correct solution in each run. This value approximates the effort required by an algorithm to solve a given problem. We measure the following statistics; median evaluations (ME), median absolute deviation (MAD), and interquartile range in evaluations (IQR). The median absolute deviation is the median absolute difference in evaluations from the median evaluations statistic. We test for significant differences in the median of the two results using the non-parametric two-tailed Mann-Whitney U test [16] and measure the effect size of significant differences using the Vargha-Delaney A test [26].

Table 2 shows the results from running these experiments. These are consistent with the results in [2], in that EGGP and its mutation operators perform statistically significantly better (with large effect size) for digital circuit synthesis problems (on all of the problems studied here) than CGP under similar conditions. These results are not intended to represent a detailed study of the application of probabilistic graph programming to EAs. Instead they give a flavour of promising results published elsewhere, and represent a possible approach to empirical evaluation of P-GP 2 programs when formal approximations of behaviour are intractable.

6 Conclusion and Future Work

We have presented P-GP 2, a conservative extension to the graph programming language GP 2 which allows a programmer to specify probabilistic executions of rule sets, with weighted distributions over rules and uniform distributions over matches. This language has been used to implement Karger's randomised algorithm for finding a minimum cut of a graph with high probability. We have also implemented a P-GP 2 program for the $G(n, p)$ random graph model which generates n-node graphs in which edges between nodes exist with probability p. Finally, we have described the application of P-GP 2 to evolutionary algorithms in our approach EGGP. The program of this case study was evaluated empirically on common circuit benchmark problems and found to significantly outperform a publicly available implementation of Cartesian Genetic Programming.

There are a number of possible directions for future work. We would like to explore which algorithms from the areas of randomised graph algorithms and random graph generation can be described in P-GP 2. Obvious examples include randomised algorithms for checking graph connectedness [18], generating minimum spanning trees [13] and generating random graphs according to the model of [7]. Additionally, it would be interesting to investigate the efficiency of using incremental pattern matching [5] in the implementation as an alternative method for identifying all matches. Turning to evolutionary algorithms, there are various avenues to be explored, such as using graph programming to represent crossover operators, and combining domain knowledge and graph representations to possibly achieve even better performance.

References

1. Atkinson, T., Plump, D., Stepney, S.: Probabilistic graph programming. In: Pre-Proceedings of Graph Computation Models (GCM 2017) (2017)
2. Atkinson, T., Plump, D., Stepney, S.: Evolving graphs by graph programming. In: Castelli, M., Sekanina, L., Zhang, M., Cagnoni, S., García-Sánchez, P. (eds.) EuroGP 2018. LNCS, vol. 10781, pp. 35–51. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77553-1_3
3. Bak, C.: GP 2: efficient implementation of a graph programming language. Ph.D. thesis, Department of Computer Science, University of York (2015). http://etheses.whiterose.ac.uk/12586/
4. Bak, C., Plump, D.: Compiling graph programs to C. In: Echahed, R., Minas, M. (eds.) ICGT 2016. LNCS, vol. 9761, pp. 102–117. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40530-8_7
5. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 396–410. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_27

6. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Natural Computing Series, Second edn. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-05094-1
7. Erdős, P., Rényi, A.: On random graphs. Publ. Math. (Debrecen) **6**, 290–297 (1959)
8. Fernández, M., Kirchner, H., Pinaud, B.: Strategic port graph rewriting: an interactive modelling and analysis framework. In: Proceedings of 3rd Workshop on Graph Inspection and Traversal Engineering (GRAPHITE 2014). Electronic Proceedings in Theoretical Computer Science, vol. 159, pp. 15–29 (2014). https://doi.org/10.4204/EPTCS.159.3
9. Gilbert, E.N.: Random graphs. Ann. Math. Stat. **30**(4), 1141–1144 (1959)
10. Heckel, R., Lajios, G., Menge, S.: Stochastic graph transformation systems. Fundamenta Informaticae **74**(1), 63–84 (2006)
11. Hristakiev, I., Plump, D.: Checking graph programs for confluence. In: Seidl, M., Zschaler, S. (eds.) STAF 2017. LNCS, vol. 10748, pp. 92–108. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74730-9_8
12. Karger, D.R.: Global min-cuts in RNC, and other ramifications of a simple mincut algorithm. In: Proceedings of 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1993), pp. 21–30. Society for Industrial and Applied Mathematics (1993)
13. Karger, D.R.: Random sampling in matroids, with applications to graph connectivity and minimum spanning trees. In: Proceedings of 34th Annual Symposium on Foundations of Computer Science (FOCS 1993), pp. 84–93 (1993). https://doi.org/10.1109/SFCS.1993.366879
14. Krause, C., Giese, H.: Probabilistic graph transformation systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 311–325. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_21
15. Galván-López, E., Rodríguez-Vázquez, K.: Multiple interactive outputs in a single tree: an empirical investigation. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 341–350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71605-1_32
16. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18**(1), 50–60 (1947)
17. Miller, J.F. (ed.): Cartesian Genetic Programming. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17310-3_2
18. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
19. Plump, D.: Reasoning about graph programs. In: Proceedings of Computing with Terms and Graphs (TERMGRAPH 2016). Electronic Proceedings in Theoretical Computer Science, vol. 225, pp. 35–44 (2016). https://doi.org/10.4204/EPTCS.225.6
20. Plump, D.: From imperative to rule-based graph programs. J. Logical Algebraic Methods Program. **88**, 154–173 (2017). https://doi.org/10.1016/j.jlamp.2016.12.001
21. Poli, R.: Parallel distributed genetic programming. In: Corne, D., Dorigo, M., Glover, F. (eds.) New Ideas in Optimization, pp. 403–431. McGraw-Hill, New York (1999)
22. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 33–48. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09108-2_3

23. Stanley, K.O., Miikkulainen, R.: Efficient reinforcement learning through evolving neural network topologies. In: Proceedings of Annual Conference on Genetic and Evolutionary Computation (GECCO 2002), pp. 569–577. Morgan Kaufmann, Burlington (2002)
24. Turner, A.J., Miller, J.F.: Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In: Proceedings of GECCO 2013, pp. 1005–1012. ACM (2013). https://doi.org/10.1145/2463372.2463484
25. Turner, A.J., Miller, J.F.: Introducing a cross platform open source Cartesian genetic programming library. Genet. Program. Evol. Mach. **16**(1), 83–91 (2015). https://doi.org/10.1007/s10710-014-9233-1
26. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000)

Graph-Rewriting Petri Nets

Géza Kulcsár(✉), Malte Lochau, and Andy Schürr

Real-Time Systems Lab, TU Darmstadt, Darmstadt, Germany
{geza.kulcsar,malte.lochau,andy.schuerr}@es.tu-darmstadt.de

Abstract. Controlled graph rewriting enhances expressiveness of plain graph-rewriting systems (i.e., sets of graph-rewriting rules) by introducing additional constructs for explicitly controlling graph-rewriting rule applications. In this regard, a formal semantic foundation for controlled graph rewriting is inevitable as a reliable basis for tool-based specification and automated analysis of graph-based algorithms. Although several promising attempts have been proposed in the literature, a comprehensive theory of controlled graph rewriting capturing semantic subtleties of advanced control constructs provided by practical tools is still an open challenge. In this paper, we propose graph-rewriting Petri nets (GPN) as a novel foundation for unifying control-flow and rule-application semantics of controlled graph rewriting. GPN instantiate coloured Petri nets with categorical DPO-based graph-rewriting theory where token colours denote typed graphs and graph morphisms and transitions define templates for guarded graph-rewriting rule applications. Hence, GPN enjoy the rich body of specification and analysis techniques of Petri nets including inherent notions of concurrency. To demonstrate expressiveness of GPN, we present a case study by means of a topology-control algorithm for wireless sensor networks.

1 Introduction

Graph-rewriting systems provide an expressive and theoretically founded, yet practically applicable and tool-supported framework for the specification and automated analysis of complex software systems [5]. Due to their declarative nature, graph-rewriting systems are inherently non-deterministic as (1) multiple graph-rewriting rules might be simultaneously applicable to an input graph and (2) the input graph might contain multiple matches for a rule application. The lack of expressiveness in *controlling* rule applications may obstruct a precise specification of more complicated graph-based algorithms involving sequential, conditional, iterative (and intentionally non-terminating), or even concurrent composition of rule applications. To tackle this problem, various tools (e.g., PROGRES [18], Fujaba [7] and eMoflon [14]) incorporate *controlled* (or, programmed) graph rewriting enriching declarative graph rewriting with additional constructs for explicitly restricting the order and matches of rule applications.

This work has been partially funded by the German Research Foundation (DFG) as part of project A1 within CRC1053–MAKI.

L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 79–96, 2018.
https://doi.org/10.1007/978-3-319-92991-0_6

Bunke was one of the first to lay a formal foundation for controlled graph rewriting by proposing a generalization of programmed string grammars to graph grammars [2]. Since then, several promising attempts have been proposed to investigate essential semantic aspects of graph-rewriting processes. Schürr proposes a semantic domain for controlled graph-rewriting systems with sequencing, choice and recursion using a *denotational* fix-point characterization of valid input-output graph pairs [17]. The *transformation units* of Kreowski et al. [13] provide a high-level, flexible formalism to describe controlled graph-rewriting processes, however, as in the previous case, operational semantics and formal analysis aspects are out-of-scope. In contrast, Plump and Steinert propose an *operational* semantics for the programmed graph-rewriting language *GP*, mostly focusing on algorithmic aspects and data structures for executing *GP* programs on input graphs [16]. Further existing approaches, like those of Guerra and de Lara [9] as well as Wimmer et al. [20], integrate graph-based model transformations and control structures of coloured Petri nets for purposes similar to those pursued in this paper. Nevertheless, those approaches rely on out-of-the-box model-transformation engines and, thus, do not address issues arising from graph-rewriting theory. While *algebraic higher-order nets* by Hoffmann and Mossakowski [10] provide a means to handle graphs attached to tokens in Petri nets models, they do not extend PN transition semantics for algebraic graph rewriting as in our work. Corradini et al. propose a *trace* notion and a corresponding trace-based equivalence relation for (non-controlled) graph-rewriting processes [3]. Thereupon, Corradini et al. propose *graph processes* as a compact representation of sequences of rule applications incorporating a static notion of independence inspired by Petri nets [1,4].

To summarize, existing formalisms mostly focus on particular formal aspects concerning (mostly operational) semantics of (controlled) graph rewriting, whereas a comprehensive theory of controlled graph rewriting is still an open issue. From a theoretical point of view, recent approaches often lack a proper trade-off between *separation* and *integration* of the two competing, yet partially overlapping, theories involved in controlled graph rewriting, namely categorical graph-rewriting theory (e.g., DPO/SPO style) and process theory (e.g., operational/denotational semantics). As a result, essential theoretical concepts established in both worlds in potentially different ways (e.g., notions of non-determinism, conflict, independence and concurrency) may be obscured when being combined into one formal framework, owing to the different interpretations of those concepts in their respective originating theory. As a crucial example, *concurrency* in graph rewriting usually amounts to independently composed, yet operationally interleaved rule application steps, thus leaving open the *truly* concurrent aspects of operational step semantics. From a practical point of view, recent approaches are often *incomplete*, missing semantic subtleties of advanced constructs provided by recent tools (e.g., negative application conditions, propagation of sub-graph bindings to synchronize matches of rule applications etc. [7,14,19]). (Note that another important aspect of graph rewriting in practice, namely attributed graphs, is out-of-scope for this paper.)

In this paper, we propose *graph-rewriting Petri nets* (*GPN*) as a novel foundation of controlled graph rewriting unifying control-flow and rule-application semantics of those systems in a comprehensive and intuitive way. To this end, we aim at integrating constructs for expressing control-flow specifications, sub-graph bindings and concurrency in controlled graph rewriting into a unified formal framework which allows for automated reasoning using state-of-the-art analysis techniques. GPN are based on coloured Petri nets (CPN) [11], a backward-compatible extension of Petri nets introducing notions of typed data attached to tokens (so-called colours) being processed by guarded transitions. In particular, GPN instantiate CPN with categorical DPO-based graph-rewriting theory where token colours denote type graphs and graph morphisms and transitions define templates for guarded graph-rewriting rule applications. In this way, rule applications are composeable in arbitrary ways in GPN, including inherent notions of concurrency. In addition, our GPN theory incorporates advanced constructs including complex transition guards and (concurrent) propagation of multiple graphs and/or sub-graph matches among rule applications. The resulting GPN theory preserves major theoretical properties and accompanying reasoning techniques of the underlying (coloured) Petri net framework such as reachability analysis as well as natural notions of conflict and independence of controlled graph-rewriting rule applications. To demonstrate expressiveness of GPN, we present a case study by means of a topology-control algorithm for wireless sensor networks and we describe how to reason about essential correctness properties of those systems by utilizing the underlying Petri net theory.

2 An Illustrative Example: WSN Topology Control

We first illustrate controlled graph rewriting by an example: a simplified *wireless sensor network* (*WSN*) in which autonomous, mobile sensors communicate through wireless channels. WSN communication topologies evolve over time (e.g., due to unpredictable status changes of nodes or links) thus potentially deteriorating topology quality. The quality of topologies is usually measured by performance metrics and other properties like energy consumption of nodes. *Topology control* (*TC*) is a widely used technique to maintain or improve quality of topologies in a proactive manner, by continuously adapting the status of communication links [15]. In a realistic WSN scenario, TC is necessarily employed in a decentralized setting, where an external *environment* sporadically impacts the current topology in an unpredictable manner. When analyzing WSN evolution, those changes are interleaved with proactively performed TC actions which might lead to inconsistent states or violations of essential WSN properties (e.g., network connectedness). Nevertheless, existing approaches assume that TC operates on a consistent snapshot of the entire topology thus abstracting from realistic environment behaviors and concurrency aspects [12].

Figure 1 shows four stages in the evolution of a sample WSN topology, starting from topology T_0 (Fig. 1a). We denote the current physical distance between sensor nodes by edges either being labeled *short* or *long* (`S,L`). From stage T_0

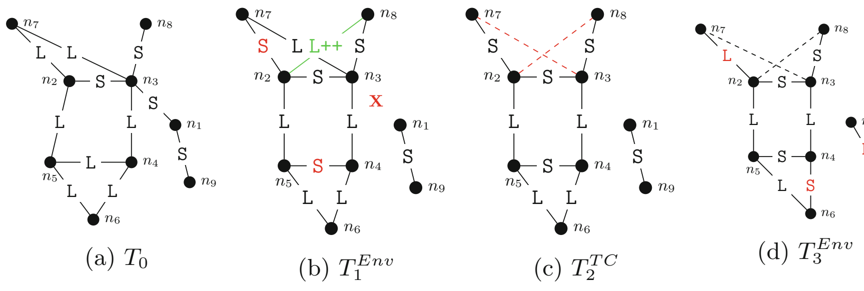
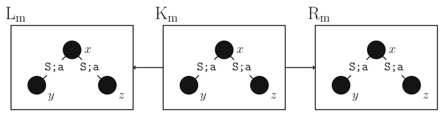


Fig. 1. WSN example: topology control and environmental changes

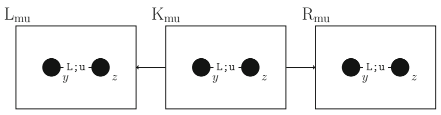
to T_1^{Env}, the environment causes the following changes: edges n_2n_7 and n_4n_5 become short (indicated by the new labels in red), a new long edge n_2n_8 appears (in green with label `L++`), and edge n_1n_3 falls out due to communication failure or insufficient range (indicated by a red cross), causing the topology to lose *connectedness*. In the next stage T_2^{TC}, a TC action *inactivates* the presumably redundant long edges n_3n_7 and n_2n_8 (dashed edges) as both can be replaced by an alternative path via two short edges with better quality measures [12]. Besides, no further long edges exist in the current topology matching this pattern. In the next stage T_3^{Env}, the environment changes n_1n_9 and n_2n_7 into long edges while n_4n_6 becomes short. The change of n_1n_9 does not lead to any TC action, whereas the change of n_2n_7 poses a problem now as the path which previously caused the inactivation of n_3n_7 is not short anymore. In contrast, edges n_4n_5 and n_4n_6 now provide a short path enabling inactivation of edge n_5n_6. In a realistic setting, the change of the distance of n_2n_7 might even cause a *deadlock*: if the TC inactivation action is performed concurrently to the environmental changes, the two short edges required for inactivation are not present anymore. To summarize, TC processes should fulfill the following basic *correctness properties*:

- **(P1a)** Redundancy reduction: TC should eventually inactivate long edges for which there are alternative paths via 2 short edges.
- **(P1b)** Connectedness preservation: TC must preserve connectedness of input-topology graphs.
- **(P2)** Liveness: TC must not deadlock due to concurrent interactions with environmental behaviors.

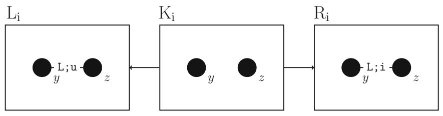
We next present a specification of TC actions as *graph-rewriting rules* to investigate those properties in more detail. As shown in Fig. 1, we use graphs for representing WSN topologies, with nodes denoting sensors (names given as single letters) and edges denoting bidirectional communication links via (volatile) channels. In addition to length attributes `S` (short) and `L` (long), a further edge attribute represents link status (*active*, *inactive* or *unclassified*) thus leading to six possible edge types `S;a, S;i, S;u, L;a, L;i` and `L;u`, respectively. Status *active* indicates that the link is currently used for communication, whereas



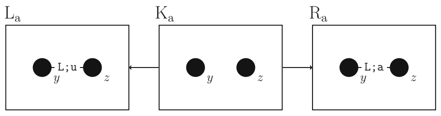
(a) Rule ρ_m: Match Two Short Edges



(b) Rule ρ_{mu}: Find Long `u`-Edge



(c) Rule ρ_i: Inactivate Long Edge

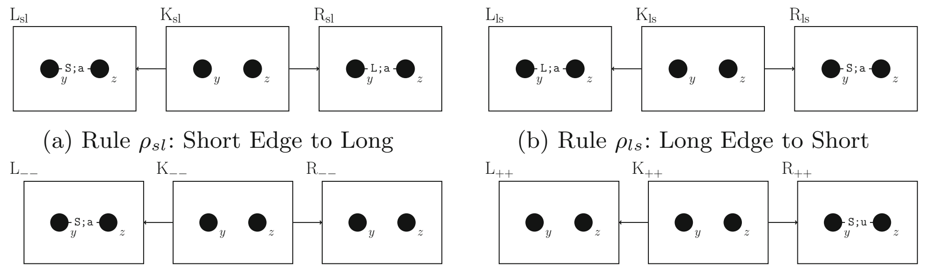


(d) Rule ρ_a: Activate Long Edge

Fig. 2. Graph-rewriting rules for topology-control actions

inactive links are currently not in use. Edges with status *unclassified* are in use but require status revision, usually due to environmental changes. Based on this model, the DPO graph-rewriting rules in Fig. 2a–d specify (simplified) TC actions. A DPO-rule ρ consists of three graphs, where L is called *left-hand side*, R *right-hand side* and K *interface*. Graphs L and K specify *deletion* of those parts of L not appearing in K while R and K specify *creation* of those parts of R not appearing in K. An *application* of rule ρ on input graph G consists in (1) finding an occurrence (*match)* of L in G, (2) deleting elements of G according to L and K, and (3) creating new elements in G according to R and K, thus, yielding an *output graph* H. As a common restriction in the WSN domain, nodes in a WSN topology graph are only visible to their direct neighbor nodes thus limiting granularity of graph patterns within graph-rewriting rules, accordingly. For instance, rule ρ_m in Fig. 2a matches two neighboring active short edges (indicated by the label `S;a`) in topology graph G without performing any changes to G. Similarly, rule ρ_{mu} (Fig. 2b) matches a long unclassified (`L;u`) edge. The actual adaptations (rewritings) of the topology graph following those rule applications are performed by rule ρ_i and rule ρ_a (Fig. 2c–d), by inactivating or activating an (`L;u`) edge, respectively. A TC algorithm based on these rules defines a non-terminating TC process repeatably triggered by environmental changes of links as modeled by the rules in Fig. 3. In Fig. 3a–b, we only present the rules for status `a` and in Fig. 3c–d for edge type `S;a` (`S;u`); whereas the complete specification includes 14 such rules (six for length change as well as six for deletions and two for creations of links).

TC continuously searches for active short edge pairs (ρ_m) as well as long unclassified edges (ρ_{mu}), and then *either* inactivates a long unclassified edge (ρ_i) *if it forms a triangle exactly containing the previously matched short edges and the long edge or* it activates such an edge (ρ_a), otherwise. However, a TC algorithm specified in a purely declarative manner by means *uncontrolled* graph-rewriting rule applications is not sufficient for faithfully reasoning about the aforementioned properties. Hence, we conclude four major challenges to be addressed by a comprehensive *controlled* graph-rewriting formalism as will be presented in the following.



(a) Rule ρ_{sl}: Short Edge to Long (b) Rule ρ_{ls}: Long Edge to Short

(c) Rule ρ_{--}: Delete Edge (d) Rule ρ_{++}: Create Edge

Fig. 3. Graph-rewriting rules for the environment

(C1: Control-Flow Specification) Control-flow constructs for graph-rewriting processes, namely sequential-, negated-, conditional-, and iterative rule composition to restrict *orderings* of graph-rewriting rule applications.

(C2: Subgraph Binding) Data-flow constructs for propagating subgraph bindings among rules to restrict *matches* of graph-rewriting rule applications (cf. dependencies between rule ρ_m and the subsequent rules in Fig. 2).

(C3: Concurrency) Control-flow and data-flow constructs for concurrent rule composition and synchronization of graph-rewriting processes being compatible with solutions for **C1** and **C2**.

(C4: Automated Analysis) Techniques for automated analysis of correctness properties for algorithms specified as controlled graph-rewriting processes including **C1**, **C2**, **C3**.

3 Foundations

We first recall basic notions of graph rewriting based on category theory [5].

Definition 1 (Graphs, Typed Graphs). *A* (directed) graph *is a tuple* $G = \langle N, E, s, t\rangle$*, where* N *and* E *are sets of* nodes *and* edges*, and* $s, t : E \to N$ *are the* source *and* target functions*. The components of a graph* G *are denoted by* N_G*,* E_G*,* s_G*,* t_G*. A* graph morphism $f : G \to H$ *is a pair of functions* $f = \langle f_N : N_G \to N_H, f_E : E_G \to E_H\rangle$ *such that* $f_N \circ s = s' \circ f_E$ *and* $f_N \circ t = t' \circ f_E$*; it is an* isomorphism *if both* f_N *and* f_E *are bijections. We denote by* **Graph** *the category of graphs and graph morphisms.*

A *typed graph* is a graph whose elements are "labelled" over a fixed *type graph* TG. The category of *graphs typed over* TG is the slice category (**Graph** $\downarrow TG$), also denoted $\mathbf{Graph}_{TG}$ [4].

Definition 2 (Typed Graph-Rewriting System). *A* (TG-typed graph) DPO rule *is a span* $(L \xleftarrow{l} K \xrightarrow{r} R)$ *in* $\mathbf{Graph}_{TG}$*. The typed graphs* L*,* K*, and* R *are called* left-hand side, interface*, and* right-hand side*. A* (TG-typed) graph-rewriting system *is a tuple* $\mathcal{G} = \langle TG, \mathcal{R}, \pi\rangle$*, where* $\mathcal{R}$ *is a set of* rule names *and* π *maps rule names in* $\mathcal{R}$ *to rules.*

We write $\rho : (L \xleftarrow{l} K \xrightarrow{r} R)$ for rule $\pi(\rho)$ with $\rho \in \mathcal{R}$. A *rule application* of rule ρ on input graph G amounts to a construction involving two pushouts by (1) finding a *match* of L in G, represented by morphism m, (2) choosing a pushout complement $K \xrightarrow{k} D \xrightarrow{f} G$ for $K \xrightarrow{l} L \xrightarrow{m} G$, representing the application interface, and (3) creating a pushout $R \xrightarrow{n} H \xleftarrow{g} D$ over $K \xrightarrow{k} D \xrightarrow{f} G$, yielding *output graph* H by deleting and creating elements of G according to the rule span and match m. By $G \overset{\rho@m}{\Longrightarrow} H$ we denote application of rule ρ on input graph G at match m, producing output graph H.

We next revisit *coloured Petri nets* (*CPN*), constituting a conservative extension of plain *Petri nets* (*PN*). To this end, we employ the notations of *multisets*.

Definition 3 (Multiset). *A* multiset *ranged over a non-empty set S is a function $M : S \to \mathbb{N}$ for which the following holds.*

- $M(s)$ *is the* coefficient (number of appearances) *of $s \in S$ in M,*
- $s \in M \Leftrightarrow M(s) > 0$ (membership)*,*
- $\mid M \mid = \sum_{s \in S} M(s)$ (size)*,*
- $(M_1{+}{+}\, M_2)(s) = M_1(s) + M_2(s)$ (summation)*,*
- $M_1 \lll= M_2 \Leftrightarrow \forall s \in S : M_1(s) \leq M_2(s)$ (comparison)*,*
- $(M_1{-}{-}\, M_2)(s) = M_1(s) - M_2(s)$ *iff* $M_2 \lll= M_1$ (subtraction)*, and*
- $(n{*}{*}\, M)(s) = n * M(s)$ (scalar multiplication)

The symbol $\sum^{++}$ is defined on multisets according to summation ++. By $\mathbb{N}^S$ we refer to the *set of all multisets* over set S and by $M_\emptyset$ we denote the *empty multiset* (i.e., $M_\emptyset(s) = 0$ for each $s \in S$). By *i's* we refer to the ith appearance of element s in multiset M, where $1 \leq i \leq M(s)$. As depicted, for instance, in Fig. 6, a PN consists of a set of *places* (circles), a set of *transitions* (rectangles) and a set of (directed) *arcs* (arrows) each leading from a place to a transition or vice versa. *Markings* of places are depicted as filled circles within places.

Definition 4 (Petri Net). *A* Petri net *is a tuple (P, T, A, M_0), where P is a finite set* (places)*, T is a finite set* (transitions) *such that $P \cap T = \emptyset$, $A \subseteq (P \times T) \cup (T \times P)$ is a set* (directed arcs)*, and $M_0 \in \mathbb{N}^P$ is a multiset* (initial marking)*. We use the notations ${}^\bullet t = \{p \in P \,|\, pAt\}$* (preplaces) *and $t^\bullet = \{p \in P \,|\, tAp\}$* (postplaces) *for $t \in T$ as well as ${}^\bullet p = \{t \in T \,|\, tAp\}$ and $p^\bullet = \{t \in T \,|\, pAt\}$ for $p \in P$, respectively. These notations naturally extend to multisets $X : P \cup T \to \mathbb{N}$ by ${}^\bullet X := \sum_{x \in S \cup T} X(x){*}{*}{}^\bullet x$ and $X^\bullet := \sum_{x \in S \cup T} X(x){*}{*}x^\bullet$.*

Transition $t \in T$ is *enabled* by marking $M \in \mathbb{N}^P$ if all preplaces of t are marked by at least one token. For each *occurrence* of enabled transition t in a *step*, one token is consumed from all preplaces and one token is added to each postplace of t.

Definition 5 (PN Step). *Let N be a PN and $M \in \mathbb{N}^P$ be a* marking*, then $M \xrightarrow{X} M'$ with non-empty $X \in \mathbb{N}^T$ and $M \in \mathbb{N}^P$ is a* step *of N iff (1) ${}^\bullet X \lll= M$* (enabledness) *and (2) $M' = (M{-}{-}\,{}^\bullet X){+}{+}\,X^\bullet$* (occurrence)*.*

We may omit X as well as M' in denoting steps $M \xrightarrow{X} M'$ if not relevant.

Definition 6. *Let N be a Petri net.*

- *Marking M' is* reachable *from marking M in N iff there exists a sequence $M \rightarrow M'' \rightarrow \cdots \rightarrow M'$. By $\mathcal{R}_N(M) \subseteq \mathbb{N}^P$ we denote the set of markings reachable from M in N.*
- *Two transitions $t, u \in T$ are* concurrent*, denoted $t \smile u$, iff $\exists M \in \mathcal{R}_N(M_0) : M \xrightarrow{\{t,u\}} M'$.*
- *N is k-*bounded *iff $\forall M \in \mathcal{R}_N(M_0) : \forall p \in P : M(p) \leq k$. A 1-bounded Petri net is also called* 1-safe*.*
- *N is* live *iff $\forall M \in \mathcal{R}_N(M_0) \setminus \{M_h\} : M \rightarrow$*

Coloured Petri nets (CPN) extend PN by assigning typed data (*colours*) to tokens and by augmenting net elements with *inscriptions* denoting conditions and operations involving typed variables over token data. Let Σ be a finite non-empty set *(types or colours)*, V a finite set *(variables)* and $\mathit{Type} : V \rightarrow \Sigma$ a function *(type declaration)*. By INSCR_V we denote the set of *inscriptions* over V and by $\mathit{Type}[v]$ we denote the set of values of variable v. Subscript MS as in $C(p)_{MS}$ denotes the object (e.g., $C(p)$) to be a multiset.

Definition 7 (Coloured Petri Net). *A* Coloured Petri Net (CPN) *is a tuple $(P, T, A, \Sigma, V, C, \Gamma, E, \Xi)$, where*

- *P is a finite set* (places)*,*
- *T is a finite set* (transitions) *such that $P \cap T = \emptyset$,*
- *$A \subseteq (P \times T) \cup (T \times P)$ is a set* (directed arcs)*,*
- *Σ is a finite, non-empty set* (colours)*,*
- *V is a finite set* (typed variables) *such that $\forall v \in V : \mathit{Type}[v] \in \Sigma$,*
- *$C : P \rightarrow \Sigma$ is a function* (place colours)*,*
- *$\Gamma : T \rightarrow \mathit{INSCR}_V$ is a function* (transition guards) *such that $\forall t \in T : \mathit{Type}[\Gamma(t)] = \mathit{Bool}$,*
- *$E : A \rightarrow \mathit{INSCR}_V$ is a function* (arc expressions) *such that $\forall a \in A : \mathit{Type}[E(a)] = C(p)_{MS}$ with $p \in P$ being the place related to a,*
- *$\Xi : P \rightarrow \mathit{INSCR}_\emptyset$ is a function* (initialization expressions) *such that $\forall p \in P : \mathit{Type}[\Xi(p)] = C(p)_{MS}$.*

CPN steps extend PN steps by *binding elements* (t, b) of transitions $t \in T$, where b is a function *(binding)* such that $\forall v \in \mathit{Var}(t) : b(v) \in \mathit{Type}[v]$ (i.e., assignments of values to variables occurring in inscriptions of the transition according to the type of the variable). By $B(t)$, we denote the set of all bindings of $t \in T$ and by BE, we denote the set of all binding elements of CPN $\mathcal{N}$. By $\langle \mathit{var}_1 = \mathit{value}_1, \mathit{var}_2 = \mathit{value}_2, \ldots \rangle$, or $\langle b \rangle$ for short, we denote value assignments to variables in binding b. Furthermore, by $\mathit{Var}(t)$, we denote the set of *free variables* of $t \in T$ (i.e., $v \in \mathit{Var}(t)$ either if v occurs in guard $\Gamma(t)$ of t or in inscription $E(a)$ of some arc $a \in A$ having t as source or target).

Definition 8 (CPN Step). *Let $\mathcal{N}$ be a CPN and $\mathcal{M} : P \to \mathbb{N}^{\Sigma}$ a* coloured marking*, then $\mathcal{M} \overset{Y}{\Rightarrow} \mathcal{M}'$ with $Y \in \mathbb{N}^{BE}$ and $\mathcal{M} : P \to \mathbb{N}^{\Sigma}$ is a* step *of $\mathcal{N}$ iff (1) $\forall (t,b) \in Y : \Gamma(t)\langle b\rangle$ and $\forall p \in P : \sum^{++}_{(t,b)\in Y} E(p,t)\langle b\rangle \ll= \mathcal{M}(p)$* (enabledness) *and (2) $\forall p \in P : \mathcal{M}'(p) = (\mathcal{M}(p) -- \sum^{++}_{(t,b)\in Y} E(p,t)\langle b\rangle) ++ \sum^{++}_{(t,b)\in Y} E(t,p)\langle b\rangle$* (occurrence).

As an example, Fig. 4 shows a basic CPN with one transition, whose inscription will be descried in the next section. The passing of input/output values in a CPN step is handled by arc and guard variables: variable G is bound to the value of input token 1‘**G**, and the produced output token is being bound to the value of variable H.

Coloured marking $\mathcal{M}'$ is *reachable* from $\mathcal{M}$ in $\mathcal{N}$ iff there exists a sequence of CPN steps from $\mathcal{M}$ to $\mathcal{M}'$, denoted as $\mathcal{M} \Rightarrow \mathcal{M}'' \Rightarrow \cdots \Rightarrow \mathcal{M}'$. By $\mathcal{R}^C_{\mathcal{N}}(\mathcal{M})$ we denote the set of coloured markings reachable from $\mathcal{M}$ in $\mathcal{N}$. The set of *reachable coloured markings* of $\mathcal{N}$ is denoted as $\mathcal{R}^C_{\mathcal{N}}(\mathcal{M}_0)$ where the *initial coloured marking* $\mathcal{M}_0$ of $\mathcal{N}$ is defined as $\forall p \in P : \mathcal{M}_0(p) = I(p)\langle\rangle$. Thereupon, all further PN notions of Definition 6 are adaptable to CPN, accordingly.

4 Graph-Rewriting Petri Nets

We now introduce *graph-rewriting Petri nets* (*GPN*) as an instantiation of CPN. To this end, we define *graph-rewriting inscriptions* to specify controlled graph-rewriting processes as GPN.

4.1 GPN Syntax

In GPN, token colours represent objects (i.e., graphs) and morphisms, respectively, from category $\mathbf{Graph}_{TG}$ and transitions correspond to DPO rule applications. Although the distinction between graphs and morphisms is not really necessary from a categorical point of view (as objects may be represented as identity morphisms), it is useful to handle *subgraph bindings* as first-class entity in GPN being distinguished from complete graphs when specifying inputs and outputs of GPN transitions. Notationally, colour set

$$\Sigma_{TG} = \{Obj(\mathbf{Graph}_{TG}), Mor(\mathbf{Graph}_{TG})\}$$

consists of two colours: objects (denoted in capital letters) and morphisms (denoted in non-capital letters) in $\mathbf{Graph}_{TG}$. In the following, we use the same typing for elements of $\mathbf{Graph}_{TG}$) in an obvious way for convenience.

By $INSCR_{TG,V}$, we denote the set of *graph-rewriting inscriptions* over a set of variables V, typed over Σ_{TG}. Transition guards in GPN are given as graph-rewriting inscriptions representing templates for diagrams in $\mathbf{Graph}_{TG}$ having a *bound* part as well as *free* variables thus expressing DPO rule applications potentially involving subgraph bindings. In an inscription I, the diagram template containing variable names is given by a finite category $\mathbf{FV_I}$, where the bound part (i.e., elements already fixed in $\mathbf{Graph}_{TG}$) is given by a binding functor $B_I : \mathbf{BV_I} \rightarrow \mathbf{Graph}_{TG}$ with $\mathbf{BV_I}$ being a finite category. The structure of the diagram is given by the type of the variables as well as further structural properties of the inscription (cf. Φ_I in the next paragraph). The distinction between bound and free names directly corresponds to the respective notion of variables in CPN such that firing a GPN transition t consists in binding free variables of t by extending the binding functor to a complete diagram. Bound elements (from $\mathbf{BV_I}$) are denoted by bold letters (e.g., $\mathbf{L}, \mathbf{r}$) and free variables by sans-serif letters (e.g., G, b).

In addition, a proper graph-rewriting inscription has to involve further categorical properties to hold for the diagram for yielding the intended DPO semantics, including, for instance, commutativity of arrows, pushout squares, as well as non-existence or partiality of morphisms. We refer to this set of properties of inscriptions I by Φ_I. For instance, when utilizing the DPO diagram in Fig. 4b as transition guard I_ρ in which span $(\mathbf{L} \xleftarrow{\mathbf{l}} \mathbf{K} \xrightarrow{\mathbf{r}} \mathbf{R})$ is mapped by B_{I_ρ} to concrete graphs and morphisms according to rule ρ, Φ_{I_ρ} requires both rectangles to be pushouts (and thus to commute), indicated by (PO). In a concurrent setting, GPN inscriptions will be further generalized to compound operations with potentially multiple graphs and subgraph bindings involved. Adapting step semantics of CPN to GPN means that (inscription) *bindings* are functors mapping variable set $\mathbf{FV_I}$ into $\mathbf{Graph}_{TG}$ in a way that conforms to B_I thus completing the bound parts of the inscription.

Definition 9 (Graph-Rewriting Inscription). *A* graph-rewriting inscription *$I \in INSCR_{TG,V}$ is based on the following components:*

(i) a finite category of bound *variables $\mathbf{BV_I}$ with $v \in \mathbf{BV_I} \Rightarrow v \in V$,*
(ii) a finite category of free *variables $\mathbf{FV_I}$ with $v \in \mathbf{FV_I} \Rightarrow v \in V$, $\mathbf{BV_I} \subseteq \mathbf{FV_I}$,*
(iii) a binding functor $B_I : \mathbf{BV_I} \rightarrow \mathbf{Graph}_{TG}$ such that $\forall v \in \mathbf{BV_I} : Type(v) = Type(B_I(v))$ and
(iv) a set of categorical properties Φ_I for (the images of) $\mathbf{FV_I}$.

A binding *B for I is a functor $B : \mathbf{FV_I} \rightarrow \mathbf{Graph}_{TG}$ such that $B \mid_{\mathbf{BV_I}} = B_I$ and $\forall v \in \mathbf{FV_I} : Type(v) = Type(B(v))$. $I\langle B \rangle$ evaluates to true if $B(\mathbf{FV_I})$ satisfies Φ_I.*

A GPN defined over type graph TG is a CPN over Σ_{TG} with inscriptions $I \in INSCR_{TG,V}$.

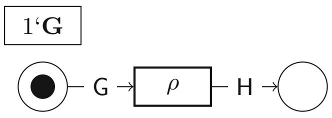
Definition 10 (Graph-Rewriting Petri Net). *A* graph-rewriting Petri net *(*GPN*) over type graph TG is a CPN with set of colours Σ_{TG}, transition guard function Γ and arc expression function E ranged over $INSCR_{TG,V}$, initial marking function Ξ ranged over $INSCR_{TG,\emptyset}$.*

We further require each transition of a GPN to consume and produces at least one graph token (i.e., having non-empty sets of pre- and postplaces).

4.2 GPN Semantics

We first illustrate GPN semantics intuitively by describing the GPN pattern corresponding to a basic DPO rule application. Then, we provide a generic pattern for GPN transitions. To avoid bloat of notation, we simplify transition guards using a compact graphical notation.

DPO Rule Application. The most basic GPN pattern in Fig. 4 consists of a transition applying rule $\rho : (L \xleftarrow{l} K \xrightarrow{r} R)$ to input graph G, producing output graph H (both typed over TG). We denote the GPN transition guard simply



(a) Net Structure: DPO Rule Application with Input Graph G

$$\begin{tikzcd}
\mathbf{L} \arrow[d, "\mathsf{m}"'] & \mathbf{K} \arrow[l, "\mathbf{l}"'] \arrow[r, "\mathbf{r}"] \arrow[d, "\mathsf{k}"] \arrow[dl, phantom, "(PO)"] & \mathbf{R} \arrow[d, "\mathsf{n}"] \arrow[dl, phantom, "(PO)"] \\
\mathsf{G} & \mathsf{D} \arrow[l, "\mathsf{f}"] \arrow[r, "\mathsf{g}"'] & \mathsf{H}
\end{tikzcd}$$

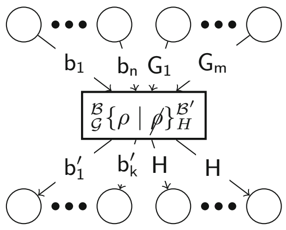
(b) Guard ρ: DPO Application of Rule ρ

Fig. 4. Basic GPN pattern: DPO rule application

by ρ which is satisfied if $G \overset{\rho @ m}{\Longrightarrow} H$ for some match m (cf. Definition 2ff). In GPN terms, guard ρ is satisfied if there is a respective inscription binding such that the complete diagram becomes a DPO rule application (indicated by (PO) as imposed in Φ_ρ).

Hence, input variable G is bound to the graph corresponding to the input token ($1'\mathbf{G}$ denoting a multiset consisting of a single graph $\mathbf{G}$) and the output token corresponds to the output graph bound to variable H. Sequential compositions of this basic GPN pattern specifies controlled sequences of DPO rule applications (incl. loops) restricted to the given ordering, as well as control-flow branches (choices) and joins (i.e., places with multiple outgoing/incoming arcs).

Generic GPN Transition Pattern. Figure 5 shows the most general form (net structure and transition guard) of GPN transitions including *concurrent* behaviors. As represented by the transition label in Fig. 5a, each GPN transition either denotes an application of a rule ρ or a *non-applicability condition* of ρ (i.e., requiring the left-hand side of ρ to *not* have any match in G, where output graph H is omitted in the transition label as G remains unchanged). The set of preplaces of transitions might include a (non-empty) set $\mathcal{G}$ of (simultaneously consumed) input graphs $G_1, G_2, \ldots, G_l$ to each of which either ρ is applied or a non-applicability condition is imposed. Application of rule ρ to a set of input graphs is performed by means of a non-deterministic *merge* as follows: (1) selecting (non-deterministically) a family of injective arrows $G_\cap \xrightarrow{e_l} G_l$, $l = 1 \ldots m$ from an arbitrary graph $G_\cap$ representing a common overlapping of each input graph, and (2) creating a co-limit object over that family in the category of typed


b1
bn
G1
Gm
$^{\mathcal{B}}_{\mathcal{G}}\{\rho \mid \not\rho\}^{\mathcal{B}'}_{H}$
b'1
b'k
H
H

(a) Generic Transition

$$\begin{tikzcd}
 & G_{\cap} \arrow[dl, "e_1"'] \arrow[dr, "e_m"] & \\
G_1 \arrow[dr] & \bullet\bullet\bullet & G_m \arrow[dl] \\
 & G &
\end{tikzcd}$$

(b) Generic Guard: Graph Merge

$$\begin{tikzcd}
\overset{i = 1 \ldots n}{\mathbf{S}_i} \arrow[r, "\mathbf{s}_i^{\mathbf{L}}"] \arrow[d, leftrightarrow, "\mathsf{i}_i^{\mathsf{L}}"'] \arrow[dr, phantom, "(=)"] & \mathbf{L} \arrow[d, "\mathsf{m}"] \arrow[dr, phantom, "(PO)"] & \mathbf{K} \arrow[l, "\mathbf{l}"'] \arrow[r, "\mathbf{r}"] \arrow[d, "\mathsf{k}"] \arrow[dr, phantom, "(PO)"] & \mathbf{R} \arrow[d, "\mathsf{n}"] \arrow[dr, phantom, "(=)"] & \overset{j = 1 \ldots k}{\mathbf{S}'_j} \arrow[l, "\mathbf{s}_j^{\mathbf{R}}"'] \arrow[d, leftrightarrow, "\mathsf{i}_j^{\mathsf{R}}"] \\
\mathsf{B}_i \arrow[r, "\mathsf{b}_i^{\mathsf{L}}"] \arrow[d, "\mathsf{b}_i"'] \arrow[dr, phantom, "(=)"] & \mathsf{G} & \mathsf{D} \arrow[l, "\mathsf{f}"'] \arrow[r, "\mathsf{g}"] & \mathsf{H} & \mathsf{B}'_j \arrow[l, "\mathsf{b}'_j"'] \\
\mathsf{G}'_i \arrow[ur, dashed, "\mathsf{p}"'] & & & &
\end{tikzcd}$$

(c) Generic Guard: Rule Application with Subgraphs

Fig. 5. Generic pattern for GPN transitions

graphs, as indicated in Fig. 5b. This diagram is part of the transition guard and the merged graph G is used as input for applying ρ to produce output H. Correspondingly, the set of postplaces might include *forks* into multiple (at least one) concurrent instances of H.

Subgraph Binding and Matching. In addition to control-flow dependencies restricting the *orderings* of rule applications, GPN further allow to express data-flow dependencies among rule applications for restricting the *matches* to which particular rules are to be applied in a given input graph (cf. e.g., Fig. 2, rules ρ_m and ρ_i (ρ_a)). GPN therefore allow to propagate parts of output graphs from preceding rule applications (*subgraph binding*) to be matched within input graphs of subsequent rule applications (*subgraph matching*) via additional tokens coloured by respective matching morphisms. To this end, GPN transitions are equipped with *synchronization interfaces* $R_1 \xleftarrow{s_R} S \xrightarrow{s_L} L_2$ between sequentially applied rules ρ_1 and ρ_2 to be a span relating the right-hand side of ρ_1 with the left-hand side of ρ_2. In Sect. 2, the rules ρ_i and ρ_a require left-hand side synchronization interfaces to ρ_m (cf. Fig. 2) in order to ensure that exactly those edges are matched for a node pair which have been previously matched by ρ_m. This concept is, however, not limited to GPN net structures with sequentially applied rules, but rather generalize to any kind of rule composition. In Fig. 5a, $\mathcal{B}'$ denotes the (possibly empty) set of outgoing *subgraph binding* morphisms, while

$\mathcal{B}$ denotes the (possibly empty) set of incoming *subgraph matching* morphisms of a GPN transition. The rightmost rectangle in Fig. 5c represents *subgraph binding* where the rule-application structure is extended by right-hand side synchronization interfaces of ρ (for some subsequent rule(s)). For each right-hand interface S'_j (indexed by $j = 1 \dots k$), the application of ρ produces an additional token $\mathsf{b'}_j$ coloured as morphism for that subgraph of H which is (1) within the co-match of the rule application and (2) corresponds to that part of the co-match designated by $S'_j \xrightarrow{s_j^R} R$. This property is captured by the commutation of the arrows $n \circ s_j^R$ and $b'_j \circ i_j^R$, denoted by (=). The bidirectionality of i_j^R denotes that i_j^R is an isomorphism. Conversely, the leftmost part (rectangle and triangle) of Fig. 5c represents *subgraph matching* for left-hand side synchronization interfaces (indexed by $i = 1 \dots n$, where an application of rule ρ additionally consumes a token coloured as morphism to be checked for compatibility with the input graph. The additional commutative diagram expresses that (1) the bound part B_i of the previous graph G'_i should be also present in the current input graph G (*partial* morphism p commuting with b in the lower triangle) and (2) the occurrence b_i^L of B_i in G should be isomorphic to the left-hand side synchronization interface (the leftmost commutative rectangle). Here, a *partial* graph morphism p between G' and G is a graph morphism $p : G_0 \to G$ where $\exists g : G_0 \to G'$ injective.

Properties of GPN Processes. We utilize the notion of reachable coloured markings of CPN (cf. Sect. 3) to characterize semantic correctness properties on GPN models as described in Sect. 2. In particular, we are interested in *completed* markings only consisting of tokens being coloured over $Obj(\mathbf{Graph}_{TG})$, whereas markings additionally containing tokens coloured over $Mor(\mathbf{Graph}_{TG})$ correspond to intermediate steps of GPN processes.

Definition 11 (GPN Language). *A marking $\mathcal{M}$ of GPN $\mathcal{N}$ is* completed *if for all $p \in P$ with $\mathcal{M}(p) \neq \emptyset$ it holds that $C(p) = Obj(\mathbf{Graph}_{TG})$. The* language *of $\mathcal{N}$ for initial completed marking $\mathcal{M}_0$ is defined as $\mathcal{L}(\mathcal{N}, \mathcal{M}_0) = \mathcal{R}_{\mathcal{N}}^{C\checkmark}(\mathcal{M}_0)$, where $\mathcal{R}_{\mathcal{N}}^{C\checkmark}(\mathcal{M})$ denotes the subset of completed markings reachable from marking $\mathcal{M}$.*

This definition characterizes GPN semantics in terms of input/output correspondences according to the definition of the graph language as the set of graphs generated by a set of rules applied to a given start graph [5]. Note that GPN is able to simulate the standard semantics of plain graph-rewriting systems, i.e., sets of rules: For a given rule set, such a GPN canonically consists of a single place (initially holding a single start graph token) and one transition for every rule, each having this place as its pre- and postplace.

Thereupon, a more elaborated process-centric characterization of GPN semantics might employ a labeled transition system (LTS) representation.

Definition 12 (GPN Processes). *The* LTS *of GPN $\mathcal{N}$ for initial completed marking $\mathcal{M}_0$ is a tuple $(S, s_0, \to, \mathcal{P})$, where $S = \mathbb{N}^{\Sigma_{TG}}$* (processes)*, $s_0 = \mathcal{M}_0$*

(initial process), $\mathcal{M} \xrightarrow{Y} \mathcal{M}'$ *iff* $\mathcal{M} \overset{Y}{\Rightarrow} \mathcal{M}'$ (labeled transitions) *and* $\mathcal{P} = \{pred \mid pred : Obj(\mathbf{Graph}_{TG}) \rightarrow Bool\}$ (predicates).

Predicates $pred \in \mathcal{P}$ might be used to define (computable) correctness properties $pred : Obj(\mathbf{Graph}_{TG}) \rightarrow Bool$ to be checked on graphs in reachable completed markings (e.g., connectedness as described in Sect. 2). In the next section, we provide an elaborated case study illustrating the intuition of the above definitions, by means of a GPN model for our WSN running example (cf. Sect. 2).

5 Case Study

We now demonstrate how GPN address the challenges in controlled graph-rewriting by revisiting our WSN and TC case study from Sect. 2.

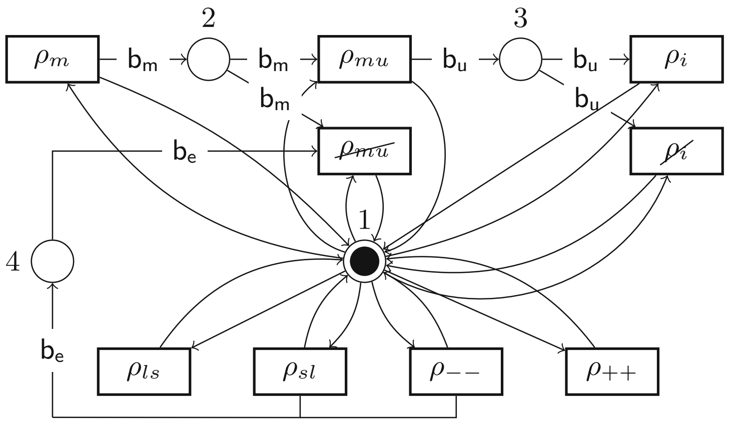


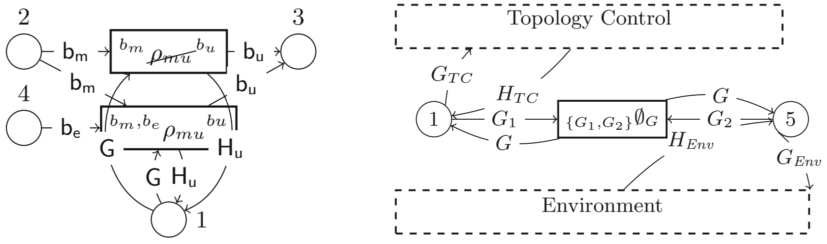
Fig. 6. GPN model for the WSN and TC scenario

Control-Flow Specification (C1). A possible GPN specification of the WSN and TC scenario is shown in Fig. 6, where we omit arc labels denoting graph variables (but still include binding variables like $\mathsf{b_m}$) and we annotate places by numbers 1–4 for the sake of clarity. The environment behavior affecting the WSN topology graph comprises the following actions (lower part of the GPN): ρ_{ls} turns long edges into short, ρ_{sl} short edges into long, ρ_{--} deletes an edge while ρ_{++} creates a fresh (unclassified) edge. Moreover, the rules ρ_{sl} and ρ_{--} propagate their matches as subgraph bindings to the TC process (upper part of the GPN) as those changes influence TC decisions. This specification represents an idealistic (i.e., *synchronous*) model of TC, as all actions (TC as well as environmental) are assumed to be atomic (i.e., non-interfered and instantaneously globally visible). Thus, each transition/rule application consumes the global token (current topology) at place 1 and immediately produces a token (evolved/adapted topology) to the same place.

Subgraph Matching/Binding (C2). Regarding the TC process, transition ρ_m first matches a pair of neighboring short edges in the current graph and passes this match as subgraph binding to place 2 for the next step (using the binding variable $\mathsf{b_m}$). Place 3 plays a similar role for passing the long edge to be inactivated from ρ_{mu} to ρ_i. For those rules, the parallel presence of their positive and negative forms amounts to a deterministic conditional *if-else* construct based on rule applicability. We omitted rule ρ_a in Fig. 2d from the model due to space restrictions. Transition $\cancel{\rho_{mu}}$ handles those case where the subgraph binding previously matched by ρ_m becomes outdated in the current graph due to concurrent changes by the environment (i.e., either one of the matched edges of the edge pair becomes long by action ρ_{sl} or it disappears by action ρ_{--}). However, the model might *deadlock* if ρ_{mu} is not applicable but place 4 has no token. This can be solved by introducing two new transitions as in Fig. 7a (where place numbers correspond to Fig. 6). With that extension, subgraph binding guarantees *liveness* as the new transitions capture the case above.

Concurrency (C3). The idealized (synchronous) GPN specification considered so far above does not yet sufficiently capture the concurrent (or even distributed) semantics aspects of TC for WSN, which shall be reflected (still in a simplified manner) in the GPN in Fig. 7b. Here, the dashed boxes *Topology Control* and *Environment* refer to the GPN components as depicted in upper and lower part of Fig. 6, respectively, such that place 1 retains its role (i.e., all its arcs are combined into $G_{TC}, H_{TC}, G_{Env}, H_{Env}$). As a consequence, both TC and the environment fork the current graph (place 1 and place 5, respectively) while performing their (concurrent and thus potentially conflicting) changes to the topology. Afterwards, the *merge transition* (cf. Fig. 5b) labeled ${}_{\{G_1,G_2\}}\emptyset_G$ (where $\emptyset$ is the empty rule leaving the input graph unmodified) integrates both local graphs into the new global topology graph. In this GPN model, the (non-deterministic) merge transition potentially yields many different subsequent GPN steps as both subprocesses do not synchronize their matches as done in the first GPN model. In a future work, we plan to study different possible trade-offs between both extrema presented here, by considering different possible degrees of granularity of synchronization of subgraph-matches and -merges using GPN. In this regard, the detection of *concurrent transitions* (cf. Definition 6) can be employed to refine the (inherently imprecise) notion of *critical pair analysis* techniques [5] in the context of controlled graph-rewriting semantics.

Automated Analysis (C4). Correctness properties like **P1a-b** and **P2** in Sect. 2 can be specified on the LTS process semantics of GPN using appropriate *predicates* according to Definition 12. For instance, a predicate for property **P1a** holds for topology graph G reached in a marking iff for each *inactive* edge xy, there is a node z with *short* edges xz and yz. Similarly, **P1b** (i.e., connectedness) holds iff for each pair of nodes x and y, there is a path between x and y not containing any inactive edge. Predicates **P1a** and **P1b** constitutes



(a) New Transitions for **P2** (b) Concurrent WSN Model

Fig. 7. Extensions of the WSN and TC model

correctness properties in terms of *invariants* to be checked for any reachable completed marking, whereas **P2** requires further capabilities to express *liveness* properties on the LTS semantics of the GPN model (e.g., using temporal logics). For automated reasoning, we may choose as initial marking any possible initial topology graph G_0 on place 1 to explore the set of reachable markings. In this regard, *1-safety* of the underlying PN of a GPN is of particular interest as most essential PN analysis problems become decidable or computationally cheaper [6]. Here, the GPN model in Fig. 6 is 1-safe due to the centralized structure with place 1, whereas the GPN model in Fig. 7b is unbounded as the environment may continuously produce an arbitrary number of tokens to place 5. Thus, applying existing PN analysis tools to GPN at least allows for exploring *uncoloured* reachable markings (e.g., for over-approximating pairs of potentially concurrent transitions). Concerning reachability analysis of *coloured* markings, CPN Tools constitutes a state-of-the-art simulation and analysis tool for CPN which we currently integrate in our graph-rewriting framework eMoflon in an ongoing work.

To summarize, while certain basic aspects of the case study are directly expressible in other recent modeling approaches (e.g., control flow in Henshin [19] and eMoflon [14], predicate analysis in Groove [8]), neither of those approaches addresses all the challenges **C1-4** (cf. Sect. 2). Crucially, explicit concurrency is considered by none of the existing approaches, whereas Petri nets and therefore also GPN are equipped with a natural notion of true concurrency, thus being particularly well-suited for our purposes.

6 Conclusion and Future Work

We proposed graph-rewriting Petri nets (GPN) to build a comprehensive theoretical foundation for modeling and automated analysis of controlled graph-rewriting processes. In particular, GPN provides a control-flow specification language for graph rewriting processes with a natural notion of concurrency. Additionally, GPN provide an explicit means to handle *sub-graph bindings*, being prevalent in practical applications of controlled graph rewriting (cf. **C1-3** in Sect. 2). Furthermore, GPN constitutes an appropriate basis for automated semantic analysis of controlled graph-rewriting systems (cf. **C4** in Sect. 2).

Our plans for future work are twofold. First, we want to explore further theoretical properties of GPN based on existing (C)PN results (e.g., revised notion of parallel independence in the context of concurrent graph-rewriting processes as well as equivalence notions beyond input/output equivalence). Second, we plan to use GPN as back-end for existing controlled graph-rewriting modeling tools and languages (e.g., SDM in eMoflon [14]) as well as for developing novel modeling approaches directly based on GPN. This would allow us to apply existing analysis tools like CPN Tools [11] for automatically analyzing crucial properties of graph-based algorithms like our WSN case study.

References

1. Baldan, P., Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Löwe, M.: Concurrent semantics of algebraic graph transformation. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 3, pp. 107–187. World Scientific (1999). https://doi.org/10.1142/9789812814951_0003
2. Bunke, H.: Programmed graph grammars. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 155–166. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0025718
3. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Rossi, F.: Abstract graph derivations in the double pushout approach. In: Schneider, H.J., Ehrig, H. (eds.) Graph Transformations in Computer Science. LNCS, vol. 776, pp. 86–103. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57787-4_6
4. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae **26**(3–4), 241–265 (1996). https://doi.org/10.3233/FI-1996-263402
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
6. Esparza, J.: Decidability and complexity of Petri net problems — an introduction. In: Reisig, W., Rozenberg, G. (eds.) ACPN 1996. LNCS, vol. 1491, pp. 374–428. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-65306-6_20
7. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the unified modeling language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46464-8_21
8. Ghamarian, A.H., de Mol, M.J., Rensink, A., Zambon, E., Zimakova, M.V.: Modelling and analysis using GROOVE. Int. J. Softw. Tools Technol. Transf. **14**, 15–40 (2012). https://doi.org/10.1007/s10009-011-0186-x
9. Guerra, E., de Lara, J.: Colouring: execution, debug and analysis of QVT-relations transformations through coloured petri nets. Softw. Syst. Model. **13**(4), 1447–1472 (2014). https://doi.org/10.1007/s10270-012-0292-6
10. Hoffmann, K., Mossakowski, T.: Algebraic higher-order nets: graphs and petri nets as tokens. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 253–267. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_14

11. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Science & Business Media, Heidelberg (2009). https://doi.org/10.1007/b95112
12. Kluge, R., Stein, M., Varró, G., Schürr, A., Hollick, M., Mühlhäuser, M.: A systematic approach to constructing families of incremental topology control algorithms using graph transformation. Softw. Syst. Model. 1–41 (2017). https://doi.org/10.1007/s10270-017-0587-8
13. Kreowski, H.-J., Kuske, S., Rozenberg, G.: Graph transformation units – an overview. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 57–75. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68679-8_5
14. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 138–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_10
15. Li, M., Li, Z., Vasilakos, A.V.: A survey on topology control in wireless sensor networks: taxonomy, comparative study, and open issues. Proc. IEEE **101**(12), 2538–2557 (2013). https://doi.org/10.1109/JPROC.2013.2257631
16. Plump, D., Steinert, S.: The semantics of graph programs. In: RULE 2009 (2009). https://doi.org/10.4204/EPTCS.21.3
17. Schürr, A.: Logic-based programmed structure rewriting systems. Fundam. Inf. **26**(3,4), 363–385 (1996). https://doi.org/10.3233/FI-1996-263407
18. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES-approach: language and environment. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 487–550. World Scientific (1999). https://doi.org/10.1142/9789812815149_0013
19. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: a usability-focused framework for EMF model transformation development. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 196–208. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_12
20. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Right or wrong?–verification of model transformations using colored petri nets. In: DSM 2009 (2009)

Parallel Independence and Conflicts

On the Essence and Initiality of Conflicts

Guilherme Grochau Azzi[1](✉), Andrea Corradini[2], and Leila Ribeiro[1]

[1] Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{ggazzi,leila}@inf.ufrgs.br
[2] Università di Pisa, Pisa, Italy
andrea@di.unipi.it

Abstract. Understanding conflicts between transformations and rules is an important topic in algebraic graph transformation. A conflict occurs when two transformations are not parallel independent, that is, when after applying one of them the other can no longer occur. We contribute to this research thread by proposing a new characterization of the root causes of conflicts, called "conflict essences". By exploiting a recently proposed characterization of parallel independence we easily show that the conflict essence of two transformations is empty iff they are parallel independent. Furthermore we show that conflict essences are smaller than the "conflict reasons" previously proposed, and that they uniquely determine the so-called "initial conflicts". All results hold in categories of Set-valued functors, which include the categories of graphs and typed graphs, and several of them hold in the more general adhesive categories.

Keywords: Graph transformation · Double-pushout
Parallel independence · Conflict · Critical pair · Initial conflict

1 Introduction

Graph transformation is a formal model of computation with an intuitive graphical interpretation. Graphs are used to represent states of the system, while possible transitions are represented by transformation rules. The algebraic approach is not restricted to a particular notion of graph: using category theory, its definitions and results can be instantiated for different notions of graph (e.g. labelled, attributed) whose categories satisfy certain axioms.

An important topic in algebraic graph transformation is the study of parallel independence and conflicts. When two transformations are parallel independent they may be applied in any order; if a transformation can no longer happen after applying another, they are in conflict [5]. Understanding conflicts between transformations and rules provides great insight into the behaviour of a transformation system. Indeed, the potential conflicts between two rules, in a minimal context, can be enumerated by *Critical Pair Analysis* (CPA) and used to check local confluence of the system [9]. This has many applications, particularly in Model-Driven Development (e.g. [6,16,17]).

L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 99–117, 2018.
https://doi.org/10.1007/978-3-319-92991-0_7

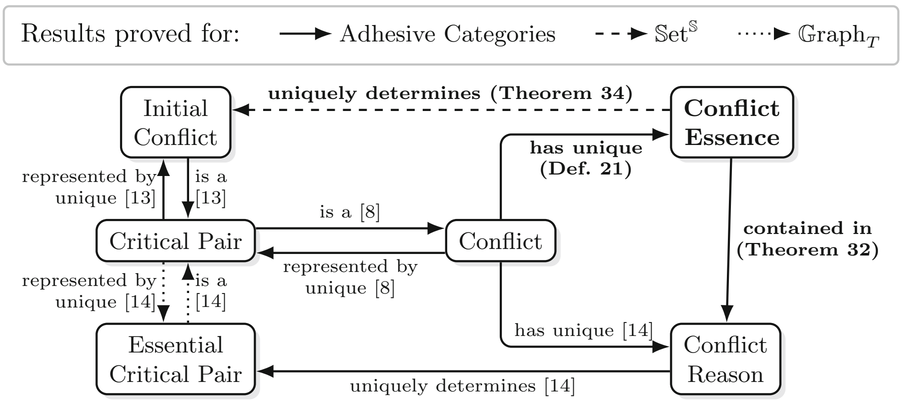


Fig. 1. Overview of conflicts and their root causes, where new concepts and results are in bold.

When two transformations are in conflict, understanding its root causes is often difficult. The formal characterization of *conflict reasons* [14] helps, but it may include elements unrelated to the conflict and lacks a direct connection to the definition of parallel independence. Moreover, critical pairs generated by any two rules are numerous and often redundant, hindering the application of CPA.

An overview of the results presented in this paper and of related concepts from the literature is depicted in Fig. 1. Based on conflict reasons, *essential critical pairs* were proposed as a subset of critical pairs. They were proven complete (for the categories of graphs and typed graphs [14]), in the sense that every critical pair is the embedding of some essential critical pair into a larger context. More recently, *initial conflicts* were also proposed and proven to be a complete subset of critical pairs (in any adhesive category [13]), and were proven to exist for the categories of graphs and typed graphs. Relations between initial conflicts and conflict reasons were not reported, to the best of our knowledge.

In this paper, we contribute to this research thread by proposing a new characterization for the root causes of conflicts, called *conflict essences*, based on a recently proposed characterization of parallel independence [3]. In any adhesive category with strict initial object we show that having an empty conflict essence is equivalent to parallel independence, and that conflict essences are smaller than conflict reasons. In categories of $\mathbb{S}\mathrm{et}$-valued functors, we show that conflict essences uniquely determine initial conflicts. Furthermore, we identify sufficient conditions for this to hold in any adhesive category.

The reader should be familiar with basic concepts of Category Theory and with the DPO approach [5]. Some background notions and a motivating example are introduced in Sect. 2. We define conflict essences in Sect. 3, proving important properties and comparing them to conflict reasons. In Sect. 4 we show that essences uniquely determine initial conflicts, and in Sect. 5 we conclude.

2 Preliminaries

2.1 Algebraic Graph Transformation

In this section we briefly review the basic definitions of algebraic graph transformation, according to the Double-Pushout (DPO) approach [5]. We follow the generalization of DPO to work with objects of any adhesive category [12], which include variations of graphs (typed, labelled, attributed), and several other structures. A more recent and detailed introduction to algebraic graph transformation is available in [8], where the theory is generalized to $\mathcal{M}$-adhesive categories, which also encompass structures like Petri nets and algebraic specifications.

We begin by reviewing the notion of adhesive category, which underlies several other definitions, as well as some of its properties. For proofs and a detailed discussion we refer to [12].

Definition 1 (Adhesive Category). *A category $\mathbb{C}$ is called* **adhesive** *if (i) it has all pullbacks, (ii) it has all pushouts along monos, and (iii) such pushouts are* van Kampen (VK) *squares, which implies that they are preserved and reflected by pullbacks [12].*

Fact 2 (Properties of Adhesive Categories). *Let $\mathbb{C}$ be an adhesive category.*

1. *Pushouts along monos in $\mathbb{C}$ are pullbacks.*
2. *For every object C of C, let $\mathbf{Sub}(C)$ be the partial order of its subobjects, i.e. equivalence classes of monic arrows with target C, where $f : A \rightarrowtail C$ and $g : B \rightarrowtail C$ are equivalent if there is an isomorphism $h : A \to B$ making the triangle commute. Then $\mathbf{Sub}(C)$ is a distributive lattice: intersection of subobjects is obtained as the pullback of the corresponding monos, union is obtained from a pushout over the intersection.*

We proceed by reviewing the basic concepts of DPO rewriting in an arbitrary category $\mathbb{C}$. Assumptions on $\mathbb{C}$ will be made explicit when needed.

Definition 3 (Rule, Match and Transformation). *A* **rule** *ρ is a span $\rho = L \overset{l}{\leftarrowtail} K \overset{r}{\rightarrowtail} R$, with monic l and r. We call L and R the* left- *and* right-hand sides*, respectively, while K is called the* interface*. A* **transformation system** *$\mathcal{G}$ is a finite set of rules.*

A **match** *for a rule ρ in an object G is a monic arrow $m : L \rightarrowtail G$. Given a match $m : L \to G$ for rule ρ, a* **transformation** *$G \overset{\rho,m}{\Longrightarrow} H$ corresponds to a diagram (1), where both squares are pushouts. In an adhesive category, the left pushout is guaranteed to be unique up to isomorphism, if it exists.*

$$\begin{tikzcd} L \arrow[d, tail, "m"'] \arrow[dr, phantom, "\urcorner"] & K \arrow[l, tail, "l"'] \arrow[r, tail, "r"] \arrow[d, tail, "k"] \arrow[dr, phantom, "\ulcorner"] & R \arrow[d, tail, "m'"] \\ G & D \arrow[l, tail, "l'"] \arrow[r, tail, "r'"'] & H \end{tikzcd} \quad (1)$$

In the technical development that follows we will use the notions of *strict initial objects* and of *initial pushouts* [9] that we recall here.

Definition 4 (Strict Initial Object). *An object* $\mathbf{0}$ *is* **initial** *in a category* $\mathbb{C}$ *if for each object* C *of* $\mathbb{C}$ *there is a unique arrow* $!_C : \mathbf{0} \to C$*. An initial object* $\mathbf{0}$ *is* **strict** *if for any arrow* $f : X \to \mathbf{0}$*, the source* X *is also initial. If any initial object is strict, all initial objects are, since they are isomorphic. Furthermore, if* $\mathbf{0}$ *is strict, then every arrow* $!_C : \mathbf{0} \to C$ *is mono.*

Many adhesive categories of interest, including the categories of functors presented in the next section, have a strict initial object, but not all of them: for example, the category of sets and partial functions is adhesive, but the initial object is not strict [12].

Definition 5 (Initial Pushout). *Given a morphism* $f : X \to Y$*, the outer rectangle of diagram (2) is an* **initial pushout (over** f**)** *when, for any pushout* ① *with monic* x *and* y*, there exist unique arrows* b^* *and* c^* *making the diagram commute. The subobject* $b : B \to X$ *is the* **boundary** *of* f*, while* $c : C \to Y$ *is the* **context***.*

$$\begin{tikzcd}
B \arrow[r, tail, dotted, "b^*"] \arrow[d, "f'"'] \arrow[rr, tail, bend left, "b"] & U \arrow[r, tail, "x"] \arrow[d, "g"] \arrow[dr, phantom, "①"] & X \arrow[d, "f"] \\
C \arrow[r, tail, dotted, "c^*"] \arrow[rr, tail, bend right, "c"'] & V \arrow[r, tail, "y"] & Y
\end{tikzcd} \qquad (2)$$

The next two properties of initial pushouts will be useful later. For the proof of Lemma 7 we refer to [9].

Lemma 6. *In any category with a strict initial object* $\mathbf{0}$*, the square of diagram (3) is an initial pushout of* f *if and only if* f *is an isomorphism.*

$$\begin{tikzcd}
\mathbf{0} \arrow[r, tail, "!_X"] \arrow[d, "\mathrm{id}_\mathbf{0}"'] & X \arrow[d, "f"] \\
\mathbf{0} \arrow[r, tail, "!_Y"'] & Y
\end{tikzcd} \qquad (3)$$

Lemma 7. *Initial pushouts are preserved and reflected by pushouts along monos. That is, assuming in diagram (4) that square* ② *is a pushout with monic* h_L *and* h_K*, and squares* ① *and* ③ *are initial pushouts, then there exist unique isomorphisms* b^* *and* c^* *making the diagram commute.*

$$\begin{tikzcd}
\overline{B} \arrow[r, tail, "\overline{b}"] \arrow[d, dashed, "\overline{d}"'] \arrow[rrr, dotted, bend left, "b^*"] & \overline{X} \arrow[r, tail, "h_K"] \arrow[d, dashed, "\overline{f}"] \arrow[l, phantom, "①" near end, shift right=6] & X \arrow[d, dashed, "f"] \arrow[l, phantom, "②", shift right=6] & B \arrow[l, tail, "b"'] \arrow[d, dashed, "d"] \arrow[l, phantom, "③", shift right=6] \\
\overline{C} \arrow[r, tail, "\overline{c}"] \arrow[rrr, dotted, bend right, "c^*"'] & \overline{Y} \arrow[r, tail, "h_L"] & Y & C \arrow[l, tail, "c"']
\end{tikzcd} \qquad (4)$$

The categories of graphs and of typed graphs, which are now introduced, are adhesive. Thus, the theory of DPO transformation applies to those categories.

Definition 8 (Categories of Graphs). *A* **graph** $G = (V, E, s, t)$ *has sets* V *of nodes and* E *of edges, along with source and target functions* $s, t : E \to V$*. A* **graph morphism** $f : G \to G'$ *is a pair of functions* $f = (f_V : V \to V', f_E : E \to E')$ *that preserve incidence, that is,* $f_V \circ s = s' \circ f_E$ *and* $f_V \circ t = t' \circ f_E$*. Graphs along with graph morphisms determine the* **category of graphs** $\mathbb{G}\mathrm{raph}$.

Given graph T*, called a* type graph*, we can view arrows* $g : G \to T$ *as graphs typed over* T*. The* **category of** T**-typed graphs** *is* $\mathbb{G}\mathrm{raph}_T = \mathbb{G}\mathrm{raph} \downarrow T$*.*

Example 9. As a motivating example, we use a model of an elevator system, which is based on the type graph of Fig. 2. The system is composed of multiple floors (◇) and elevators (⇅). Solid edges between floors indicate the next floor up, while dashed edges indicate that the source floor is below the target. Solid edges from an elevator to a floor indicate its position, while a self-loop edge of type ⌃ or ⌄ indicates its direction of motion. People on a floor may request an elevator, which is represented as a self-loop edge of type ⬆ or ⬇ , depending on the direction the elevator is expected to go. People inside an elevator may request a stop at a specific floor, which is represented by a dashed arrow from the elevator to the floor.

Due to limited space, we present in Fig. 2 only some rules, related to moving the elevator up. Only left- and right-hand sides are shown, their intersection is the interface. The elevator should only move up when given a reason to do so, i.e. some request involving a higher floor. The rule move-up-AS, for example, moves the elevator up one floor given a stop request for some floor above.

Since matches are required to be injective, this rule is not applicable when the requested floor is y. Thus, we must define another rule move-up-NS when the stop request involves the next floor up; then the request is fulfilled and the corresponding edge is deleted. We also depict the rules applicable when the next floor has requested an elevator to move up (move-up-NU) or down (move-up-ND).

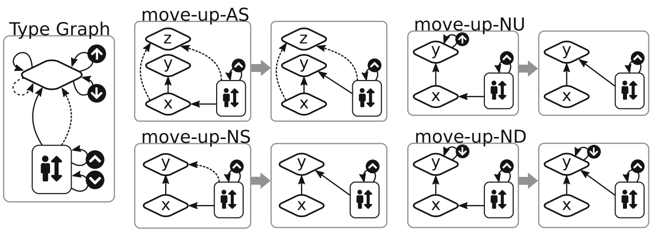


Fig. 2. Type graph and some transformation rules for an elevator system.

2.2 Categories of Set-Valued Functors

Not all results of this paper are proven in terms of adhesive categories, some rely on categories of $\mathbb{S}$et-valued functors. In this section, we show how such categories generalize many important graph models and review some of their properties.

Definition 10 (Set-Valued Functor Category). *Given category $\mathbb{S}$, the category $\mathbb{S}\mathrm{et}^{\mathbb{S}}$ has functors $\mathbb{S} \to \mathbb{S}\mathrm{et}$ as objects and natural transformations as arrows. If $t : F \dot{\to} G$ is a natural transformation between functors $F, G : \mathbb{S} \to \mathbb{S}\mathrm{et}$ and S is an object of $\mathbb{S}$, we denote by $t_S : F(S) \to G(S)$ the component of t on S.*

It is easy to see that the category of graphs is isomorphic to $\mathbb{S}\mathrm{et}^{\mathbb{G}}$, where $\mathbb{G}$ is depicted on diagram (5). A functor $G : \mathbb{G} \to \mathbb{S}\mathrm{et}$ selects two sets $G(V)$ and $G(E)$ as well as two functions $G(s), G(t) : G(E) \to G(V)$. A natural transformation $f : G \dot{\to} G'$ has two components $f_V : G(V) \to G'(V)$ and $f_E : G(E) \to G'(E)$, then naturality corresponds to preservation of incidence.

$$\begin{tikzcd} \mathbb{G} = V \arrow[r, shift left, "s"] \arrow[r, shift right, "t"'] & E \end{tikzcd} \quad (5)$$

$\mathbb{S}\mathrm{et}$-valued functors generalize *graph structures*, which in turn generalize graphs and many variations (e.g. labelled graphs, hypergraphs, E-graphs) [10]. They are also closed w.r.t. the construction of slice categories.

Definition 11. *A* **graph structure signature** *is an algebraic signature Σ containing only unary operator symbols. A* **graph structure** *is a Σ-algebra for a graph structure signature Σ. The category of algebras for a graph structure signature is then a category of graph structures.*

Lemma 12. *Every category of graph structures is isomorphic to $\mathbb{S}\mathrm{et}^{\mathbb{S}}$ for some small, free category $\mathbb{S}$.*

Proof. A graph structure signature Σ defines a graph by taking sorts as nodes and operation symbols as edges. Let $\mathbb{S}$ be the free category generated by this graph. It is easy to see that the category of Σ-algebras is isomorphic to $\mathbb{S}\mathrm{et}^{\mathbb{S}}$. $\square$

Fact 13. *For any functor category $\mathbb{S}\mathrm{et}^{\mathbb{S}}$ and any object $C : \mathbb{S} \to \mathbb{S}\mathrm{et}$ in it, the slice category $\mathbb{S}\mathrm{et}^{\mathbb{S}} \downarrow C$ is equivalent to a $\mathbb{S}\mathrm{et}$-valued functor category [15].*

Functor categories $\mathbb{S}\mathrm{et}^{\mathbb{S}}$ are particularly well-behaved. They inherit a lot of structure from $\mathbb{S}\mathrm{et}$, and many categorical concepts can be considered pointwise for each object of $\mathbb{S}$. When dealing with such concepts, we will apply set-theoretical reasoning to $\mathbb{S}\mathrm{et}^{\mathbb{S}}$. Then, given $X, Y \in \mathbb{S}\mathrm{et}^{\mathbb{S}}$ and $f : X \to Y$, we will write X, Y and f instead of $X(S)$, $Y(S)$ and f_S for an implicit, universally quantified S.

Fact 14. *In any category of functors $\mathbb{S}\mathrm{et}^{\mathbb{S}}$, limits and colimits are constructed pointwise for each object of $\mathbb{S}$ [15]. In particular, the initial object $\mathbf{0}$ of $\mathbb{S}\mathrm{et}^{\mathbb{S}}$ is strict, composed only of empty sets.*

Fact 15. *In any functor category $\mathbb{S}\mathrm{et}^{\mathbb{S}}$, a morphism $f : X \to Z$ is monic (epic) iff each component f_S is injective (surjective) [15]. A pair of morphisms $X \xrightarrow{f} Z \xleftarrow{g} Y$ is jointly epic iff each pair of components (f_S, g_S) is jointly surjective.*

Fact 16. *Given a small category $\mathbb{S}$, the category of functors $\mathbb{S}\mathrm{et}^{\mathbb{S}}$ is a topos [11]. Then it is adhesive [12] and has unique epi-mono factorisations [11].*

In the context of graph transformation, we often have commutative squares that are both a pullback and a pushout. A set-theoretic characterization will be useful in the following. It underlies a construction of initial pushouts, which we omit due to limited space.

Lemma 17. *In any category of functors* $\mathbb{S}\text{et}^{\mathbb{S}}$*, if square (6) is a pullback, then it is also a pushout iff both of the following hold for any element* $z \in Z$*.*

(i) If there is no $x \in X$ *with* $z = f(x)$*, then there is a unique* $y \in Y$ *with* $z = g(y)$*.*
(ii) If there is no $y \in Y$ *with* $z = g(y)$*, then there is a unique* $x \in X$ *with* $z = f(x)$*.*

$$\begin{tikzcd}
W \arrow[r, "g'"] \arrow[d, "f'"'] & X \arrow[d, "f"] \\
Y \arrow[r, "g"'] & Z
\end{tikzcd} \tag{6}$$

Lemma 18. *Any category of functors* $\mathbb{S}\text{et}^{\mathbb{S}}$ *has initial pushouts for all arrows.*

3 The Essence of Conflicting Transformations

An important tool for understanding the behaviour of a transformation system is the notion of *parallel independence*, which ensures that two transformations don't interfere with each other. Essentially, if two transformations $H_1 \overset{t_1}{\Longleftarrow} G \overset{t_2}{\Longrightarrow} H_2$ are parallel independent, then there exist transformations $H_1 \overset{t_2'}{\Longrightarrow} H$ and $H_2 \overset{t_1'}{\Longrightarrow} H$ reaching the same state. If they are not parallel independent, it is said they are *in conflict.*

Understanding the root causes of such conflicts is a subject of ongoing research [2,14]. In this section, we propose a formal characterization of these root causes and compare it to previous work. We show that our characterization has many useful properties, including a direct connection to the definition of parallel independence and being preserved by extension into larger contexts.

We start introducing parallel independence according to the so-called *essential definition* [3].

Definition 19 (Parallel Independence). *A pair of transformations* $(t_1, t_2) : H_1 \overset{\rho_1,m_1}{\Longleftarrow} G \overset{\rho_2,m_2}{\Longrightarrow} H_2$ *is* **parallel independent** *when, building the pullbacks ①, ② and ③ as in diagram (7), the arrows* $K_1L_2 \to L_1L_2$ *and* $L_1K_2 \to L_1L_2$ *are isomorphisms.*[1]

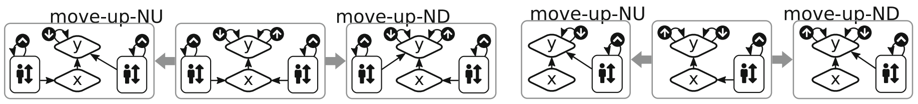
If t_1 *and* t_2 *are not parallel independent, we say they are* **in conflict**. *When* $K_1L_2 \to L_1L_2$ *is not an isomorphism, we say* t_1 **disables** t_2*; when* $L_1K_2 \to L_1L_2$ *is not an isomorphism, we say* t_2 *disables* t_1*.*

$$\begin{tikzcd}
 & K_1L_2 \arrow[rr, "\cong"] \arrow[d] \arrow[dr, phantom, "\lrcorner\ ②"] & & L_1L_2 \arrow[dl, "p_1"'] \arrow[dr, "p_2"] \arrow[d, phantom, "\vee"] & & L_1K_2 \arrow[ll, "\cong"'] \arrow[d] \arrow[dl, phantom, "③\ \llcorner"] & \\
R_1 \arrow[d, "n_1"'] & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"] & L_1 \arrow[dr, "m_1"] & ① & L_2 \arrow[dl, "m_2"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] \arrow[d, "k_2"] & R_2 \arrow[d, "n_2"] \\
H_1 & D_1 \arrow[l, "h_1"'] \arrow[rr, "g_1"] & & G & & D_2 \arrow[ll, "g_2"'] \arrow[r, "h_2"] & H_2
\end{tikzcd} \tag{7}$$

[1] Since l_1 and l_2 are monos, this is equivalent to requiring existence of arrows $L_1L_2 \to K_1$ and $L_1L_2 \to K_2$ making the resulting triangles commute. However, this simpler condition is not helpful for the characterization of conflicts.

We refer the reader to [3] for a proof that this definition is equivalent to the traditional one (see e.g. [8]) if diagrams are taken in an adhesive category and rule morphisms are monic.

Example 20. Figure 3a shows a pair of parallel independent transformations obtained by applying the rules move-up-NU and move-up-ND of Example 9 to two different elevators. In Fig. 3b the transformations caused by the same rules are in conflict, since the edge between elevator and floor is deleted by both rules.



(a) Parallel independent transformations. (b) Transformations in conflict.

Fig. 3. Examples of parallel independence.

A disabling occurs when some element is matched by both rules and deleted by at least one of them, as illustrated in Example 20. Since matches are monic, they can be interpreted as subobjects of G and the pullback L_1L_2 as their intersection. Pulling it back along l_1 removes exactly the elements that would be deleted by the transformation using ρ_1 and that are matched by m_2.

In order to determine such elements, we use an initial pushout over arrow $K_1L_2 \to L_1L_2$ (Definition 5). In fact in a functor category $\mathbb{S}\mathrm{et}^{\mathbb{S}}$ the context of a mono $f : X \rightarrowtail Y$ is the smallest subobject of Y containing all the elements which are not in the image of f. This brings us to the following definition.

$$\begin{tikzcd}
B_1 \arrow[r, "d_1"] \arrow[d, tail, "b_1"] \arrow[dr, phantom, "\textcircled{4}"] & C_1 \arrow[d, tail, "c_1"] & \\
K_1L_2 \arrow[r, "q_2"] \arrow[d, "q_1"] \arrow[dr, phantom, "\textcircled{2}"] & L_1L_2 \arrow[r, tail, "p_2"] \arrow[d, tail, "p_1"] \arrow[dr, phantom, "\textcircled{1}"] & L_2 \arrow[d, tail, "m_2"] \\
K_1 \arrow[r, "l_1"] & L_1 \arrow[r, tail, "m_1"] & G
\end{tikzcd} \qquad (8)$$

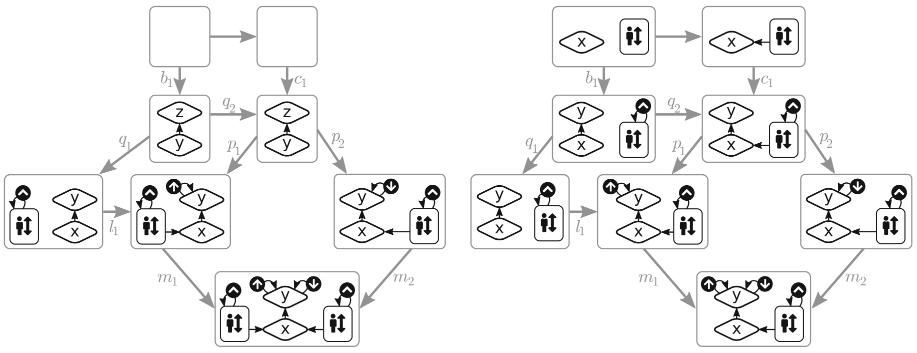
Definition 21 (Conflict and Disabling Essence). *In any adhesive category, let* $(t_1, t_2) : H_1 \overset{\rho_1,m_1}{\Longleftarrow} G \overset{\rho_2,m_2}{\Longrightarrow} H_2$ *be a pair of transformations.*

The **disabling essence** $c_1 : C_1 \rightarrowtail L_1L_2$ *for* (t_1, t_2) *is obtained by taking pullbacks* $\textcircled{1}$ *and* $\textcircled{2}$ *as in diagram (8), then the initial pushout* $\textcircled{4}$ *over* q_2*.*

The **conflict essence** $c : C \to L_1L_2$ *is the union in* $\mathbf{Sub}(L_1L_2)$ *of the disabling essences* c_1 *for* (t_1, t_2) *and* c_2 *for* (t_2, t_1)*. Recall from Fact 2 that this is obtained as a pushout over the pullback of* (c_1, c_2)*.*

Remark 22. The disabling and conflict essences are also subobjects of G, since the composite $m_1 \circ p_1 \circ c = m_2 \circ p_2 \circ c$ is a monomorphism $C \rightarrowtail G$.

Example 23. The disabling essences for Example 20 are constructed in Fig. 4. In Fig. 4a, where transformations are independent, the essence is empty. In Fig. 4b, where a disabling exists, the essence contains an edge from elevator to floor.



(a) Disabling essence for Figure 3a. (b) Disabling essence for Figure 3b.

Fig. 4. Examples of disabling essence.

From Example 23 we may expect that an empty disabling essence is equivalent to having no disabling. More generally, we can show that this holds whenever the essence is the strict initial object, establishing a direct correspondence between conflict essences and the definition of parallel independence.

Theorem 24. *In any adhesive category with a strict initial object, let* $(t_1, t_2) : H_1 \overset{\rho_1, m_1}{\Longleftarrow} G \overset{\rho_2, m_2}{\Longrightarrow} H_2$ *be a pair of transformations. Then the disabling essence is initial iff* t_1 *doesn't disable* t_2*, and the conflict essence is initial iff* t_1 *and* t_2 *are parallel independent.*

Proof. The case for disabling essences follows directly from Lemma 6: the disabling essence is initial if and only if q_2 in diagram (8) is an isomorphism, which means that t_1 doesn't disable t_2, according to Definition 19. For conflict essences, recall that the strict initial object is the bottom element of the lattice $\mathbf{Sub}(L_1L_2)$. Since $c = c_1 \cup c_2$, C is initial iff C_1 and C_2 are initial, which is equivalent to having no disablings and thus parallel independence. □

Another important property of the disabling essence is that it factors uniquely through the context of the (initial pushout over) arrow $K_1 \xrightarrow{l_1} L_1$. This means that the essence only contains deleted elements, or elements incident to them. The next result is exploited in Subsect. 3.2 to relate our notion to *disabling reasons* [14]. It would also be the basis for a precise comparison of conflict essences with the *basic conflict conditions* introduced in [2], which is left for future work.

$$\begin{tikzcd}
B_{l1} \arrow[r] \arrow[d, tail] \arrow[dr, phantom, "①"] & C_{l1} \arrow[d, tail, "c_{l1}"'] & C_1 \arrow[l, tail, dotted, "h"'] \arrow[d, tail, "c_1"] \\
K_1 \arrow[r, "l_1"'] & L_1 & L_1L_2 \arrow[l, tail, "p_1"]
\end{tikzcd} \tag{9}$$

Lemma 25. *In any adhesive category, let* $(t_1, t_2) : H_1 \overset{\rho_1, m_1}{\Longleftarrow} G \overset{\rho_2, m_2}{\Longrightarrow} H_2$ *be a transformation pair, let* $c_1 : C_1 \rightarrowtail L_1L_2$ *be its disabling essence and let square* ① *be the initial pushout over* l_1 *in diagram (9). Then the disabling essence of* (t_1, t_2) *factors uniquely through the context* C_{l1} *over* l_1*, i.e., there is a unique mono* $h : C_1 \rightarrowtail C_{l1}$ *with* $p_1 \circ c_1 = c_{l1} \circ h$.

3.1 Conflict Essence and Extension

The *extension* of a transformation into a larger context underlies the concept of *completeness* of critical pairs and initial conflicts: any pair of conflicting transformations is the extension of a critical pair [8,9] and of an initial conflict [13]. It ensures that checking each critical pair (or each initial conflict) for strict confluence guarantees local confluence for the entire transformation system [8,9].

Definition 26 (Extension Diagram). *An* **extension diagram** *over transformation* $t : G \overset{\rho,m}{\Longrightarrow} H$ *and* extension morphism $e : G \to \overline{G}$ *is a diagram (10) where* $\overline{m} = e \circ m$ *is monic and there is a transformation* $\bar{t} : \overline{G} \overset{\rho,\overline{m}}{\Longrightarrow} \overline{H}$ *defined by the four pushout squares of diagram (11).*

$$\begin{tikzcd} G \arrow[r, Rightarrow, "t"] \arrow[d, "e"] & H \arrow[d, "f"] \\ \overline{G} \arrow[r, Rightarrow, "\bar{t}"] & \overline{H} \end{tikzcd} \tag{10}$$

$$\begin{tikzcd} L \arrow[d, tail, "m"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, tail, "k"] & R \arrow[d, tail, "n"] \\ G \arrow[d, "e"'] & D \arrow[l, "g"'] \arrow[r, "h"] \arrow[d, "d"] & H \arrow[d, "f"] \\ \overline{G} & \overline{D} \arrow[l, "g'"'] \arrow[r, "\overline{h}"] & \overline{H} \end{tikzcd} \tag{11}$$

It was already shown that conflicts are *reflected* by extension [13], i.e. when the extension of a transformation pair is in conflict, the original transformation pair is in conflict as well. It turns out they are also *preserved* by extension in categories of set-valued functors, and in particular of graphs and typed graphs. Furthermore, conflict *essences* are also preserved, which means the root causes of a conflict don't change with extension.

Lemma 27 (Essence Inheritance). *In a category of functors* $\mathbb{Set}^{\mathbb{S}}$*, let* $(\overline{t_1}, \overline{t_2}) : \overline{H_1} \overset{\rho_1,\overline{m_1}}{\Longleftarrow} \overline{G} \overset{\rho_2,\overline{m_2}}{\Longrightarrow} \overline{H_2}$ *and* $(t_1, t_2) : H_1 \overset{\rho_1,m_1}{\Longleftarrow} G \overset{\rho_2,m_2}{\Longrightarrow} H_2$ *be two pairs of transformations such that both extension diagrams of (12) exist for some* $f : \overline{G} \to G$.
Then the transformation pairs have isomorphic conflict and disabling essences. That is, given the mediating morphism $h_L : \overline{L_1L_2} \rightarrowtail L_1L_2$ *between the pullback objects and the conflict (disabling) essences* $\bar{c}$ *of* $(\overline{t_1}, \overline{t_2})$ *and* c *of* (t_1, t_2)*, then* $h_L \circ \bar{c} \cong c$ *in* $\mathbf{Sub}(L_1L_2)$.

$$\begin{tikzcd} \overline{H_1} \arrow[d] & \overline{G} \arrow[l, Rightarrow, "\overline{t_1}"'] \arrow[r, Rightarrow, "\overline{t_2}"] \arrow[d, dashed, "f"] & \overline{H_2} \arrow[d] \\ H_1 & G \arrow[l, Rightarrow, "t_1"'] \arrow[r, Rightarrow, "t_2"] & H_2 \end{tikzcd} \tag{12}$$

Proof. **[Disabling]** Consider diagram (13) where $\overline{L_1L_2}$ and L_1L_2 are pullback objects for $(\overline{m_1}, \overline{m_2})$ and (m_1, m_2), respectively, and h_L their unique mediating morphism. Constructing the squares ② and ① + ② as pullbacks, by decomposition there is a unique monomorphism h_K making ① a pullback. We will show that ① is also a pushout. Then, since initial pushouts are reflected by pushouts along monomorphisms (Lemma 7), there is an isomorphism $h_C : \overline{C_1} \to C_1$ between the contexts of $\overline{q_2}$ and q_2, with $c_1 \circ h_C = h_L \circ \overline{c_1}$. This makes c_1 and $h_L \circ \overline{c_1}$ isomorphic in $\mathbf{Sub}(L_1L_2)$.

$$\begin{tikzcd}
\overline{K_1L_2} \arrow[r, tail, "h_K"] \arrow[d, tail, "\overline{q_2}"'] \arrow[rr, bend left, tail, "\overline{q_1}"] \arrow[dr, phantom, "①"] & K_1L_2 \arrow[r, tail, "q_1"] \arrow[d, tail, "q_2"] \arrow[dr, phantom, "②"] & K_1 \arrow[r, "\overline{k}"] \arrow[d, tail, "l_1"] \arrow[rr, bend left, tail, "k"] \arrow[dr, phantom, "③"] & \overline{D_1} \arrow[r, "f'"] \arrow[d, tail, "\overline{g}"] \arrow[dr, phantom, "④"] & D_1 \arrow[d, tail, "g"] \\
\overline{L_1L_2} \arrow[r, tail, "h_L"] \arrow[rr, bend right, tail, "\overline{p_1}"'] & L_1L_2 \arrow[r, tail, "p_1"] & L_1 \arrow[r, tail, "\overline{m_1}"] \arrow[rr, bend right, tail, "m_1"'] & \overline{G} \arrow[r, "f"] & G
\end{tikzcd} \tag{13}$$

Note that, because the left square of (12) is an extension diagram, there exist pushouts ③ and ④ in diagram (13). Without loss of generality, assume all vertical morphisms of (13) are inclusions, as well as h_L and h_K.

In order to show that pullback ① is also a pushout, by Lemma 17 it suffices to show that (h_L, q_2) is jointly epic. We will show that every element $x \in L_1L_2$ that is not in $\overline{L_1L_2}$ must be in K_1L_2. That is, it must be preserved by step t_1.

So assume such an $x \in L_1L_2 \setminus \overline{L_1L_2}$ and consider the two elements $y_1 = \overline{m_1}(p_1(x)) \in \overline{G}$ and $y_2 = \overline{m_2}(p_2(x)) \in \overline{G}$. These elements of $\overline{G}$ are distinct, but identified by f. In fact, $x \notin \overline{L_1L_2}$ implies that $\overline{m_1}(p_1(x)) \neq \overline{m_2}(p_2(x))$, that is, $y_1 \neq y_2$; instead, since (p_1, p_2) is a pullback of (m_1, m_2), we have $m_1(p_1(x)) = m_2(p_2(x))$ which is equivalent to $f(y_1) = f(y_2)$.

Since ④ is a pushout and pullback, and $f(y_1) \in G$ has two distinct preimages by f, it follows from Lemma 17 that $f(y_1)$ has a unique preimage by g. That is, $f(y_1) \in D_1 \subseteq G$. Then, since ② + ③ + ④ is a pullback and $f(\overline{m_1}(p_1(x))) = f(y_1) = g(f(y_1))$, x must have a preimage by q_2. That is, $x \in K_1L_2 \subseteq L_1L_2$.

In conclusion, every $x \in L_1L_2 \setminus \overline{L_1L_2}$ is such that $x \in K_1L_2$ and therefore ① is a pushout. This implies that q_2 and $\overline{q_2}$ have isomorphic initial pushouts, making $h_L \circ \overline{c_1}$ isomorphic to c_1 in $\mathbf{Sub}(L_1L_2)$.

[Conflict] Follows directly from the previous point. Given the unions c and $\overline{c}$ of $c_1, c_2 \in \mathbf{Sub}(L_1L_2)$ and $\overline{c_1}, \overline{c_2} \in \mathbf{Sub}(\overline{L_1L_2})$, it is trivial to show that $\overline{c} \circ h_L$ is isomorphic to c when $\overline{c_j} \circ h_L$ is isomorphic to c_j for $j \in \{1, 2\}$.

Corollary 28. *In a category of functors* $\mathbb{S}\mathrm{et}^{\mathbb{S}}$*, assume the extension diagrams of (12) exist. Then* t_1 *disables* t_2 *if and only if* $\overline{t_1}$ *disables* $\overline{t_2}$*. Furthermore,* t_1 *and* t_2 *are in conflict if and only if* $\overline{t_1}$ *and* $\overline{t_2}$ *are in conflict.*

3.2 Comparing Reasons and Essences

Conflict essences provide some advantages over the *conflict reason spans* proposed in [14]. In order to simplify the comparison, we introduce conflict and disabling reasons in a slightly different but equivalent way, characterizing them as subobjects of L_1L_2, the pullback object of the matches.

Definition 29 (Disabling Reason). *In an adhesive category, let* $(t_1, t_2) : H_1 \overset{\rho_1, m_1}{\Longleftarrow} G \overset{\rho_2, m_2}{\Longrightarrow} H_2$ *be a pair of transformations.*

The **disabling reason** $s_1 : S_1 \rightarrowtail L_1L_2$ *for* (t_1, t_2) *is the mediating morphism obtained by constructing the initial pushout over* l_1 *as in diagram (14), then the pullbacks* (o_1, s_{12}) *of* $(m_1 \circ c_{l1}, m_2)$ *and* (p_1, p_2) *of the matches.*

A disabling reason satisfies the **conflict condition** *if there is no morphism* $b^* : S_1 \to B_{l1}$ *making diagram (14) commute.*

If $s_1 : S_1 \rightarrowtail L_1L_2$ *and* $s_2 : S_2 \rightarrowtail L_1L_2$ *are the disabling reasons of* (t_1, t_2) *and* (t_2, t_1)*, the* **conflict reason subobject** $s : S \rightarrowtail L_1L_2$ *is constructed as follows. If both* s_1 *and* s_2 *satisfy the conflict condition, then* $s = s_1 \cup s_2$ *in* $\mathbf{Sub}(L_1L_2)$*. If only* s_1 *satisfies the conflict condition, then* $s = s_1$*; analogously for* s_2*.*

$$\begin{tikzcd}
 & & S_1 \arrow[dll, dotted, "b^*"'] \arrow[dl, "o_1"'] \arrow[d, dotted, "s_1"] \arrow[ddr, bend left, "s_{12}"] & \\
B_{l1} \arrow[r, "l'_1"] \arrow[d, "b_{l1}"'] & C_{l1} \arrow[d, "c_{l1}"'] & L_1L_2 \arrow[dl, "p_1"'] \arrow[dr, "p_2"] & \\
K_1 \arrow[r, "l_1"'] & L_1 \arrow[dr, "m_1"'] & & L_2 \arrow[dl, "m_2"] \\
 & & G &
\end{tikzcd} \tag{14}$$

A conflict reason *span* is defined in [14] as the span $L_1 \overset{c_{l1} \circ o_1}{\leftarrow} S_1 \overset{s_{12}}{\rightarrow} L_2$ if only s_1 satisfies the conflict condition, symmetrically if only s_2 satisfies it, and as a span obtained by building a pushout over a pullback if both satisfy the condition. The equivalence with Definition 29 follows by the bijection between spans of monos commuting with $L_1 \overset{m_1}{\to} G \overset{m_2}{\leftarrow} L_2$ and monos to L_1L_2, and observing that the construction in the third case is identical to that of unions of subobjects.

Fact 30. *Given morphisms* $L_1 \overset{m_1}{\to} G \overset{m_1}{\leftarrow} L_2$ *with pullback object* L_1L_2*, the set of spans* $L_1 \overset{f}{\leftarrowtail} S \overset{g}{\rightarrowtail} L_2$ *commuting with* (m_1, m_2) *is isomorphic to the set of monos* $h : S \rightarrowtail L_1L_2$*.*

Proof. Given a span $L_1 \overset{f}{\leftarrowtail} S \overset{g}{\rightarrowtail} L_2$ commuting with (m_1, m_2), there is a unique h making (15) commute. Since $p_1 \circ h$ is monic, h is also monic, thus we constructed a unique mono corresponding to (f, g). Given monic $h : S \rightarrowtail L_1L_2$, we can construct the span $(p_1 \circ h, p_2 \circ h)$. These constructions establish a bijection.

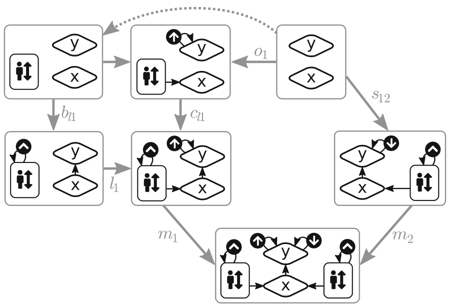
$$\begin{tikzcd}
 & S \arrow[d, "h"] \arrow[ddl, bend right, "f"'] \arrow[ddr, bend left, "g"] & \\
 & L_1L_2 \arrow[dl, "p_1"'] \arrow[dr, "p_2"] & \\
L_1 \arrow[dr, "m_1"'] & & L_2 \arrow[dl, "m_2"] \\
 & G &
\end{tikzcd} \tag{15}$$

□

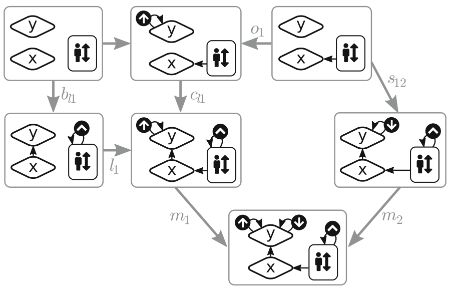
Note that the relation between disabling reasons and any condition of parallel independence is not very direct. It has been shown that a disabling exists (in the sense of Definition 19) if and only if the reason satisfies the conflict condition [14], but the proof is much more involved than that of Theorem 24.

Interestingly, both Definitions 29 and 21 use the same operations, but in reversed orders. More explicitly, the disabling reason is obtained by first taking the context of (the initial pushout over) l_1, containing all elements deleted by ρ_1 (and the boundary), and then the intersection with the image of m_2. Instead, the disabling essence first restricts on the elements which are matched by both transformations, and then takes the context, thus filtering out boundary elements of l_1 that are not relevant for the conflict. This suggests that disabling essences are in general smaller than disabling reasons, as illustrated by the following example.

Example 31. The disabling reasons for Example 20 are constructed in Fig. 5. In Fig. 5a, even though the transformations were independent, the reason contains both floors. In Fig. 5b, the floor y is part of the reason despite not being involved in the conflict.

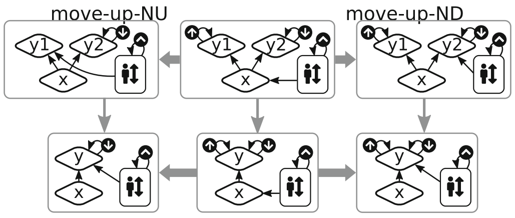


(a) Disabling reason for Figure 3a.

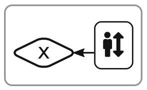


(b) Disabling reason for Figure 3b.

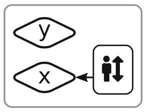
Fig. 5. Examples of disabling reason.



(a) Extension diagrams.



(b) Reason for upper pair.



(c) Reason for lower pair.

Fig. 6. Extended transformations with distinct disabling reasons.

As Example 31 shows, the disabling and conflict reasons may contain elements that aren't directly related to the conflict. In the case of graphs, these are *isolated boundary nodes* [13], i.e. nodes adjacent to a deleted edge where this deletion does not cause a disabling. A comparison with Example 23 indicates that this is not the case for the essences.

The presence of isolated boundary nodes provides another disadvantage: extending a transformation pair may modify the disabling reason by introducing new isolated boundary nodes, as shown in Fig. 6. This cannot happen for conflict essences, as proved in Lemma 27.

We conclude this section with a formal proof that every conflict essence is more precise than the corresponding reason, since the former factors uniquely through the latter. Indeed, essences are subobjects of reasons, since the unique factoring is a monomorphism.

Theorem 32 (Precision of Essences). *In any adhesive category, let* $(t_1, t_2) : H_1 \overset{\rho_1,m_1}{\Longleftarrow} G \overset{\rho_2,m_2}{\Longrightarrow} H_2$ *be a pair of transformations with conflict (disabling) essence* $c : C \rightarrowtail L_1L_2$ *and reason* $s : S \rightarrowtail L_1L_2$*. Then* $c \subseteq s$ *in* $\mathbf{Sub}(L_1L_2)$*, that is, there is a unique monomorphism* $h : C \rightarrowtail S$ *such that* $c = s \circ h$*.*

Proof. [**Disabling**] By Lemma 25, there is a unique monomorphism $f : C_1 \rightarrowtail C_{l1}$ with $p_1 \circ c_1 = c_{l1} \circ f$, where C_{l1} is the context of l_1. By Definition 29, the rectangle of diagram (16) is a pullback. Then there is a unique h with $p_2 \circ s_1 \circ h = p_2 \circ c_1$. Since p_2 is monic, we have $s_1 \circ h = c_1$.

[**Conflict**] If only s_1 satisfies the conflict condition, then $s = s_1$ by Definition 29. In this case t_1 does not disable t_2, thus by Theorem 24 c_2 is the bottom of $\mathbf{Sub}(L_1L_2)$, which implies $c = c_1 \cup \bot = c_1$, and thus $c = c_1 \subseteq s_1 = s$.

$$\begin{tikzcd}
C_1 \arrow[r, dotted, "h"] \arrow[rr, bend left, "c_1"] \arrow[dr, "f"'] & S_1 \arrow[r, "s_1"] \arrow[d, "o_1"] & L_1L_2 \arrow[r, "p_2"] \arrow[d, "p_1"] & L_2 \arrow[d, "m_2"] \\
& C_{l1} \arrow[r, "c_{l1}"] & L_1 \arrow[r, "m_1"] & G
\end{tikzcd} \tag{16}$$

The case when only s_2 satisfies the conflict condition is symmetrical. If both s_1 and s_2 satisfy it, then $s = s_1 \cup s_2$. Since $c_1 \subseteq s_1$ and $c_2 \subseteq s_2$, then it follows from distributivity of $\mathbf{Sub}(L_1L_2)$ that $c = c_1 \cup c_2 \subseteq s_1 \cup s_2 = s$. □

4 Conflict Essences and Initial Conflicts

In this section we show how conflict essences can be used to characterize initial conflicts. Although this is only proven for categories of $\mathbb{S}\mathrm{et}$-valued functors, the characterization is stated completely in categorical terms, unlike the previous element-based characterization in the category of typed graphs [13]. Thus, this formulation may be applicable to other categories.

In order to curb the redundancy of critical pairs, a notion of initiality with respect to embedding was introduced in [13].

Definition 33 (Initial Transformation Pair). *Given the pair of transformation steps* $(t_1, t_2) : H_1 \overset{\rho_1,m_1}{\Longleftarrow} G \overset{\rho_2,m_2}{\Longrightarrow} H_2$*, a pair* $(s_1, s_2) : J_1 \overset{\rho_1,n_1}{\Longleftarrow} I \overset{\rho_2,n_2}{\Longrightarrow} J_2$ *is an* **initial transformation pair** *for* (t_1, t_2) *if it satisfies both of the following.*

(i) The pair (s_1, s_2) *can be embedded into* (t_1, t_2) *as in diagram (17).*

(ii) For every pair $(\overline{t_1}, \overline{t_2}) : \overline{H_1} \overset{\rho_1,\overline{m_1}}{\Longleftarrow} \overline{G} \overset{\rho_2,\overline{m_2}}{\Longrightarrow} \overline{H_2}$ *that can be embedded into* (t_1, t_2) *as in diagram (18), then* (s_1, s_2) *can be embedded into* $(\overline{t_1}, \overline{t_2})$*.*

$$\begin{tikzcd}
J_1 \arrow[d] & I \arrow[l, Leftarrow, "s_1"'] \arrow[r, Rightarrow, "s_2"] \arrow[d, dashed, "f"] & J_2 \arrow[d] \\
H_1 & G \arrow[l, Leftarrow, "t_1"'] \arrow[r, Rightarrow, "t_2"] & H_2
\end{tikzcd} \tag{17}$$

$$\begin{tikzcd}
J_1 \arrow[d] & I \arrow[l, Leftarrow, "s_1"'] \arrow[r, Rightarrow, "s_2"] \arrow[d, dashed, "h"] & J_2 \arrow[d] \\
\overline{H_1} \arrow[d] & \overline{G} \arrow[l, Leftarrow, "\overline{t_1}"'] \arrow[r, Rightarrow, "\overline{t_2}"] \arrow[d, dashed, "g"] & \overline{H_2} \arrow[d] \\
H_1 & G \arrow[l, Leftarrow, "t_1"'] \arrow[r, Rightarrow, "t_2"] & H_2
\end{tikzcd} \tag{18}$$

Initial transformation pairs are not guaranteed to exist in any category, but they exist in the category of typed graphs [13]. It turns out this is also true for any category of $\mathbb{S}$et-valued functors, where initial conflicts can be constructed as pushouts of the conflict essence.

Theorem 34 (Construction of Initial Transformation Pairs). *In any category of functors* $\mathbb{S}\mathrm{et}^{\mathbb{S}}$*, let* $(t_1, t_2) : H_1 \overset{\rho_1, m_1}{\Longleftarrow} G \overset{\rho_2, m_2}{\Longrightarrow} H_2$ *be a pair of transformations with conflict essence* $c : C \rightarrowtail L_1L_2$*. Then the pushout* $L_1 \overset{n_1}{\rightarrow} I \overset{n_2}{\leftarrow} L_2$ *of* $L_1 \overset{p_1 \circ c}{\leftarrow} C \overset{p_2 \circ c}{\rightarrow} L_2$ *determines an initial transformation pair* $(s_1, s_2) : J_1 \overset{\rho_1, n_1}{\Longleftarrow} I \overset{\rho_2, n_2}{\Longrightarrow} J_2$ *for* (t_1, t_2)*.*

Proof. We have to show that $L_1 \overset{n_1}{\rightarrow} I \overset{n_2}{\leftarrow} L_2$ (i) determines a transformation pair (s_1, s_2) which (ii) can be embedded into (t_1, t_2) via some morphism f and (iii) can be embedded into any other pair of transformation steps $(\overline{t_1}, \overline{t_2})$ that is embedded into (t_1, t_2) via some morphism g.

Note that (iii) follows from (ii). In fact, consider the mediating morphism $p : \overline{L_1L_2} \to L_1L_2$. By essence inheritance, the conflict essence of $(\overline{t_1}, \overline{t_2})$ is $\overline{c} : C \rightarrowtail \overline{L_1L_2}$ with $c = p \circ \overline{c}$. Then (n_1, n_2) is also the pushout of $(\overline{p_1} \circ \overline{c}, \overline{p_2} \circ \overline{c})$, since $\overline{p_1} \circ \overline{c} = p_1 \circ p \circ \overline{c} = p_1 \circ c$ and analogously $\overline{p_2} \circ \overline{c} = p_2 \circ c$. Then by (ii) the transformation determined by (n_1, n_2) can be embedded into $(\overline{t_1}, \overline{t_2})$.

To prove (i) and (ii), note that (n_1, n_2) is jointly epic, since it is a pushout. There is also a unique morphism $f : I \to G$ making diagram (19) commute.

Now consider the diagram (20), where (p_1, p_2) is the pullback of (m_1, m_2) and ① is the initial pushout of q_2. From the transformation step t_1, there is also a pushout ③ + ④, and we can construct a pullback ④. We can also show $n_1 \circ p_1 \circ c_1 = n_1 \circ p_2 \circ c_1$. In fact, since the conflict essence is the union of the disablings, there is $h : C_1 \rightarrowtail C$ with $c_1 = c \circ h$. Then $n_j \circ p_j \circ c_1 = n_j \circ p_j \circ c \circ h$ for $j \in \{1, 2\}$, and $n_1 \circ p_1 \circ c = n_2 \circ p_2 \circ c$ by construction of (n_1, n_2).

$$\begin{tikzcd}
& C \arrow[dl, tail, "p_1\circ c"'] \arrow[dr, tail, "p_2\circ c"] & \\
L_1 \arrow[dr, tail, "n_1"] \arrow[ddr, tail, bend right, "m_1"'] & & L_2 \arrow[dl, tail, "n_2"'] \arrow[ddl, tail, bend left, "m_2"] \\
& I \arrow[d, dotted, "f"] & \\
& G &
\end{tikzcd} \tag{19}$$

$$\begin{tikzcd}
& & L_2 \arrow[dr, tail, "n_2"] & & \\
C_1 \arrow[r, tail, "c_1"] & L_1L_2 \arrow[ur, tail, "p_2"] \arrow[dr, tail, "p_1"] & & I \arrow[r, "f"] & G \\
& & L_1 \arrow[ur, tail, "n_1"] & & \\
B_1 \arrow[uu, tail, "d_1"] \arrow[r, tail, "b_1"'] & K_1L_2 \arrow[uu, tail, "q_2"] \arrow[dr, tail, "q_1"'] & & D_1' \arrow[uu, tail, "\overline{g}"] \arrow[dr, "f'"] & \\
& & K_1 \arrow[uu, tail, "l_1"] \arrow[ur, tail, dotted, "k_1'"] \arrow[rr, tail, "k_1"] & & D_1 \arrow[uuu, tail, "g"']
\arrow[from=2-1, to=4-2, phantom, "\textcircled{1}"]
\arrow[from=2-2, to=5-3, phantom, "\textcircled{2}"]
\arrow[from=3-3, to=4-4, phantom, "\textcircled{3}"]
\arrow[from=2-4, to=5-5, phantom, "\textcircled{4}"]
\end{tikzcd} \tag{20}$$

Then, by the following lemma, ③ and ④ are pushouts in diagram (20). Pushout ③ ensures the existence of the transformation step s_1, as required by (i). Pushout ④ ensures that s_1 can be embedded into t_1, as required by (ii). The proof that a transformation step s_2 exists and can be embedded into t_2 is analogous, using the disabling reason of (t_2, t_1). □

Lemma 35 (PO-PB Decomposition by Disabling Essence). *In any category of functors $\mathbb{S}\mathrm{et}^{\mathbb{S}}$, assume that in diagram (20) the triangle and squares ①–④ commute and all morphisms except f and f' are monic. Let $m_1 = f \circ n_1$ and $m_2 = f \circ n_2$. Assume also: (n_1, n_2) is jointly epic; (p_1, p_2) is the pullback of (m_1, m_2); $n_1 \circ p_1 \circ c_1 = n_2 \circ p_2 \circ c_2$; ③ + ④ is a pushout; ② and ④ are pullbacks; ① is the initial pushout of q_2. Then both squares ③ and ④ are pushouts.*

Proof. We will show that square ③ is a pushout, which by decomposition implies that ④ is also a pushout. Without loss of generality, assume that all vertical morphisms of diagram (20) as well as c_1 and b_1 are inclusions.

By Lemma 17, it suffices to show that every element $x \in I \setminus n_1(L_1)$ has a unique preimage by $\overline{G}$, that is $x \in D_1' \subseteq I$. Since (n_1, n_2) is jointly epic and $x \notin n_1(L_1)$, we must have $x \in n_2(L_2)$. Thus, there is $y_2 \in L_2$ with $x = n_1(y_2)$.

Now, consider $f(x) \in G$. Recall that (m_1, g) is a pushout and thus jointly epic, then $f(x)$ must be in the image of m_1 or g. We will show that it is always in the image of g, that is, $f(x) \in D_1$. Then, since ④ is a pullback, $x \in D_1'$.

When $f(x)$ is in the image of m_1 there is $y_1 \in L_1$ with $m_1(y_1) = f(x) = m_2(y_2)$. Then since (p_1, p_2) is a pullback there is a unique $z \in L_1L_2$ with $p_1(z) = y_1$ and $p_2(z) = y_2$. But z cannot be in C_1, otherwise we would have $x = n_2(p_2(c_1(z))) = n_1(p_1(c_1(z)))$, contradicting the assumption that $x \notin n_1(L_1)$. Thus, we must have $z \in K_1L_2$. Then, since squares ②–④ commute in diagram (20), we have $f(x) = k_1(q_1(z)) \in D_1$. □

Note that the proof of Theorem 34 doesn't directly depend on details of the category of functors. Thus, it holds in any category with essence inheritance and PO-PB decomposition by disabling essence.

Conflicting transformation pairs that are themselves initial are called initial conflicts. They provide a suitable subset of critical pairs, being complete in the sense that any conflicting transformation pair is a unique extension of some initial conflict [13]. We provide a simple characterization for initial conflicts.

Definition 36 (Initial Conflict). *A pair of conflicting transformation steps* $(t_1, t_2) : H_1 \overset{\rho_1,m_1}{\Longleftarrow} G \overset{\rho_2,m_2}{\Longrightarrow} H_2$ *is an* initial conflict *when it is isomorphic to its initial transformation pair.*

Corollary 37. *In a category of functors* $\mathbb{S}\mathrm{et}^{\mathbb{S}}$*, let* $(t_1, t_2) : H_1 \overset{\rho_1,m_1}{\Longleftarrow} G \overset{\rho_2,m_2}{\Longrightarrow} H_2$ *be a pair of transformations with non-empty conflict essence. Then the following are equivalent:*

(i) (t_1, t_2) *is an initial conflict*
(ii) the conflict essence of (t_1, t_2) *is isomorphic to* L_1L_2
(iii) the pullback square to the right is also a pushout.

$$\begin{tikzcd}
L_1L_2 \arrow[r, "p_1"] \arrow[d, "p_2"] & L_1 \arrow[d, "m_1"] \\
L_2 \arrow[r, "m_2"'] & G
\end{tikzcd}$$

5 Conclusions

In this paper we have introduced conflict essences as a formal characterization of the root causes of conflicts. We have shown that, in adhesive categories, they are empty if and only if the transformations are parallel independent, and they are not larger than the previously proposed conflict reasons. In categories of set-valued functors, which include the categories of graphs and typed graphs, we have shown that essences are preserved by extension, and that they uniquely determine initial conflicts. We have also identified two sufficient conditions for the existence of initial conflicts in an adhesive category: essence inheritance (Lemma 27) and PO-PB decomposition by disabling essence (Lemma 35).

As future work, we intend to adapt the definitions of this paper to the Sesqui-Pushout approach [4]. In the context of the DPO approach, we intend to apply the conflict conditions of [2] to the essences. From a more practical perspective, it should be possible to improve the efficiency of Critical-Pair Analysis, as implemented by AGG [18] and Verigraph [1] by enumerating conflict essences and initial conflicts instead of critical pairs. Furthermore, integrating this work with constraints and application conditions [7] is important for practical applications.

Acknowledgements. The authors would like to acknowledge the brazilian agency CAPES for their support in the form of scholarships.

References

1. Azzi, G.G., Bezerra, J.S., Ribeiro, L., Costa, A., Rodrigues, L.M., Machado, R.: The verigraph system for graph transformation. In: Heckel, R., Taentzer, G. (eds.) Graph Transformation, Specifications, and Nets. LNCS, vol. 10800, pp. 160–178. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75396-6_9
2. Born, K., Lambers, L., Strüber, D., Taentzer, G.: Granularity of conflicts and dependencies in graph transformation systems. In: de Lara, J., Plump, D. (eds.) ICGT 2017. LNCS, vol. 10373, pp. 125–141. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61470-0_8
3. Corradini, A., et al.: On the essence of parallel independence for the double-pushout and sesqui-pushout approaches. In: Heckel, R., Taentzer, G. (eds.) Graph Transformation, Specifications, and Nets. LNCS, vol. 10800, pp. 1–18. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75396-6_1
4. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_4
5. 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. Foundations, vol. 1, pp. 163–246. World Scientific (1997). https://doi.org/10.1142/9789812384720_0003
6. Cota, É.F., Ribeiro, L., Bezerra, J.S., Costa, A., da Silva, R.E., Cota, G.: Using formal methods for content validation of medical procedure documents. Int. J. Med. Inf. **104**, 10–25 (2017). https://doi.org/10.1016/j.ijmedinf.2017.04.012
7. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H.: Constraints and application conditions: from graphs to high-level structures. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 287–303. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_21
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. MTCSAES. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
9. Ehrig, H., Habel, A., Padberg, J., Prange, U.: Adhesive high-level replacement categories and systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 144–160. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_12
10. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation - part II: single pushout approach and comparison with double pushout approach. In: Handbook of Graph Grammars, pp. 247–312. World Scientific (1997). https://doi.org/10.1142/9789812384720_0004
11. Johnstone, P.T.: Sketches of an Elephant: A Topos Theory Compendium, vol. 2. Oxford University Press, Oxford (2002)
12. Lack, S., Sobocinski, P.: Adhesive and quasiadhesive categories. ITA **39**(3), 511–545 (2005). https://doi.org/10.1051/ita:2005028
13. Lambers, L., Born, K., Orejas, F., Strüber, D., Taentzer, G.: Initial conflicts and dependencies: critical pairs revisited. In: Heckel, R., Taentzer, G. (eds.) Graph Transformation, Specifications, and Nets. LNCS, vol. 10800, pp. 105–123. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75396-6_6

14. Lambers, L., Ehrig, H., Orejas, F.: Efficient conflict detection in graph transformation systems by essential critical pairs. ENTCS **211**, 17–26 (2008). https://doi.org/10.1016/j.entcs.2008.04.026
15. MacLane, S., Moerdijk, I.: Sheaves in Geometry and Logic: A First Introduction to Topos Theory. Springer, New York (2012). https://doi.org/10.1007/978-1-4612-0927-0
16. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Softw. Syst. Model. **6**(3), 269–285 (2007). https://doi.org/10.1007/s10270-006-0044-6
17. Oliveira, M., Ribeiro, L., Cota, É., Duarte, L.M., Nunes, I., Reis, F.: Use case analysis based on formal methods: an empirical study. In: Codescu, M., Diaconescu, R., Ţuţu, I. (eds.) WADT 2015. LNCS, vol. 9463, pp. 110–130. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28114-8_7
18. Taentzer, G.: AGG: a graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_35

Characterisation of Parallel Independence in AGREE-Rewriting

Michael Löwe(✉)

FHDW Hannover, Freundallee 15, 30173 Hannover, Germany
michael.loewe@fhdw.de

Abstract. AGREE is a new approach to algebraic graph transformation. It provides sophisticated mechanisms for object cloning and non-local manipulations. It is for example possible to delete all edges in a graph by a single rule application. Due to these far-reaching effects of rule application, Church-Rosser results are difficult to obtain. Currently, there are only sufficient conditions for parallel independence, i.e. there are independent situations which are not classified independent by the existing conditions. In this paper, we characterise *all* independent situations.

1 Introduction

The AGREE approach to algebraic graph transformation has been introduced in [1] as a rewrite mechanism in categories with partial arrow classifiers.[1] It comes equipped with comfortable control mechanisms for item cloning. Besides that, it allows global manipulations of the objects that are rewritten. In the category of graphs for example, a rule ϱ_E and a rule ϱ_D can be formulated that erases respectively duplicates all edges in the graph it is applied to.[2] These global effects complicate the analysis of AGREE-rewrites wrt. parallel independence and Church-Rosser properties. Currently, there are only parallel independence results available for the sub-set of so-called local AGREE-rewrites, compare [2].

But there are non-local *and* independent rewrites. A simple example in the category of graphs is the pair of rules ϱ_E and ϱ_D mentioned above: If we duplicate all edges and thereafter erase all edges or if we erase all edges and thereafter duplicate all remaining (none) edges, we obtain the same result, namely a discrete graph with the same set of vertices as in the beginning. In this paper, we develop better criteria for parallel independence than in [2] that are also able to handle rules with such global effects. The major tool for this purpose is the modelling of AGREE-rewrites as gluing constructions that have been introduced in [10].

Definition 1 (Gluing). *The gluing of two pairs of morphisms* $(l : K \to L, r : K \to R)$ *and* $(m : L' \to L, n : L' \to G)$ *in a category is given by the diagram in*

[1] For details compare Definition 2 below.

[2] This is mainly due to a pullback construction that is part of definition of rewrite in AGREE. For details compare Sect. 3.

L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 118–133, 2018.
https://doi.org/10.1007/978-3-319-92991-0_8

$$\begin{tikzcd}
L & & K \arrow[ll, "l"'] \arrow[rr, "r"] & & R \\
& \text{(PB)} & & \text{(FPC)} & \\
L' \arrow[uu, "m"] \arrow[dd, "n"'] & & K' \arrow[ll, dashed, "l'"'] \arrow[rr, dashed, "r'"] \arrow[uu, dashed, "t"] \arrow[dd, dashed, "u"'] & & R' \arrow[uu, dashed, "p"'] \arrow[dd, dashed, "q"'] \\
& \text{(FPC)} & & \text{(PO)} & \\
G & & D \arrow[ll, dashed, "g"'] \arrow[rr, dashed, "h"] & & H
\end{tikzcd}$$

Fig. 1. Gluing

Fig. 1, where (i) (l', t) *is pullback of* (l, m), (ii) (u, g) *and* (r', p) *are final pullback complements of* (l', n) *resp.* (t, r), *and* (iii) (h, q) *is pushout of* (u, r').

The gluing construction provides a general framework into which all algebraic approaches to graph transformation can be integrated. If we interpret the span (l, r) as rule and the span (m, n) as match,

1. a double-pushout rewrite [4] requires monic l, isomorphic m, and that (u, g) is not only final pullback complement but also pushout complement,
2. a single-pushout rewrite at conflict-free match [8,9] requires monic l and isomorphic m, and
3. a sesqui-pushout rewrite [3] requires isomorphic m only.

In Sect. 3, we integrate AGREE-rewriting with *right-linear* rules into this framework.[3] AGREE is the first approach that does not require the m-part of a match to be an isomorphism. Instead the n-part must be monic. By this integration, we obtain a straightforward notion of parallel independence in Sect. 4: two matches in the same object are parallel independent, if there are residuals for both matches after the rewrite with the other rule. Finally, Sect. 5 provides "syntactical", easy to check, and characterising criteria for parallel independence. The conclusion in Sect. 6 interprets the obtained results on an intuitive level and presents the independence of the rules ϱ_E and ϱ_D (see above).

2 Preliminaries

This section introduces fundamental notions, requirements, and results that make up the categorical framework for the following sections. We assume for the underlying category $\mathcal{C}$

P1 that it has all finite limits and colimits,
P2 that it has all partial arrow classifiers, and
P3 that pushouts along monomorphisms are stable under pullbacks.

A pushout (f', g') of (g, f) as depicted in sub-diagram (1) of Fig. 2 is *stable under pullbacks*, if, for all commutative situations as in Fig. 2, the pair (f'_m, g'_m) is pushout of the span (g_m, f_m), if sub-diagrams (2)–(5) are pullbacks.[4]

[3] As in [2], we consider right-linear rules only.
[4] This is only half of the usually required adhesivity, compare [4].

$$\begin{tikzcd}
P \arrow[dr, "m_P"] \arrow[ddd, "f'_m"'] & & & D \arrow[lll, "g_m"'] \arrow[dl, "m"'] \arrow[ddd, "f_m"] \\
& B \arrow[d, "f'"] & A \arrow[l, "g"] \arrow[d, "f"'] & \\
& E & C \arrow[l, "g'"'] & \\
U \arrow[ur, "m_U"] & & & Q \arrow[lll, "g'_m"] \arrow[ul, "m_Q"']
\arrow[from=1-1, to=2-3, phantom, "(2)"]
\arrow[from=2-2, to=3-3, phantom, "(1)"]
\arrow[from=1-4, to=3-3, phantom, "(3)"]
\arrow[from=1-1, to=3-2, phantom, "(4)"]
\arrow[from=3-2, to=4-4, phantom, "(5)"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A^\bullet & X \arrow[l, dashed, "(m,f)^\bullet"'] \\
A \arrow[u, tail, "\eta_A"] & D \arrow[l, "f"] \arrow[u, tail, "m"'] \arrow[ul, phantom, "(PB)"]
\end{tikzcd}$$

Fig. 2. Stability and partial arrow classification

Definition 2 (Partial arrow classifiers). *A category has partial arrow classifiers, if there is a monomorphism $\eta_A : A \rightarrowtail A^\bullet$ for every object A with the following universal property: For every pair $(m : D \rightarrowtail X, f : D \to A)$ of morphisms with monic m, there is a unique morphism $(m,f)^\bullet : X \to A^\bullet$ such that (m,f) is the pullback of $(\eta_A, (m,f)^\bullet)$, compare right part of Fig. 2. In the following, the unique morphism $(m,f)^\bullet$ is also called* totalisation *of (m,f).*

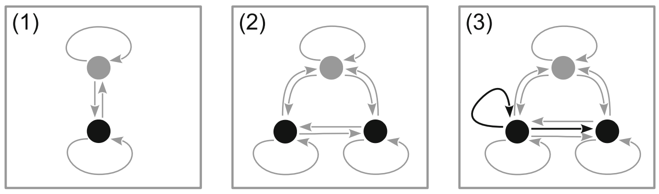


Fig. 3. Sample partial arrow classifiers in the category of graphs

Figure 3 depicts three sample partial arrow classifiers in the category of graphs: (1) for a single vertex, (2) for a graph with two vertices, and (3) for a graph with two vertices, a loop, and an edge between the vertices. The graph A that is classified is painted black, the grey parts are added by the classifier $A^\bullet$. The classifier provides the additional structure that is needed to map the objects that are not in the image of m in arbitrarily given pair $(m : D \rightarrowtail X, f : D \to A)$.

In a category with partial arrow classifiers, each monomorphism $m : A \rightarrowtail B$ induces a morphism $(m, \mathrm{id}_A)^\bullet : B \to A^\bullet$ in "the opposite" direction. We abbreviate $(m, \mathrm{id}_A)^\bullet$ by $m^\bullet$. The morphisms $\eta_A : A \rightarrowtail A^\bullet$, $m : A \rightarrowtail B$, and $m^\bullet : B \to A^\bullet$ produce a commutative triangle $m^\bullet \circ m = \eta_A$ such that (m, id_A) is pullback of $(\eta_A, m^\bullet)$. In this situation, we say that η_A is *reflected* along $m^\bullet$.

Definition 3 (Pullback reflection). *A morphism $b : B \to A$* is reflected along *a morphism $r : R \to A$ if there is morphism $\beta : B \to R$ such that (β, id_B) is pullback of (b, r). The morphism β is called the r-*reflection *of b.*

$$\begin{tikzcd}
B^\bullet & A \arrow[l, "b^\bullet"] & C \arrow[l, "c"] \arrow[ll, bend right, "(e,d)^\bullet"'] \\
B \arrow[u, tail, "\eta_B"] \arrow[ur, phantom, "\text{(PB)}"] & B \arrow[l, "\mathrm{id}_B"] \arrow[u, tail, "b"'] \arrow[ur, phantom, "\text{(PB)}"] & D \arrow[l, tail, "d"'] \arrow[l, dashed, tail, bend left, "x"] \arrow[u, tail, "e"'] \arrow[ll, bend left, "d"]
\end{tikzcd}$$

Fig. 4. Diagram for Fact F3

Categories with partial arrow classifiers possess many interesting properties. Some of these properties that are used below are:[5].

F1: Pushouts preserve monomorphisms.
F2: Pushouts along monomorphisms are pullbacks.
F3: Let $(b : B \rightarrowtail A, c : C \to A)$ be a co-span such that b is monic. Then $(e : D \rightarrowtail C, d : D \to B)$ is pullback of (b, c), if and only if $b^\bullet \circ c = (e, d)^\bullet$.
F4: If $e : B \rightarrowtail C$ and $c : C \to A$ are two morphisms such that e and $c \circ e$ are monic, then e is the c-reflection of $c \circ e$, if and only if $e^\bullet = (c \circ e)^\bullet \circ c$.[6]
F5: For $c : C \rightarrowtail B, b : B \to A$, and $a : C \rightarrowtail A$, $a^\bullet \circ b = c^\bullet$ implies $b \circ c = a$.[7]

Proof. (Fact F3) The situation is depicted in Fig. 4: ($\Rightarrow$) Since (e, d) is pullback of (c, b) and (id_B, b) is pullback of $(\eta_B, b^\bullet)$, pullback composition provides (e, d) as pullback of $(\eta_B, b^\bullet \circ c)$. Since (e, d) is also pullback of $(\eta_B, (e, d)^\bullet)$, the uniqueness of the totalisation provides $b^\bullet \circ c = (e, d)^\bullet$. ($\Leftarrow$) If $b^\bullet \circ c = (e, d)^\bullet$, then (b, id_B) is pullback of $(\eta_B, b^\bullet)$ and (e, d) is pullback of $(\eta_B, (e, d)^\bullet)$. Since $b^\bullet \circ (c \circ e) = (e, d)^\bullet \circ e = \eta_B \circ d$, there is $x : D \to B$ with (x, e) as pullback of (b, c) by pullback decomposition and $x \circ \mathrm{id}_B = d$ implying $x = d$. □

These facts imply some additional properties. Important for the rest of the paper are facts about the existence of certain *final pullback complements.*

Definition 4 (Final pullback complement). *Let (a, d) be the pullback of (b, c) as depicted in the left part of Fig. 5. The pair (d, c) is the final pullback complement of the pair (a, b) if for every collection (x, y, z, w) of morphisms such that (x, y) is pullback of (b, z) and $a \circ w = x$, there is unique morphism w^* providing $w^* \circ y = d \circ w$ and $c \circ w^* = z$.*

[5] Compare [6,7]. Facts F1 and F2 are consequences of the fact that all pushouts in a category with partial arrow classifiers are *hereditary. A pushout* (f', g') *of* (g, f) *as depicted in sub-diagram* (1) *of Fig.* 2 *is hereditary, if for all commutative situations as presented in Fig.* 2 *in which sub-diagrams* (2) *and* (3) *are pullbacks, and* m, m_P, and m_Q *are monomorphisms the following property is valid: the co-span* (f'_m, g'_m) *is pushout of the span* (g_m, f_m), *if and only if sub-diagrams* (4) *and* (5) *are pullbacks and* m_U *is monomorphism.* Hereditariness implies stability of pushouts under pullbacks along monomorphisms. For some of the following results, however, we need requirement P3 which does not require *monic* m, m_P, m_Q, and m_U.
[6] Special case of F3 for $d = \mathrm{id}_B$ and $b = c \circ e$.
[7] Consequence of F3 for $d = \mathrm{id}_B$.

$$\begin{tikzcd}
 & & P \arrow[lld, bend right, "x"'] \arrow[ld, "w"] \arrow[dd, "y"] \\
A \arrow[d, "b"] \arrow[rd, phantom, "(\mathrm{PB})"] & B \arrow[l, "a"] \arrow[d, "d"] & \\
C & D \arrow[l, "c"] & Q \arrow[l, dashed, "w^*"] \arrow[ll, bend left, "z"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A' \arrow[rd, "i"] \arrow[ddd, "b'"] \arrow[rrrd, phantom, "(3)"] & & & B' \arrow[lll, "a'"] \arrow[ld, "j"] \arrow[ddd, "d'"] \\
 & A \arrow[d, "b"] \arrow[rd, phantom, "(1)"] \arrow[ld, phantom, "(2)"] & B \arrow[l, "a"] \arrow[d, "d"] \arrow[rd, phantom, "(5)"] & \\
 & C \arrow[rd, phantom, "(4)"] & D \arrow[l, "c"] & \\
C' \arrow[ru, "q"] & & & D' \arrow[lll, "c'"] \arrow[lu, "p"]
\end{tikzcd}$$

Fig. 5. Final pullback complement

The next proposition states that final pullback complements and pullbacks are mutually dependent in some situations.[8]

Proposition 5 (Pullbacks and final pullback complements). *Let a commutative diagram as in Fig. 5 (right part) be given such that (d, c) is final pullback complement of (a, b) in sub-diagram (1), and sub-diagrams (2) and (3) are pullbacks. Then sub-diagram (4) is pullback, if and only if (d', c') is final pullback complement of (a', b').*

Partial arrow classifiers produce certain final pullback complements.

Proposition 6 (Final pullback complement by classification). *If (a, η_B) is pullback of $(\eta_A, (\eta_B, a)^\bullet)$ for a given morphism $a : A \to B$, then the pair $(\eta_B, (\eta_B, a)^\bullet)$ is the final pullback complement of (a, η_A).*

Proof. Let the tuple (x, y, z, w) in Fig. 6 be given such that (x, y) is pullback of (η_A, z) and $a \circ w = x$. Then y is monomorphism. Construct $w^* = (y, w)^\bullet$. This provides $\eta_B \circ w = w^* \circ y$ and (w, y) as pullback of (η_B, w^*). And, since (y, x) is the pullback of (η_A, z) and the pullback of $(\eta_A, (\eta_B, a)^\bullet \circ w^*)$, we can conclude $z = (\eta_B, a)^\bullet \circ w^*$, due to uniqueness of partial arrow totalisation for (y, x). Uniqueness of w^* is provided by the uniqueness of the totalisation of (y, w). □

Corollary 7 (Final pullback complements). *Every pair $(b : B \rightarrowtail C, a : A \to B)$ of morphisms with monic b has a final pullback complement.*

Proof. Consequence of Propositions 5 and 6. A proof can be found in [1].

The existence of certain final pullback complements which is guaranteed by Corollary 7 leads to another result which is important for the rest of the paper.

Corollary 8 (Weak final pullback complement adhesivity). *In each commutative diagram as in Fig. 7 in which sub-diagram* (5) *is a pushout along monic*

[8] For the proof, see [11].

$$\begin{tikzcd}
A^\bullet & B^\bullet \arrow[l, "(\eta_B,a)^\bullet"'] & Q \arrow[l, dashed, "w^*=(y,w)^\bullet"] \arrow[ll, dashed, bend right, "z"'] \\
A \arrow[u, tail, "\eta_A"'] & B \arrow[l, "a"] \arrow[u, tail, "\eta_B"] & P \arrow[l, dashed, "w"] \arrow[u, dashed, "y"'] \arrow[ll, dashed, bend left, "x"]
\end{tikzcd}$$

Fig. 6. Classifier provides final pullback complement

a, sub-diagram (1) *is pullback, and* (a', f) *is final pullback complement of* (e, a) *in sub-diagram* (2)*, the following property is valid:* (b', d') *is pushout of* (a', c') *if and only if sub-diagram* (3) *is pullback and in sub-diagram* (4) (d', h) *is final pullback complement of* (g, d).

$$\begin{tikzcd}
F \arrow[ddddddd, "b'"'] & & & & & & E \arrow[llllll, tail, "a'"'] \arrow[ddl, "e"'] \arrow[ddddddd, "c'"] \\
& & & (2) & & & \\
& & B \arrow[from=1-1, "f"] \arrow[dd, "b"] & & A \arrow[ll, tail, "a"] \arrow[dd, "c"'] & & \\
& (3) & & (5) & & (1) & \\
& & D & & C \arrow[ll, tail, "d"'] & & \\
& & & (4) & & & \\
& & & & & & \\
H \arrow[uuurr, "h"] & & & & & & G \arrow[llllll, tail, "d'"] \arrow[uuull, "g"']
\end{tikzcd}$$

Fig. 7. Weak final pullback complement adhesivity

Proof. Due to Corollary 7, we can *always construct* sub-diagram (4) as final pullback complement. By Fact F2, (5) is pullback which implies that (2)+(5) is pullback. Since $g \circ c' = c \circ e$, final pullback complement (4) provides morphism b' such that $h \circ b' = b \circ f$ and $b' \circ a' = d' \circ c'$. Now the premises for Proposition 5 (*if-part*) are satisfied, namely (2) and (4) are final pullback complements and (5) and (1) are pullbacks. Thus, also sub-diagram (3) is pullback. Finally, stability of pushouts under pullbacks guarantees that (b', d') is pushout of (a', c'). □

3 AGREE Rewrites as Gluing Constructions

In this section, we show that AGREE-rewrites are gluing constructions in the sense of [10]. As in [2], we only consider rules whose right-hand sides are monic.

Definition 9 (AGREE rule, match, and rewrite). *A right-linear AGREE* rule ϱ *is a triple* $\varrho = (l : K \to L, t : K \rightarrowtail T_K, r : K \rightarrowtail R)$ *of morphism such that* t *and* r *are monic. A* match *in an object* G *is a monic morphism* $m : L \rightarrowtail G$*. A* rewrite *with a rule at a match is constructed as depicted in Fig. 8:*

$$\begin{tikzcd}
& L^\bullet & & T_K \arrow[ll, "(t,l)^\bullet"'] & & \\
L \arrow[ur, tail, "\eta_L"] \arrow[dr, tail, "m"] & & K \arrow[ll, "l"'] \arrow[ur, tail, "t"] \arrow[rr, tail, "r"] \arrow[dr, tail, "n"] \arrow[dl, phantom, "\text{(PB)}"] \arrow[dr, phantom, "\text{(PO)}", shift left=6] & & R \arrow[dr, tail, "p"] & \\
& G \arrow[uu, "m^\bullet"] & & D \arrow[ll, "g"] \arrow[uu, "n'"'] \arrow[rr, tail, "h"'] & & H
\end{tikzcd}$$

Fig. 8. AGREE rewrite

1. *Let $\eta_L : L \rightarrowtail L^\bullet$ be the partial arrow classifier for the rule's left-hand side.*
2. *Construct pullback (g, n') of the pair $((t,l)^\bullet : K' \to L^\bullet, m^\bullet : G \to L^\bullet)$.*
3. *Let $n : K \rightarrowtail D$ be the unique morphism such that $g \circ n = m \circ l$ and $n' \circ n = t$.*
4. *Construct $(h : D \rightarrowtail H, p : R \rightarrowtail H)$ as pushout of $(r : K \rightarrowtail R, n : K \rightarrowtail D)$.*

The span $(g : D \to G, h : D \rightarrowtail H)$ is called the trace *of the derivation. The trace of a derivation with rule ϱ at match m is also denoted by $\varrho\langle m \rangle$.* □

The right-hand side of every trace is monic, since pushouts preserve monomorphisms by Fact F1. By the next definition, AGREE-rules and matches are made compatible to the gluing construction of Definition 1. The following theorem proves that AGREE-rewrites are special gluings.

Definition 10 (AGREE-flavoured rule and global match). *A span of morphisms $\sigma = (l : K \to L, r : K \rightarrowtail R)$ is an AGREE-flavoured gluing rule if there is an AGREE rule $\varrho = (l' : K' \to L', t : K' \rightarrowtail K, r' : K' \rightarrowtail R')$ such that:*[9]

1. *$\eta_L : L' \rightarrowtail L$ is the partial arrow classifier for L', i.e. $L = L'^\bullet$,*
2. *l is the totalisation of the partial morphism (t, l'), i.e. $l = (t, l')^\bullet$, and*
3. *(r, t') is pushout of (t, r').*

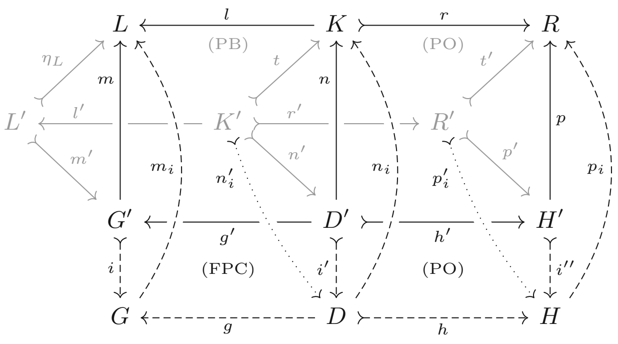
A span $(m : G' \to L, i : G' \to G)$ is a global match *for an AGREE-flavoured rule in an object G, if:*[10]

1. *i is the identity on G, i.e. $G' = G$ and $i = \mathrm{id}_G$, and*
2. *there is a monomorphism $m' : L' \rightarrowtail G'$ such $m = m'^\bullet$ which is called* base match *of m (for ϱ) in the following.* □

Theorem 11. *An AGREE rewrite is a gluing construction.*

Proof. For the proof consider Fig. 9 which depicts an AGREE rule (l', t, r') together with the induced AGREE-flavoured gluing rule (l, r), compare Definition 10. A given match $m' : L \rightarrowtail G$ for the AGREE rule (l', t, r') induces a global match $(m : G \to L, \mathrm{id}_G : G \to G)$ for (l, r). Thus in Fig. 9, $G = G'$ and

[9] Compare top part of Fig. 9.
[10] Compare left part of Fig. 9.

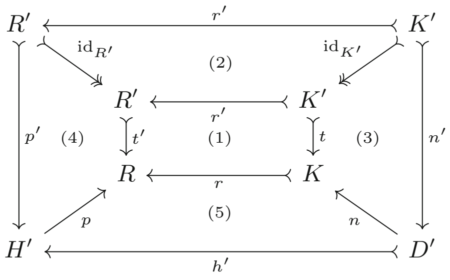


$$\begin{tikzcd}
 & L \arrow[rr, phantom, "\text{(PB)}"] & & K \arrow[ll, "l"'] \arrow[rr, "r"] \arrow[rr, phantom, "\text{(PO)}"] & & R \\
L' \arrow[ur, "\eta_L"] \arrow[dr, "m'"'] & & K' \arrow[ll, "l'"'] \arrow[ur, "t"] \arrow[dr, "n'"] \arrow[rr, "r'"] \arrow[ddrr, dotted, "n'_i"] & & R' \arrow[ur, "t'"] \arrow[dr, "p'"] \arrow[ddrr, dotted, "p'_i"] & \\
 & G' \arrow[uu, "m"] \arrow[d, dashed, "i"] & & D' \arrow[ll, "g'"] \arrow[rr, "h'"] \arrow[uu, "n"] \arrow[d, dashed, "i'"] & & H' \arrow[uu, "p"] \arrow[d, dashed, "i''"] \\
 & G \arrow[uuu, dashed, bend right, "m_i"'] \arrow[rr, phantom, "\text{(FPC)}", shift left=4] & & D \arrow[ll, dashed, "g"] \arrow[rr, dashed, "h"'] \arrow[uuu, dashed, bend right, "n_i"'] \arrow[rr, phantom, "\text{(PO)}", shift left=4] & & H \arrow[uuu, dashed, bend right, "p_i"']
\end{tikzcd}$$

Fig. 9. AGREE-flavoured gluing construction

$i = \mathrm{id}_G$ for the rest of the proof. The pair (n, g') is constructed as pullback of (l, m) in both rewrite approaches and the morphism $n' : K' \rightarrowtail D$ making the diagram commutative provides $(\mathrm{id}_{K'}, n')$ as pullback of (n, t) due to composition and decomposition properties of the pullbacks $(\mathrm{id}_{L'}, m')$ of (m, η_L), (t, l') of (η_L, l), (g', n) of (m, l), and (l', n') of (m', g').

Next, let (p', h') be the right-hand side of the AGREE rewrite step with rule (l', t, r') at match m', i.e. (p', h') is the pushout of (r', n'). Then we get a unique morphism $p : H' \to R$ making the diagram in Fig. 9 commutative.



$$\begin{tikzcd}
R' \arrow[dr, "\mathrm{id}_{R'}"] \arrow[ddd, "p'"'] & & & K' \arrow[lll, "r'"'] \arrow[dl, "\mathrm{id}_{K'}"'] \arrow[ddd, "n'"] \\
 & R' \arrow[d, "t'"] \arrow[r, phantom, "(2)", shift left=5] & K' \arrow[l, "r'"] \arrow[d, "t"'] & \\
 & R \arrow[r, phantom, "(1)", shift left=5] \arrow[l, phantom, "(4)"] & K \arrow[l, "r"'] \arrow[r, phantom, "(3)"] & \\
H' \arrow[ur, "p"] \arrow[rrr, phantom, "(5)", shift left=5] & & & D' \arrow[lll, "h'"] \arrow[ul, "n"']
\end{tikzcd}$$

Fig. 10. Rewrite details

The details of the resulting (right-hand) situation are depicted in Fig. 10:

1. Sub-diagram (1) is pushout along monic r'.
2. The pair (p', h') in the outer rectangle is pushout of (r', n').
3. In sub-diagram (2), $(r', \mathrm{id}_{R'})$ is final pullback complement of $(\mathrm{id}_{K'}, r')$.
4. Sub-diagram (3) is pullback.

With these premises, Corollary 8 guarantees that sub-diagram (4) is pullback and the pair (h', p) is final pullback complement of (n, r) in sub-diagram (5).

Since we have chosen the match such that $G' = G$ and $i = \mathrm{id}_G$, we can choose $i' = \mathrm{id}_D$, $i'' = \mathrm{id}_H$, $g' = g$, and $h' = h$ and immediately obtain the remaining final pullback complement (i', g) and pushout (i'', h) in Fig. 9. □

Corollary 12. *An AGREE rewrite step is a gluing construction consisting of a pullback and a pushout (complement).*

Proof. In the commutative diagram in Fig. 9, the pair (r, p) is pushout, since (r, t') and (p', h') are pushouts and pushouts decompose.

We conclude this section by a notion of *local match*. Local matches provide a better integration of AGREE-rewrites into the gluing framework of Definition 1, since local matches are real spans of morphisms.

Definition 13 (Local match). *If $\sigma = (l : K \to L, r : K \rightarrowtail R)$ is a rule that is flavoured by the AGREE rule $\varrho = (l' : K' \to L', t : K' \rightarrowtail K, r' : K' \rightarrowtail R')$, a span $(m : G' \to L, i : G' \rightarrowtail G)$ is a* local match, *if*

1. *i is monic,*
2. *there is a* base match *$m' : L' \rightarrowtail G'$ such that $m = m'^{\bullet}$, and*
3. *given the pullbacks $(g' : D' \to G', n : D' \to K)$ and $(g : D \to G, n_i : D \to K)$ of (m, l) and $((i \circ m')^{\bullet}, l)$ resp. and the induced unique morphism $i' : D' \to D$ for $(i \circ m')^{\bullet} \circ i = m$, (i', g) is final pullback complement of (g', i).*[11]

The global match that is induced by a local match $(m : G' \to L, i : G' \rightarrowtail G)$ is $\big((i \circ m')^{\bullet} : G \to L, \mathrm{id}_G : G \to G\big)$. □

In the following, the totalisation $(i \circ m')^{\bullet}$ is also called m_i, compare Fig. 9.

Proposition 14 (Local match). *The gluing construction for a rule σ that is flavoured by an AGREE rule ϱ at a local match (m, i) results in the same trace as the gluing construction for σ at the induced global match m_i.*

Proof. Compare Fig. 9. The gluing of (l, r) at the global match produces pullback (g, n_i) of (l, m_i) on the left-hand and final pullback complement (h, p_i) of (r, n_i) on the right-hand side. Since the rule is AGREE-flavoured, we get morphisms n'_i and p'_i such that $n_i \circ n'_i = t$ and (h, p'_i) is pushout of (n'_i, r').

Building the gluing at the local match, we construct an AGREE rewrite with (l', t, r') at match m'. Thus, we get pullbacks (n, g') of (l, m) and (n', l') of (g', m'), final pullback complement (h', p) of (n, r) and pushout (p', h') of (r', n').

Since i is monic, m' is the i-reflection of $i \circ m'$ and $m_i \circ i = m$ by Fact F4. Thus, there is i' such that $n_i \circ i' = n$ and (g', i') is pullback of (i, g). Definition 13(3) makes sure that (g, i') is final pullback complement of (g', i).

Since $g \circ n'_i = i \circ m' \circ l' = i \circ g' \circ n' = g \circ (i' \circ n')$ and $n_i \circ (i' \circ n') = n \circ n' = t = n_i \circ n'_i$, (n_i, g) being pullback guarantees that $i' \circ n' = n'_i$. Due to (h', p') being pushout and $h \circ i' \circ n' = h \circ n'_i = p'_i \circ r'$, there is i'' such that $i'' \circ p' = p'_i$ and $i'' \circ h' = h \circ i'$. Pushout decomposition guarantees that (i'', h) is pushout of (i', h'). □

[11] Compare Fig. 9 where $(i \circ m')^{\bullet}$ is called m_i!.

4 Parallel Independence

Any form of parallel independence analysis investigates situations in which rewrite rules $\chi_1 : L_1 \rightsquigarrow R_1$ and $\chi_2 : L_2 \rightsquigarrow R_2$ can be applied to object G at matches $m_1 :: L_1 \rightsquigarrow G$ resp. $m_2 : L_2 \rightsquigarrow G$ leading to derivations $\chi_1 \langle m_1 \rangle :: G \rightsquigarrow H_1$ and $\chi_2 \langle m_2 \rangle :: G \rightsquigarrow H_2$. The aim is to formulate conditions that make sure that

- χ_1 can be applied at a *residual match* $m_1^* :: L_1 \rightsquigarrow H_2$ after χ_2 has been applied at m_2 resulting in object H_2,
- χ_2 can be applied at a *residual match* $m_2^* :: L_2 \rightsquigarrow H_1$ after χ_1 has been applied at m_1 resulting in object H_1, and
- both sequences, namely $G \overset{\chi_1\langle m_1\rangle}{\rightsquigarrow} H_1 \overset{\chi_2\langle m_2^*\rangle}{\rightsquigarrow} H_{12}$ and $G \overset{\chi_2\langle m_2\rangle}{\rightsquigarrow} H_2 \overset{\chi_1\langle m_1^*\rangle}{\rightsquigarrow} H_{21}$, produce the *same result*.

The notion "same result" means in its *weak* form that H_{12} and H_{21} are equal or at least structurally equal.[12] In a *stronger* form, "same result" means that the two derivation sequences are equal, i.e. $\chi_2 \langle m_2^* \rangle \circ \chi_1 \langle m_1 \rangle = \chi_1 \langle m_1^* \rangle \circ \chi_2 \langle m_2 \rangle$. The strong form can only be formulated in a setup where derivations are represented by some sort of morphisms for which a composition is well-defined.

The central notion in this analysis is the notion of *residual match*. Intuitively, a match $m_1^* : L_1 \rightsquigarrow H_2$ is a residual for $m_1 : L_1 \rightsquigarrow G$ in the transformation result of $\chi_2 \langle m_2 \rangle :: G \rightsquigarrow H_2$, if m_1 and m_1^* are the "same" match or $\chi_2 \langle m_2 \rangle$ maps m_1 to m_1^* by some suitable mapping mechanism of matches in G to matches in H_2.

In a setup in which matches and derivations are represented by morphisms in a suitable category, the most straightforward idea for a match mapping is morphism composition: If matches $m_1 : L_1 \to G$ and $m_2 : L_2 \to G$ for rules χ_1 and χ_2 are morphisms and the derivations of G to H_1 via rule χ_1 at match m_1 and to H_2 via rule χ_2 at match m_2 are represented by morphisms $\chi_1 \langle m_1 \rangle : G \to H_1$ and $\chi_2 \langle m_2 \rangle : G \to H_2$, the residual of m_1 after $\chi_2 \langle m_2 \rangle$ is $m_1^* = \chi_2 \langle m_2 \rangle \circ m_1$ and the residual of m_2 after $\chi_1 \langle m_1 \rangle$ is $m_2^* = \chi_1 \langle m_1 \rangle \circ m_2$.

In our context of AGREE-flavoured rules, traces of rewrites are represented by special spans, compare Definition 9. Thus, we have to pass to the category of spans in order to exploit the ideas for residuals described above:

Definition 15 (Span category). *A* concrete (right-linear) span *is a pair of morphisms* $\langle l : K \to L, r : K \rightarrowtail R \rangle$ *such that* r *is monic. Two concrete spans* $\langle l : K \to L, r : K \rightarrowtail R \rangle$ *and* $\langle l' : K' \to L, r' : K' \rightarrowtail R \rangle$ *are equivalent, if there is an isomorphism* $i : K \leftrightarrow K'$ *such that* $l' \circ i = l$ *and* $r' \circ i = r$. *An* abstract span $(l : K \to L, r : K \rightarrowtail R)$ *is the class of all concrete spans equivalent to* $\langle l : K \to L, r : K \rightarrowtail R \rangle$. *The* category of spans *has the same objects as the underlying category and abstract spans as morphisms. The* identity *for object* A *is defined by* $(\mathrm{id}_A : A \to A, \mathrm{id}_A : A \to A)$. Composition *of two morphisms* $(l : K \to L, r : K \rightarrowtail R)$ *and* $(p : H \to R, q : H \rightarrowtail S)$ *is defined by* $(p, q) \circ (l, r) = (l \circ p', q \circ r')$ *where* (p', r') *is the pullback of* (r, p).

[12] Equal up to isomorphism.

Fact 16 (Gluing construction). *A gluing construction is a commutative diagram in the category of abstract spans.*

With these prerequisites, we can precisely express what we mean by residual in AGREE-flavoured rewriting.

Definition 17 (Residual). *A* global *match $m : G \to L$ for a gluing rule with AGREE-flavour* has a residual *m^* in a trace $(g : D \to G, h : D \rightarrowtail H)$, if $m^* = (m \circ g, h) = (g, h) \circ (m, \mathrm{id})$ is* local *match for p.*

By Theorem 11 and Proposition 14, we know that AGREE rewrites at global and local matches are gluing constructions. Exploiting composition and decomposition properties of gluings, we obtain the following crucial result.

Theorem 18 (Church-Rosser). *Let $\sigma_1 \langle m_1 \rangle = (g_1, h_1)$ and $\sigma_2 \langle m_2 \rangle = (g_2, h_2)$ be two traces starting at the same object G, one for the application of AGREE-flavoured rule σ_1 at global match m_1 and the other for the application of rule σ_2 at global match m_2. If m_1^* and m_2^* are residuals for m_1 and m_2 respectively, then $\sigma_1 \langle m_1^* \rangle \circ \sigma_2 \langle m_2 \rangle = \sigma_2 \langle m_2^* \rangle \circ \sigma_1 \langle m_1 \rangle$, where $\sigma_1 \langle m_1^* \rangle$ and $\sigma_2 \langle m_2^* \rangle$ are the traces of the rewrites with σ_1 and σ_2 at m_1^* and m_2^* respectively.*

For the proof, we need the following lemmata (proofs can be found in [11]).[13]

Lemma 19 (Gluing decomposition). *If the pair $[(i, j), (h, k)]$ is gluing of $[(a, b), (c, d)]$, $[(x, y), (p, q)]$ is gluing of $[(a, b), (m, n) \circ (c, d)]$, and b, d, and n are monic, then there is a span (r, s) such that $(r, s) \circ (i, j) = (x, y)$ and $[(r, s), (p, q)]$ is gluing of $[(h, k), (m, n)]$, compare Fig. 11.*

Lemma 20 (Gluing composition). *If $[(i, j), (h, k)]$ is gluing of $[(a, b), (c, d)]$, $[(r, s), (p, q)]$ is gluing of $[(h, k), (m, n)]$, and b, d, and n are monic, then the pair $[(r, s) \circ (i, j), (p, q)]$ is gluing of $[(a, b), (m, n) \circ (c, d)]$, compare again Fig. 11.*

Proof (for Theorem 18). Let $[(g_{12}, h_{12}), (u, v)]$ be the gluing construction for rewrite rule σ_1 at the residual $m_1^* = (m_1 \circ g_2, h_2)$ such that (g_{12}, h_{12}) is the trace $\sigma_1 \langle m_1^* \rangle$. By Lemma 19, there is a span (x, y) such that $[(x, y), (g_{12}, h_{12})]$ is the gluing of $[(g_1, h_1), (g_2, h_2)]$. Due to Lemma 20, the composition of this gluing square with the gluing construction which stands for the application of rule σ_2 at match m_2 represents the application of rule σ_2 at residual $m_2^* = (m_2 \circ g_1, h_1)$. Thus, the span (x, y) is the trace $\sigma_2 \langle m_2^* \rangle$ of the rewrite step with rule σ_2 at residual m_2^*. Due to Fact 16, $\sigma_1 \langle m_1^* \rangle \circ \sigma_2 \langle m_2 \rangle = \sigma_2 \langle m_2^* \rangle \circ \sigma_1 \langle m_1 \rangle$. □

Theorem 18 justifies the following definition of *parallel independence*.

Definition 21 (Parallel independence). *Matches m_1 and m_2 for AGREE-flavoured rules p_1 and p_2 in the same object are* parallel independent*, if they possess mutual residuals, i.e. if there are residuals for m_1 and m_2 in the trace of the rewrite with p_2 at m_2 and in the trace of the rewrite with p_1 at m_1 respectively.*

[13] These two lemmata demonstrate that gluing constructions in categories of abstract (right linear) spans possess the same composition and decomposition properties as simple pushouts in arbitrary categories.

$$\begin{tikzcd}
 & A & & B \arrow[ll, tail, "a"'] \arrow[rr, tail, "b"] & & C \\
 & D \arrow[u, tail, "c"] \arrow[d, tail, "d"] & & E \arrow[ll, tail, "a'"'] \arrow[rr, tail, "b'"] \arrow[u, tail, "e"] \arrow[d, tail, "f"] & & F \arrow[u, tail, "i"] \arrow[d, tail, "j"] \\
S \arrow[ur, dashed, "m'"] \arrow[uur, dashed, bend left, "com'"] \arrow[ddr, dashed, tail, "d'"'] \arrow[dddr, dashed, tail, bend right, "n\circ d'"'] & & T \arrow[ll, dashed, tail, "t"'] \arrow[rr, dashed, tail, "u"] \arrow[uur, dashed, "v"] \arrow[ur, "\alpha"] \arrow[ddr, tail, "\gamma"] \arrow[dddr, dashed, tail, "w"'] & & U \arrow[uur, dashed, "x"] \arrow[ur, "\beta"] \arrow[ddr, tail, "\varepsilon"] \arrow[dddr, dashed, tail, "y"'] & \\
 & G & & H \arrow[ll, tail, "h"'] \arrow[rr, tail, "k"] & & K \\
 & L \arrow[u, tail, "m"] \arrow[d, tail, "n"] & & M \arrow[ll, dotted, tail, "p'"] \arrow[rr, dotted, tail, "\lambda"] \arrow[u, dotted, tail, "\delta"] \arrow[d, dotted, tail, "n'"] & & N \arrow[u, dashed, tail, "r"'] \arrow[d, dashed, tail, "s"] \\
 & P & & Q \arrow[ll, tail, "p"] \arrow[rr, tail, "q"'] & & R
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "{(a,b)}"] \arrow[d, "{(c,d)}"] \arrow[dd, bend right=60, "{(m,n)\circ(c,d)}"'] & C \arrow[d, "{(i,j)}"'] \arrow[dd, bend left=60, "{(x,y)}"] \\
G \arrow[r, "{(h,k)}"] \arrow[d, "{(m,n)}"] & K \arrow[d, dashed, "{(r,s)}"'] \\
P \arrow[r, "{(p,q)}"'] & R
\end{tikzcd}$$

Fig. 11. Composition and decomposition of gluings

5 Characterisation of Parallel Independence

In this section, we analyse the notion of parallel independence given by Definition 21. The aim is to find characterising conditions that are easy to check without constructing (parts of) the rewrites and taking only the structure of the participating rules and matches into account.

We start by investigating necessary conditions for parallel independence. First we analyse the structure of the left-hand sides of independent rule application. For this purpose consider Fig. 12. It depicts the pullback $(\pi_1 : L_{12} \rightarrowtail L'_1, \pi_2 : L_{12} \rightarrowtail L'_2)$ of two base matches m'_1 and m'_2 for the left-hand sides $\eta_{L_1} : L'_1 \rightarrowtail L_1$ and $\eta_{L_2} : L'_2 \rightarrowtail L_2$ of two rules in the same object G. Both pullback morphisms are monic since the base matches are required to be monic.

Proposition 22 (Necessary condition – left-hand sides). *If a global match m_2 for an AGREE-flavoured rule (l_2, r_2) has a residual in the trace of rule (l_1, r_1) at global match m_1 and $(\pi_1 : L_{12} \rightarrowtail L'_1, \pi_2 : L_{12} \rightarrowtail L'_2)$ is the pullback of the base matches $m'_1 : L'_1 \rightarrowtail G$ and $m'_2 : L'_2 \rightarrowtail G$, then $(\pi_2, \pi_1)^\bullet$ has a l_1-reflection (α_1 in Fig. 12) and π_1 has a monic l'_1-reflection (β_1 in Fig. 12).*

Proof. For the proof compare Fig. 12. If match m_2 has residual in the trace $(g_1 : D_1 \rightarrow G, h_1 : D_1 \rightarrowtail H_1)$, there is a base match $\gamma_1 : L'_2 \rightarrowtail D_1$ such that $m_2 \circ g_1 = \gamma_1^\bullet$, compare Definition 13(2). Since $m_2 = {m'_2}^\bullet$, we obtain (i) ${m'_2}^\bullet \circ g_1 = \gamma_1^\bullet$, (ii) $m'_2 = g_1 \circ \gamma_1$ and $(g_1 \circ \gamma_1)^\bullet \circ g_1 = \gamma_1^\bullet$ by Fact F5, and (iii) that γ_1 is the g_1-reflection of m'_2 by Fact F4. By Fact F3, $m_1 \circ m'_2 = {m'_1}^\bullet \circ m'_2 = (\pi_2, \pi_1)^\bullet$. Since (n_1, g_1) is pullback of (l_1, m_1) and pullbacks compose, $(n_1 \circ \gamma_1, \mathrm{id}_{L'_2})$ is pullback of $(l_1, m_1 \circ m'_2) = (l_1, (\pi_2, \pi_1)^\bullet)$. Thus, $(\pi_2, \pi_1)^\bullet$ has a l_1-reflection, namely $n_1 \circ \gamma_1$ which is also called α_1 in Fig. 12.

Since (t_1, l'_1) is pullback of $(\eta_{L'_1}, l_1)$ and $l_1 \circ (\alpha_1 \circ \pi_2) = (\pi_2, \pi_1)^\bullet \circ \pi_2 = m_1 \circ m'_2 \circ \pi_2 = m_1 \circ m'_1 \circ \pi_1 = \eta_{L'_1} \circ \pi_1$, there is morphism $\beta_1 : L_{12} \rightarrowtail K'_1$

$$\begin{tikzcd}
 & & L_1 & K_1 \arrow[l, "l_1"] \\
L'_2 \arrow[rru, "(\pi_2,\pi_1)^\bullet"] \arrow[rrru, dashed, bend left, "\alpha_1"] \arrow[dd, "\eta_{L'_2}"'] \arrow[rrdd, "m'_2"] \arrow[rrrddd, dashed, bend right, "\gamma_1"'] & & & \\
 & L_{12} \arrow[lu, "\pi_2"'] \arrow[r, "\pi_1"] \arrow[rr, dashed, bend right, "\beta_1"'] & L'_1 \arrow[uu, "\eta_{L'_1}"] \arrow[d, "m'_1"] & K'_1 \arrow[l, "l'_1"'] \arrow[uu, "t_1"] \arrow[dd, "n'_1"] \\
L_2 & & G \arrow[ll, "m_2"] \arrow[uuu, bend right, "m_1"'] & \\
 & & & D_1 \arrow[lu, "g_1"] \arrow[uuuu, bend right, "n_1"']
\end{tikzcd}$$

Fig. 12. Parallel independence: left-hand side

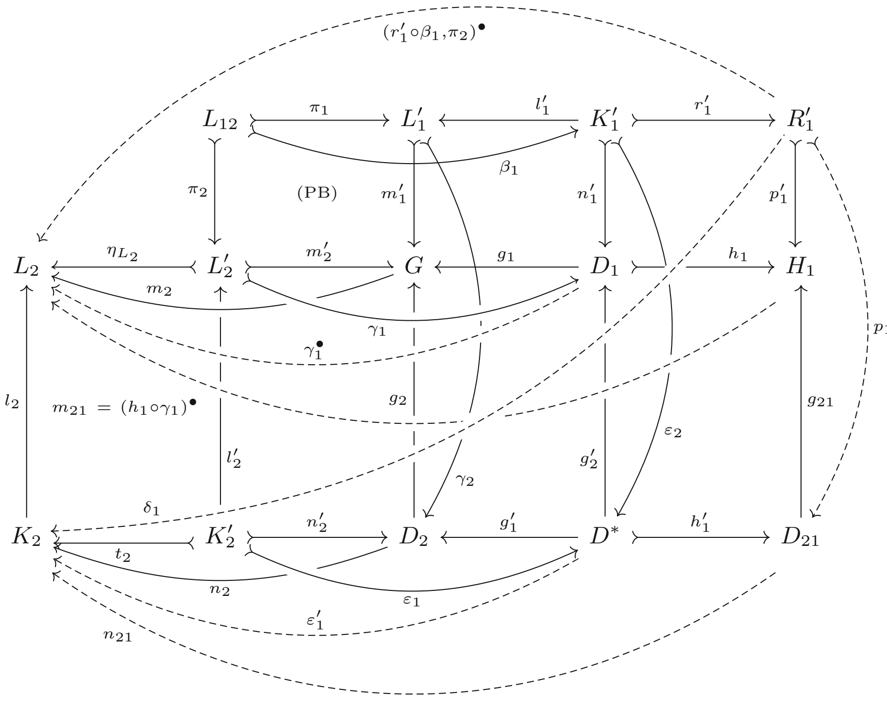
with $l'_1 \circ \beta_1 = \pi_1$ and $t_1 \circ \beta_1 = \alpha_1 \circ \pi_2$. Pullback decomposition [of (π_1, π_2) by (l'_1, n'_1)] guarantees that (π_2, β_1) is pullback of (γ_1, n'_1). Pullbacks $(\alpha, \mathrm{id}_{L'_2})$ of $(l_1, (\pi_2, \pi_1)^\bullet)$ and $(\pi_2, \mathrm{id}_{L_{12}})$ of $(\mathrm{id}_{L'_2}, \pi_2)$ can be composed to pullback $(\alpha_1 \circ \pi_2, \mathrm{id}_{L_{12}})$ of $(l_1, (\pi_2, \pi_1)^\bullet \circ \pi_2)$. Since $t_1 \circ \beta_1 = n_1 \circ n'_1 \circ \beta_1 = n_1 \circ \gamma_1 \circ \pi_2 = \alpha \circ \pi_2$ and $(\pi_2, \pi_1)^\bullet \circ \pi_2 = \eta_{L'_1} \circ \pi_1$, $(t_1 \circ \beta_1, \mathrm{id}_{L_{12}})$ is pullback of $(l_1, \eta_{L'_1} \circ \pi_1)$. Since (l'_1, t_1) is pullback of $(l_1, \eta_{L'_1})$, $(\beta_1, \mathrm{id}_{L_{12}})$ is pullback of (l'_1, π_1) by pullback decomposition such that β_1 is the l'_1-reflection of π_1. Since π_1 is monic, β_1 is monic as well. □

In a second step, we investigate necessary conditions that are stipulated by the right hand sides of rules and rewrites. For this purpose consider Fig. 13. Again, (π_1, π_2) is the pullback of the base matches. And the morphisms β_1 and γ_1 are the reflections induced by Proposition 22, if match m_2 has a residual in the trace (g_1, h_1) of the rewrite with rule (l_1, r_1) at match m_1, and morphism γ_2 is one of the reflections induced by Proposition 22, if match m_1 has a residual in the trace (g_2, h_2) of the rewrite with rule (l_2, r_2) at match m_2.

Proposition 23 (Necessary condition – right-hand sides). *If matches m_1 and m_2 for AGREE-flavoured rules (l_1, r_1) and (l_2, r_2) have residuals in the trace of the other rule, (π_1, π_2) is the pullback of the base matches m'_1 and m'_2, and β_1 is the l'_1-reflection of π_1, then $(r'_1 \circ \beta_1, \pi_2)^\bullet$ has a l_2-reflection (δ_1 in Fig. 13).*

Proof. For the proof, consider Fig. 13 which depicts the complete situation. Since both matches have residuals, there are reflections β_1, γ_1, and γ_2. Since $(m_2 \circ g_1, h_1)$ is the residual for m_2, (h'_1, g_{21}) is final pullback complement of (g'_2, h_1) where (ε'_1, g'_2) is pullback of $\gamma_1^\bullet = m_2 \circ g_1$ and l_2, (g_{21}, n_{21}) is pullback of $m_{21} = (h_1 \circ \gamma_1)^\bullet$ and l_2, and h'_1 is the unique morphism making the diagram commutative, compare Definition 13(3).

Since also (g_2, n_2) is pullback of (m_2, l_2) due to the construction of the rewrite for rule (l_2, r_2) at match m_2, there is unique morphism g'_1 such that $n_2 \circ g'_1 = \varepsilon'_1$ and (g'_1, g'_2) is pullback of (g_1, g_2) by pullback decomposition. Since $g_1 \circ n'_1 =$



$$\begin{tikzcd}
 & L_{12} \arrow[r, tail, "\pi_1"] \arrow[d, tail, "\pi_2"'] \arrow[rr, tail, bend right, "\beta_1"'] \arrow[rd, phantom, "\text{(PB)}"] & L_1' \arrow[d, tail, "m_1'"'] \arrow[dd, tail, bend left, "\gamma_2"] & K_1' \arrow[l, tail, "l_1'"'] \arrow[r, tail, "r_1'"] \arrow[d, tail, "n_1'"'] \arrow[dd, tail, bend left, "\varepsilon_2"] & R_1' \arrow[d, tail, "p_1'"'] \arrow[dd, tail, dashed, bend left, "p_{12}"] \arrow[llll, dashed, bend right, "(r_1'\circ\beta_1{,}\pi_2)^\bullet"'] \arrow[lllldd, dashed, "\delta_1"] \\
L_2 & L_2' \arrow[l, tail, "\eta_{L_2}"'] \arrow[r, tail, "m_2'"] \arrow[rr, tail, bend right, "\gamma_1"'] & G \arrow[ll, bend left, "m_2"] & D_1 \arrow[l, tail, "g_1"'] \arrow[r, tail, "h_1"] \arrow[lll, dashed, bend left, "\gamma_1^\bullet"] & H_1 \arrow[llll, dashed, bend left, "m_{21} = (h_1\circ\gamma_1)^\bullet"] \\
K_2 \arrow[u, "l_2"] & K_2' \arrow[l, tail, "t_2"'] \arrow[u, "l_2'"] \arrow[r, tail, "n_2'"] \arrow[rr, tail, bend right, "\varepsilon_1"'] & D_2 \arrow[u, "g_2"] \arrow[ll, bend left, "n_2"] & D^* \arrow[l, tail, "g_1'"'] \arrow[u, "g_2'"] \arrow[r, tail, "h_1'"] \arrow[lll, dashed, bend left, "\varepsilon_1'"] & D_{21} \arrow[u, "g_{21}"] \arrow[llll, dashed, bend left, "n_{21}"]
\end{tikzcd}$$

Fig. 13. Parallel independence: right-hand side

$m_1' \circ l_1' = g_2 \circ (\gamma_2 \circ l_1')$, there is ε_2 with $g_1' \circ \varepsilon_2 = \gamma_2 \circ l_1'$ and $g_2' \circ \varepsilon_2 = n_1'$. By pullback composition and decomposition properties, ε_2 is the g_2'-reflection of n_1'.

Since (h_1', g_{21}) is final pullback complement of (g_2', h_1), (r_1', n_1') is pullback of (h_1, p_1') by Fact F2, and $g_2' \circ \varepsilon_2 = n_1'$, there is unique morphism p_{12} such that $g_{21} \circ p_{12} = p_1'$ and $p_{12} \circ r_1' = h_1' \circ \varepsilon_2$. Now, we have that

1. (h_1', g_{21}) is final pullback complement,
2. $(r_1', \mathrm{id}_{R_1'})$ is trivially final pullback complement of $(\mathrm{id}_{K_1'}, r_1')$,
3. $(\varepsilon_2, \mathrm{id}_{K_1'})$ is pullback of (n_1', g_2'), and
4. (n_1', r_1') is pullback of (h_1, p_1').

In this situation, Proposition 5 guarantees that p_{12} is g_{21}-reflection of p_1'. Since (g_{21}, n_{21}) is pullback of (m_{21}, l_2), $n_{21} \circ p_{12}$ is the l_2-reflection of $m_{21} \circ p_1'$. Since pushout (p_1', h_1) of (r_1', n_1') is also pullback, composition of pullbacks provides $(\pi_2, r_1' \circ \beta_1)$ as pullback of $(p_1', h_1 \circ \gamma_1)$.[14] By Fact F3, $m_{21} \circ p_1' = (h_1 \circ \gamma_1)^\bullet \circ p_1' = (r_1' \circ \beta_1, \pi_2)^\bullet$. Therefore, $n_{21} \circ p_{12}$ is l_2-reflection of $(r_1' \circ \beta_1, \pi_2)^\bullet$. $\square$

The final step in this section is the proof that the necessary conditions of Propositions 22 and 23 are sufficient as well.

[14] (π_2, β_1) is pullback of (α_1, t_1), compare proof of Proposition 22.

Theorem 24 (Characterisation of parallel independence). *Two global matches m_1 and m_2 for rules $\sigma_1 = (l_1, r_1)$ and $\sigma_2 = (l_2, r_2)$, which are AGREE-flavoured by (l'_1, t_1, r'_1) and (l'_2, t_2, r'_2) respectively, are parallel independent, if and only if:*

1. *$(\pi_2, \pi_1)^\bullet$ has a l_1-reflection (and π_1 has a derived l'_1-reflection β_1) and, vice versa, $(\pi_1, \pi_2)^\bullet$ has a l_2-reflection (and π_2 has a derived l'_2-reflection β_2) and*
2. *$(r'_2 \circ \beta_2, \pi_1)^\bullet$ has a l_1-reflection and $(r'_1 \circ \beta_1, \pi_2)^\bullet$ has a l_2-reflection*

where (π_1, π_2) is pullback of the base matches m'_1 and m'_2 for m_1 and m_2 resp.

Proof. ($\Rightarrow$) Immediate consequence of Propositions 22 and 23. ($\Leftarrow$) We have to show properties (2) and (3) of Definition 13, property 1 is satisfied since h_1 and h_2 in the traces $\sigma_1 \langle m_1 \rangle = (g_1, h_1)$ and $\sigma_2 \langle m_2 \rangle = (g_2, h_2)$ are monic by Fact F1.

For property (2) compare again Fig. 12 and let α_1 be the l_1-reflection of $(\pi_2, \pi_1)^\bullet$. Since (g_1, n_1) is constructed as pullback of (m_1, l_1) in a rewrite and we have $m_1 \circ m'_2 = {m'_1}^\bullet \circ m'_2 = (\pi_2, \pi_1)^\bullet = l_1 \circ \alpha_1$ by Fact F3, we get γ_1 with $g_1 \circ \gamma_1 = m'_2$ and $n_1 \circ \gamma_1 = \alpha_1$ such that, by pullback decomposition, γ_1 is the g_1-reflection of $m'_2 = g_1 \circ \gamma_1$. The reflection γ_1 is monic, since m'_2 is. In this situation, Fact F4 provides $\gamma_1^\bullet = (g_1 \circ \gamma_1)^\bullet \circ g_1 = {m'_2}^\bullet \circ g_1 = m_2 \circ g_1$ as desired. Now, β_1 can be derived as the l'_1-reflection of π_1 as in the proof of Proposition 22.

For property (3), we can presuppose that there are reflections β_1, γ_1 and β_2, γ_2. Consider Fig. 13 in which δ_1 is the l_2-reflection of $(r'_1 \circ \beta_1, \pi_2)^\bullet$. Since $(r'_1 \circ \beta_1, \pi_2)$ is pullback of $(p'_1, h_1 \circ \gamma_1)$, compare proof of Proposition 23, we obtain $(r'_1 \circ \beta_1, \pi_2)^\bullet = m_{21} \circ p'_1$ by Fact F3. Therefore, δ_1 is the l_2-reflection of $m_{21} \circ p'_1$ and $m_{21} \circ p'_1 = l_2 \circ \delta_1$. Since (n_{21}, g_{21}) is pullback of (l_2, m_{21}), there is p_{12} such that $g_{21} \circ p_{12} = p'_1$, $n_{21} \circ p_{12} = \delta_1$, and p_{12} is the g_{21}-reflection of p'_1 by pullback decomposition. As in the proof of Proposition 23, ε_2 can be constructed as the g'_2-reflection of n'_1. Now, we have pullbacks $(\mathrm{id}_{R'_1}, p_{12})$, $(\mathrm{id}_{K'_1}, r'_1)$, $(\varepsilon_2, \mathrm{id}_{K'_1})$, and (h'_1, g'_2) such that 2 in Sect. 2 guarantees that (h'_1, p_{12}) is pushout of r'_1 and ε_2. Since $(r'_1, \mathrm{id}_{R'_1})$ is trivially final pullback complement of $(\mathrm{id}_{K'_1}, r'_1)$, we conclude by Corollary 8 that (h'_1, g_{21}) is final pullback complement of (g'_2, h_1). $\square$

6 Conclusion

Theorem 24 states that parallel independence of two rewrites $\sigma_1 \langle m_1 \rangle$ and $\sigma_2 \langle m_2 \rangle$ is guaranteed, if the AGREE-rule ϱ_1 which flavours σ_1 is completely (left- *and* right-hand side) preserved by the left-hand side of σ_2 and vice versa. These conditions can be easily checked without construction of the rewrites: On the basis of the pullback of the participating base matches, certain reflections of the left- and the right-hand side of one rule wrt. the left-hand side morphism of the other rule must be checked. In most application categories, like graphs, the pullback of a suitable pair of monic morphisms is a simple intersection and the reflection property is equivalent to partial injectivity of the rules left-hand sides.

As an example, we formalise the two AGREE-rules ϱ_E and ϱ_D which are mentioned in the introduction:

$$\varrho_E = (\emptyset, t_E, \emptyset) \text{ with } t_E : \emptyset \to (\{v\}, \emptyset, s, t) \text{ with } \mathrm{s} = \mathrm{t} = \emptyset$$
$$\varrho_D = (\emptyset, t_D, \emptyset) \text{ with } t_D : \emptyset \to (\{v\}, \{e, e'\}, s, t) \text{ where } s \text{ and } t \text{ are final mappings}$$

Since $L_E = L_D = \emptyset$, $\eta_{L_E} = \eta_{L_D} = \emptyset : \emptyset \rightarrowtail 1$, where 1 is the final graph. There is only one base match $m_E = \emptyset$ for ϱ_E and $m_D = \emptyset$ for ϱ_D in any graph. Their intersection (pullback) is empty and both projections (π_1 and π_2) are empty morphisms. Thus, $(\pi_1, \pi_2)^\bullet$ and $(\pi_2, \pi_1)^\bullet$ are empty morphisms which are reflected by every graph morphism. Since the right-hand side morphisms r_E and r_D of both rules are empty, $(r_D \circ \beta_1, \pi_2)^\bullet$ and $(r_E \circ \beta_2, \pi_1)^\bullet$ are empty morphisms and reflected by any other graph morphism. Thus, every two rewrites with ϱ_E and ϱ_D are parallel independent and the corresponding derivations sequences commute by Theorem 18.

It is the task of future research to apply the developed parallel independence condition in more complex case studies and to figure out, if and how non-local effects of rewrites can be used in practical examples.

References

1. Corradini, A., Duval, D., Echahed, R., Prost, F., Ribeiro, L.: AGREE – algebraic graph rewriting with controlled embedding. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 35–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_3
2. Corradini, A., Duval, D., Prost, F., Ribeiro, L.: Parallelism in AGREE transformations. In: Echahed, R., Minas, M. (eds.) ICGT 2016. LNCS, vol. 9761, pp. 37–53. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40530-8_3
3. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_4
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. MTCSAES. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
5. Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.): Graph Transformations. LNCS, vol. 6372. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2
6. Goldblatt, R.: Topoi. Dover Publications, Mineola (1984)
7. Heindel, T.: Hereditary pushouts reconsidered. In: Ehrig et al. [5], pp. 250–265 (2010)
8. Kennaway, R.: Graph rewriting in some categories of partial morphisms. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 490–504. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0017408
9. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. **109**(1&2), 181–224 (1993)
10. Löwe, M.: Graph rewriting in span-categories. In: Ehrig et al. [5], pp. 218–233 (2010)
11. Löwe, M.: Characterisation of parallel independence in agree-rewriting. Technical report 2018/01, FHDW Hannover (2018)

Equivalence and Independence in Controlled Graph-Rewriting Processes

Géza Kulcsár[1(✉)], Andrea Corradini[2], and Malte Lochau[1]

[1] Real-Time Systems Lab, TU Darmstadt, Darmstadt, Germany
{geza.kulcsar,malte.lochau}@es.tu-darmstadt.de
[2] Dipartimento di Informatica, University of Pisa, Pisa, Italy
andrea@di.unipi.it

Abstract. Graph transformation systems (GTS) are often defined as sets of rules that can be applied repeatedly and non-deterministically to model the evolution of a system. Several semantics proposed for GTSs are relevant in this case, providing means for analysing the system's behaviour in terms of dependencies, conflicts and potential parallelism among the relevant events. Several other approaches equip GTSs with an additional control layer useful for specifying rule application strategies, for example to describe graph manipulation algorithms. Almost invariably, the latter approaches consider only an input-output semantics, for which the above mentioned semantics are irrelevant.

We propose an original approach to controlled graph transformation, where we aim at bridging the gap between these two complementary classes of approaches. The control is represented by terms of a simple process calculus. Expressiveness is addressed by encoding in the calculus the Graph Processes defined by Habel and Plump, and some initial results are presented relating parallel independence with process algebraic notions like bisimilarity.

1 Introduction

Graph-rewriting systems are used for many different purposes and in various application domains. They provide an expressive and theoretically well founded basis for the specification and the analysis of concurrent and distributed systems [7]. Typically, a set of graph-rewriting rules describes the potential changes of graph-based, abstract representations of the states of a system under consideration. Each rule can be applied when a certain pattern occurs in the state, producing a local change to it. Thus, graph-rewriting systems are inherently non-deterministic regarding both the rule sequencing and the selection of the match, i.e. the pattern to rewrite. Several semantics have been proposed for graph transformation systems (GTS), which emphasize the parallelism that naturally arises between rules that are applied to independent parts of the distributed state.

This work has been partially funded by the German Research Foundation (DFG) as part of project A1 within CRC 1053–MAKI.

They include, among others, the *trace-based*, the *event structure* and the *process semantics* summarized and compared in [2]. The common intuition is that the semantic domain should be rich enough to describe the computations of a system (i.e., not only the reachable states), but also abstract enough to avoid distinguishing computations that differ for irrelevant details only, or for the order in which independent rule applications are performed.

Sometimes however, mainly in the design of graph manipulation algorithms, a finer control on the order of application of graph rules is desirable, for example including sequential, conditional, iterative or even concurrent composition operators. To address this problem, several approaches to *programmed graph grammars* or *controlled graph rewriting* have been proposed, which generalize the *controlled string grammars* originally introduced with the goal of augmenting language generation with constraints on the order of application of productions [6]. One of the first approaches to programmed graph grammars is due to Bunke [3]. Regarding the semantics of controlled graph rewriting, Schürr proposes a semantic domain including possible input/output graph pairs [14], which has been the basis for the development of several tools (such as PROGRES [15], Fujaba (SDM) [9] and eMoflon [12]). Habel and Plump [10] propose a minimal language for controlled graph rewriting (see Sect. 4) with the goal of showing its computational completeness, for which an input/output semantics is sufficient. Also Plump and Steinert propose an input/output semantics, presented in an operational style, for the controlled graph rewriting language *GP* [13].

In this paper, we propose an original approach to controlled graph rewriting, where control is specified with terms of a simple process calculus able to express non-deterministic choice, parallel composition, and prefixing with non-applicability conditions, thus constraining in a strict but not necessarily sequential way the order of application of rules of a given system. In the present paper we introduce the relevant definitions and start exploring the potentialities of the approach, while the long-term goal is to equip such systems with an abstract truly-concurrent semantics, suitable as foundation of efficient analysis techniques (like for example [1] for contextual Petri nets). We start by introducing process terms which specify a labeled transition system (LTS) where transitions represent potential applications of possibly parallel rules. This corresponds conceptually to the code of an algorithm, or to an unmarked net in the theory of Petri nets, an analogy that we adopt by calling those processes *unmarked*. Next, we define the operational semantics of such process terms when applied to a graph, yielding the *marked* LTS where transitions become concrete rule applications. The LTS semantics involves explicit handling of parallel independence (i.e., arbitrary sequentialization) of rule applications, if they are fired by controlled processes running in parallel: our notion of *synchronization* allows for single shared transitions of parallel processes whenever there are parallel independent rule applications of the involved processes. Particularly, synchronized actions represent a first step towards adequately capturing true concurrency of independent rule applications in a controlled setting.

From an expressiveness perspective, in order to enrich controlled graph-rewriting processes by conditional branching constructs as necessary in any control mechanism, we introduce *non-applicability conditions* formulated over rules, corresponding to the condition that a given rule is not applicable. We also prove, as a sanity check, that our control language including non-applicability conditions is able to encode in a precise way the language proposed by Habel and Plump in [10]: an input/output semantics is sufficient to this aim.

The LTS framework is exploited next to start exploring other potentialities of the approach. We introduce an abstract version of the marked LTS showing that it is finite branching under mild assumptions, and define trace equivalence and bisimilarity among marked processes. Besides some pretty obvious results concerning such equivalence, we show that in a simple situation bisimilarity can be used to check that two derivations are not parallel independent: a link between the classical theories of GTSs and of LTSs that we intend to explore further.

2 Preliminaries

We introduce here the basic definitions related to (typed) graphs, algebraic Double-Pushout (DPO) rewriting, parallel derivations and shift equivalence [7].

Definition 1 (Graphs and Typed Graphs)**.** *A* (directed) graph *is a tuple* $G = \langle N, E, s, t\rangle$*, where* N *and* E *are finite sets of nodes and edges, and* $s, t : E \to N$ *are the source and target functions. The components of a graph* G *are often denoted by* N_G*,* E_G*,* s_G*,* t_G*. A* graph morphism $f : G \to H$ *is a pair of functions* $f = \langle f_N : N_G \to N_H, f_E : E_G \to E_H\rangle$ *such that* $f_N \circ s_G = s_H \circ f_E$ *and* $f_N \circ t_G = t_H \circ f_E$*; it is an isomorphism if both* f_N *and* f_E *are bijections.*

Graphs G *and* H *are isomorphic, denoted* $G \cong H$*, if there is an isomorphism* $f : G \to H$*. We denote by* $[G]$ *the class of all graphs isomorphic to* G*, and we call it an* abstract graph*. We denote by* **Graph** *the category of graphs and graph morphisms, by* $|\mathbf{Graph}|$ *the set of its objects, that is all graphs, and by* $[|\mathbf{Graph}|]$ *the set of all abstract graphs.*

The category of typed graphs *over a* type graph T *is the slice category* $(\mathbf{Graph} \downarrow T)$*, also denoted* $\mathbf{Graph}_T$ *[4]. That is, objects of* $\mathbf{Graph}_T$ *are pairs* (G, t) *where* $t : G \to T$ *is a* typing *morphism, and an arrow* $f : (G, t) \to (G', t')$ *is a morphism* $f : G \to G'$ *such that* $t' \circ f = t$*.*

Along the paper we will mostly work with typed graphs, thus when clear from the context we omit the word "typed" and the typing morphisms.

Definition 2 (Graph Transformation System)**.** *A* (DPO T-typed graph) rule *is a span* $(L \xleftarrow{l} K \xrightarrow{r} R)$ *in* $\mathbf{Graph}_T$ *where* l *is mono. The graphs* L*,* K*, and* R *are called the* left-hand side*, the* interface*, and the* right-hand side *of the rule, respectively. A* graph transformation system (GTS) *is a tuple* $\mathcal{G} = \langle T, \mathcal{R}, \pi\rangle$*, where* T *is a type graph,* $\mathcal{R}$ *is a finite set of* rule names*, and* π *maps each rule name in* $\mathcal{R}$ *into a rule.*

The categorical framework allows to define easily the parallel composition of rules, by taking the coproduct of the corresponding spans.

Definition 3 (Parallel Rules). *Given a* GTS $\mathcal{G} = \langle T, \mathcal{R}, \pi\rangle$, *the set of* parallel rule names $\mathcal{R}^*$ *is the free commutative monoid generated by* $\mathcal{R}$, $\mathcal{R}^* = \{p_1|\dots|p_n \mid n \geq 0, p_i \in \mathcal{R}\}$, *with monoidal operation "$|$" and unit* ε. *We use* ρ *to range over* $\mathcal{R}^*$. *Each element of* $\mathcal{R}^*$ *is associated with a span in* $\mathbf{Graph}_T$, *up to isomorphism, as follows:*

1. ε: $(\emptyset \leftarrow \emptyset \rightarrow \emptyset)$, *where* $\emptyset$ *is the empty graph;*
2. p: $(L \xleftarrow{l} K \xrightarrow{r} R)$ *if* $p \in \mathcal{R}$ *and* $\pi(p) = (L \xleftarrow{l} K \xrightarrow{r} R)$;
3. $\rho_1|\rho_2$: $(L_1 + L_2 \xleftarrow{l_1+l_2} K_1 + K_2 \xrightarrow{r_1+r_2} R_1 + R_2)$ *if* ρ_1: $(L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ *and* ρ_2: $(L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2)$, *where* $G + H$ *denotes the* coproduct *(disjoint union) of graphs* G *and* H, *and if* $g : G \rightarrow G'$ *and* $h : H \rightarrow H'$ *are morphisms, then* $g + h : G + H \rightarrow G' + H'$ *denotes the obvious mediating morphism.*

For $\rho \in \mathcal{R}^*$, *we denote by* $\langle\rho\rangle$ *the set of rule names appearing in* ρ, *defined inductively as* $\langle\varepsilon\rangle = \emptyset$, $\langle p\rangle = \{p\}$ *if* $p \in \mathcal{R}$, *and* $\langle\rho_1|\rho_2\rangle = \langle\rho_1\rangle \cup \langle\rho_2\rangle$.

Note that the above definition is well-given because coproducts are associative and commutative up to isomorphism. Clearly, the same rule name can appear several times in a parallel rule name. In the following, we assume that $\mathcal{G} = \langle T, \mathcal{R}, \pi\rangle$ denotes an arbitrary but fixed GTS.

Definition 4 *(Rule Application, Derivations). Let* G *be a graph, let* $\rho : (L \xleftarrow{l} K \xrightarrow{r} R)$ *be a possibly parallel rule, and let* m *be a* match, *i.e., a (possibly non-injective) graph morphism* $m : L \rightarrow G$. *A* DPO rule application *from* G *to* H *via* ρ *(based on* m*) is a diagram* δ *as in (1), where both squares are pushouts in* $\mathbf{Graph}_T$. *In this case we write* $G \overset{\delta}{\Rightarrow} H$ *or simply* $G \overset{\rho@m}{\Longrightarrow} H$. *We denote by* $\mathcal{D}$ *the set of* DPO *diagrams, ranged over by* δ. *For a rule* $p \in \mathcal{R}$ *and a graph* G, *we write* $G \overset{p}{\not\Rightarrow}$ *if there is no match* m *such that* $G \overset{p@m}{\Longrightarrow} H$ *for some graph* H.

A (parallel) derivation φ *from a graph* G_0 *is a finite sequence of rule applications* $\varphi = G_0 \overset{\delta_1}{\Rightarrow} G_1 \cdots G_{n-1} \overset{\delta_n}{\Rightarrow} G_n$, *via* $\rho_1, \dots, \rho_n \in \mathcal{R}^*$. *A derivation is* linear *if* $\rho_1, \dots, \rho_n \in \mathcal{R}$.

$$\begin{tikzcd} L \arrow[d, "m"'] \arrow[dr, phantom, "(PO)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "n"] \\ G & D \arrow[l, "f"] \arrow[r, "g"'] & H \end{tikzcd} \qquad (1)$$

Intuitively, two rule applications starting from the same graph are *parallel independent* if they can be sequentialized arbitrarily with isomorphic results. This property is captured categorically by the following definition [5].

Definition 5 (Parallel Independence)**.** *Given two (possibly parallel) rules* $\rho_1 : (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ *and* $\rho_2 : (L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2)$ *and two matches* $L_1 \xrightarrow{m_1} G \xleftarrow{m_2} L_2$ *in a graph* G*, the rule applications* $\rho_1@m_1$ *and* $\rho_2@m_2$ *are* parallel independent *if there exist arrows* $a_1 : L_1L_2 \to K_1$ *and* $a_2 : L_1L_2 \to K_2$ *such that* $l_1 \circ a_1 = \pi_1$ *and* $l_2 \circ a_2 = \pi_2$ *as in Diagram (2), where* L_1L_2 *is the pullback object over* $L_1 \xrightarrow{m_1} G \xleftarrow{m_2} L_2$*.*

$$\begin{tikzcd}
& L_1 \arrow[dl, "m_1"'] & & K_1 \arrow[ll, "l_1"'] \\
G & (PB) & L_1L_2 \arrow[ul, "\pi_1"'] \arrow[ur, "a_1"'] \arrow[dl, "\pi_2"] \arrow[dr, "a_2"] & \\
& L_2 \arrow[ul, "m_2"] & & K_2 \arrow[ll, "l_2"]
\end{tikzcd} \qquad (2)$$

As discussed in [5], this definition is equivalent to others proposed in literature, but does not need to compute the pushout complements to be checked.

As recalled by the next result, two parallel independent rule applications can be applied in any order to a graph G obtaining the same resulting graph, up to isomorphism. Furthermore, the same graph can be obtained by applying to G the parallel composition of the two rules, at a match uniquely determined by the coproduct construction.

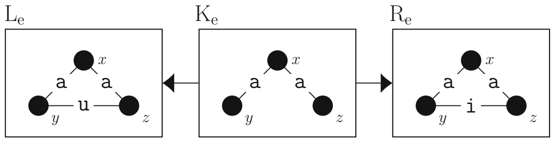
Proposition 1 (Local Church-Rosser and Parallelism Theorems [7])**.** *Given two rule applications* $H_1 \overset{\rho_1@m_1}{\Longleftarrow} G \overset{\rho_2@m_2}{\Longrightarrow} H_2$ *with parallel independent matches* $m_1 : L_1 \to G$ *and* $m_2 : L_2 \to G$*, there exist matches* $m_1' : L_1 \to H_2$*,* $m_2' : L_2 \to H_1$ *and* $m : L_1 + L_2 \to G$ *such that there are rule applications* $H_1 \overset{\rho_2,m_2'}{\Longrightarrow} H_{12}$*,* $H_2 \overset{\rho_1,m_1'}{\Longrightarrow} H_{21}$ *and* $G \overset{\rho_1|\rho_2,m}{\Longrightarrow} H$*, and graphs* H_{12}*,* H_{21} *and* H *are pairwise isomorphic.*

3 Controlled Graph-Rewriting Processes

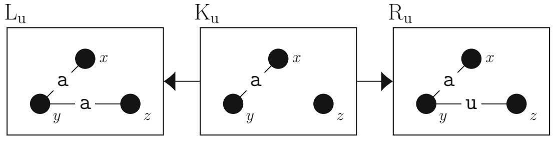
In this section, we first motivate diverse aspects of our approach by presenting a (simplified) application of controlled graph-rewriting processes (Sec. 3.1). Then, in Sec. 3.2, we start to develop the theory by first introducing *unmarked* processes, representing a process-algebraic control-flow specification, i.e., a process whose executions specify permitted derivations. Afterwards (Sec. 3.3), we introduce *marked processes* as pairs of unmarked processes and graphs, and compare marked traces to parallel derivations [2].

3.1 An Illustrative Example: WSN Topology Control

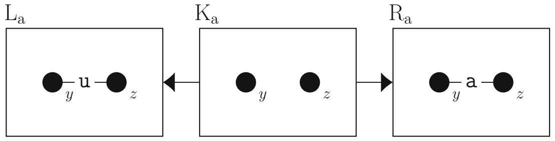
We illustrate controlled graph-rewriting processes by an example: a simplified *wireless sensor network* (WSN) model in which autonomous sensors communicate through wireless channels, represented as typed graphs where nodes denote sensors and edges denote bidirectional communication links via channels. We use different edge types to represent the link status: *active* (`a`) indicates that the link is currently used for communication, whereas link status *inactive* (`i`)



(a) Rule p_e: Eliminate Active Triangle



(b) Rule p_u: Unclassify Active Neighbor



(c) Rule p_a: Activate Edge

Fig. 1. Topology control operations as DPO rules

denotes links currently not in use. Links with status *unclassified* (u) require status revision.

The DPO rules shown in Fig. 1a–c represent *topology control* (TC) operations [11]: p_e and p_u reduce link redundancy either conservatively by eliminating u-edges from active triangles (p_e), or through unclassifying edges with active neighbors (p_u), whereas p_a is a stability counter-measure, activating unclassified edges. We use the rules in Fig. 1 to specify *controlled graph-rewriting processes* expressing different topology control strategies. Due to the decentralized nature of WSN, both *sequential rule control with non-applicability conditions* and *parallel processes* are inherent in topology control strategies. As a concrete example for a TC strategy, let us consider (using a yet informal process-algebraic notation).

$$P_{TC} := P_e \,||\, P_u \qquad P_e := p_e.P_e + (p_a, \{p_e\}).P_e \qquad P_u := p_u.P_u$$

Here, each P (with a subscript) is a process name that can appear in other processes, allowing to express recursion. The dot (".") operator represents prefixing, while "+" represents non-deterministic choice and "$||$" parallel composition. Actions can be either plain rule applications (like p_e in P_e) or rule applications with additional *non-applicability conditions* (as in $(p_a, \{p_e\})$). The second component of the action is a set of rule names (here, containing only p_e), denoting that p_a should be applied *only if* p_e *is not applicable*. (Here, as also later in the paper, we omit the second component of an action if it is empty, writing for example p_e for $(p_e, \emptyset)$.)

P_{TC} defines a strategy where, in parallel, unclassified edges get inactivated if being part of a triangle or activated otherwise (P_e), while P_u repeatedly unclassifies edges with active neighbors. Although this strategy specification provides an intuitive separation of classification and unclassification, and guarantees using a non-applicability condition that no a-triangles arise, still, the possibly overlapping applications of p_e and p_u might create unwanted triangles in our concurrent setting. The above example illustrates the need for a formal analysis methodology to reason about controlled parallel graph-rewriting processes.

3.2 Unmarked Processes

As suggested by the example in the previous section, unmarked processes are terms of a process calculus including prefixing of actions, non-deterministic choice, parallel composition, as well as recursion to express iteration and intended non-termination (e.g., for specifying reactive behaviors).

An action $\gamma = (\rho, N)$ consists of a (possibly parallel) rule name $\rho \in \mathcal{R}^*$ and a set N of rule names, $N = \{p_1, \dots, p_k\}$. Intuitively, given a graph G such an action can be fired by applying ρ to G only if none of the rules in N is applicable to G. For the definition of unmarked processes, we use the following sets: $\mathcal{K}$ is a set of *process identifiers*, ranged over by A, and $\mathcal{P}$ is the set of *(unmarked) processes*, ranged over by P, Q.

Definition 6 (Unmarked Process Terms)**.** *The syntax of an unmarked process term $P \in \mathcal{P}$ is inductively defined as*

$$P, Q ::= \mathbf{0} \;\mid\; \gamma.P \;\mid\; A \;\mid\; P + Q \;\mid\; P \,||\, Q$$

where $A \in \mathcal{K}$ and γ ranges over $\mathcal{R}^* \times 2^{\mathcal{R}}$.

The process $\mathbf{0}$ is the *inactive* process incapable of actions. Given a process P, $\gamma.P$ represents an *action prefix*, meaning that this process can perform an action γ and then continue as P. Process identifiers are used to represent process terms through *defining equations*, and thus might be used to describe recursive process behavior. A defining equation for $A \in \mathcal{K}$ is of the form $A := P$ with $P \in \mathcal{P}$. We assume that each $A \in \mathcal{K}$ has a unique defining equation. $P + Q$ represents a process which non-deterministically behaves either as P or as Q. The *parallel composition* of P and Q, denoted as $P \,||\, Q$, is a process which might interleave the actions of P and Q or even execute them *in parallel.*

There are some syntactically different processes which we treat as equivalent in each context. This relation is called *structural congruence* and denoted $\equiv$.

Definition 7 (Structural Congruence of Unmarked Processes)**.** *The relation $\equiv$ on unmarked process terms is the least equivalence relation s.t.*

$$P \,||\, \mathbf{0} \equiv P \qquad P + Q \equiv Q + P \qquad P \,||\, Q \equiv Q \,||\, P$$

The semantics of an unmarked process term is a *labeled transition system* having processes as states and moves labeled by actions as transitions. We first recall the standard definition of labeled transition systems and of their traces.

$$\text{STRUCT}\,\frac{P \equiv Q \quad P \xrightarrow{\alpha} P'}{Q \xrightarrow{\alpha} P'} \qquad \text{PRE}\,\frac{}{\gamma.P \xrightarrow{\gamma} P} \qquad \text{STOP}\,\frac{}{\mathbf{0} \xrightarrow{\checkmark} \mathbf{0}}$$

$$\text{CHOICE}\,\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{PAR}\,\frac{P \xrightarrow{\gamma} P'}{P \,||\, Q \xrightarrow{\gamma} P' \,||\, Q} \qquad \text{REC}\,\frac{A := P \quad P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}$$

$$\text{SYNC}\,\frac{P \xrightarrow{(\rho_1, N_1)} P' \quad Q \xrightarrow{(\rho_2, N_2)} Q' \quad \langle \rho_1 \rangle \cap N_2 = \emptyset \quad \langle \rho_2 \rangle \cap N_1 = \emptyset}{P \,||\, Q \xrightarrow{(\rho_1 | \rho_2, N_1 \cup N_2)} P' \,||\, Q'}$$

Fig. 2. Transition rules of unmarked processes

Definition 8 (Labeled Transition System, Trace, Trace Equivalence)**.** *A* labeled transition system *(LTS) is a tuple* $(S, A, \rightarrow_X)$*, where* S *is a set of* states*,* A *is a set of* actions *containing the distinguished element "*$\checkmark$*" representing successful termination, and* $\rightarrow_X \subseteq S \times A \times S$ *is a* transition relation*. As usual, we will write* $s \xrightarrow{a} s'$ *if* $(s, a, s') \in \rightarrow_X$*.*

A trace $t = a_1 a_2 \dots a_n \in A^*$ *of a state* $s \in S$ *is a sequence of actions such that there exist states and transitions with* $s \xrightarrow{a_1}_X s_1 \xrightarrow{a_2}_X \dots \xrightarrow{a_n}_X s_n$*. A trace is* successful *if its last element, and only it, is equal to* $\checkmark$*.*

States $s, s' \in S$ *are* trace equivalent *w.r.t.* $\rightarrow_X$*, denoted as* $\simeq^T_X$*, if* s *and* s' *have the same set of traces.*

The LTS for unmarked process terms is defined by inference rules as follows.

Definition 9 (Unmarked Transition System)**.** *The* unmarked transition system *(UTS) of* $\mathcal{G}$ *is an LTS* $(\mathcal{P}, (\mathcal{R}^* \times 2^{\mathcal{R}}) \cup \{\checkmark\}, \rightarrow)$ *with* $\rightarrow$ *being the least relation satisfying the rules in Fig. 2, where* α *ranges over* $(\mathcal{R}^* \times 2^{\mathcal{R}}) \cup \{\checkmark\}$*,* γ *ranges over* $\mathcal{R}^* \times 2^{\mathcal{R}}$*, and* N *over* $2^{\mathcal{R}}$*.*

Rule STRUCT expresses that structural congruent processes share every transition. Rule PRE states that any action γ appearing as a prefix induces a transition labeled by γ and then the process continues as specified. Rule REC says that process identifiers behave as their defining processes. Rule CHOICE expresses that $P + Q$ can proceed as P or Q by firing any of their transitions (commutativity of $+$ is provided by STRUCT). In the case of *parallel composition*, interleaved actions as in rule PAR mean that one side of the composition proceeds independently of the other. In contrast, SYNC represents *synchronization*, i.e., that the two sides agree on performing their respective actions *in parallel*, which in the case of rules amounts to performing the *composed rule*, where the non-applicability conditions of both sides hold, while both sides proceed. Finally, STOP introduces the special $\checkmark$-transition to denote termination, i.e. that the empty process $\mathbf{0}$ was reached. Note that termination is global, in the sense that a process of the shape $P \,||\, \mathbf{0}$ does not have a $\checkmark$-transition unless $P \equiv \mathbf{0}$.

3.3 Marked Processes

Now, we extend unmarked process specifications by letting them not only specify potential rule sequences, but also operate on a given graph. The states of a marked process are pairs containing an unmarked process and a graph, while the marked transitions correspond to rule applications. Since a concrete rule application is characterized by a DPO diagram (as in Diagram (1)), we include in the labels of the marked transition system not only the names of the applied (parallel) rule and of the rules in the non-applicability condition, but also the resulting DPO diagram.

Definition 10 (Marked Transition System). *The* marked transition system *(*MTS*) is an LTS* $(\mathcal{P} \times |\mathbf{Graph}_T|, \mathcal{R} \times \mathcal{D} \times 2^{\mathcal{R}} \cup \{\checkmark\}, \rightarrow_{\mathcal{D}})$ *where* $\rightarrow_{\mathcal{D}}$ *is the least relation satisfying the following rules and* δ *is a* DPO *diagram over* ρ*:*

$$\mathit{MARK}\frac{P \xrightarrow{(\rho,N)} P' \quad G \overset{\rho@m}{\Longrightarrow} H \quad \forall p \in N : G \overset{p}{\not\Rightarrow}}{(P,G) \xrightarrow{(\rho,\delta,N)}_{\mathcal{D}} (P',H)} \quad \mathit{STOP}\frac{P \xrightarrow{\checkmark} P'}{(P,G) \xrightarrow{\checkmark}_{\mathcal{D}} (P',G)}$$

It easily follows from the definition that, as desired, traces of controlled graph-rewriting processes correspond to the parallel derivations of Definition 4. In particular, Proposition 2.1 states that every successful trace of a marked process naturally determines a(n underlying) parallel derivation; Proposition 2.2 provides a process definition by recursive choice, which has a successful trace for each linear derivation starting from a given graph, while Proposition 2.3 does the same for parallel (i.e., not necessarily linear) derivations by providing a recursive process allowing for arbitrary parallel composition of the rules as well.

Proposition 2 (Traces and Derivations).

1. *Given a marked process* (P, G)*, each of its successful traces uniquely identifies an* underlying parallel derivation *of* $\mathcal{G}$ *starting from* G*. In particular, if* $(\rho_1, \delta_1, N_1) \cdots (\rho_n, \delta_n, N_n)\checkmark$ *is a successful trace of* (P, G)*, then* $\delta_1; \cdots; \delta_n$ *is its underlying derivation.*
2. *Let* $P_{\mathcal{R}}$ *be the unmarked process defined as follows:*

$$P_{\mathcal{R}} = \mathbf{0} + \sum\nolimits_{p \in \mathcal{R}} p.P_{\mathcal{R}}$$

Then for each graph G *and for each linear derivation* φ *starting from* G *there is a successful trace of* $(P_{\mathcal{R}}, G)$ *such that* φ *is its underlying derivation.*

3. *Let* $Q_{\mathcal{R}}$ *be the unmarked process defined as follows:*

$$Q_{\mathcal{R}} = \mathbf{0} + \left(\left(\sum\nolimits_{p \in \mathcal{R}} p.\mathbf{0} + \varepsilon.\mathbf{0}\right) \,||\, Q_{\mathcal{R}}\right)$$

Then for each graph G *and for each parallel derivation* φ *starting from* G *there is a successful trace of* $(Q_{\mathcal{R}}, G)$ *such that* φ *is its underlying derivation.*

4 On the Expressiveness of Unmarked Processes

Unmarked processes are intended to provide a high-level, declarative language for specifying the evolution of systems modeled using graphs and rewriting rules on them. In a trace of the corresponding marked system, as it results from Proposition 2, all the relevant information about the computation are recorded, and this can be exploited for analyses concerned with the truly concurrent aspects of such computations, including causalities, conflicts and parallelism among the individual events. Some preliminary results in this direction are presented in the next section.

In this section, as a proof of concept for the choice of our unmarked processes, we consider an alternative control mechanism for graph rewriting and discuss how it can be encoded into ours. The chosen approach is declarative and abstract like ours, therefore the encoding is pretty simple. Still, it may provide some insights for encoding more concrete and expressive control structures (like those of [15]) which is left as future work.

Habel and Plump [10] interpret *computational completeness* as the ability to compute every computable partial function on labelled graphs: we refer the reader to the cited paper for motivations and details of this notion. They show in [10] that three programming constructs suffice to guarantee computational completeness: (1) non-deterministic choice of a rule from a set of DPO rules, (2) sequential composition, and (3) *maximal iteration*, in the sense that a program is applied repeatedly as long as possible. *Graph Programs* are built using such constructs, and their semantics is defined as a binary relation on abstract graphs relating the start and end graphs of derivations, as recalled by the following definitions.

Definition 11 (Graph Programs [10]). Graph programs *over a label alphabet $\mathcal{C}$ are inductively defined as follows:*

(1) A finite set of DPO *rules over $\mathcal{C}$ is an* elementary *graph program.*
(2) If GP_1 and GP_2 are graph programs, then $GP_1; GP_2$ is a graph program.
(3) If GP is a graph program by (1) or (2), then $GP\downarrow$ is a graph program.

The set of graph programs is denoted as $\mathcal{GP}$.

Notice that Graph Programs are based on graphs labeled on a label alphabet $\mathcal{C} = \langle \mathcal{C}_E, \mathcal{C}_N \rangle$, and the main result of completeness exploits constructions based on this assumption. It is an easy exercise to check that such graphs are one-to-one with graphs typed over the type graph $T_{\mathcal{C}} = \langle \mathcal{C}_N, \mathcal{C}_N \times \mathcal{C}_E \times \mathcal{C}_N, \pi_1, \pi_3 \rangle$. It follows that $\mathcal{A}_{\mathcal{C}}$, the class of abstract graphs labeled over $\mathcal{C}$ introduced in [10], is actually isomorphic to $[|\mathbf{Graph}_{T_{\mathcal{C}}}|]$. Nevertheless, we still use $\mathcal{A}_{\mathcal{C}}$ in definitions and results of the rest of this section, when they depend on the concrete representation of graphs as defined in [10].

Definition 12 (Semantics of Graph Programs [10]). *Given a program GP over a label alphabet $\mathcal{C}$, the semantics of GP is a binary relation $\rightarrow_{GP}$ on $\mathcal{A}_{\mathcal{C}}$, which is inductively defined as follows:*

(1) $\rightarrow_{GP} = \Rightarrow_{GP}$ *if* $GP = \{p_1, \ldots, p_n\}$ *is an elementary program;*
(2) $\rightarrow_{GP_1;GP_2} = \rightarrow_{GP_2} \circ \rightarrow_{GP_1}$*;*
(3) $\rightarrow_{GP\downarrow} = \big\{\langle G, H\rangle \mid G \rightarrow^{*}_{GP} H$ *and* H *is a normal form w.r.t.* $\rightarrow_{GP}\big\}$.

We show that there is an encoding of Graph Programs in unmarked processes that preserves the semantics.

Definition 13 (Encoding Graph Programs as Processes)**.** *Given unmarked processes* P *and* Q*, their* sequentialization *is the process* $P \mathbin{⨟} Q := P[A_Q/\mathbf{0}]$ *where* $A_Q \in \mathcal{K}$ *is a fresh identifier with* $A_Q := Q$ *and* $t[x/y]$ *denotes the syntactic substitution of* x *for* y *in a term* t*.*

The encoding function $[\![\_]\!] : \mathcal{GP} \rightarrow \mathcal{P}$ *is defined as follows:*

- *If* $GP = \{p_1, \ldots, p_n\}$ *is an elementary graph program, then* $[\![GP]\!] := \sum_{i=1}^{n} p_i.\mathbf{0}$.
- $[\![GP_1; GP_2]\!] := [\![GP_1]\!] \mathbin{⨟} [\![GP_2]\!]$.
- $[\![GP\downarrow]\!] := A_{GP\downarrow} \in \mathcal{K}$ *where* $A_{GP\downarrow} := [\![GP]\!] \mathbin{⨟} A_{GP\downarrow} + \widehat{[\![GP]\!]}$.

Process $\widehat{[\![GP]\!]}$ *is a process which acts as the identity (and terminates successfully) on all and only the graphs which are normal forms with respect to* $[\![GP]\!]$*. It is defined inductively as follows:*

- $\widehat{[\![GP]\!]} := (\varepsilon, \{p_1, \ldots, p_n\}).\mathbf{0}$ *if* $GP = \{p_1, \ldots, p_n\}$ *is an elementary program;*
- $\widehat{[\![GP_1; GP_2]\!]} := \widehat{[\![GP_1]\!]} + [\![GP_1]\!] \mathbin{⨟} \widehat{[\![GP_2]\!]}$*;*
- $\widehat{[\![GP\downarrow]\!]} := (p, \{p\}).\mathbf{0}$*, where* p *is any rule.*

Proposition 3 (Encoding Preserves Semantics)**.** *For each graph program* GP *and graph* G *in* $Graph_{T_{\mathcal{C}}}$ *it holds* $G \rightarrow_{GP} H$ *iff* $([\![GP]\!], G) \rightarrow^{*}_{\mathcal{D}} (\mathbf{0}, H)$.

An easy consequence of this precise encoding is that the main result of [10], stating the computational completeness of graph programs, also holds for unmarked processes.

Corollary 1. *Given a label alphabet* $\mathcal{C}$ *and subalphabets* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*, for every computable partial function* $f : \mathcal{A}_{\mathcal{C}_1} \rightarrow \mathcal{A}_{\mathcal{C}_2}$*, there exists an unmarked process that computes* f*.*

5 Equivalence and Independence

In this section, we elaborate on the semantics of marked processes by first introducing an *abstraction* of DPO diagrams to provide a more compact representation of marked transition systems. Afterwards, we investigate different equivalence notions (trace, bisimilarity) and re-interpret parallel independence of actions in marked processes.

Abstract Labels. When reasoning about a system in terms of graphs representing its possible states and of graph transformations modeling its evolution, a natural attitude is to abstract from irrelevant details like the identity of the involved nodes and edges. Formally, this corresponds to considering individual graphs, or also diagrams in the category of graphs, up to isomorphism. By applying this standard abstraction technique to our marked transition systems we define an abstract variant of them having the advantage of exhibiting a state space where branching is bounded under some obvious, mild assumptions. This is certainly valuable for the analysis of such systems, but this is left as future work. On the contrary, note that the marked transition systems of Definition 10 are infinitely branching even for a single rule and a single state, because the pushout object is defined only up to isomorphism.

Definition 14 (Abstract DPO Diagrams)**.** *Given two* DPO *diagrams* δ_1 *and* δ_2 *as in Fig. 1 with each graph indexed by* 1 *and* 2*, respectively, they are equivalent, denoted as* $\delta_1 \cong \delta_2$*, if there exist isomorphisms* $L_1 \to L_2, K_1 \to K_2, R_1 \to R_2, G_1 \to G_2, D_1 \to D_2, H_1 \to H_2$*, such that each arising square commutes.*

An abstract DPO diagram *is an equivalence class* $[\delta] = \{\delta' \mid \delta \cong \delta'\}$*. We denote by* $[\mathcal{D}]$ *the set of abstract* DPO *diagram.*

Definition 15 (Abstract Marked Transition System)**.** *The* abstract marked transition system *(*AMTS*) of* $\mathcal{G}$ *is an LTS* $(\mathcal{P} \times [|\mathbf{Graph}_T|], \mathcal{R} \times [\mathcal{D}] \times 2^{\mathcal{R}} \cup \{\checkmark\}, \to_{[\mathcal{D}]})$ *with* $\to_{[\mathcal{D}]}$ *being the least relation satisfying the following rule as well as rule STOP from Definition 10:*

$$MARK\,\frac{P \xrightarrow{(\rho,N)} P' \qquad G \overset{\rho @ m}{\Longrightarrow} H \qquad \forall p \in N : G \overset{p}{\not\Rightarrow}}{(P,[G]) \xrightarrow{(\rho,[\delta],N)}_{[\mathcal{D}]} (P',[H])}$$

where $[\delta]$ *is an abstract DPO diagram over* ρ*.*

Proposition 4 (AMTS is Finite Branching)**.** *If for each rule* $p\colon (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *in* $\mathcal{R}$ *the left-hand side* l *is not surjective, then the* AMTS *of* $\mathcal{G}$ *is finite-branching, i.e., for each unmarked process* P *and* T*-typed graph* G *there is a finite number of transitions from* $(P, [G])$*.*

Considering graphs and DPO diagrams only up to isomorphism is safe for the kind of equivalences of systems considered below, based on traces or on bisimilarity, but it is known to be problematic for example for a truly concurrent semantics of GTSs [2]. For instance, rule p_u in Sec. 3.1 has two different matches in a graph identical to its left-hand side, but it induces a single abstract DPO diagram. If only the latter is given, it could be impossible to determine whether such rule application is parallel independent or not from another one.

Equivalences. To capture the equivalence of reactive (non-terminating) processes in a branching-sensitive manner, finer equivalence notions are required. *Bisimilarity* is a well-known branching-sensitive equivalence notion.

Definition 16 (Simulation, Bisimulation)**.** *Given an LTS* $(S, A, \rightarrow_X)$ *and* $s, t \in S$*. A* simulation *is a relation* $\mathsf{R} \subseteq S \times S$ *s.t. whenever* $s \mathrel{\mathsf{R}} t$*, for each transition* $s \xrightarrow{\alpha}_X s'$ *(with* $\alpha \in A$*), there exists a transition* $t \xrightarrow{\alpha}_X t'$ *with* $s' \mathrel{\mathsf{R}} t'$*. State* s *is* simulated by t *if there is a simulation relation* R *such that* $s \mathrel{\mathsf{R}} t$*.*

A bisimulation *is a symmetric simulation. States* s *and* t *are* bisimilar*, denoted* $s \simeq_X^{BS} t$*, if there is a bisimulation* R *such that* $s \mathrel{\mathsf{R}} t$*.*

Note that as we compare labels containing full DPO diagrams involving also input graphs, in order for two MTS processes to be bisimilar, their graphs should be the same concrete graphs. In the case of AMTS, both graphs should be in the same isomorphism class, i.e., they are isomorphic.

First, we state that simulation is "faithful" to synchronization, i.e., a parallel process simulates a sequential one if the latter can apply the corresponding parallel rule.

Proposition 5. *Given unmarked processes* P_1, P_2, P_3, Q *with bisimilar associated UTSs and actions* $\gamma_1 = (\rho_1, N_1), \gamma_2 = (\rho_2, N_2), \gamma_c = (\rho_1|\rho_2, N_1 \cup N_2)$*. Let* $P_0 := \gamma_1.\gamma_2.P_1 + \gamma_2.\gamma_1.P_2 + \gamma_c.P_3$ *and* $Q_0 := \gamma_1.Q \,||\, \gamma_2.\mathbf{0}$*.*

Then, the process (P_0, G) *is simulated by* (Q_0, G)*. Moreover,* (P_0, G) *and* (Q_0, G) *are bisimilar if* $P_1 := \mathbf{0}$*,* $P_2 := \mathbf{0}$*,* $P_3 := \mathbf{0}$ *and* $Q := \mathbf{0}$*.*

The following proposition states that, as expected, "concrete" equivalence implies abstract equivalence w.r.t. trace equivalence as well as bisimilarity.

Proposition 6. *Given* $P, Q \in \mathcal{P}$ *and* $G, H \in |\mathbf{Graph}_T|$*, (1)* $(P, G) \simeq_{\mathcal{D}}^T (Q, H)$ *implies* $(P, [G]) \simeq_{[\mathcal{D}]}^T (Q, [H])$ *and (2)* $(P, G) \simeq_{\mathcal{D}}^{BS} (Q, H)$ *implies* $(P, [G]) \simeq_{[\mathcal{D}]}^{BS} (Q, [H])$*.*

Now, to conclude the different kinds of transition systems and their equivalences, we show that unmarked process equivalence and graph isomorphism implies marked process equivalence for both trace equivalence and bisimilarity, in both a concrete and an abstract setting, as expected. (The abstract case is a direct consequence of Proposition 6.)

Proposition 7. *For any* $P, Q \in \mathcal{P}$ *and* $G \in |\mathbf{Graph}_T|$*, (1)* $P \simeq^T Q$ *implies* $(P, G) \simeq_{\mathcal{D}}^T (Q, G)$ *and (2)* $P \simeq^{BS} Q$ *implies* $(P, G) \simeq_{\mathcal{D}}^{BS} (Q, G)$*.*

For a bisimilarity relation, it is an important property if the processes retain bisimilarity if put into different contexts, i.e., if it is a *congruence*. In the following, we show that our abstract AMTS bisimilarity has the desired property of being retained if the control processes are expanded by a further context. (We have a similar result for $\simeq_{\mathcal{D}}^{BS}$ if we set the same concrete starting graph G for both sides.)

Theorem 1. *Given* $P, Q \in \mathcal{P}$ *with* $P \simeq^{BS} Q$ *as well as* $G \in |\mathbf{Graph}_T|$*. Then, the following hold (with* $R \in \mathcal{P}$*):*

1. $(P + R, [G]) \simeq^{BS}_{[\mathcal{D}]} (Q + R, [G])$,
2. $(P \,||\, R, [G]) \simeq^{BS}_{[\mathcal{D}]} (Q \,||\, R, [G])$, *and*
3. $(\gamma.P, [G]) \simeq^{BS}_{[\mathcal{D}]} (\gamma.Q, [G])$ *for any* $\gamma \in \mathcal{R}^* \times 2^{\mathcal{R}}$.

Proof. First, we prove that UTS bisimilarity ($\simeq^{BS}$) is a congruence w.r.t. those operators. In particular, $P \simeq^{BS} Q$ implies the following:

1. $P + R \simeq^{BS} Q + R$: The resulting transition system on both sides arises as a union (i.e., "gluing" at the root) of P and R on the left-hand side as well as Q and R on the right-hand side. Then, the statement follows from $P \simeq^{BS} Q$.
2. $P \,||\, R \simeq^{BS} Q \,||\, R$: The proof is done by coinduction. At the starting state, there are three possibilities for firing transitions: (i) P (Q) fires, (ii) R fires or (iii) synchronization (cf. rule SYNC in Fig. 2) takes place.
   Transitions of case (i) are covered w.r.t. bisimilarity through the assumption. If R fires as in case (ii), proceeding to R', the resulting marked states constitute a pair which is of the same form as our starting pair: $P \,||\, R'$ and $Q \,||\, R'$. Thus, the statement holds by coinduction. In case (iii), we have the same situation as in case (ii): if each of the partaking processes $X \in \{P, Q, R\}$ proceeds to X' and synchronization takes place, the resulting processes are $P' \,||\, R'$ and $Q' \,||\, R'$. Moreover, $P' \simeq^{BS} Q'$ due to $P \simeq^{BS} Q$. Thus, the statement holds by coinduction.
3. $\gamma.P \simeq^{BS} \gamma.Q$: Here, on both sides, the outgoing transitions of the starting state correspond to transitions over γ, after which the two sides become P and Q, respectively. Thus, the statement follows from $P \simeq^{BS} Q$. □

The statements in the Theorem are easy consequences of the fact that UTS bisimilarity is a congruence, as well as Propositions 6 and 7.

Note that a similar result would hold only for choice if we do not require unmarked bisimilarity; assuming only marked bisimilarity, parallel composition and prefixing might introduce fresh rule applications, leading to fresh graph states where the behavior of the two sides might diverge. For instance, using rules from Sec. 3.1, $(\rho_u.\mathbf{0}, G) \simeq^{BS}_{[\mathcal{D}]} (\rho_u.\mathbf{0} + \rho_e.\mathbf{0}, G)$ if G has no triangle to apply ρ_e on; however, a parallel (or prefix) context where ρ_a might create a match for ρ_e ruins bisimilarity as the right-side process has a ρ_e-transition that the other cannot simulate.

Summarizing, an abstract representation of a marked transition system might be a useful tool in order to gain a finite representation. Investigating equivalence notions, AMTS exhibits a trade-off between hiding some execution details on the one hand, but enabling a bisimilarity congruence for control processes on the other hand.

Independence. In this section, we demonstrate how abstract marked processes allow for a novel characterization of parallel independence in the context of controlled graph-rewriting processes. Intuitively, two rule applications available simultaneously are *parallel independent* if after performing any of the applications, the other rule is still applicable *on the same match image as in the original* *rule application* (cf. Proposition 1). In the following, we refer to a 4-tuple of DPO diagrams $\delta'_2, \delta_1, \delta_2, \delta'_1$ as *strictly confluent* [7] if they correspond to some matches m'_2, m_1, m_2, m'_1 as in Proposition 1. Now, we are ready to re-interpret the notion of parallel independence for marked transitions.

Definition 17 (Parallel Transition Independence)**.** *Given an abstract marked process* $(P, [G])$ *with outgoing transitions* $(P, [G]) \xrightarrow{(\rho_1,[\delta_1],N_1)}_{[\mathcal{D}]} (P_1, [H_1])$ *and* $(P, [G]) \xrightarrow{(\rho_2,[\delta_2],N_2)}_{[\mathcal{D}]} (P_2, [H_2])$*, these outgoing* transitions *are* parallel independent *if there exist the following transitions:*

(i) $(P_1, [H_1]) \xrightarrow{(\rho_2,[\delta'_2],N_2)}_{[\mathcal{D}]} (P_{12}, [H_{12}])$*, and*
(ii) $(P_2, [H_2]) \xrightarrow{(\rho_1,[\delta'_1],N_1)}_{[\mathcal{D}]} (P_{21}, [H_{21}])$

such that there is a 4-tuple of (representative) elements of $[\delta'_2], [\delta_1], [\delta_2], [\delta'_1]$ *which is strict confluent and* $(P_{12}, [H_{12}]) \simeq^{BS}_{[\mathcal{D}]} (P_{21}, [H_{21}])$*.*

The following proposition states that parallel *transition* independence implies parallel independence of the involved rule applications. Note that the inverse implication does not hold, as parallel independence is defined only for rules and their matches; thus, it might happen that some non-applicability conditions prevent a subsequent rule application even if the applications themselves are parallel independent.

Proposition 8. *Given two parallel independent transitions* $(P, [G]) \xrightarrow{(\rho_1,[\delta_1],N_1)}_{[\mathcal{D}]} (P_1, [H_1])$ *and* $(P, [G]) \xrightarrow{(\rho_2,[\delta_2],N_2)}_{[\mathcal{D}]} (P_2, [H_2])$*, the corresponding rule applications* $\rho_1@m_1$ *and* $\rho_2@m_2$*, respectively, are parallel independent.*

For instance, in our example in Sec. 3.1, if (P_{TC}, G) has outgoing transitions for both p_a and p_u (on some graph G), then we know that the rule applications inducing those transitions were parallel independent. In that case, those applications can be executed also as a synchronized action. Note, instead, that if we consider a different pair of rules like p_e and p_u, not each of their applications to the same graph is independent as their left-hand sides have common elements and thus their matches might overlap.

Finally, we elaborate on the consequences of parallel independence in the presence of synchronization. Particularly, (1) for actions without non-applicability conditions, the absence of a synchronized transition indicates the parallel *dependence* of transitions, and (2) parallel transition independence is equivalent to the existence of a synchronized action in parallel processes, i.e., synchronization implies strict confluence.

Theorem 2 (Bisimilarity and Parallel Independence). *Given bisimilar unmarked processes* P_1, P_2, Q_1, Q_2 *and actions* $\gamma_1 = (\rho_1, N_1)$*,* $\gamma_2 = (\rho_2, N_2)$ *with rules* ρ_1, ρ_2*.*

1. *Let* $P'_0 := \rho_1.(\rho_2 \,||\, P_1) + \rho_2.(\rho_1 \,||\, P_2)$ *and* $Q_0 := \rho_1.Q_1 \,||\, \rho_2.\mathbf{0}$*. There exist no parallel independent applications* $\rho_1@m_1, \rho_2@m_2$ *on* G*, if and only if* $(P'_0, [G]) \simeq^{BS}_{\mathcal{D}} (Q_0, [G])$*.*
2. *Let* $(Q'_0, [G]) := (\gamma_1.Q_1 \,||\, \gamma_2.Q_2, [G])$*. Two transitions* $(Q'_0, [G]) \xrightarrow{(\rho_1,\delta_1,N_1)}_{[\mathcal{D}]} (Q_1 \,||\, \gamma_2.Q_2, [H_1]), (Q'_0, [G]) \xrightarrow{(\rho_2,\delta_2,N_2)}_{[\mathcal{D}]} (\gamma_1.Q_1 \,||\, Q_2, [H_2])$ *are parallel independent if and only if there are transitions* $(Q'_0, [G]) \xrightarrow{(\rho_1|\rho_2,\delta_c,N_1\cup N_2)}_{[\mathcal{D}]} (Q_1 \,||\, Q_2, [H])$*,* $(Q_1 \,||\, \gamma_2.Q_2, [G]) \xrightarrow{(\rho_2,\delta'_2,N_2)}_{[\mathcal{D}]} (Q_1 \,||\, Q_2, [H])$*, and* $(\gamma_1.Q_1 \,||\, Q_2, [H_2]) \xrightarrow{(\rho_1,\delta'_1,N_1)}_{[\mathcal{D}]} (Q_1 \,||\, Q_2, [H])$*.*

Proof. 1. **If:** We prove the statement indirectly. First, let us observe that if there is a pair of outgoing transitions over ρ_1 and ρ_2 in $(P'_0, [G])$, then those transitions are also present in (Q_0, G). Thus, let us assume that there are $\rho_1@m_1, \rho_2@m_2$ parallel independent. Then, there is also an outgoing transition $(\rho_1|\rho_2, [\delta_c], \emptyset)$: We set $m_c : L_1 + L_2 \to G$ of δ_c such that $m_c = m_1 + m_2$. This transition cannot be mimicked by $(P'_0, [G])$, a contradiction.

Only if: If there are no parallel independent transitions of ρ_1 and ρ_2, then $(Q_0, [G])$ is unable to synchronize: If there would be a match m_c of $\rho_1|\rho_2$, then there also would be parallel independent matches m_1, m_2 of ρ_1, ρ_2 separately, by taking $m_1 = m_c \circ e_1$ and $m_2 = m_c \circ e_2$, where e_1 and e_2 are the obvious embeddings of the left-hand sides in the coproduct, i.e., $L_1 \xrightarrow{e_1} L_1 + L_2 \xleftarrow{e_2} L_2$. Thus, the transition sequences induced by ρ_1, ρ_2 are the same in $(P'_0, [G])$ and $(Q_0, [G])$.

2. **If:** The existence of a transition $(\rho_1|\rho_2, [\delta_c], N_1 \cup N_2)$ implies the existence of parallel independent matches m_1, m_2 for ρ_1, ρ_2 due to the construction in Clause 1 above. Thus, there are parallel independent transitions from (Q'_0, G) over $\rho_1@m_1$ and $\rho_2@m_2$, respectively, as we know from the synchronized transition that both N_1 and N_2 hold in G. By the assumption, we also know that there is at least one application of ρ_1 after which ρ_2 is applicable and N_2 holds, and the same vice versa. If those applications were using other matches than m_1, m_2, i.e., if the corresponding DPO diagrams were not isomorphic to $\delta_1, \delta_2, \delta'_1, \delta'_2$, then at least one of the graphs resulting from the sequences $\rho_1.\rho_2$ and $\rho_2.\rho_1$ were not isomorphic to H, the result of the synchronized rule application.

Only if: This is a direct consequence of Definition 17 and the construction of m_c in Clause 1 above.

6 Conclusions and Future Work

In this paper we have introduced an original approach to controlled graph-rewriting, where the control layer is described using terms of a simple process calculus, instead of standard programming constructs as in most other approaches. We have presented an operational semantics for processes, enabling a novel perspective on the equivalence of those processes on the one hand, and (in)dependence of processes running in parallel on the other hand. Among other things, we have shown that congruence of the bisimilarity relation is achieved by abstracting from concrete graph details. Furthermore, we have re-interpreted the notion of parallel independence in our operational setting and shown that synchronization and bisimulation captures parallel (in)dependence as present in controlled graph-rewriting processes.

Among the several topics that we intend to address in future work, we mention (i) to study conditions of bisimilarity and/or simulation among marked processes which are more interesting than those pretty elementary addressed in this paper; (ii) to compare our notion of *process* bisimulation with the *graph-interface* bisimulation of Ehrig and König [8] by including their generalized notion of graph-rewriting steps in our framework; (iii) to investigate a Petri net interpretation, particularly the connection between the non-applicability conditions introduced here and *inhibitor arcs*; (iv) exploiting the process calculus framework, to explore composition (or synchronization) operations that are not conservative with respect to linear derivations, like for example amalgamation; (v) to consider a more elaborate notion of transition independence for capturing true concurrency of rule applications more faithfully.

References

1. Baldan, P., Bruni, A., Corradini, A., König, B., Rodríguez, C., Schwoon, S.: Efficient unfolding of contextual Petri nets. Theor. Comput. Sci. **449**, 2–22 (2012). https://doi.org/10.1016/j.tcs.2012.04.046
2. Baldan, P., Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Löwe, M.: Concurrent semantics of algebraic graph transformation. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 3, pp. 107–187. World Scientific (1999). https://doi.org/10.1142/9789812814951_0003
3. Bunke, H.: Programmed graph grammars. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 155–166. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0025718
4. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundam. Inform. **26**(3/4), 241–265 (1996). https://doi.org/10.3233/FI-1996-263402
5. Corradini, A., et al.: On the essence of parallel independence for the double-pushout and sesqui-pushout approaches. In: Heckel, R., Taentzer, G. (eds.) Graph Transformation, Specifications, and Nets. LNCS, vol. 10800, pp. 1–18. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75396-6_1
6. Dassow, J., Păun, G., Salomaa, A.: Grammars with controlled derivations. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 2, pp. 101–154. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-662-07675-0_3
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
8. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In: Walukiewicz, I. (ed.) FoSSaCS 2004. LNCS, vol. 2987, pp. 151–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24727-2_12

9. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the unified modeling language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46464-8_21
10. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Honsell, F., Miculan, M. (eds.) FoSSaCS 2001. LNCS, vol. 2030, pp. 230–245. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45315-6_15
11. Kluge, R., Stein, M., Varró, G., Schürr, A., Hollick, M., Mühlhäuser, M.: A systematic approach to constructing families of incremental topology control algorithms using graph transformation. Softw. Syst. Model. **38**, 47–83 (2017). https://doi.org/10.1016/j.jvlc.2016.10.003
12. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 138–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_10
13. Plump, D., Steinert, S.: The semantics of graph programs. In: RULE. EPTCS, vol. 21 (2009). https://doi.org/10.4204/EPTCS.21.3
14. Schürr, A.: Logic-based programmed structure rewriting systems. Fundam. Inform. **26**(3, 4), 363–385 (1996). https://doi.org/10.3233/FI-1996-263407
15. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES-approach: language and environment. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 487–550. World Scientific (1999). https://doi.org/10.1142/9789812815149_0013

Graph Conditions and Verification

Verifying Graph Transformation Systems with Description Logics

Jon Haël Brenas[1](✉), Rachid Echahed[2](✉), and Martin Strecker[3](✉)

[1] UTHSC - ORNL, Memphis, TN, USA
`jhael@uthsc.edu`
[2] CNRS and University Grenoble-Alpes, LIG Lab., Grenoble, France
`rachid.echahed@imag.fr`
[3] Université de Toulouse, IRIT, Toulouse, France
`martin.strecker@irit.fr`

Abstract. We address the problem of verification of graph transformations featuring actions such as node *merging* and *cloning*, *addition* and *deletion* of nodes and edges, node or edge *labeling* and edge *redirection*. We introduce the considered graph rewrite systems following an algorithmic approach and then tackle their formal verification by proposing a Hoare-like weakest precondition calculus. Specifications are defined as triples of the form {`Pre`}(`R`, `strategy`){`Post`} where `Pre` and `Post` are conditions specified in a given Description Logic (DL), `R` is a graph rewrite system and `strategy` is an expression stating in which order the rules in `R` are to be performed. We prove that the proposed calculus is sound and characterize which DL logics are suited or not for the targeted verification tasks, according to their expressive power.

1 Introduction

Graphs, as well as their transformations, play a central role in modelling data in various areas such as chemistry, civil engineering or computer science. In many such applications, it may be desirable to be able to prove that graph transformations are correct, i.e., from any graph (or state) satisfying a given set of conditions, only graphs satisfying another set of conditions can be obtained.

The correctness of graph transformations has attracted some attention in recent years, see e.g., [3,5,6,9,16,19,21,24]. In this paper, we provide a Hoare-like calculus to address the problem of correctness of programs defined as strategy expressions over graph rewrite rules. Specifications are defined as triples of the form {`Pre`}(`R`, `strategy`){`Post`} where `Pre` and `Post` are conditions, `R` is a graph rewrite system and `strategy` is an expression stating how rules in `R` are to be performed. Our work is thus close to [9,16,21] but differs both on the class of the considered rewrite systems as well as on the logics used to specify `Pre` and `Post` conditions.

The considered rewrite rules follow an algorithmic approach where the left-hand sides are attributed graphs and the right-hand sides are sequences of elementary actions [14]. Among the considered actions, we quote node and edge

L. Lambers and J. Weber (Eds.): ICGT 2018, LNCS 10887, pp. 155–170, 2018.
https://doi.org/10.1007/978-3-319-92991-0_10

addition or *deletion*, node and edge *labelling* and edge *redirection*, in addition to node *merging* and *cloning*. To our knowledge, the present work is the first to consider the verification of graph transformations including node cloning.

Hoare-like calculi for the verification of graph transformations have already been proposed with different logics to express the pre- and post-conditions. Among the most prominent approaches figure nested conditions [16,21] that are explicitly created to describe graph properties. The considered graph rewrite transformations are based on the double pushout approach with linear spans which forbid actions such as node merging and node cloning.

Other logics might be good candidates to express graph properties which go beyond first-order definable properties such as monadic second-order logic [13,22] or the dynamic logic defined in [5]. These logics are undecidable in general and thus either cannot be used to prove correctness of graph transformations in an automated way or only work on limited classes of graphs.

Starting from the other side of the logical spectrum, one could consider the use of decidable logics such as fragments of Description Logics (DLs) to specify graph properties [1,8]. DLs [4] are being used heavily in formal knowledge representation languages such as Owl [2]. In this paper we consider the case where `Pre` and `Post` conditions are expressed in DLs and show which Description Logic can be used or not for the targeted verification problems.

The use of decidable logics contributes to the design of a push-button technology that gives definite and precise answers to verification problems. Model-checking [24] provides an alternative to our approach with ready to use tools, such as Alloy [7,18] or GROOVE [15]. The main issue with those tools is that they are restricted to finding counter-examples, instead of a full verification, when the set of possible models is infinite (or too large to be checked in a timely manner), and thus provide only part of the solution. On the other hand, techniques based on abstract interpretation (such as [23]) are not guaranteed to give correct answers and have a risk of false positive or negative.

The paper is organized as follows. Section 2 provides the considered definitions of graphs and the elementary graph transformation actions. Section 3 recalls useful notions of Descriptions Logics. In Sect. 4, we define the considered graph rewrite systems and strategies. Section 5 provides a sound Hoare-like calculus and states which DLs can be used or not for the considered program verification problems. Section 6 concludes the paper. The missing proofs and definitions can be consulted in [10].

2 Preliminaries

We first define the notion of *decorated graphs* we consider in this paper.

Definition 1 (Decorated Graph). *Let $\mathcal{C}$ (resp. $\mathcal{R}$) be a set of node labels (resp. edge labels). A* decorated graph *G over a graph alphabet $(\mathcal{C}, \mathcal{R})$ is a tuple $(N, E, \Phi_N, \Phi_E, s, t)$ where N is a set of* nodes*, E is a set of* edges*, Φ_N is a* node labeling *function, $\Phi_N : N \to \mathcal{P}(\mathcal{C})$, Φ_E is an* edge labeling *function,*

$\Phi_E : E \rightarrow \mathcal{R}$, s is a source function *$s : E \rightarrow N$ and t is a* target function *$t : E \rightarrow N$.*

Notice that nodes are decorated by means of subsets of $\mathcal{C}$ while edges are labeled with a single element in $\mathcal{R}$.

Graph transformation systems considered in this paper follow an algorithmic approach based on the notion of *elementary actions* introduced below.

Definition 2 (Elementary action, action). *Let $\mathcal{C}_0$ (resp. $\mathcal{R}_0$) be a set of node (resp. edge) labels. An* elementary action*, say a, may be of the following forms:*

- *a* node addition *$add_N(i)$ (resp.* node deletion *$del_N(i)$) where i is a new node (resp. an existing node). It creates the node i. i has no incoming nor outgoing edge and it is not labeled (resp. it deletes i and all its incident edges).*
- *a* node label addition *$add_C(i, c)$ (resp.* node label deletion *$del_C(i, c)$) where i is a node and c is a label in $\mathcal{C}_0$. It adds the label c to (resp. removes the label c from) the labeling of node i.*
- *an* edge addition *$add_E(e, i, j, r)$ (resp.* edge deletion *$del_E(e, i, j, r)$) where e is an edge, i and j are nodes and r is an edge label in $\mathcal{R}_0$. It adds the edge e with label r between nodes i and j (resp. removes all edges with source i and target j with label r).*
- *a* global edge redirection *$i \gg j$ where i and j are nodes. It redirects all incoming edges of i towards j.*
- *a* merge action *$mrg(i, j)$ where i and j are nodes. This action merges the two nodes. It yields a new graph in which the first node i is labeled with the union of the labels of i and j and such that all incoming or outgoing edges of any of the two nodes are gathered.*
- *a* clone action *$cl(i, j, L_{in}, L_{out}, L_{l\_in}, L_{l\_out}, L_{l\_loop})$ where i and j are nodes and L_{in}, L_{out}, L_{l_in}, L_{l_out} and L_{l_loop} are subsets of $\mathcal{R}_0$. It clones a node i by creating a new node j and connects j to the rest of a host graph according to different information given in the parameters $L_{in}, L_{out}, L_{l\_in}, L_{l\_out}, L_{l\_loop}$ as specified further below.*

The result of performing an elementary action *a on a graph $G = (N^G, E^G, \Phi_N^G, \Phi_E^G, s^G, t^G)$, written $G[a]$, produces the graph $G' = (N^{G'}, E^{G'}, \Phi_N^{G'}, \Phi_E^{G'}, s^{G'}, t^{G'})$ as defined in Fig. 1. A (composite)* action*, say α, is a sequence of elementary actions of the form $\alpha = a_1; a_2; \ldots; a_n$. The result of performing α on a graph G is written $G[\alpha]$. $G[a; \alpha] = (G[a])[\alpha]$ and $G[\epsilon] = G$ where ϵ is the empty sequence.*

The elementary action $cl(i, j, L_{in}, L_{out}, L_{l\_in}, L_{l\_out}, L_{l\_loop})$ might be not easy to grasp at first sight. It thus deserves some explanations. Let node j be a clone of node i. What would be the incident edges of the clone j? Answering this question is not straightforward. There are indeed different possibilities to connect j to the neighborhood of i. Figure 2 illustrates such a problem where node q_1', a clone of node q_1, has indeed different possibilities to be connected to the other nodes. In order to provide a flexible clone action, the user may tune the way the edges connecting a clone are treated through the five parameters $L_{in}, L_{out}, L_{l\_in}, L_{l\_out}, L_{l\_loop}$. All these parameters are subsets of the set of edge labels $\mathcal{R}_0$ and are explained informally below:

- L_{in} (resp. L_{out}) indicates that every incoming (resp. outgoing) edge e of i, which is not a loop, and whose label is in L_{in} (resp. L_{out}) is cloned as a new edge e' such that $s(e') = s(e)$ and $t(e') = j$ (resp. $s(e') = j$ and $t(e') = t(e)$).
- L_{l_in} indicates that every self-loop e over i whose label is in L_{l_in} is cloned as a new edge e' with $s(e') = i$ and $t(e') = j$. (see the blue arrow in Fig. 2).
- L_{l_out} indicates that every self-loop e over i whose label is in L_{l_out} is cloned as a new edge e' with $s(e') = j$ and $t(e') = i$. (see the red arrow in Fig. 2).
- L_{l_loop} indicates that every self-loop e over i whose label is in L_{l_loop} is cloned as a new edge e' which is a self-loop over j, i.e., $s(e') = j$ and $t(e') = j$. (see the self-loop over node q_1' in Fig. 2).

The semantics of the cloning action $cl(i, j, L_{in}, L_{out}, L_{l\_in}, L_{l\_out}, L_{l\_loop})$ as defined in Fig. 1 use some auxiliary pairwise disjoint sets representing the new edges that are created according to how the clone j should be connected to the neighborhood of node i. These sets of new edges are denoted $E'_{in}, E'_{out}, E'_{l\_in}, E'_{l\_out}$ and E'_{l_loop}. They are provided with the auxiliary bijective functions *in*, *out*, *l_in*, *l_out* and *l_loop* as specified below.

1. E'_{in} is in bijection through function *in* with the set $\{e \in E^G |\ t^G(e) = i \wedge s^G(e) \neq i \wedge \Phi_E^G(e) \in L_{in}\}$,
2. E'_{out} is in bijection through function *out* with the set $\{e \in E^G |\ s^G(e) = i \wedge t^G(e) \neq i \wedge \Phi_E^G(e) \in L_{out}\}$,
3. E'_{l_in} is in bijection through function *l_in* with the set $\{e \in E^G |\ s^G(e) = t^G(e) = i \wedge \Phi_E^G(e) \in L_{l\_in}\}$,
4. E'_{l_out} is in bijection through function *l_out* with the set $\{e \in E^G |\ s^G(e) = t^G(e) = i \wedge \Phi_E^G(e) \in L_{l\_out}\}$,
5. E'_{l_loop} is in bijection through function *l_loop* with the set $\{e \in E^G |\ s^G(e) = t^G(e) = i \wedge \Phi_E^G(e) \in L_{l\_loop}\}$).

Informally, the set E'_{in} contains a copy of every incoming edge e of node i (i.e., $t^G(e) = i$), which is not a self-loop (i.e., $s^G(e) \neq i$), and having a label in L_{in}, (i.e., $\Phi_E^G(e) \in L_{in}$). L_{in} is thus used to select which incoming edges of node i are cloned. The other sets E'_{out}, E'_{l_in}, E'_{l_out} and E'_{l_loop} are defined similarly.

Example 1. Let $\mathcal{A}$ be the graph of Fig. 2a, over an alphabet $(\mathcal{C}_0, \mathcal{R}_0)$ such that $\{a, b\} \subseteq \mathcal{R}_0$. Performing the action $cl(q_1, q_1', \mathcal{R}_0, \mathcal{R}_0, X, Y, Z)$ yields the graph presented in Fig. 2b where the blue-plain (resp. red-dashed, purple-dotted) edge exists iff X (resp. Y, Z) contains the label $\{a\}$.

Readers familiar with algebraic approaches to graph transformation may recognize the cloning flexibility provided by the recent AGREE approach [11]. The parameters of the clone action reflect somehow the embedding morphisms of AGREE-rules. Cloning a node according to the approach of Sesquipushout [12] could be easily simulated by instantiating all the parameters by the full set of edge labels, i.e., $cl(i, j, \mathcal{R}_0, \mathcal{R}_0, \mathcal{R}_0, \mathcal{R}_0, \mathcal{R}_0)$.

If $\alpha = \mathit{add}_C(i, c)$ **then:**
$N^{G'} = N^G, E^{G'} = E^G,$
$$\Phi_N^{G'}(n) = \begin{cases} \Phi_N^G(n) \cup \{c\} & \text{if } n = i \\ \Phi_N^G(n) & \text{if } n \neq i \end{cases}$$
$\Phi_E^{G'} = \Phi_E^G,\ s^{G'} = s^G,\ t^{G'} = t^G$

If $\alpha = \mathit{add}_E(e, i, j, r)$ **then:**
$N^{G'} = N^G,\ \Phi_N^{G'} = \Phi_N^G$
$E^{G'} = E^G \cup \{e\}$
$$\Phi_E^{G'}(e') = \begin{cases} r & \text{if } e' = e \\ \Phi_E^G(e') & \text{if } e' \neq e \end{cases}$$
$s^{G'}(e') = s^G(e')$ if $e' \neq e, s^{G'}(e) = i$
$t^{G'}(e') = t^G(e')$ if $e' \neq e, t^{G'}(e) = j$

If $\alpha = \mathit{add}_N(i)$ **then:**
$N^{G'} = N^G \cup \{i\}$ where i is a new node
$E^{G'} = E^G,\ \Phi_E^{G'} = \Phi_E^G, s^{G'} = s^G,\ t^{G'} = t^G$
$$\Phi_N^{G'}(n) = \begin{cases} \emptyset & \text{if } n = i \\ \Phi_N^G(n) & \text{if } n \neq i \end{cases}$$

If $\alpha = \mathit{del}_N(i)$ **then:**
$N^{G'} = N^G \backslash \{i\}$
$E^{G'} = E^G \backslash \{e | s^G(e) = i \vee t^G(e) = i\}$
$\Phi_N^{G'}$ is the restriction of Φ_N^G to $N^{G'}$
$\Phi_E^{G'}$ is the restriction of Φ_E^G to $E^{G'}$
$s^{G'}$ is the restriction of s^G to $E^{G'}$
$t^{G'}$ is the restriction of t^G to $E^{G'}$

If $\alpha = i \gg j$ **then:**
$N^{G'} = N^G,\ E^{G'} = E^G$
$\Phi_N^{G'} = \Phi_N^G,\ \Phi_E^{G'} = \Phi_E^G, s^{G'} = s^G$
$$t^{G'}(e) = \begin{cases} j & \text{if } t^G(e) = i \\ t^G(e) & \text{if } t^G(e) \neq i \end{cases}$$

If $\alpha = \mathit{mrg}(i, j)$ **then:**
$N^{G'} = N^G \backslash \{j\}, E^{G'} = E^G, \Phi_E^{G'}(e) = \Phi_E^G(e)$
$$\Phi_N^{G'}(n) = \begin{cases} \Phi_N^G(i) \cup \Phi_N^G(j) & \text{if } n = i \\ \Phi_N^G(n) & \text{otherwise} \end{cases}$$
$$s^{G'}(e) = \begin{cases} i & \text{if } s^G(e) = j \\ s^G(e) & \text{otherwise} \end{cases}$$
$$t^{G'}(e) = \begin{cases} i & \text{if } t^G(e) = j \\ t^G(e) & \text{otherwise} \end{cases}$$

If $\alpha = \mathit{del}_C(i, c)$ **then:**
$N^{G'} = N^G, E^{G'} = E^G,$
$$\Phi_N^{G'}(n) = \begin{cases} \Phi_N^G(n) \backslash \{c\} & \text{if } n = i \\ \Phi_N^G(n) & \text{if } n \neq i \end{cases}$$
$\Phi_E^{G'} = \Phi_E^G,\ s^{G'} = s^G,\ t^{G'} = t^G$

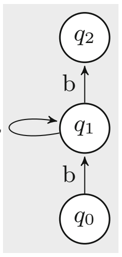
If $\alpha = \mathit{del}_E(e, i, j, r)$ **then:**
$N^{G'} = N^G,\ \Phi_N^{G'} = \Phi_N^G$
$E^{G'} = E^G \backslash \{a \in E^G \mid$
$s^G(a) = i,\ t^G(a) = j$ and $\Phi_E^G(a) = r\}$
$\Phi_E^{G'}$ is the restriction of Φ_E^G to $E^{G'}$
$s^{G'}$ is the restriction of s^G to $E^{G'}$
$t^{G'}$ is the restriction of t^G to $E^{G'}$

If $\alpha = \mathit{cl}(i, j, L_{in}, L_{out}, L_{l\_in}, L_{l\_out}, L_{l\_loop})$ **then:**
$N^{G'} = N^G \cup \{j\}$
$E^{G'} = E^G \cup E'_{in} \cup E'_{out} \cup E'_{l\_in} \cup E'_{l\_out} \cup E'_{l\_loop}$
$$\Phi_N^{G'}(n) = \begin{cases} \Phi_N^G(i) & \text{if } n = j \\ \Phi_N^G(n) & \text{otherwise} \end{cases}$$
$$\Phi_E^{G'}(e) = \begin{cases} \Phi_E^G(\mathit{in}(e)) & \text{if } e \in E'_{in} \\ \Phi_E^G(\mathit{out}(e)) & \text{if } e \in E'_{out} \\ \Phi_E^G(\mathit{l\_in}(e)) & \text{if } e \in E'_{l\_in} \\ \Phi_E^G(\mathit{l\_out}(e)) & \text{if } e \in E'_{l\_out} \\ \Phi_E^G(\mathit{l\_loop}(e)) & \text{if } e \in E'_{l\_loop} \\ \Phi_E^G(e) & \text{otherwise} \end{cases}$$
$$s^{G'}(e) = \begin{cases} s^G(\mathit{in}(e)) & \text{if } e \in E'_{in} \\ j & \text{if } e \in E'_{out} \\ i & \text{if } e \in E'_{l\_in} \\ j & \text{if } e \in E'_{l\_out} \\ j & \text{if } e \in E'_{l\_loop} \\ s^G(e) & \text{otherwise} \end{cases}$$
$$t^{G'}(e) = \begin{cases} j & \text{if } e \in E'_{in} \\ t^G(\mathit{out}(e)) & \text{if } e \in E'_{out} \\ j & \text{if } e \in E'_{l\_in} \\ i & \text{if } e \in E'_{l\_out} \\ j & \text{if } e \in E'_{l\_loop} \\ t^G(e) & \text{otherwise} \end{cases}$$

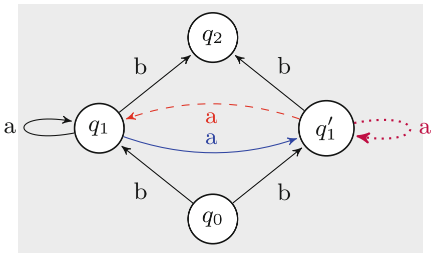
Fig. 1. $G' = G[\alpha]$, summary of the effects of the elementary actions: $\mathit{add}_N(i)$, $\mathit{del}_N(i)$, $\mathit{add}_C(i, c)$, $\mathit{del}_C(i, c)$, $\mathit{add}_E(e, i, j, r)$, $\mathit{del}_E(e)$, $i \gg j$, $\mathit{mrg}(i, j)$ and $\mathit{cl}(i, j, L_{in}, L_{out}, L_{l\_in}, L_{l\_out}, L_{l\_loop})$.

3 DL Logics

Description Logics (DLs) are a family of logic based knowledge representation formalisms. They constitute the basis of well known ontology languages such as the Web Ontology Language OWL [2]. Roughly speaking, a DL syntax allows one to define *Concept* names, which are equivalent to classical first-order logic unary predicates, *Role* names, which are equivalent to binary predicates and



(a) A graph and



(b) the possible results of cloning node q_1 as node q_1'

Fig. 2. Example of application of the elementary action *Clone* (Color figure online)

Individuals, which are equivalent to classical constants. There are various DLs in the literature, they mainly differ by the logical operators they offer to construct concept and role expressions or axioms. In this paper we are interested in extensions of the prototypical DL $\mathcal{ALC}$. We recall that these extensions are named by appending a letter representing additional constructors to the logic name. We focus on nominals (represented by $\mathcal{O}$), counting quantifiers ($\mathcal{Q}$), self-loops (*Self*), inverse roles ($\mathcal{I}$) and the universal role ($\mathcal{U}$). For instance, the logic $\mathcal{ALCUO}$ extends $\mathcal{ALC}$ with the universal role and nominals (see [4], for details about DL names). Below we recall the definitions of DL concepts and roles we consider in this paper.

Definition 3 (Concept, Role). *Let $\mathcal{A} = (\mathcal{C}_0, \mathcal{R}_0, \mathcal{O})$ be an alphabet where $\mathcal{C}_0$ (resp. $\mathcal{R}_0$ and $\mathcal{O}$) is a set of atomic concepts (resp. atomic roles and nominals). Let $c_0 \in \mathcal{C}_0$, $r_0 \in \mathcal{R}_0$, $o \in \mathcal{O}$, and n an integer. The set of concepts C and roles R are defined by:*

$$C := \top \mid c_0 \mid \exists R.C \mid \neg C \mid C \vee C \mid o \ (nominals) \mid \exists R.Self \ (self\ loops)$$
$$\mid (<\ n\ R\ C)\ (counting\ quantifiers)$$
$$R := r_0 \mid U\ (universal\ role) \mid R^-(inverse\ role)$$

For the sake of conciseness, we define $\bot \equiv \neg\top$, $C \wedge C' \equiv \neg(\neg C \vee \neg C')$, $\forall R.C \equiv \neg(\exists R.\neg C)$ and $(\geq\ n\ R\ C) \equiv \neg(<\ n\ R\ C)$.

The concepts C_{alc} and roles R_{alc} of the logic $\mathcal{ALC}$, which stand for "Attributive concept Language with Complement", are subsets of the concepts and roles given above. They can be defined as follows:

$$C_{alc} := \top \mid c_0 \mid \exists R.C_{alc} \mid \neg C_{alc} \mid C_{alc} \vee C_{alc} \qquad \text{and} \qquad R_{alc} := r_0$$

Definition 4 (Interpretation). *An interpretation over an alphabet $(\mathcal{C}_0, \mathcal{R}_0, \mathcal{O})$ is a tuple $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\cdot^{\mathcal{I}}$ is a function such that $c_0^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, for every atomic concept $c_0 \in \mathcal{C}_0$, $r_0^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, for every atomic role $r_0 \in \mathcal{R}_0$, $o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ for every nominal $o \in \mathcal{O}$. The interpretation function is extended to concept and role descriptions by the following inductive definitions:*

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
- $(C \vee D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\exists R.C)^{\mathcal{I}} = \{n \in \Delta^{\mathcal{I}} | \exists m, (n,m) \in R^{\mathcal{I}} \textit{ and } m \in C^{\mathcal{I}}\}$
- $(\exists R.Self)^{\mathcal{I}} = \{n \in \Delta^{\mathcal{I}} | (n,n) \in R^{\mathcal{I}}\}$
- $(< \ n \ R \ C)^{\mathcal{I}} = \{\delta \in \Delta^{\mathcal{I}} | \#(\{m \in \Delta^{\mathcal{I}} | (\delta, m) \in R^{\mathcal{I}} \textit{ and } m \in C^{\mathcal{I}}\}) < n\}$
- $(R^{-})^{\mathcal{I}} = \{(n,m) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} | (m,n) \in R^{\mathcal{I}}\}$
- $U^{\mathcal{I}} = \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

Definition 5 (Interpretation induced by a decorated graph). *Let $G = (N, E, \Phi_N, \Phi_E, s, t)$ be a graph over an alphabet $(\mathcal{C}, \mathcal{R})$ such that $\mathcal{C}_0 \cup \mathcal{O} \subseteq \mathcal{C}$ and $\mathcal{R}_0 \subseteq \mathcal{R}$. The interpretation induced by the graph G, denoted $(\Delta^{\mathcal{G}}, \cdot^{\mathcal{G}})$, is such that $\Delta^{\mathcal{G}} = N$, $c_0^{\mathcal{G}} = \{n \in N | c_0 \in \Phi_N(n)\}$, for every atomic concept $c_0 \in \mathcal{C}_0$, $r_0^{\mathcal{G}} = \{(n,m) \in N \times N | \exists e \in E. s(e) = n \textit{ and } t(e) = m \textit{ and } r_0 = \Phi_E(e)\}$, for every atomic role $r_0 \in \mathcal{R}_0$, $o^{\mathcal{G}} = \{n \in N | o \in \Phi_N(n)\}$ for every nominal $o \in \mathcal{O}$. We say that a node n of a graph G satisfies a concept c, written $n \models c$ if $n \in c^{\mathcal{G}}$. We say that a graph G satisfies a concept c, written $G \models c$ if $c^{\mathcal{G}} = N$, that is every node of G belongs to the interpretation of c induced by G. We say that a concept c is valid if for all graphs G, $G \models c$.*

To illustrate the different notions introduced in this paper, we use a running example inspired from ontologies related to the Malaria surveillance.

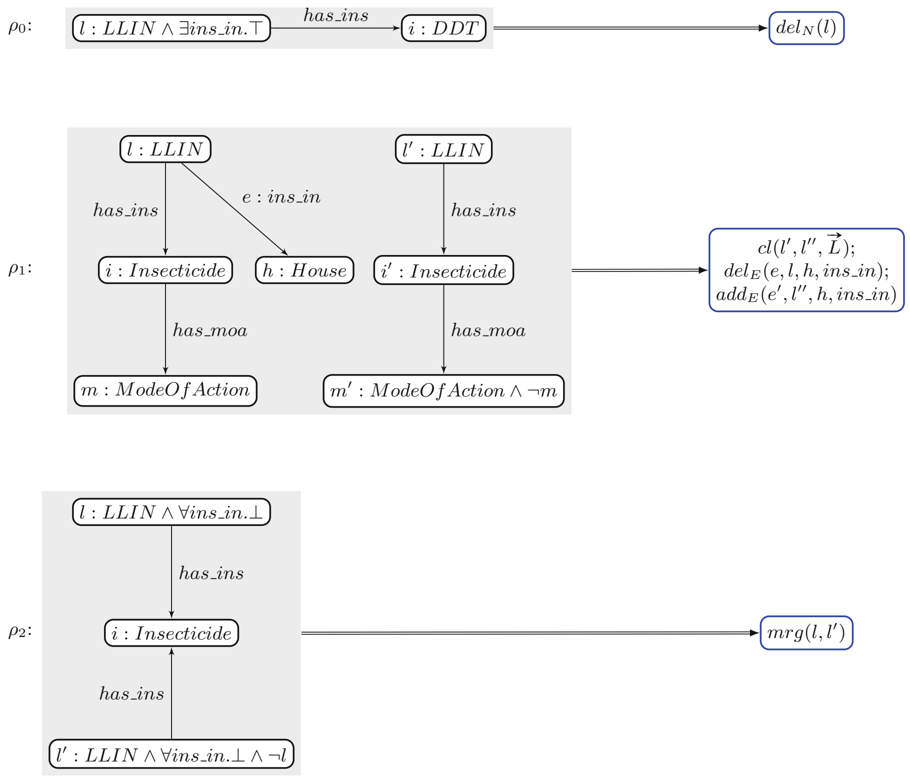
Example 2. Malaria is an infectious, vector-borne disease that overwhelmingly affects Sub-Saharan Africa. In order to reduce its incidence, one of the most widely used techniques is to install long lasting insecticide-treated nets (LLINs). Several materials can be used to produce LLINs and they can be treated with many different insecticides. Each insecticide has a mode of action that characterizes how it affects mosquitoes. In order to avoid the appearance of insecticide resistances in mosquito populations, it is required to use LLINs with different modes of actions.

In this example, we start by giving examples of concepts, roles and nominals related to Malaria surveillance. Let $\mathcal{A}_{mal} = (\mathcal{C}_{mal}, \mathcal{R}_{mal}, \mathcal{O}_{mal})$, be an alphabet such that $\{LLIN, Insecticide, Material, House, ModeOfAction\} \subseteq \mathcal{C}_{mal}$, $\{has\_ins, has\_mat, has\_moa, ins\_in\} \subseteq \mathcal{R}_{mal}$ and $\{i_0, l, mat, DDT\} \subseteq \mathcal{O}_{mal}$. We can then express as concepts that i_0 is an insecticide $(\exists U.i_o \wedge Insecticide)$, that there exists a LLIN using the material mat $(\exists U.mat \wedge \exists has\_mat^{-}.LLIN)$ or that all LLINs except for l are installed in at most one house $(\forall U.LLIN \Rightarrow ((< \ 2 \ ins\_in \ House) \vee l))$.

4 Graph Rewrite Systems and Strategies

In this section, we introduce the notion of *DL decorated graph rewrite systems*. These are extensions of the graph rewrite systems defined in [14] featuring new actions over graph structures decorated over an alphabet $(\mathcal{C}, \mathcal{R})$ consisting of concepts $\mathcal{C}$ and roles $\mathcal{R}$ of a given DL logic.

Definition 6 (Rule, DLGRS). *A* rule *ρ is a pair (L, α) where L, called the left-hand side (lhs), is a decorated graph over $(\mathcal{C}, \mathcal{R})$ and α, called the right-hand side (rhs), is an action. Rules are usually written $L \rightarrow \alpha$. A DL decorated graph rewrite system, DLGRS, is a set of rules.*



ρ_0:
$l : LLIN \wedge \exists ins\_in.\top$
has_ins
$i : DDT$
$del_N(l)$
ρ_1:
$l : LLIN$
has_ins
$e : ins\_in$
$i : Insecticide$
$h : House$
has_moa
$m : ModeOfAction$
$l' : LLIN$
has_ins
$i' : Insecticide$
has_moa
$m' : ModeOfAction \wedge \neg m$
$cl(l', l'', \overrightarrow{L});$
$del_E(e, l, h, ins\_in);$
$add_E(e', l'', h, ins\_in)$
ρ_2:
$l : LLIN \wedge \forall ins\_in.\bot$
has_ins
$i : Insecticide$
has_ins
$l' : LLIN \wedge \forall ins\_in.\bot \wedge \neg l$
$mrg(l, l')$

Fig. 3. Rules used in Example 3. In rule ρ_1, $\overrightarrow{\mathcal{L}} = \{\emptyset, \{has\_ins, has\_mat\}, \emptyset, \emptyset, \emptyset\}$.

Example 3. We now define some rules that can be applied to our malaria example. These rules are given in Fig. 3. Rule ρ_0 searches for a LLIN (l) using DDT as an Insecticide and installed in a place, i.e., such that $l \models \exists ins\_in.\top$. As DDT is highly dangerous to human health, l is then deleted.

Rule ρ_1 changes the LLIN installed in a House. As already said in Example 2, one has to avoid using insecticides with the same mode of action twice in a row in the same House. ρ_1 thus searches for a LLIN (l) that is installed in a House (h) which has an Insecticide (i) with a given ModeOfAction (m). It also searches for a LLIN (l') that has an Insecticide (i') with a different ModeOfAction (m' where $m' \models \neg m$). A new LLIN (l'') is then created by cloning l', the edge, e, between l and h is removed (it loses its label) and a new one, e', is created between l'' and h labeled with ins_in.

The idea behind rule ρ_2 is that there are two kinds of LLINs: those that are currently used, i.e. installed in some House, and those that are used as templates for creating new LLINs by cloning with ρ_1. In order to limit the number of templates, if two LLINs which are not installed anywhere, i.e., they are models of $\forall ins\_in.\bot$, and they use the same Insecticide, then they can be considered as the same, and thus can be merged into one template by rule ρ_2.

Definition 7 (Match). *A* match *h between a lhs L and a graph G is a pair of functions $h = (h^N, h^E)$, with $h^N : N^L \to N^G$ and $h^E : E^L \to E^G$ such that:*
1. $\forall n \in N^L, \forall c \in \Phi_N^L(n), h^N(n) \models c$ *2.* $\forall e \in E^L, \Phi_E^G(h^E(e)) = \Phi_E^L(e)$
3. $\forall e \in E^L, s^G(h^E(e)) = h^N(s^L(e))$ *4.* $\forall e \in E^L, t^G(h^E(e)) = h^N(t^L(e))$.

The third and the fourth conditions are classical and say that the source and target functions and the match have to agree. The first condition says that for every node, n, of the lhs, the node to which it is associated, $h(n)$, in G has to satisfy every concept in $\Phi_N^L(n)$. This condition clearly expresses additional negative and positive conditions which are added to the "structural" pattern matching. The second condition ensures that the match respects edge labeling.

Definition 8 (Rule application). *Let G be a graph decorated over an alphabet $(\mathcal{C}_0, \mathcal{R}_0)$ consisting of atomic concepts $\mathcal{C}_0$ and roles $\mathcal{R}_0$ of a given DL logic. G rewrites into graph G' using a rule $\rho = (L, \alpha)$ iff there exists a match h from L to G. G' is obtained from G by performing actions in $h(\alpha)$*[1]*. Formally, $G' = G[h(\alpha)]$. We write $G \to_\rho G'$.*

Very often, *strategies* are used to control the use of possible rules in rule-based programs (e.g. [20,26]). Informally, a strategy specifies the application order of different rules. It does not indicate where the matches are to be tried nor does it ensure unique normal forms.

Definition 9 (Strategy). *Given a graph rewrite system* R*, a* strategy *is a word of the following language defined by s, where ρ is any rule in* R*:*

$s := \epsilon$ *(Empty Strategy)*	ρ *(Rule)*	$s \oplus s$ *(Choice)*
$s; s$ *(Composition)*	s^* *(Closure)*	$\rho?$ *(Rule Trial)*
$\rho!$ *(Mandatory Rule)*		

Informally, the strategy "$s_1; s_2$" means that strategy s_1 should be applied first, followed by the application of strategy s_2. The expression $s_1 \oplus s_2$ means that either the strategy s_1 or the strategy s_2 is applied. The strategy ρ^* means that rule ρ is applied as many times as possible. Notice that the closure is the standard "while" construct, that is the strategy s^* applies s as much as possible.

Example 4. Let us assume that we want to get rid of DDT-treated LLINs, change the LLINs in 1 or 2 Houses and then remove duplicate templates. In such a situation, one can use the strategy: $\rho_0^*; (\rho_1 \oplus (\rho_1; \rho_1)); \rho_2^*$.

[1] $h(\alpha)$ is obtained from α by replacing every node name, n, of L by $h(n)$.

We write $G \Rightarrow_{\mathsf{s}} G'$ to denote that graph G' is obtained from G by applying the strategy s. In Fig. 4, we provide the rules that specify how strategies are used to rewrite a graph. For that we use the formula $\mathbf{App}(\mathsf{s})$ such that for all graphs G, $G \models \mathbf{App}(\mathsf{s})$ iff the strategy s can perform at least one step over G. This formula is specified below.

- $G \models \mathbf{App}(\rho)$ iff there exists a match h from the left-hand side of ρ to G
- $G \models \mathbf{App}(\rho!)$ iff there exists a match h from the left-hand side of ρ to G
- $G \models \mathbf{App}(\epsilon)$ • $G \models \mathbf{App}(s_0 \oplus s_1)$ iff $G \models \mathbf{App}(s_0)$ or $G \models \mathbf{App}(s_1)$
- $G \models \mathbf{App}(s_0^*)$ • $G \models \mathbf{App}(s_0; s_1)$ iff $G \models \mathbf{App}(s_0)$
- $G \models \mathbf{App}(\rho?)$

Whenever $G \models \mathbf{App}(s)$, this does not mean that the whole strategy, s, can be applied on G, but it rather ensures that at least one step of the considered strategy can be applied.

$$\frac{}{G \Rightarrow_{\epsilon} G}\ \text{(Empty rule)} \qquad \frac{G \Rightarrow_{s_0} G'' \quad G'' \Rightarrow_{s_1} G'}{G \Rightarrow_{s_0;s_1} G'}\ \text{(Strategy composition)}$$

$$\frac{G \Rightarrow_{s_0} G'}{G \Rightarrow_{s_0 \oplus s_1} G'}\ \text{(Choice left)} \qquad \frac{G \Rightarrow_{s_1} G'}{G \Rightarrow_{s_0 \oplus s_1} G'}\ \text{(Choice right)}$$

$$\frac{G \not\models \mathbf{App}(s)}{G \Rightarrow_{s^*} G}\ \text{(Closure false)} \qquad \frac{G \Rightarrow_{s} G'' \quad G'' \Rightarrow_{s^*} G' \quad G \models \mathbf{App}(s)}{G \Rightarrow_{s^*} G'}\ \text{(Closure true)}$$

$$\frac{G \not\models \mathbf{App}(\rho)}{G \Rightarrow_{\rho} \top}\ \text{(Rule False)} \qquad \frac{G \models \mathbf{App}(\rho) \quad G \rightarrow_{\rho} G'}{G \Rightarrow_{\rho} G'}\ \text{(Rule True)}$$

$$\frac{G \not\models \mathbf{App}(\rho)}{G \not\Rightarrow_{\rho!}}\ \text{(Mandatory Rule False)} \qquad \frac{G \models \mathbf{App}(\rho) \quad G \rightarrow_{\rho} G'}{G \Rightarrow_{\rho!} G'}\ \text{(Mandatory Rule True)}$$

$$\frac{G \not\models \mathbf{App}(\rho)}{G \Rightarrow_{\rho?} G}\ \text{(Rule Trial False)} \qquad \frac{G \models \mathbf{App}(\rho) \quad G \rightarrow_{\rho} G'}{G \Rightarrow_{\rho?} G'}\ \text{(Rule Trial True)}$$

Fig. 4. Strategy application rules

Notice that the three strategies using rules (i.e. ρ, $\rho!$ and $\rho?$) behave the same way when $G \models \mathbf{App}(\rho)$ holds, as shown in Fig. 4, but they do differ when $G \not\models \mathbf{App}(\rho)$. In such a case, ρ can yield any graph, denoted by $\top$, (i.e. the process stops without an error), $\rho!$ stops the rewriting process with failure and $\rho?$ ignores the rule application and moves to the next step to be performed, if any, of the considered strategy.

The formula **App** has to be able to express the existence of a match in the considered logic. However, assuming the nodes of the lhs are explicitly named, $L = (\{n_0, \ldots, n_k\}, E, \Phi_N, \Phi_E, s, t)$, for a rule ρ, one may specify the existence of a match in a more direct way when using explicitly one nominal o_i for each node n_i of the lhs. That is to say, one can define a predicate $App(\rho, \{o_0, \ldots, o_k\})$, also noted $App(\rho)$, such that $G \models App(\rho, \{o_0, \ldots, o_k\})$ iff there exists a match h such that $h_N(n_0) = o_0^{\mathcal{G}}, \ldots, h_N(n_k) = o_k^{\mathcal{G}}$. This requires less expressive power to express than **App**. We also define $NApp(\rho) \equiv \neg \mathbf{App}(\rho)$.

Example 5. For the rule ρ_2 of Fig. 3, $App(\rho_2, \{l_0, i_0, l'_0\}) \equiv \exists U.(l_0 \wedge LLIN \wedge \forall ins\_in.\bot \wedge \exists has\_ins.(i_0 \wedge Insecticide \wedge \exists has\_ins^{-}.(l'_0 \wedge LLIN \wedge \neg l \wedge \forall ins\_in.\bot)))$.

5 Verification

In this section, we follow a Hoare style verification technique to specify properties of DLGRS's for which we establish a sound proof procedure.

Definition 10 (Specification). *A* specification *SP is a triple* $\{\mathtt{Pre}\}(\mathtt{R}, \mathtt{s})$ $\{\mathtt{Post}\}$ *where* `Pre` *and* `Post` *are DL formulas,* `R` *is a DLGRSand* `s` *is a strategy.*

Example 6. Continuing with the malaria example, we give a simple example of a specification. We first define a precondition as *every LLIN is installed in at most one House*: $\mathtt{Pre} \equiv \forall U.LLIN \Rightarrow (<2\ ins\_in\ House)$. We can consider a post-condition having the same constraints as the precondition augmented with the fact that *no House is equipped with a LLIN using DDT*: $\mathtt{Post} \equiv (\forall U.LLIN \Rightarrow (<2\ ins\_in\ House)) \wedge (\forall U.House \Rightarrow \forall ins\_in^{-}.\forall has\_ins.\neg DDT)$. To complete the specification example, we consider the DLGRS given Fig. 3 and the strategy as proposed in Example 4.

Definition 11 (Correctness). *A specification SP is said to be* correct *iff for all graphs G, G' such that $G \Rightarrow_s G'$ and $G \models \mathtt{Pre}$, then $G' \models \mathtt{Post}$.*

In order to show the correctness of a specification, we follow a Hoare-calculus style [17] by computing weakest preconditions. For that, we give in Fig. 5 the definition of the function wp which yields the weakest precondition of a formula Q w.r.t. actions and strategies.

The weakest precondition of an elementary action, say a, and a postcondition Q is defined as $wp(a, Q) = Q[a]$ where $Q[a]$ stands for the precondition consisting of Q to which is applied a substitution induced by the action a that we denote by $[a]$. The notion of substitution used here follow the classical ones from Hoare-calculi (e.g., [25]).

Definition 12 (Substitutions). *A* substitution*, written $[a]$, is associated to each elementary action a, such that for all graphs G and DL formula ϕ, $(G \models \phi[a]) \Leftrightarrow (G[a] \models \phi)$.*

When writing a formula of the form $\phi[a]$, the substitution $[a]$ is used as a new formula constructor whose meaning is that the weakest preconditions for elementary actions, as defined above, are correct. DL logics are not endowed with such substitution constructor. The addition of such a substitution constructor to a given description logic is not harmless in general. That is to say, if ϕ is a formula of a DL logic $\mathcal{L}$, $\phi[a]$ is not necessarily a formula of $\mathcal{L}$. Hence, only closed DL logics under substitutions can be used for verification purposes. The two following theorems characterize non trivial fragments of DL logics which are closed, respectively not closed, under substitutions.

Theorem 1. *The description logics $\mathcal{ALCUO}$, $\mathcal{ALCUOI}$, $\mathcal{ALCQUOI}$, $\mathcal{ALCUOS}elf$, $\mathcal{ALCUOIS}elf$, and $\mathcal{ALCQUOIS}elf$ are closed under substitutions.*

The proof of this theorem consists in providing a rewrite system which transforms any formula with substitutions into an equivalent substitution free formula in the considered logic. Details of such a rewrite system can be found in [10].

Theorem 2. *The description logics $\mathcal{ALCQUO}$ and $\mathcal{ALCQUOS}elf$ are not closed under substitutions.*

The proof of the above theorem is not straightforward. It uses notions of bisimulations induced by the considered logics. Two bisimilar models are provided which do not fulfill the same set of formulas. Details of the proof can be found in [10].

$$
\begin{array}{ll}
wp(a,\ Q) = Q[a] & wp(a;\alpha,\ Q) = wp(a, wp(\alpha, Q)) \\
wp(\epsilon,\ Q) = Q & wp(s_0; s_1,\ Q) = wp(s_0, wp(s_1,\ Q)) \\
wp(s_0 \oplus s_1,\ Q) = wp(s_0, Q) \wedge wp(s_1, Q) & wp(s^*,\ Q) = inv_s \\
wp(\rho,\ Q) = App(\rho) \Rightarrow wp(\alpha_\rho, Q) & wp(\rho!, Q) = App(\rho) \wedge wp(\alpha_\rho, Q) \\
wp(\rho?, Q) = (App(\rho) \Rightarrow wp(\alpha_\rho, Q)) \wedge (\neg App(\rho) \Rightarrow Q) &
\end{array}
$$

Fig. 5. Weakest preconditions w.r.t. actions and strategies, where a (resp. α, α_ρ) stands for an elementary action (resp. action, the right-hand side of a rule ρ) and Q is a formula

In presence of DL logics closed under substitutions, the definitions of $wp(s, Q)$ for strategy expressions consisting of the *Empty Strategy*, the *Composition* or the *Choice* operators are quite direct (see, Fig. 5). The definitions of $wp(s, Q)$ when strategy s is a *Rule*, *Mandatory Rule* or *Rule Trial* are not the same depending on what happens if the considered rewrite rule cannot be applied. When a rule ρ can be applied, then applying it should lead to a graph satisfying Q. When the rule ρ cannot be applied, $wp(\rho,\ Q)$ indicates that the considered specification is correct; while $wp(\rho!,\ Q)$ indicates that the specification is not correct and $wp(\rho?,\ Q)$ leaves the postcondition unchanged and thus transformations can move to possible next steps.

As for the computation of weakest preconditions for the *Closure* of strategies, it is close to the *while* statement. It requires an invariant inv_s to be defined, $wp(s^*,\ Q) = inv_s$, which means that the invariant has to be true when entering the iteration for the first time. On the other hand, it is obviously not enough to be sure that Q will be satisfied when exiting the iteration or that the invariant will be maintained throughout the execution. To make sure that iterations behave correctly, we need to introduce some additional *verification conditions* computed by means of a function vc, defined in Fig. 6.

As the computation of wp and vc requires the user to provide invariants, we now introduce the notion of annotated strategies and specification.

$$\begin{array}{|ll|}\hline vc(\epsilon,\ Q) & = vc(\rho,\ Q) = vc(\rho!, Q) = vc(\rho?, Q) = \top \text{ (true)} \\ vc(s_0; s_1,\ Q) & = vc(s_0, wp(s_1,\ Q)) \wedge vc(s_1, Q) \\ vc(s_0 \oplus s_1,\ Q) & = vc(s_0, Q) \wedge vc(s_1, Q) \\ vc(s^*,\ Q) & = vc(s, inv_s) \wedge (inv_s \wedge App(s) \Rightarrow wp(s, inv_s)) \wedge (inv_s \wedge \neg App(s) \Rightarrow Q) \\ \hline \end{array}$$

Fig. 6. Verification conditions for strategies.

Definition 13 (Annotated strategy, Annotated specification). *An* annotated strategy *is a strategy in which every iteration s^* is annotated with an invariant inv_s. It is written $s^*\{inv_s\}$. An* annotated specification *is a specification whose strategy is an annotated strategy.*

Example 7. As the strategy introduced in Example 4 contains two closures, we need to define two invariants. We choose $inv_0 \equiv \texttt{Pre}$ and $inv_2 \equiv \texttt{Post}$. The annotated strategy we use is thus $\mathcal{AS} \equiv \rho_0^*\{inv_0\}; (\rho_1 \oplus (\rho_1; \rho_1)); \rho_2^*\{inv_2\}$.

Definition 14 (Correctness formula). *We call* correctness formula *of an annotated specification $SP = \{\texttt{Pre}\}(\mathcal{R}, \texttt{s})\{\texttt{Post}\}$, the formula : $correct(SP) = (\texttt{Pre} \Rightarrow wp(\texttt{s}, \texttt{Post})) \wedge vc(\texttt{s}, \texttt{Post})$.*

Example 8. The specification of the considered running example is thus $\{\texttt{Pre}\}(\mathcal{R}, \texttt{s})\{\texttt{Post}\}$, where Pre and Post are those introduced in Example 6, the rules $\mathcal{R}$ are those of Example 3 and the annotated strategy s is the strategy $\mathcal{AS}$ as defined in Example 7.

Theorem 3 (Soundness). *Let $SP = \{\texttt{Pre}\}(\texttt{R}, \texttt{s})\{\texttt{Post}\}$ be an annotated specification. If $correct(SP)$ is valid, then for all graphs G, G' such that $G \Rightarrow_s G'$, $G \models \texttt{Pre}$ implies $G' \models \texttt{Post}$.*

The proof is done by structural induction on strategy expressions (see [10]).

Example 9. We now compute the correctness formula of the specification of Example 8: $corr \equiv (\texttt{Pre} \Rightarrow wp(\mathcal{AS}, \texttt{Post})) \wedge vc(\mathcal{AS}, \texttt{Post})$.

By applying the weakest precondition rules, $wp(\mathcal{AS}, \texttt{Post})) \equiv wp(\rho_0^*, wp((\rho_1 \oplus \rho_1; \rho_1); \rho_2^*, \texttt{Post})) \equiv inv_0$. Thus: $corr \equiv (\texttt{Pre} \Rightarrow inv_0) \wedge vc(\mathcal{AS}, \texttt{Post})$.

Let us now focus on $vc(\mathcal{AS}, \texttt{Post})$. For ease of reading, let us write $\mathcal{S}_1 \equiv (\rho_1 \oplus (\rho_1; \rho_1)); \rho_2^*\{inv_2\}$. By applying the rules for the verification conditions, one gets that $vc(\mathcal{AS}, \texttt{Post}) \equiv vc(\rho_0^*, wp(\mathcal{S}_1, \texttt{Post})) \wedge vc(\mathcal{S}_1, \texttt{Post})$. We will discuss the resulting subformulas.

1. We now focus on the first formula, $vc(\rho_0^*, wp(\mathcal{S}_1, \texttt{Post}))$. By applying the rules for the verification conditions, one gets that $vc(\rho_0^*, wp(\mathcal{S}_1, \texttt{Post})) \equiv vc(\rho_0, inv_0) \wedge (inv_0 \wedge App(\rho_0, \{l_0, i_0\}) \Rightarrow wp(\rho_0, inv_0)) \wedge (inv_0 \wedge NApp(\rho_0) \Rightarrow wp(\mathcal{S}_1, \texttt{Post}))$. The rules state that $vc(\rho, Q) = \top$; thus it is possible to get rid of the first formula $vc(\rho_0, inv_0)$.
   Similarly, $wp(\rho_0, inv_0) = App(\rho_0, \{l_0, i_0\}) \Rightarrow inv_0[del_N(l_0)]$.

Altogether: $vc(\rho_0^*, wp(\mathcal{S}_1, \texttt{Post})) \equiv$
$(inv_0 \wedge App(\rho_0, \{l_0, i_0\}) \Rightarrow inv_0[del_N(l_0)]) \wedge (inv_0 \wedge NApp(\rho_0) \Rightarrow wp(\mathcal{S}_1, \texttt{Post}))$.
Let us now focus on the last wp. Unfolding the definition of $wp(\mathcal{S}_1, \texttt{Post}) \equiv wp(\rho_1 \oplus \rho_1; \rho_1, wp(\rho_2^*, \texttt{Post}))$. Applying the rules of weakest preconditions yields $wp(\rho_1, wp(\rho_2^*, \texttt{Post})) \wedge wp(\rho_1, wp(\rho_1, wp(\rho_2^*, \texttt{Post})))$. From the definition of the weakest precondition for a closure, one gets that $wp(\rho_2^*, \texttt{Post}) = inv_2$. By applying the rule for the weakest precondition of a rule application, one gets that:
$wp(\rho_1, inv_2) \equiv App(\rho_1, \{l_1, h_1, i_1, m_1, l_1', i_1', m_1'\}) \Rightarrow inv_2\sigma_1$
and
$wp(\rho_1, wp(\rho_1, inv_2)) \equiv App(\rho_1, \{l_2, h_2, i_2, m_2, l_2', i_2', m_2'\}) \Rightarrow$
$(App(\rho_1, \{l_3, h_3, i_3, m_3, l_3', i_3', m_3'\}) \Rightarrow inv_2\sigma_3)\sigma_2$
where $\sigma_i \equiv [add_E(l_i'', h_i, ins\_in)][del_E(l_i'', h_i, ins\_in)][cl(l_i', l_i'', \overrightarrow{L})]$.

2. Let us now focus on the second formula. Let us apply the verification conditions rules, $vc(\mathcal{S}_1, \texttt{Post}) \equiv vc(\rho_1 \oplus \rho_1; \rho_1, wp(\rho_2^*, \texttt{Post})) \wedge vc(\rho_2^*, \texttt{Post})$. More applications of those rules yield $vc(\rho_1 \oplus \rho_1; \rho_1, wp(\rho_2^*, \texttt{Post})) \equiv \top$.
Thus: $vc(\mathcal{S}_1, \texttt{Post}) \equiv vc(\rho_2^*, \texttt{Post})$
We apply again the rule for closures to $vc(\rho_2^*, \texttt{Post})$ and get $vc(\rho_2, inv_2) \wedge (App(\rho_2, \{l_4, i_4, l_4'\}) \wedge inv_2 \Rightarrow wp(\rho_2, inv_2)) \wedge (inv_2 \wedge NApp(\rho_2) \Rightarrow \texttt{Post})$.
Applying the verification rules yields $vc(\rho_2, inv_2) \equiv \top$.
Applying the weakest precondition to $wp(\rho_2, inv_2)$ yields $App(\rho_2, \{l_4, i_4, l_4'\}) \Rightarrow inv_2[mrg(l_4, l_4')]$ and thus: $vc(\mathcal{S}_1, \texttt{Post}) \equiv$
$(App(\rho_2, \{l_4, i_4, l_4'\}) \wedge inv_2 \Rightarrow inv_2[mrg(l_4, l_4')]) \wedge (inv_2 \wedge NApp(\rho_2) \Rightarrow \texttt{Post})$

To sum up, the correctness formula is $corr \equiv (\texttt{Pre} \Rightarrow inv_0) \wedge$
$(inv_0 \wedge App(\rho_0, \{l_0, i_0\}) \Rightarrow inv_0[del_N(l_0)]) \wedge$
$(inv_0 \wedge NApp(\rho_0) \wedge App(\rho_1, \{l_1, h_1, i_1, m_1, l_1', i_1', m_1'\}) \Rightarrow inv_2\sigma_1) \wedge$
$(inv_0 \wedge NApp(\rho_0) \wedge App(\rho_1, \{l_2, h_2, i_2, m_2, l_2', i_2', m_2'\}) \wedge$
$App(\rho_1, \{l_3, h_3, i_3, m_3, l_3', i_3', m_3'\})\sigma_2 \Rightarrow inv_1\sigma_3\sigma_2) \wedge$
$(inv_2 \wedge App(\rho_2, \{l_4, i_4, l_4'\}) \Rightarrow inv_2[mrg(l_4, l_4')]) \wedge (inv_2 \wedge NApp(\rho_2) \Rightarrow \texttt{Post})$.

The intermediate lines are the result of (1) and the last line is the result of (2). Furthermore,

- $\texttt{Pre}$, $\texttt{Post}$, inv_0 and inv_2 are defined in Examples 6 and 7;
- $App(\rho_0, \{l_0, i_0\}) \equiv \exists U.(l_0 \wedge LLIN \wedge \exists ins\_in.\top \wedge \exists has\_ins.(i_0 \wedge DDT))$;
- $NApp(\rho_0) \equiv \forall U.(\neg LLIN \vee \forall ins\_in.\bot \vee \forall has\_ins.\neg DDT)$;
- $App(\rho_1, \{l_i, h_i, i_i, m_i, l_i', i_i', m_i'\}) \equiv \exists U.(l_i \wedge LLIN \wedge \exists ins\_in.(h_i \wedge House) \wedge \exists has\_ins.(i_i \wedge Insecticide \wedge \exists has\_moa.(m_i \wedge ModeOfAction))) \wedge \exists U.(l_i' \wedge \exists has\_ins.(i_i' \wedge \exists has\_moa.(m_i' \wedge ModeOfAction \wedge \neg m_i)))$;
- $App(\rho_2, \{l_4, i_4, l_4'\}) \equiv \exists U.(l_4 \wedge LLIN \wedge \forall ins\_in.\bot \wedge \exists has\_ins.(i_4 \wedge Insecticide \wedge \exists has\_ins^-.(l_4' \wedge LLIN \wedge \forall ins\_in.\bot \wedge \neg l_4)))$ and
- $NApp(\rho_2) \equiv \forall U.Insecticide \Rightarrow (<\ 2\ has\_ins^-\ (LLIN \wedge \forall ins\_in.\bot))$.

6 Conclusion

We have presented a class of graph rewrite systems, DLGRSs, where the lhs's of the rules can express additional application conditions defined as DL logic formulas and rhs's are sequences of actions. The considered actions include node merging and cloning, node and edge addition and deletion among others. We defined computations with these systems by means of rewrite strategies. There is certainly much work to be done around such systems with logically decorated lhs's. For instance, the extension to narrowing derivations would use an involved unification algorithm taking into account the underlying DL logic. We have also presented a sound Hoare-like calculus and shown that the considered verification problem is still decidable with a large class of DL logics. Meanwhile, we pointed out two rich DL logics which are not closed under substitutions and thus cannot be candidate for verification issues. Future work includes an implementation of the proposed verification technique (work in progress) as well as the investigation of more expressive logics with connections to some SMT solvers.

References

1. Ahmetaj, S., Calvanese, D., Ortiz, M., Simkus, M.: Managing change in graph-structured data using description logics. In: Proceedings of 28th AAAI Conference on Artificial Intelligence (AAAI 2014), pp. 966–973. AAAI Press (2014)
2. Antoniou, G., van Harmelen, F.: Web ontology language: owl. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies in Information Systems, pp. 67–92. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-24750-0_4
3. Asztalos, M., Lengyel, L., Levendovszky, T.: Formal specification and analysis of functional properties of graph rewriting-based model transformation. Softw. Test. Verif. Reliabil. **23**(5), 405–435 (2013)
4. Baader, F.: Description logic terminology. In: The Description Logic Handbook: Theory, Implementation, and Applications, pp. 485–495 (2003)
5. Balbiani, P., Echahed, R., Herzig, A.: A dynamic logic for termgraph rewriting. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 59–74. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2_5
6. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Inf. Comput. **206**(7), 869–907 (2008)
7. Baresi, L., Spoletini, P.: On the use of alloy to analyze graph transformation systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 306–320. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_22
8. Brenas, J.H., Echahed, R., Strecker, M.: A hoare-like calculus using the SROIQσ logic on transformations of graphs. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) TCS 2014. LNCS, vol. 8705, pp. 164–178. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44602-7_14
9. Brenas, J.H., Echahed, R., Strecker, M.: Proving correctness of logically decorated graph rewriting systems. In: 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, pp. 14:1–14:15 (2016)
10. Brenas, J.H., Echahed, R., Strecker, M.: On the verification of logically decorated graph transformations. CoRR, abs/1803.02776 (2018)
11. Corradini, A., Duval, D., Echahed, R., Prost, F., Ribeiro, L.: AGREE – algebraic graph rewriting with controlled embedding. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 35–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_3

12. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_4
13. Courcelle, B.: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. Inf. Comput. **85**(1), 12–75 (1990)
14. Echahed, R.: Inductively sequential term-graph rewrite systems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 84–98. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_7
15. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. STTT **14**(1), 15–40 (2012)
16. Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
18. Jackson, D.: Software Abstractions. MIT Press, Cambridge (2012)
19. König, B., Esparza, J.: Verification of graph transformation systems with context-free specifications. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 107–122. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2_8
20. Plump, D.: The graph programming language GP. In: Bozapalidis, S., Rahonis, G. (eds.) CAI 2009. LNCS, vol. 5725, pp. 99–122. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03564-7_6
21. Poskitt, C.M., Plump, D.: A Hoare calculus for graph programs. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 139–154. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2_10
22. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 33–48. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09108-2_3
23. Rensink, A., Distefano, D.: Abstract graph transformation. Electron. Notes Theoret. Comput. Sci. **157**(1), 39–59 (2006). (In: Proceedings of 3rd International Workshop on Software Verification and Validation (SVV 2005))
24. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: a comparison of two approaches. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 226–241. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_17
25. Virga, R.: Efficient substitution in Hoare logic expressions. Electr. Notes Theor. Comput. Sci. **41**(3), 35–49 (2000)
26. Visser, E.: Stratego: a language for program transformation based on rewriting strategies system description of stratego 0.5. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45127-7_27

OCL2AC: Automatic Translation of OCL Constraints to Graph Constraints and Application Conditions for Transformation Rules

Nebras Nassar[1(✉)], Jens Kosiol[1], Thorsten Arendt[2], and Gabriele Taentzer[1]

[1] Philipps-Universität Marburg, Marburg, Germany
{nassarn,kosiolje,taentzer}@informatik.uni-marburg.de
[2] GFFT Innovationsförderung GmbH, Bad Vilbel, Germany
thorsten.arendt@gfft-ev.de

Abstract. Based on an existing theory, we present a tool *OCL2AC* which is able to adapt a given rule-based model transformation such that resulting models guarantee a given constraint set. *OCL2AC* has two main functionalities: First, OCL constraints are translated into semantically equivalent graph constraints. Secondly, graph constraints can further be integrated as application conditions into transformation rules. The resulting rule is applicable only if its application does not violate the original constraints. *OCL2AC* is implemented as Eclipse plug-in and enhances Henshin transformation rules.

Keywords: OCL · Nested graph constraints · Model transformation Henshin

1 Introduction

Model transformations are the heart and soul of Model-Driven Engineering (MDE). They are used for various MDE-activities including translation, optimization, and synchronization of models [13]. Resulting models should belong to the transformation's target language which means that they have to satisfy all the corresponding language constraints. Consequently, the developer has to design transformations such that they behave well w.r.t. language constraints.

Based on existing theory [8, 12], we developed a tool, called *OCL2AC*, which automatically adapts a given rule-based model transformation such that resulting models satisfy a given set of constraints. Use-cases for this tool are abundant, including instance generation [12], ensuring that refactored models do not show certain model smells (anymore), and generating model editing rules from meta-models to enable high-level model version management [9].

Our tool builds upon the following basis: The de facto standard for defining modeling languages in practice are the Eclipse Modeling Framework (EMF) [5] for specifying meta-models and the Object Constraint Language (OCL) [10] for expressing additional constraints. Graph transformation [6] has been shown to be a versatile foundation for rule-based model transformation [7] focusing on the models' underlying graph structure. To reason about graph properties, Habel and Pennemann [8] have developed (nested) graph constraints being equivalent to first-order formulas on graphs.

OCL2AC consists of two main components: The first component *OCL2GC* translates a reasonable subset of OCL constraints to graph constraints using the formally defined OCL translation in [12] as conceptual basis. The second component *GC2AC* integrates graph constraints as application conditions into transformation rules specified in Henshin, a language and tool environment for EMF model transformation [2]. The resulting application conditions ensure that EMF models resulting from applications of the enhanced rules do not violate the original constraints. The rules' actions are not changed by the integration. Each of these two components is designed to be usable on its own.

Note that our OCL translator is novel: Instead of checking satisfiability, it enhances transformation rules such that their applications can result in valid models only. To that extent, *OCL2AC* can be used to not just check constraint satisfaction but also to tell the user how to improve transformation rules.

The paper is structured as follows: In Sect. 2, we present preliminaries. Section 3 presents the two main components of the tool and their internal functionalities. Related work is given in Sect. 4 while Sect. 5 concludes the paper.

2 Preliminaries

2.1 Introductory Example

To illustrate the behaviour of our tool, we use a simple Statecharts meta-model displayed in Fig. 1.

A StateMachine contains at least one Region which may potentially contain Transitions and Vertices. Vertex is an abstract class with a concrete subclass State. FinalState inherits from State. A State can again contain Regions. Transitions connect Vertices.

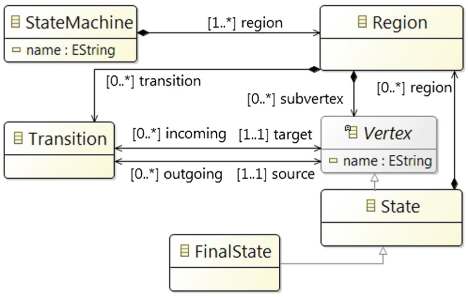


Fig. 1. A simple Statecharts meta-model

A basic constraint on Statecharts which is not expressible by just the graphical structure of the meta-model or by multiplicities is: A FinalState has no outgoing transition. We name this constraint no_outgoing_transitions.

2.2 OCL

The Object Constraint Language (OCL) [10] is a constraint language used to supplement the specification of object-oriented models. OCL constraints may be used to specify invariants, operation contracts, or queries. The constraint no_outgoing_transitions can be specified in OCL as:

```
context FinalState invariant no_outgoing_transitions:
  self.outgoing->isEmpty();
```

Our technique supports a slightly restricted subset of Essential OCL [10]. Since OCL constraints are translated to nested graph constraints and thereby get a precise semantics, we focus on OCL constraints corresponding to a first-order, two-valued logic and relying on sets as the only collection type. Also, there is only limited support for user-defined operations. Details can be found in [1,12].

2.3 Graph Rules, Graph Conditions, and Graph Constraints

Our tool currently enhances Henshin rules [2]. A rule specifies elements as to be deleted, created, or preserved at application. Additionally, it may be equipped with an *application condition* controlling its applicability. Figure 2 shows a Henshin rule insert_outgoing_transition. When applying it at chosen nodes Vertex and Transition in an instance, an edge of type outgoing is created between them.

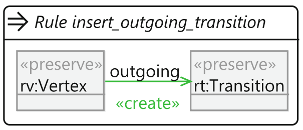


Fig. 2. Transformation rule

$$\forall\left(\boxed{\text{self:FinalState}},\ \nexists\ \boxed{\text{self:FinalState}} \xrightarrow{\text{outgoing}} \boxed{\text{var9:Transition}}\right)$$

Fig. 3. Graph constraint no_outgoing_transitions

(Nested) graph constraints are invariants which are checked for all graphs, whereas (nested) graph conditions express properties of morphisms. The primary example for that are application conditions for transformation rules. Constraints are special cases of conditions since the empty graph can be included into any graph. A version, important from the practical point of view, are *compact constraints and conditions* [12]. They allow for dense representation and obtain their semantics by *completing* them into nested constraints or conditions. All those formalisms allow to express first-order properties [8]. As an example, the graph constraint in Fig. 3 states that a FinalState does not have an outgoing transition.

3 Tooling and Architecture

We implemented an Eclipse plug-in, called *OCL2AC*, with two components: (1) *OCL2GC* takes an Ecore meta-model and a set of OCL constraints as inputs and automatically returns a set of semantically equivalent graph constraints as output. (2) *GC2AC* takes a transformation rule defined in Henshin and a graph constraint, possibly compact, as inputs, and automatically returns the Henshin rule with an updated application condition guaranteeing the given graph constraint. Each component can be used independently as an Eclipse-based tool. The tool is available for download at [1]. We introduce the architecture and internal processes of both components and highlight some additional features.

3.1 From OCL to Graph Constraints

The first component of our tool takes an Ecore meta-model and a set of OCL constraints as inputs and returns a set of semantically equivalent (nested) graph constraints as output. The translation process is composed of the steps shown in Fig. 4, which can be automatically performed.

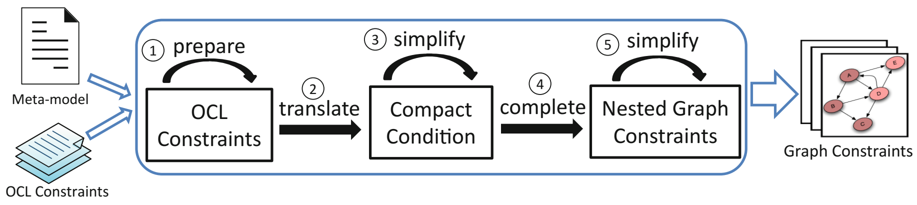


Fig. 4. From OCL to graph constraints: component design

In Step (1) an OCL constraint is prepared by refactorings. This is done to ease the translation process, especially to save translation rules for OCL constraints. The semantics of the constraint is preserved during this preparation. Step (2) translates an OCL constraint to a graph constraint along the abstract syntax structure of OCL expressions. This translation largely follows the one in [12]. Let us consider the translation of OCL constraint no_outgoing_transitions to the graph constraint displayed in Fig. 3: The expression self.outgoing→isEmpty() is refactored to not(self.outgoing→size() $\geq$ 1). Hence, a translation rule for isEmpty() is not needed. Then, this sub-expression is translated to a compact condition containing a graph with one edge of type outgoing from a node of type FinalState to a node of type Transition. The existence of such a pattern is negated.

Then a first simplification of the resulting compact condition takes place in Step (3), using equivalence rules [11,12]. Applying those can greatly simplify the representation of a condition; they can even collapse nesting levels.

Step (4) completes the compact condition to a nested graph constraint being used to compute application conditions later on. The resulting nested graph constraint is simplified in Step (5) using again equivalence rules. Our tool allows to display the resulting constraint as nested graph constraint or as compact constraint. The intermediate steps are computed internally only.

3.2 From Graph Constraints to Left Application Conditions

The second component of our tool takes a Henshin rule and a graph constraint as inputs, and returns the Henshin rule with an updated application condition guaranteeing the given graph constraint. Figure 5 gives an overview of the steps to be performed:

In Step (1) the given graph constraint is prepared; if the input is a compact constraint, it is expanded to a nested constraint. Moreover, it is refactored

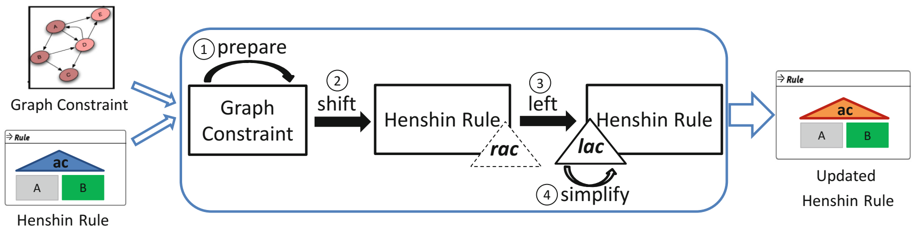


Fig. 5. Integration as application conditions: component design

to eliminate syntactic sugar. The operator ⇒ for implication, for example, is replaced with basic logic operators. Step (2) shifts a given graph constraint *gc* to the RHS of the given rule so that we get a new right application condition *rac* for this rule. The main idea of shift is to consider all possible ways in which *gc* could be satisfied after rule application. This is done by overlapping the elements of the rule's RHS with each graph of *gc* in every possible way. This overlapping is done iteratively along the nesting structure of *gc*. This algorithm is formally defined in [8] and shown to be correct. The result of this calculation is yet impractical as one would need to first apply the rule and then check the right application condition to be fulfilled. Therefore, we continue with the next step.

Step (3) translates a right application condition *rac* to the LHS of the given rule *r* to get a new left application condition *lac*. It is translated by applying the rule reversely to the right application condition *rac*, again along its nesting structure. If the inverse rule of *r* is applicable, the resulting condition is the new left application condition. Otherwise *r* gets equipped with the application condition `false`, as it is not possible to apply *r* at any instance without violating *gc*. The rule *r* with its new application condition has the property that, if it is applicable, the resulting instance fulfills the integrated constraint. Step (4) simplifies the resulting left application condition using equivalences as for graph conditions. The output is the original Henshin rule with an updated left application condition guaranteeing the given graph constraint.

For example, integrating the constraint no_outgoing_transitions into the rule insert_outgoing_transition results in the application condition displayed in Fig. 6. The upper part forbids node rv:Vertex being matched to a FinalState. The lower part requires that the rule is matched to consistent models only, i.e., not containing already a FinalState with an outgoing Transition. It may be omitted if consistent input models can always be assumed.

Tool Features. *OCL2AC* provides a wizard for selecting a rule and a graph constraint that shall be integrated. The inputs of the wizard are a Henshin model (file) and a graph constraint model being generated by *OCL2GC* or manually designed based on the meta-model for compact conditions (at [1]). *OCL2AC* additionally provides tool support for *pretty printing graph constraints and application conditions* of Henshin rules in a graphical form as shown in Figs. 3 and 6

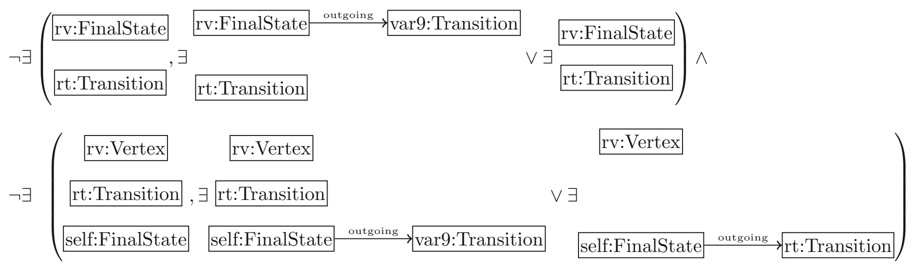


Fig. 6. Application condition for the rule insert_outgoing_transition after integrating the constraint no_outgoing_transitions

above. Pretty printing is supported for both compact and detailed representation of nested constraints and application conditions. The output of the pretty printing is a LaTeX-file being rendered as pdf-file and displayed in a developed Eclipse PDF viewer.

4 Related Work

We briefly compare our tool with the most related tools for translating OCL or calculating application conditions. To the best of our knowledge, we present the first ready-to-use tool for integrating constraints as application conditions into transformation rules.

In [3], Bergmann proposes a translation of OCL constraints into graph patterns. The correctness of that translation is not shown. The implementation covers most of that translation. In [11], Pennemann introduces ENFORCe which can check and ensure the correctness of high-level graph programs. It integrates graph constraints as left application conditions of rules as well. However, the tool is not published to try that out. Furthermore, there is no translation from OCL to graph constraints available. Clarisó et al. present in [4] how to calculate an application condition for a rule and an OCL constraint, directly in OCL. They provide a correctness proof and a partial implementation.

5 Conclusion

OCL2AC automatically translates OCL constraints into semantically equivalent graph constraints and thereafter, it takes a graph constraint and a Henshin rule as inputs and updates the application condition of that rule such that it becomes constraint-guaranteeing. *OCL2AC* is a ready-to-use tool implemented as Eclipse plug-in based on EMF and Henshin. As future works, we intend to use it for improving transformation rules for various modeling purposes such as model validation and repair.

Acknowledgement. We are grateful to Jan Steffen Becker, Annegret Habel and Christian Sandmann for their helpful support. This work was partially funded by the German Research Foundation (DFG), projects "Generating Development Environments for Modeling Languages" and "Triple Graph Grammars (TGG) 2.0".

References

1. OCL2AC: Additional material (2018). https://ocl2ac.github.io/home/
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010 Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
3. Bergmann, G.: Translating OCL to graph patterns. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 670–686. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_41
4. Clarisó, R., Cabot, J., Guerra, E., de Lara, J.: Backwards reasoning for model transformations: method and applications. J. Syst. Softw. **116**(Suppl. C), 113–132 (2016). https://doi.org/10.1016/j.jss.2015.08.017
5. Eclipse Foundation: Eclipse Modeling Framework (EMF) (2018). http://www.eclipse.org/emf/
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
7. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation - General Framework and Applications. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47980-3
8. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**, 245–296 (2009)
9. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 173–188. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42064-6_12
10. OMG: Object Constraint Language. http://www.omg.org/spec/OCL/
11. Pennemann, K.H.: Development of correct graph transformation systems. Ph.D. thesis, Carl von Ossietzky-Universität Oldenburg (2009)
12. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. Sci. Comput. Program. **152**, 38–62 (2018)
13. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)

Author Index