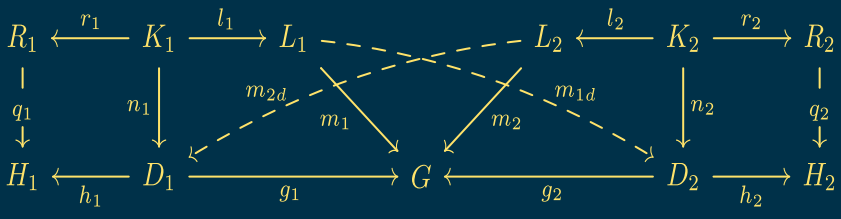
Festschrift

LNCS 10800

Reiko Heckel
Gabriele Taentzer (Eds.)

Graph Transformation, Specifications, and Nets

In Memory of Hartmut Ehrig



Springer

Lecture Notes in Computer Science 10800

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

More information about this series at http://www.springer.com/series/7407

Reiko Heckel · Gabriele Taentzer (Eds.)

Graph Transformation, Specifications, and Nets

In Memory of Hartmut Ehrig

Springer

Editors
Reiko Heckel
Department of Informatics
University of Leicester
Leicester
UK

Gabriele Taentzer
Fachbereich Mathematik und Informatik
Philipps-Universität Marburg
Marburg
Germany

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-75395-9 ISBN 978-3-319-75396-6 (eBook)
https://doi.org/10.1007/978-3-319-75396-6

Library of Congress Control Number: 2018931888

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

This volume is dedicated to the memory of Hartmut Ehrig, who passed away on March 17, 2016, at the age of 71. Hartmut was my special friend and collaborator, and so I felt honored when I was asked to write the foreword to this volume.

Hartmut was a great scientist who substantially influenced developments of several research areas. His mathematical tools, the methods he used, and the models that he created were mostly rooted in category theory and algebra.

He is in fact the father of the algebraic approach to graph transformation (graph grammars, graph rewriting). He was a co-founder of the double-pushout approach, which laid the foundations for this research area and, moreover, most of the crucial developments of the algebraic approach were either directly or indirectly influenced by him. He was also responsible for some of the most significant developments in the area of algebraic specifications. His books on algebraic specifications became the main references for both researchers and students.

A characteristic feature of Hartmut's research was the fact that, although he used and pursued very abstract (for computer science) tools and techniques, such as category theory, he was genuinely interested in and motivated by applications, especially those related to software development.

His contributions to science were not restricted to his huge scientific output. He was also very involved in developing the organizational framework for the computer science community. Here are some examples:

- He was a co-founder of the International Conference on Graph Transformations (which originated as the International Workshop on Graph Grammars and Their Applications to Computer Science).
- Because of his strong interest in connecting theoretical research with applications, he organized the first TAPSOFT conference, and was behind transforming TAPSOFT into the very prestigious European Joint Conference on Theory and Practice of Software (ETAPS).
- He also played an important role in the development of the European Association of Theoretical Computer Science (EATCS) — he was its vice president from 1997 until 2002.

Hartmut was a passionate scientist always involved in either solving technical problems or in inventing new, often pioneering, research directions. His passion and enthusiasm were contagious — working with Hartmut was always an intense experience.

I was fortunate to have Hartmut as my friend and collaborator for over 40 years. I surely miss him — I have many fond memories of the times we spent together. I am really pleased to see how his scientific ideas are still pursued today.

November 2017 Grzegorz Rozenberg

Preface

In October 2016 we held a symposium at TU Berlin commemorating the life and work of Hartmut Ehrig. This book pays tribute to Hartmut's scientific achievements. It contains contributions based on the presentations at the symposium as well as other invited papers in the areas that Hartmut was active in.

These areas include:

- Graph transformation
- Model transformation
- Concurrency theory, in particular Petri nets
- Algebraic specification
- Category theory in computer science

The editors would like to thank all authors and reviewers for their thorough and timely work, Grzegorz Rozenberg for his advice throughout the process, and Alfred Hofmann at Springer for his encouragement and support for this project.

November 2017

Gabriele Taentzer
Reiko Heckel

Organization

Program Committee

Paolo Bottoni	Sapienza University of Rome, Italy
Andrea Corradini	Dipartimento di Informatica, Università di Pisa, Italy
Zinovy Diskin	McMaster University/University of Waterloo, Canada
Dominique Duval	LJK, University of Grenoble, France
Fabio Gadducci	Università di Pisa, Italy
Annegret Habel	University of Oldenburg, Germany
Reiko Heckel	University of Leicester, UK
Berthold Hoffmann	Universität Bremen, Germany
Dirk Janssens	University of Antwerp, Belgium
Timo Kehrer	Humboldt-Universität zu Berlin, Germany
Alexander Knapp	Universität Augsburg, Germany
Hans-Joerg Kreowski	University of Bremen, Germany
Sabine Kuske	University of Bremen, Germany
Barbara König	Universität Duisburg-Essen, Germany
Harald König	FHDW Hannover, Germany
Leen Lambers	Hasso-Plattner-Institut, Universität Potsdam, Germany
Michael Löwe	FHDW Hannover, Germany
Ugo Montanari	Università di Pisa, Italy
Till Mossakowski	University of Magdeburg, Germany
Fernando Orejas	Universitat Politècnica de Catalunya, Spain
Julia Padberg	HAW Hamburg, Germany
Francesco Parisi-Presicce	Sapienza University of Rome, Italy
Detlef Plump	University of York, UK
Arend Rensink	University of Twente, The Netherlands
Leila Ribeiro	Universidade Federal do Rio Grande do Sul, Brazil
Donald Sannella	The University of Edinburgh, UK
Andy Schürr	TU Darmstadt, Germany
Pawel Sobocinski	University of Southampton, UK
Daniel Strüber	University of Koblenz and Landau, Germany
Gabriele Taentzer	Philipps-Universität Marburg, Germany
Andrzej Tarlecki	Institute of Informatics, Warsaw University, Poland
Daniel Varro	Budapest University of Technology and Economics, Hungary
Uwe Wolter	University of Bergen, Norway

Contents

On the Essence of Parallel Independence for the Double-Pushout and Sesqui-Pushout Approaches

Andrea Corradini[1(✉)], Dominique Duval[2], Michael Löwe[3], Leila Ribeiro[4], Rodrigo Machado[4(✉)], Andrei Costa[4], Guilherme Grochau Azzi[4], Jonas Santos Bezerra[4], and Leonardo Marques Rodrigues[4]

[1] Università di Pisa, Pisa, Italy
andrea@di.unipi.it
[2] Université Grenoble-Alpe, Grenoble, France
dominique.duval@imag.fr
[3] Fachhochschule für die Wirtschaft Hannover, Hannover, Germany
michael.loewe@fhdw.de
[4] Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{leila,rma,acosta,ggazzi,jsbezerra,lmrodrigues}@inf.ufrgs.br

Abstract. Parallel independence between transformation steps is a basic notion in the algebraic approaches to graph transformation, which is at the core of some static analysis techniques like Critical Pair Analysis. We propose a new categorical condition of parallel independence and show its equivalence with two other conditions proposed in the literature, for both left-linear and non-left-linear rules. Next we present some preliminary experimental results aimed at comparing the three conditions with respect to computational efficiency. To this aim, we implemented the three conditions, for left-linear rules only, in the Verigraph system, and used them to check parallel independence of pairs of overlapping redexes generated from some sample graph transformation systems over categories of typed graphs.

1 Introduction

Graph transformation is a well-developed computational model suited to describe the evolution of distributed systems. System states are represented by graphs, and rules typically describe local changes of some part of the state. One central topic in the theory of graph transformation, first addressed in [12,21], has been the identification of conditions that guarantee that two transformation steps from a given state are independent, and thus can be applied in any order generating the same result. This is known as the *Local Church-Rosser Problem*, that is presented in the following way in [8]:

> Find a condition, called *parallel independence*, such that two alternative transformation steps $H_1 \overset{\rho_1}{\Longleftarrow} G \overset{\rho_2}{\Longrightarrow} H_2$ are parallel independent if and only if there are transformation steps $H_1 \overset{\rho_2}{\Longrightarrow} X$ and $H_2 \overset{\rho_1}{\Longrightarrow} X$ such that $G \overset{\rho_1}{\Longrightarrow} H_1 \overset{\rho_2}{\Longrightarrow} X$ and $G \overset{\rho_2}{\Longrightarrow} H_2 \overset{\rho_1}{\Longrightarrow} X$ are equivalent.

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 1–18, 2018.
https://doi.org/10.1007/978-3-319-75396-6_1

The "equivalence" just mentioned informally means that the rules ρ_1 and ρ_2 consume the same items of G in the two transformation sequences. A formal definition, based on the classical *shift equivalence*, can be found in Sect. 3.5 of [8]. The above statement fixes a standard pattern for addressing the Local Church-Rosser Problem in the various approaches to algebraic graph transformation: first, a definition of *parallel independence* for transformation steps has to be provided, next a *Local Church-Rosser Theorem* proves that given two parallel independent transformation steps from a given graph, they can be applied in both orders obtaining the same result (up to isomorphism).

The efficient verification of parallel independence is important for the analysis of graph transformation systems. It is needed for example in *Critical Pair Analysis*, a static analysis technique originally introduced in term rewriting systems [15] and, starting with [20], widely used also in graph transformation and supported by some tools [9,22]. It relies on the generation of all possible *critical pairs*, i.e. pairs of transformation steps in minimal context that are not parallel independent, which can be used to prove local confluence or to provide the modeler with all possible conflicts between transformation rules. Efficient parallel independence verification could also be exploited by partial-order reduction techniques in tools supporting model checking of graph transformation systems, like GROOVE [14].

In the first part of the paper we discuss three definitions of parallel independence proposed for the classical Double-Pushout Approach (DPO) [11], and also applicable to the richer setting of the Sesqui-Pushout Approach (SQPO) [7], which extends DPO by allowing also the specification of cloning or copying of items. The third of such definitions is new, and we claim that it captures the essence of parallel independence, being simpler than the other ones. We exploit the third condition as a pivot in proving that all presented conditions are equivalent.

In the second part of the paper we report on some experimental evaluations of the complexity of verifying parallel independence according with the three conditions. They have been implemented in Verigraph, a framework for the specification and verification of graph transformation systems written in Haskell, under development at the Universidade Federal do Rio Grande do Sul [9]. Since the current version of Verigraph does not support Sesqui-Pushout transformation, only left-linear rules are considered in the evaluation. After describing the basic data structures used in Verigraph to model categorical constructions, we discuss the implementation of the three equivalent conditions and compare the time efficiency of verifying them on a collection of test cases. The newly proposed condition turns out to be the in most cases the most efficient.

The reader is assumed to be familiar with the DPO approach and with typed graphs [8]. Some background notions are introduced in Sect. 2. In Sect. 3 we introduce the three conditions for parallel independence, and Sect. 4 is devoted to the proof of their equivalence. The Verigraph system is presented in Sect. 5, which also describes how the parallel independence conditions were implemented. Experimental results are described and analysed in Sect. 6. Finally, Sect. 7 presents concluding remarks.

2 Background

In order to fix the terminology, let us recall the standard definition of Double-Pushout [11] and Sesqui-Pushout transformation [7] in a generic category $\mathbf{C}$. Conditions on $\mathbf{C}$ will be introduced when needed.

Definition 1 (Double-Pushout transformation). *A* rule $\rho = (L \xleftarrow{l} K \xrightarrow{r} R)$ *is a span of arrows in* $\mathbf{C}$. *Rule* ρ *is* left-linear *if* l *is mono,* right-linear *if* r *is mono, and* linear *if both* l *and* r *are monos. A* match *for rule* ρ *in an object* G *is an arrow* $m : L \to G$. *If the diagram in (1) exists in* $\mathbf{C}$, *where both squares are pushouts, then we say that there is a* DPO transformation step *from* G *to* H *via* (ρ, m), *and we write* $G \overset{\rho,m}{\Longrightarrow} H$. *In this case we call the pair* (ρ, m) *a* redex *in* G, *and* $K \xrightarrow{n} D \xrightarrow{g} G$ *a* pushout complement (POC) *of* $K \xrightarrow{l} L \xrightarrow{m} G$. *We write* $G \overset{\rho}{\Longrightarrow} H$ *if* $G \overset{\rho,m}{\Longrightarrow} H$ *for some match* m *for* ρ *in* G.

$$\begin{array}{ccccc} L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle n} & & \downarrow{\scriptstyle q} \\ G & \xleftarrow{\ g\ } & D & \xrightarrow{\ h\ } & H \end{array} \tag{1}$$

Therefore if (ρ, m) is a redex in G we know that ρ can be applied to match m in G. In Diagram (1), K is called the *interface* and D the *context*.

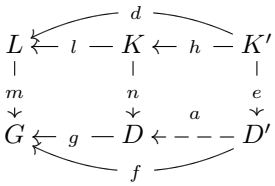
In most presentations of the DPO approach, suitable conditions are imposed to guarantee a form of determinism, i.e. that if $G \overset{\rho,m}{\Longrightarrow} H$ and $G \overset{\rho,m}{\Longrightarrow} H'$ then H and H' are isomorphic. This is often achieved by requiring rules to be left-linear, because in several categories of interest (as in *adhesive categories* [16]) if l is mono then the pushout complement of $K \xrightarrow{l} L \xrightarrow{m} G$ is uniquely determined (up to isomorphism) if it exists, and thus also object H is unique up to iso by the universal property of the right-hand side pushout.[1]

Sesqui-Pushout transformations were proposed in [7] as a little variation of DPO ones, able to guarantee the above form of determinism by the very definition. The only difference with respect to DPO is the property that the left square of (1) has to satisfy. We first recall the definition of final pullback complement.

[1] In non-adhesive categories stronger conditions might be necessary. For example, in the category of term graphs (which is not adhesive but just *quasi-adhesive* [6]), two non-isomorphic pushout complements may exist for a monic l, while uniqueness is ensured by requiring l to be a *regular mono*, i.e. the equalizer of a pair of parallel arrows. It is worth recalling here that every adhesive category is quasi-adhesive, and that in an adhesive category all monos are regular [16].

Definition 2 (final pullback complement) *In Diagram (2), $K \overset{n}{\to} D \overset{g}{\to} G$ is a* final pullback complement (FPBC) *of $K \overset{l}{\to} L \overset{m}{\to} G$ if*

1. *the resulting square is a pullback, and*
2. *for each pullback $G \overset{m}{\leftarrow} L \overset{d}{\leftarrow} K' \overset{e}{\to} D' \overset{f}{\to} G$ and arrow $K' \overset{h}{\to} K$ such that $l \circ h = d$, there is a unique arrow $D' \overset{a}{\to} D$ such that $g \circ a = f$ and $a \circ e = n \circ h$.*



(2)

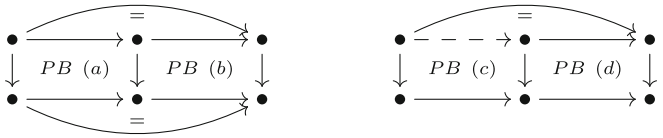
Definition 3 (Sesqui-Pushout transformation). *Under the premises of Definition 1, we say that there is a* SQPO transformation step *from G to H via (ρ, m) if the diagram in (1) can be constructed in $\mathbf{C}$, where the left square is a final pullback complement of $K \overset{l}{\to} L \overset{m}{\to} G$ and the right square is a pushout.*

The final pullback complement of two arrows is characterised by a universal property, and thus it is unique up to isomorphism, if it exists. Therefore SQPO transformation is deterministic in the above sense also for non-left-linear rules. Furthermore, as discussed in [7], in an adhesive category every DPO transformation for a left-linear rule is also an SQPO transformation, and thus SQPO rewriting can be considered as a conservative extension of DPO transformation.

Along the paper we shall often use some well-known properties of pullbacks:

Fact 1 (composition and decomposition of pullbacks)

1. *In the diagram on the left if squares (a) and (b) are pullbacks, so is the composed square:*



2. *In the diagram made of solid arrows above on the right, if square (d) and the outer square are pullbacks, then there is a unique arrow (the dotted one) such that the top triangle commutes and square (c) is a pullback.*

The following definition will be useful in the following.

Definition 4 (reflection). *Given objects X, Y, Z and arrows $f : X \to Z$, $g : Y \to Z$ we say that f* is reflected along *g if the pullback object of $Y \overset{g}{\to} Z \overset{f}{\leftarrow} X$ is isomorphic to X, as in square ① or, equivalently, if there is an arrow $h : X \to Y$ such that $Y \overset{h}{\leftarrow} X \overset{id}{\to} X$ is the pullback of $Y \overset{g}{\to} Z \overset{f}{\leftarrow} X$, as in square ②.*

Note that such an arrow $h : X \to Y$, if it exists, is necessarily unique. In this case we also say that f is reflected along *g by h.*

Intuitively, this means that g is an isomorphism when restricted to the image of f. If objects are concrete structures like graphs, then every item of the image of f in Z has exactly one inverse image along g in Y. The following facts are easy to check by properties of pullbacks.

Fact 2 (some properties of reflection)

1. *Arrow $f : X \to Z$ is reflected along $g : Y \to Z$ iff there exists an arrow $h : X \to Y$ such that for all pairs of arrows $m : W \to Y$ and $n : W \to X$, if $g \circ m = f \circ n$ then $h \circ n = m$.*
2. *If g is mono, then f is reflected along g iff there exists an arrow $h : X \to Y$ such that $f = g \circ h$.*

We will use as running example the following SQPO graph grammar, based on the category of graphs typed over the left graph of Fig. 1.

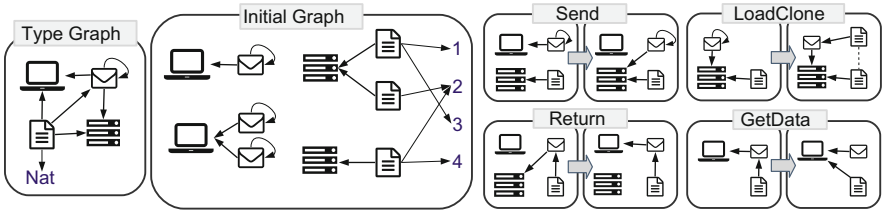


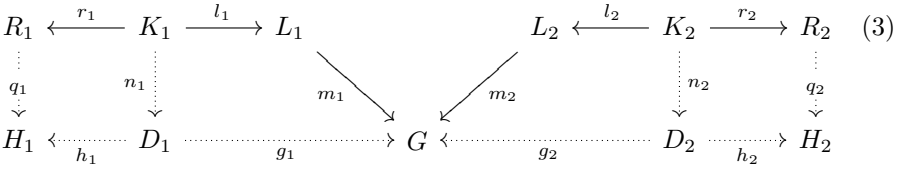
Fig. 1. Graph grammar for clients and servers

Example 1 (SQPO graph grammar for clients and servers). The graph grammar of Fig. 1 represents clients (computers) obtaining data (documents) from servers via messages (envelopes). In the model, data nodes represent subsets of $\{1, 2, 3, 4\}$ (elements of type Nat), which are initially stored in servers. Arrows represent locations and references. Loops in messages are used to ensure that only one data node is loaded at each time. Clients transmit messages to servers via rule Send. Messages are loaded with a cloned version of a data node via rule LoadClone, with a dotted line being used to represent the cloning effect. It is worth recalling that all edges from the cloned data node to numbers are also cloned as a side-effect of SQPO rewriting. Rule Return transmits messages with data back to clients. Finally, rule GetData deletes a message, placing the data node directly into the client node. Rules are presented by showing only the left- and right-hand sides: the interface is their intersection for all rules but LoadClone, which is shown in detail in Fig. 2.

3 Conditions for Parallel Independence

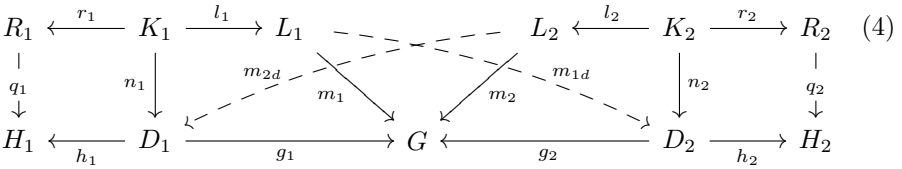
Parallel independence is a property that can be satisfied or not by two (DPO or SQPO) transformation steps from the same object G, like in the situation

depicted in Diagram (3), where we have two redexes (ρ_1, m_1) and (ρ_2, m_2) and transformation steps $G \overset{\rho_1,m_1}{\Longrightarrow} H_1$ and $G \overset{\rho_2,m_2}{\Longrightarrow} H_2$.



(3)

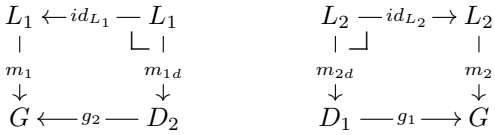
In the framework of the classical DPO approach, parallel independence was formulated in a categorical way [8,11,12] by requiring that each match factorizes through the context of the other transformation step. That is, there must exist the two dotted arrows of Diagram (4) so that the resulting triangles commute, i.e., $g_1 \circ m_{2d} = m_2$ and $g_2 \circ m_{1d} = m_1$. However, as shown explicitly with a counterexample in [4], this condition only works for DPO and SQPO with left-linear rules. For SQPO with non-left-linear rules the commutativity of the two triangles is not sufficient, but it is necessary to require the stronger condition that m_1 is reflected along g_2 by m_{1d}, and symmetrically for m_2.



(4)

Indeed, this is the condition of parallel independence that arose by addressing the Local Church-Rosser Problem in more general approaches where rules can be non-left-linear, like Reversible Sesqui-Pushout [10] and Rewriting in Span Categories [19]. We call this condition the *standard* one.

Definition 5 (Standard Condition). *Two redexes (ρ_1, m_1) and (ρ_2, m_2) as in Diagram (3) satisfy the* Standard Condition of parallel independence (STD-PI for short) *if the matches are reflected along the contexts, that is, if there are arrows $m_{1d} : L_1 \to D_2$ and $m_{2d} : L_2 \to D_1$ such that the two squares in Diagram (5) are pullbacks.*



(5)

Note that in most categories of interest, if the rules are left-linear then also morphisms $g_1 : D_1 \to G$ and $g_2 : D_2 \to G$ are mono. This holds either by Lemma 2.2 of [7] for the SQPO approach or, for the DPO approach in (quasi-)adhesive categories, because in such categories pushouts preserve (regular) monos [16].

Therefore in these cases by Fact 2(2) two redexes satisfy condition STD-PI if and only if they satisfy the more familiar condition of Diagram (4), i.e., there are arrows m_{1d} and m_{2d} making the two triangles commute.

Example 2 (the standard condition for parallel independence). Figure 2 shows a pair of parallel dependent redexes with rules LoadClone and Send, detected by the standard condition. Here and in the following, as a convention, we denote a pullback object (like $D1L2$) by concatenating the names of the cospan sources ($D1$ and $L2$ in this case), leaving implicit their morphisms to the common target (G). The conflict arises because the node cloned by LoadClone is used by Send, so the effect of Send after the application of LoadClone could differ depending on which of the clones is chosen by the match.

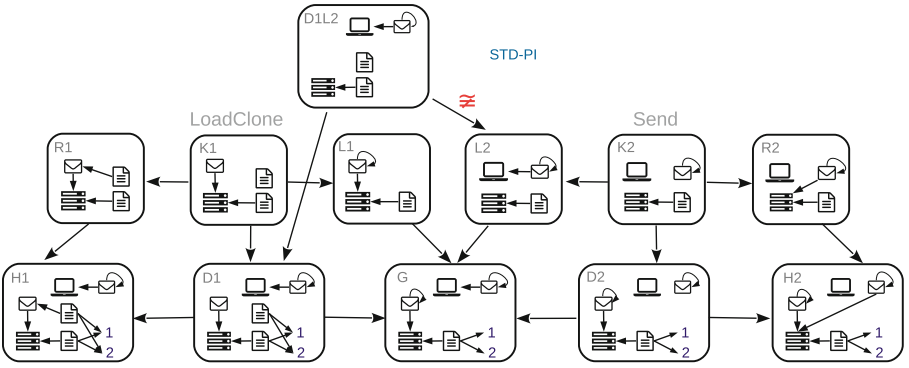


Fig. 2. Standard condition exposing a conflict between rules LoadClone and Send.

The condition of parallel independence for left-linear rules presented in [11] and shown in Diagram (4), being formulated in categorical terms, is very convenient for the proof of the Local Church-Rosser Theorem which is heavily based on diagrammatic constructions. But also a different, set-theoretical condition was proposed in [11] for DPO in the category of graphs, requiring that:

$$m_1(L_1) \cap m_2(L_2) = m_1(l_1(K_1)) \cap m_2(l_2(K_2)). \tag{6}$$

That is, each item needed by both redexes (in the image of both matches) must be preserved by both redexes (is also in the image of both interfaces).

$$\begin{array}{ccccc} K_1K_2 & & \xrightarrow{\pi_2^K} & & K_2 \\ & \searrow^{l} & \textcircled{2} & & \downarrow{\scriptstyle l_2} \\ \downarrow{\scriptstyle \pi_1^K} & & L_1L_2 & \xrightarrow{\pi_2^L} & L_2 \\ & & \downarrow{\scriptstyle \pi_1^L} & \textcircled{1} & \downarrow{\scriptstyle m_2} \\ K_1 & \xrightarrow{l_1} & L_1 & \xrightarrow{m_1} & G \end{array} \tag{7}$$

As discussed in [4], the classical condition of Diagram (4) is not a direct translation of this set-theoretical one, as the categorical counterpart of intersections are pullbacks. Diagram (7) shows the two pullbacks corresponding to the left side (①) and to the right side (②) of Eq. (6). The pullback objects are related by a unique mediating morphism $l : K_1K_2 \to L_1L_2$ such that $\pi_1^L \circ l = l_1 \circ \pi_1^K$ and $\pi_2^L \circ l = l_2 \circ \pi_2^K$, by the universal property of pullback ①.

By exploiting these pullbacks, the following condition of parallel independence was proposed in [4] as a direct categorical translation of Eq. (6).

Definition 6 (Pullback Condition). *Redexes (ρ_1, m_1) and (ρ_2, m_2) as in Diagram (3) satisfy the* Pullback Condition of parallel independence (PB-PI for short) *if in Diagram (7) mediating arrow $l : K_1K_2 \to L_1L_2$ is an isomorphism.*

Next, we propose a third condition. To our opinion, it is simpler than the two previous ones and it captures the essence of parallel independence. It works for general rules, and it can be simplified in the left-linear case.

Definition 7 (Essential Condition). *Let (ρ_1, m_1) and (ρ_2, m_2) be two redexes in an object G, as in Diagram (3), and let $(L_1L_2, \pi_1^L, \pi_2^L)$ be the pullback defined by ① in Diagram 7. Then (ρ_1, m_1) and (ρ_2, m_2) satisfy the* Essential Condition of parallel independence (ESS-PI) *if the pullback of the matches is reflected along the left-hand sides. That is, if there exist arrows $\alpha_1 : L_1L_2 \to K_1$ and $\alpha_2 : L_1L_2 \to K_2$ such that the two squares of Diagram (8) are pullbacks.*

$$\begin{array}{ccc} L_1L_2 & \xrightarrow{id} & L_1L_2 \\ {\scriptstyle \alpha_1}\downarrow & & \downarrow{\scriptstyle \pi_1^L} \\ K_1 & \xrightarrow{l_1} & L_1 \end{array} \qquad \begin{array}{ccc} L_1L_2 & \xrightarrow{id} & L_1L_2 \\ {\scriptstyle \alpha_2}\downarrow & & \downarrow{\scriptstyle \pi_2^L} \\ K_2 & \xrightarrow{l_2} & L_2 \end{array} \tag{8}$$

Example 3 (the pullback and the essential conditions). The rule LoadClone is in conflict with itself when it tries to load a clone of the same data node in two

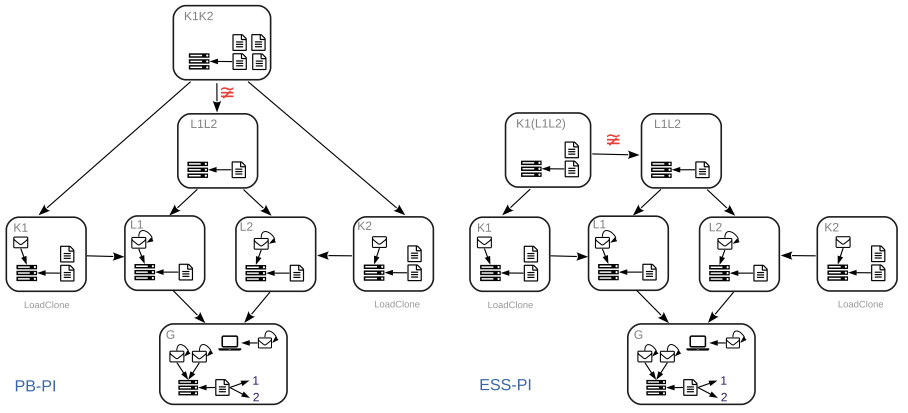


Fig. 3. Conditions PB-PI and ESS-PI show a conflict between two redexes of LoadClone.

distinct messages. Figure 3 shows how this conflict is captured by the pullback (PB-PI) and essential (ESS-PI) conditions of parallel independence.

4 Equivalence of Conditions for Parallel Independence

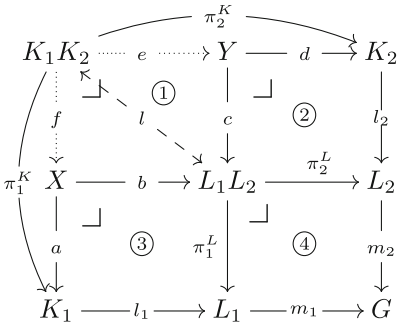
We present here explicit proofs of the equivalence of the three conditions introduced in the previous section for DPO and SQPO rewriting. The equivalence between the Standard and the Pullback Conditions was proved in an indirect way in [4], by exploiting some results of [5]. The proofs that follow are complete and in a way simpler than those in [4], by reducing both conditions to the new one. In fact, we first prove the equivalence of conditions PB-PI and ESS-PI, and next the equivalence of conditions STD-PI and ESS-PI. The proofs are presented for SQPO rewriting with possibly non-left-linear rules. They also apply to DPO rewriting with left-linear rules under mild conditions recalled in Proposition 1.

Theorem 1 (Equivalence of the Pullback and Essential Conditions). *Let* $\mathbf{C}$ *be a category with all pullbacks, and let* (ρ_1, m_1) *and* (ρ_2, m_2) *be two* SQPO *redexes in an object* G *of* $\mathbf{C}$*, as in Diagram (3). Then they satisfy condition* PB-PI *of Definition 6 if and only if they satisfy condition* ESS-PI *of Definition 7.*

Proof. **[If part].** Consider the following diagram:

$$\begin{array}{ccccc}
L_1L_2 & \xrightarrow{id} & L_1L_2 & \xrightarrow{\alpha_2} & K_2 \\
{\scriptstyle id}\downarrow & \textcircled{1} & {\scriptstyle id}\downarrow & \textcircled{2} & \downarrow{\scriptstyle l_2} \\
L_1L_2 & \xrightarrow{id} & L_1L_2 & \xrightarrow{\pi_2^L} & L_2 \\
{\scriptstyle \alpha_1}\downarrow & \textcircled{3} & {\scriptstyle \pi_1^L}\downarrow & \textcircled{4} & \downarrow{\scriptstyle m_2} \\
K_1 & \xrightarrow{l_1} & L_1 & \xrightarrow{m_1} & G
\end{array} \tag{9}$$

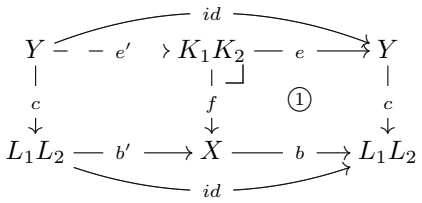
All squares are pullbacks (④ by construction, ② and ③ by condition ESS-PI, and ① trivially). Thus by uniqueness of pullbacks up to a unique isomorphism commuting with the projections, we can deduce that the mediating arrow $l : K_1K_2 \to L_1L_2$ of (7) is an isomorphism.



(10)

[Only if part]. In Diagram (10) the outer diagram and ④ are pullbacks by construction, and their mediating arrow $l : K_1K_2 \to L_1L_2$ is an isomorphism by condition PB-PI. Let ② and ③ be built as pullbacks: we have to show that b and c are isomorphism. Since ③ is a pullback and $\pi_1^L \circ l = l_1 \circ \pi_1^K$, there is a unique arrow $f : K_1K_2 \to X$ such that $b \circ f = l$ and $a \circ f = \pi_1^K$ and, symmetrically, there is a unique arrow $e : K_1K_2 \to Y$ such that $d \circ e = \pi_2^K$ and $c \circ e = l$. The resulting square ① is a pullback: in fact, the outer square and ② + ④ (by Fact 1(1)) are pullbacks, thus ① + ③ is a pullback by Fact 1(2); in turn since ③ is a pullback, also ① is, again by Fact 1(2). Since $b \circ f = l = c \circ e$ and l is an isomorphism, f and e are *sections* (that is, they have a left-inverse) and b and c are *retractions* (i.e., they have a right-inverse).

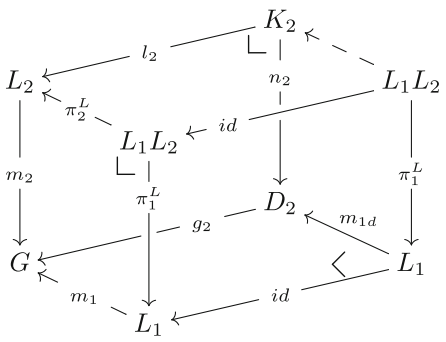
But pullbacks preserve retractions, as shown in the diagram to the right. If b is a retraction then there is a b' such that $b \circ b' = id$. Since ① is a pullback and the outer square commutes, there is an e' such that $e \circ e' = id$, thus e is a retraction as well.



Thus f and e are also retractions, and therefore they are isomorphisms. This implies that b and c are isomorphisms as well, as desired. □

Theorem 2 (Equivalence of the Standard and Essential Conditions). *Let* $\mathbf{C}$ *be a category with all pullbacks, and let* (ρ_1, m_1) *and* (ρ_2, m_2) *be two* SQPO *redexes in an object* G *of* $\mathbf{C}$*, as in Diagram (3). Then they satisfy condition* STD-PI *of Definition 5 if and only if they satisfy condition* ESS-PI *of Definition 7.*

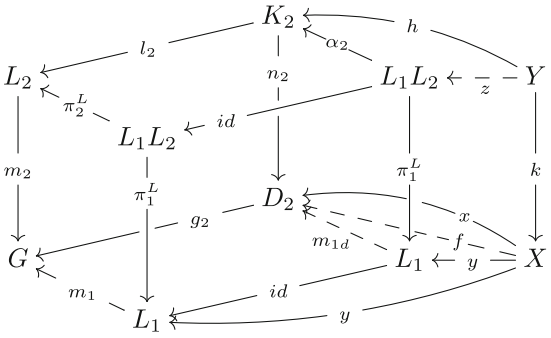
Proof. **[Only if part].** Consider the following diagram:



(11)

The back-left face is a pullback by construction, the bottom face is a pullback by condition STD-PI, and the front-right face is also trivially a pullback. Therefore since the front-left face commutes, by Fact 1(2) there is a unique arrow $L_1L_2 \to K_2$ making the upper face (i.e., the right square of Diagram (8)) a pullback. For the left square of Diagram (8) the proof is analogous.

[If part]. Consider the cube in Diagram (12), ignoring for the time being objects X, Y, and the arrows starting from them. By Definition 3 the back-left face $K_2 \stackrel{n_2}{\to} D_2 \stackrel{g_2}{\to} G$ is a final pullback complement of $K_2 \stackrel{l_2}{\to} L_2 \stackrel{m_2}{\to} G$. Therefore, since the square made of the front-left and front-right faces is obviously a pullback and the top face commutes by the essential condition, there is a unique arrow $m_{1d} : L_1 \to D_2$ such that the bottom and the back-right face commute.



(12)

In order to show that condition STD-PI holds, i.e. that the bottom face is a pullback, by Fact 2(1) it is sufficient to show that for each object X and arrows $x : X \to D_2$, $y : X \to L_1$ such that $g_2 \circ x = m_1 \circ y$, it holds (†) $m_{1d} \circ y = x$. In order to show this, let $K_2 \stackrel{h}{\leftarrow} Y \stackrel{k}{\to} X$ be the pullback of $K_2 \stackrel{n_2}{\to} D_2 \stackrel{x}{\leftarrow} X$. By composing this pullback with the back-left face we get a pullback with vertices Y, L_2, X and G, and therefore since the back-left face is a final pullback complement, there is a unique arrow $f : X \to D_2$ such that (a) $f \circ k = n_2 \circ h$ and (b) $g_2 \circ f = g_2 \circ x$. Thus (†) will follow by showing that both x and $m_{1d} \circ y$ satisfy properties (a) and (b). For x, (a') $x \circ k = n_2 \circ h$ holds by construction, and (b') is obvious. For $m_{1d} \circ y$, we have (b'') $g_2 \circ m_{1d} \circ y = m_1 \circ id \circ y = m_1 \circ y = g_2 \circ x$. In order to show (a'') $m_{1d} \circ y \circ k = n_2 \circ h$, observe that the composition of the top face (that is a pullback by the essential condition) and of the front-left face is a pullback, and that the outmost square commutes ($Y \stackrel{h}{\to} K_2 \stackrel{m_2 \circ l_2}{\to} G = Y \stackrel{y \circ k}{\to} L_1 \stackrel{m_1}{\to} G$), therefore there is a unique arrow $z : Y \to L_1L_2$ such that (c) $\alpha_2 \circ z = h$ and (d) $\pi_1^L \circ id \circ z = y \circ k$. Thus we have $m_{1d} \circ y \circ k \stackrel{(d)}{=} m_{1d} \circ \pi_1^L \circ z = n_2 \circ \alpha_2 \circ z \stackrel{(c)}{=} n_2 \circ h$, as desired. □

Proposition 1 (Equivalence of Conditions for DPO rewriting). *If category* **C** *is quasi adhesive and the left-hand sides of the rules are regular monos (i.e. they are equalizers of pairs of parallel arrows), then Theorems 1 and 2 also apply to* DPO *redexes.*

Proof. By Proposition 12 of [7], the pushout complement of a regular-mono left-hand side and a match is also a final pullback complement. Therefore a DPO redex is also a SQPO redex. □

5 Implementation in the Verigraph System

Parallel independence is important for practical applications involving graph transformation, in particular for static analysis techniques. The equivalence of conditions STD-PI, ESS-PI and PB-PI means that tools are free to implement any of them. In this context, time is usually the most critical resource, thus it should guide the choice of the algorithm. Here and in the next section we compare the performance of the three conditions based on their implementation in the Verigraph system and on their use, in various scenarios, for some concrete grammars defined in categories of typed graphs.

Verigraph [9] is implemented in Haskell, exploiting its abstraction mechanisms to promote separation between abstract and concrete code. This allows the algorithms for checking parallel independence to be implemented at an abstract level (based on arrows and composition) as an almost direct translation of categorical diagrams and definitions. Clearly, when such an abstract algorithm is applied to a concrete category of structures like (possibly typed) graphs, unary algebras, Petri nets, etc., its efficiency depends on the data structures and algorithms that implement the concrete structures, their morphisms and the primitive categorical operations on them. In general one cannot expect an instantiation of an abstract algorithm to be more efficient than an algorithm specifically designed for the concrete structures. This is why we compare only the algorithms as implemented in Verigraph, while we defer to future work a comparison with algorithms for parallel independence developed in other frameworks.

Verigraph supports the definition and analysis of DPO graph transformation systems, while the support for SQPO is under development. For this reason the experimental evaluation presented in the next section will consider left-linear rules only. Without loss of generality, we also assume that rules are right-linear, because the right-hand sides don't play a role in the conditions for parallel independence considered in this paper.

5.1 Data Structures

We briefly describe now the data structures used to represent typed graphs and related concepts. Graphs are directed and unlabeled, and nodes and edges are identified by unique integers. Each graph is made of a list of node identifiers and a list of tuples (e, s, t), which are identifiers for the edge, its source and its target, respectively. This representation is convenient as most operations on graphs will traverse all its elements. Graph morphisms are represented as pairs of finite maps. We use the datatypes provided by the standard library of Haskell, which implements finite maps using balanced binary trees. A *typed graph* over a type graph T is a graph morphism $G^T : G \to T$. A *typed graph morphism* $f : G^T \to H^T$ is just a graph morphism $f : G \to H$ which commutes with the morphisms to T. Rules are spans $L^T \hookleftarrow K^T \rightarrowtail R^T$ of injective typed graph morphisms.

5.2 Primitive Categorical Operations

Verigraph provides many categorical operations as primitives for algorithm construction. For reasons of space, we only present a brief explanation of how the most relevant operations for testing parallel independence were implemented in the category of typed graphs. For more details, we refer to the source code [1].

Pullback. Let $\mathbb{P}(X) = \{S \mid S \subseteq X\}$ and, given $f : X \to Y$, let the inverse $f^{-1} : Y \to \mathbb{P}(X)$ be defined as $f^{-1}(y) = \{x \mid f(x) = y\}$. The pullback of a cospan $(X \xrightarrow{f} Z \xleftarrow{g} Y)$ is obtained by taking the inverses $f^{-1} : Z \to \mathbb{P}(X)$ and $g^{-1} : Z \to \mathbb{P}(Y)$, then creating the disjoint union of the product of the preimages for each element of Z, that is, $\biguplus_{z \in Z} f^{-1}(z) \times g^{-1}(z)$. This is done independently for edges and nodes in order to construct the pullback graph and associated projection morphisms.

Pushout Complement. The pushout complement (see Definition 1) of $(K \overset{l}{\rightarrowtail} L \overset{m}{\to} G)$ in the category of typed graphs exists iff the *gluing conditions* are satisfied [11]. These conditions are checked set-theoretically, before the actual construction. To construct the pushout complement of l and m when it exists, we compute the elements present in the image of m but not in that of $m \circ l$, and remove them from G obtaining the subgraph D. The morphism $g : D \rightarrowtail G$ is the inclusion, while $n : K \to D$ is obtained by restricting the codomain of $m \circ l$.

Isomorphism Check. To check whether a morphism $f : X \to Y$ is an iso, we build the inverse $f^{-1} : Y \to \mathbb{P}(X)$ and then check that the image of each element of Y is a singleton.

Factorization Check. Given a cospan $(X \xrightarrow{f} Z \xleftarrow{g} Y)$ with g mono, to search for morphisms $h : X \to Y$ such that $g \circ h = f$, we first take the inverse $g^{-1} : Z \to \mathbb{P}(Y)$. Then we determine the existence of a morphism h by computing $g^{-1} \circ f$ and checking if $\forall x \in X.\, g^{-1}(f(x)) \neq \emptyset$ holds.

5.3 Parallel Independence Test

Given the primitive operations just described, testing parallel independence can be achieved by constructing the required diagram elements and testing them for desired properties, for example, if a calculated morphism is iso. In the case of left-linear rules, there are two variants for conditions STD-PI and ESS-PI: as a consequence of Lemma 2, the reflection of f along g may either be tested by (i) constructing a pullback and checking for the isomorphism of one of its components, or (ii) by checking if f factorizes through g. We refer to the former by their regular names, and to the latter by STD-F-PI and ESS-F-PI. Figure 4 summarizes diagrammatically these five conditions: the left column contains the variants based on factorization and the right one those based on isomorphism tests. They are also categorized on whether they test the diagram elements *statically* (based on the rules and the matches only) or *dynamically* (using the construction of the FPBC/POC as part of the test). For readability, Fig. 4 shows only the left part of

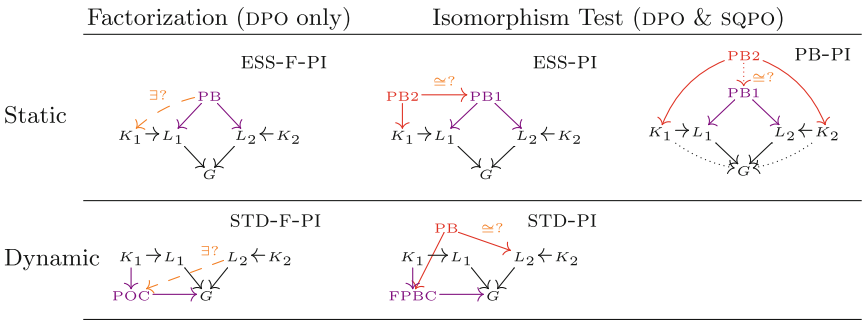


Fig. 4. Conditions for verifying parallel independence of transformations.

the conditions, although the implemented routines test both sides. The complete source code together with the tested grammars are available as a Verigraph special release [1], in file `src/library/Analysis/ParallelIndependent.hs`. Since Haskell is a lazy language, strict evaluation was enforced to provide a better measure of the overall computational effort required in each test.

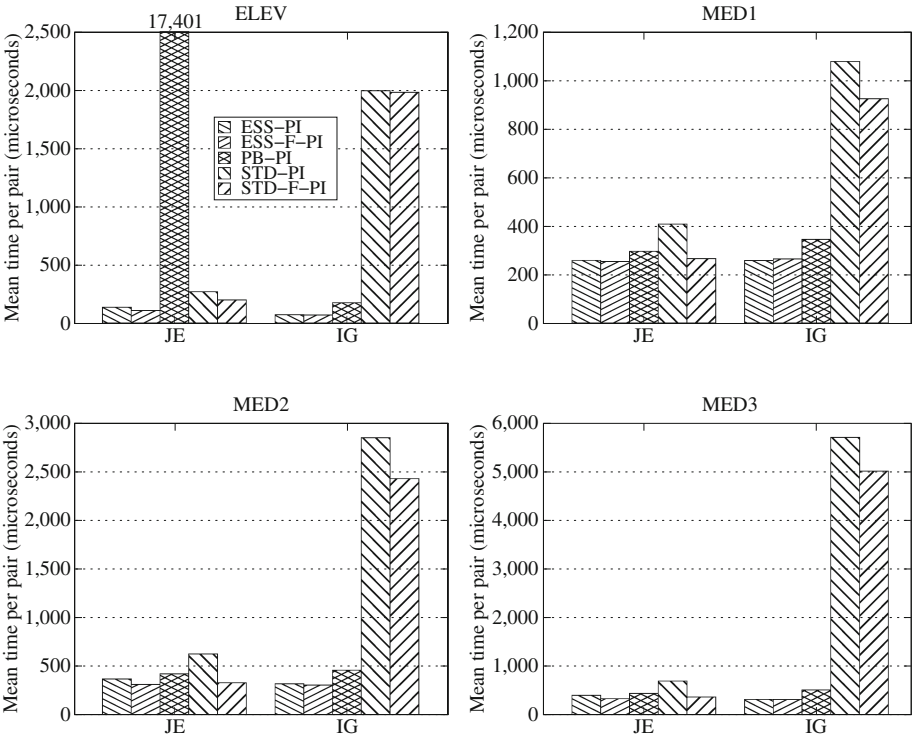


Fig. 5. Mean runtime of each algorithm, per pair of redexes, over all input sets. ELEV contains one outlier whose bar was truncated, with its numeric value written above it.

6 Experimental Evaluation

The performance of evaluating these conditions should be sensitive to characteristics of the rules, of the *instance graphs* to which they are applied, and of the matches. The size of the instance graph G, for example, should strongly affect the dynamic conditions. The static conditions, on the other hand, should be less sensitive to the size of G, since the elements outside the matches are filtered out by the first pullback, leaving smaller graphs for the subsequent operations. Conversely, the presence of non-injective matches should affect the static conditions more than the dynamic ones, because the size of the pullback of the matches grows with the number of elements identified by them. We expect that considering non-left-linear rules this multiplicative effect may be reinforced.

To compare the performance of the five algorithms, two scenarios were used:

Elevator [17]**:** A grammar with 9 rules that models the behaviour of an elevator system. It will be referred to as ELEV.

Medical guidelines [2]**:** Three grammars that model guidelines for a medical procedure, containing 36 rules in total. These grammars are referred to as MED1, MED2 and MED3.

These grammars are made of linear DPO rules. The files used to perform the benchmark and the obtained results are available at https://verites.github.io/parallel-independence-benchmarks/. The experiment compares the execution time needed to evaluate the five conditions over eight sets of inputs (pairs of redexes), two for each grammar, generated in the following ways.

Jointly epic pairs: Given a grammar GR, the input set GR-JE is obtained by considering for each pair of rules $(L_1 \leftarrow K_1 \rightarrow R_1, L_2 \leftarrow K_2 \rightarrow R_2)$, all possible partitions of $L_1 \uplus L_2$ (i.e. all epimorphisms $e : L_1 \uplus L_2 \twoheadrightarrow G$), and adding the pair $(m_1 = e \circ in_1, m_2 = e \circ in_2)$ to GR-JE iff both matches satisfy the gluing conditions. This is the standard approach when performing critical pair analysis.

Fixed instance graph: Given a grammar GR, a fixed instance graph G considerably larger than the left-hand sides of the rules of GR is constructed. Then for each pair of rules of GR all possible pairs of matches $(m_1 : L_1 \rightarrow G, m_2 : L_2 \rightarrow G)$ such that both matches satisfy the gluing conditions are added to set GR-IG. This represents testing parallel independence during concrete rewritings. For the grammars considered in the experimentation, graph G contains approximately 20 nodes.

For each input set, the five variants of the parallel independence test were executed ten times, the execution time was measured and the average time per pair of matches was calculated. The benchmark was executed on an Intel(R) Core(TM) i5-3330 machine with a 3.00GHz CPU and 16GB of RAM. Figure 5 presents the results.

From the observed results, we conclude that the static variants (ESS-PI, ESS-F-PI and PB-PI) outperform the dynamic variants (STD-PI and STD-F-PI) in most

cases, particularly with large instance graphs. One important exception was the case ELEV-JE, where PB-PI performed much worse than all other alternatives. We conjectured that this occurs due to the multiplicative effect of pullbacks in presence of non-injective matches. To confirm this, we repeated the experiment considering only the inputs with injective matches. In this case, the mean time per pair of PB-PI was 146 μs, slower than ESS-PI (114 μs) and ESS-F-PI (99 μs) but faster than STD-PI (265 μs) and STD-F-PI (211 μs).

Regarding the comparison of using factorization (ESS-F-PI, STD-F-PI) or isomorphism test (ESS-PI, STD-PI), we observed that factorization seems consistently more efficient than pullback calculation, although in some cases the results were similar, as in MED3-JE. Thus, the factorization-based algorithms can be recommended when rules are left-linear.

In general, the essential conditions (ESS-PI and ESS-F-PI) presented very good performance in all situations. ESS-F-PI was in many cases the fastest for testing parallel independence. On the other hand, ESS-PI had an overall good performance, and often the difference between it and ESS-F-PI was insignificant. Therefore, the essential conditions can be recommended for all cases.

7 Conclusions

In this paper we have considered some definitions of parallel independence proposed for the Double-Pushout and the Sesqui-Pushout Approaches to graph transformation, and we proposed a new condition, that we called the *Essential Condition* (ESS-PI). We presented explicit proofs of equivalence of condition ESS-PI with the Standard (STD-PI) and the Pullback (PB-PI) Conditions previously proposed in the literature at an abstract categorical level. Next we have implemented five variants of the parallel independence test (two being optimized versions of STD-PI and ESS-PI for left-linear rules) for grammars based on categories of typed graphs in the Verigraph system. We evaluated the runtime efficiency of each condition over a collection of test cases based on DPO transformation with linear rules only, because the support of SQPO by Verigraph is still under development. Our experiments led to the conclusion that the essential condition has the best performance in most cases.

We foresee several developments of the work presented in this paper. From the more theoretical side we intend to investigate how the intuition behind condition ESS-PI could be exploited in other frameworks. For example, it should be possible to define a stronger version of ESS-PI equivalent to, but simpler than, the *strong parallel independence* considered in [13] for DPO transformations with injective matchings. We also intend to exploit condition ESS-PI for defining and computing efficiently *minimal conflict reasons* between redexes, as studied for example in [3,18], and to evaluate to what extent ESS-PI can be exploited to improve existing algorithms for Critical Pair Analysis. This is not obvious, because several optimization techniques have been developed (see e.g. [18]) that should be adapted to our condition for independence. In this context, we also intend to check formally under which assumptions condition ESS-PI is equivalent to the

definition of parallel independence based on an initial pushout for the left-hand side of a rule, as proposed in [17,18]

Concerning the comparison of efficiency of the various conditions for parallel independence, the initial evaluations presented here should be completed to encompass non-left-linear rules as well. Next we intend to extend the comparison to algorithms for checking independence developed in other frameworks, like AGG [22].

References

1. Bezerra, J.S., Costa, A., Azzi, G., Rodrigues, L.M., Machado, R., Ribeiro, L.: Verites/verigraph: parallel independence benchmarks, June 2017. https://doi.org/10.5281/zenodo.814246
2. Bezerra, J.S., Costa, A., Ribeiro, L., Cota, É.F.: Formal verification of health assessment tools: a case study. Electr. Notes Theor. Comput. Sci. **324**, 31–50 (2016). https://doi.org/10.1016/j.entcs.2016.09.005
3. 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
4. Corradini, A.: On the definition of parallel independence in the algebraic approaches to graph transformation. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 101–111. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_8
5. 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
6. Corradini, A., Gadducci, F.: On term graphs as an adhesive category. Electr. Notes Theor. Comput. Sci. **127**(5), 43–56 (2005). https://doi.org/10.1016/j.entcs.2005.02.014
7. 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
8. 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 Publishing Co., Inc (1997). http://www.worldscientific.com/doi/abs/10.1142/9789812384720_0003
9. Costa, A., Bezerra, J., Azzi, G., Rodrigues, L., Becker, T.R., Herdt, R.G., Machado, R.: Verigraph: a system for specification and analysis of graph grammars. In: Ribeiro, L., Lecomte, T. (eds.) SBMF 2016. LNCS, vol. 10090, pp. 78–94. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49815-7_5
10. Danos, V., Heindel, T., Honorato-Zimmer, R., Stucki, S.: Reversible sesqui-pushout rewriting. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 161–176. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09108-2_11
11. Ehrig, H.: Introduction to the algebraic theory of graph grammars (a survey). In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph-Grammars and Their Application to Computer Science and Biology. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0025714

12. Ehrig, H., Kreowski, H.-J.: Parallelism of manipulations in multidimensional information structures. In: Mazurkiewicz, A. (ed.) MFCS 1976. LNCS, vol. 45, pp. 284–293. Springer, Heidelberg (1976). https://doi.org/10.1007/3-540-07854-1_188
13. Habel, A., Müller, J., Plump, D.: Double-pushout graph transformation revisited. Math. Struct. Comput. Sci. **11**(5), 637–688 (2001). https://doi.org/10.1017/S0960129501003425
14. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006). https://doi.org/10.1007/11691617_19
15. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon (1970). http://www.sciencedirect.com/science/article/pii/B9780080129754500 28X
16. Lack, S., Sobocinski, P.: Adhesive and quasiadhesive categories. Theor. Inform. Appl. **39**(3), 511–545 (2005). https://doi.org/10.1051/ita:2005028
17. Lambers, L.: Certifying rule-based models using graph transformation. Ph.D. thesis, Technische Universität Berlin (2010). https://doi.org/10.14279/depositonce-2348
18. 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
19. Löwe, M.: Graph rewriting in span-categories. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 218–233. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2_15
20. Plump, D.: Evaluation of functional expressions by hypergraph rewriting. Ph.D. thesis, University of Bremen, Germany (1993). http://d-nb.info/940423774
21. Rosen, B.K.: A Church-Rosser theorem for graph grammars. ACM SIGACT News **7**(3), 26–31 (1975). https://doi.org/10.1145/1008343.1008344
22. 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

Integration of Graph Constraints into Graph Grammars

Annegret Habel, Christian Sandmann, and Tilman Teusch(✉)

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

Abstract. We investigate the integration of graph constraints into graph grammars and consider the filter problem: Given a graph grammar and a graph constraint, does there exist a "goal-oriented" grammar that generates all graphs of the original graph language satisfying the constraint. We solve the filter problem for specific graph grammars and specific graph constraints. As an intermediate step, we construct a constraint automaton accepting exactly the graphs in the graph language that satisfy the constraint.

1 Introduction

In meta-modelling, a general problem is how to generate instances of a meta-model. There are several approaches to instance generation: most of them are logic-oriented, some are rule-based (for an overview see, e.g., [RAB+15]). Our approach to instance generation is rule-based:

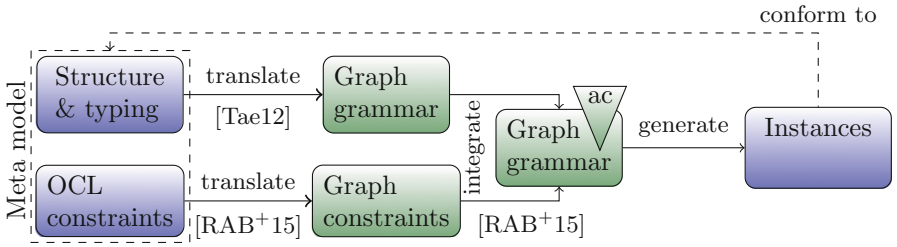
(1) Translate the meta-model without OCL constraints into a graph grammar GG and the OCL constraints into a graph constraint [Tae12,RAB+15].
(2) Integrate the graph constraint c into the graph grammar GG yielding a graph grammar GG_c generating the graphs satisfying the graph constraints.
(3) Generate instances $I \in L(GG_c)$.

In Radke et al. [RAB+15], the integration of a graph constraint c into a graph grammar GG is done by replacing the rules ϱ of the grammar by the corresponding c-guaranteeing rules. Unfortunately, this yields a grammar GG_c generating a subset of all graphs satisfying the constraint c, i.e., $L(GG_c) \subseteq L(GG) \cap [\![c]\!]$, and the inclusion is usually proper.

In this paper, we look for a construction of a graph grammar GG_c that generates exactly the set of all graphs of the original graph grammar that satisfy the constraint c: $L(GG_c) = L(GG) \cap [\![c]\!]$. We look for a "goal-oriented" grammar that "filters" exactly those graphs of the graph language that satisfy the constraint. In the following, we talk about the filter problem.

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

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 19–36, 2018.
https://doi.org/10.1007/978-3-319-75396-6_2



Filter Problem

Given: A graph grammar GG and a graph constraint c.
Question: Does there exist a goal-oriented graph grammar GG_c such that $L(GG_c) = L(GG) \cap [\![c]\!]$?

We solve the filter problem for specific graph grammars GG and specific graph constraints c. The construction is done in two steps.

(1) Construct a goal-oriented constraint automaton $\mathcal{A}_c$ with $L(\mathcal{A}_c)=L(GG)\cap[\![c]\!]$.
(2) Construct a goal-oriented graph grammar GG_c with $L(GG_c) = L(\mathcal{A}_c)$.

We illustrate our approach with Petri nets as the modeling language. We consider a graph grammar for generating Petri-nets and graph constraints for restricting Petri-nets. The example is a simplification of the one in [RAB+15] for typed attributed graphs. Due to space restrictions, we do not consider types and attributes here.

Example 1. The Petri-net grammar and Petri-net constraints are given below.

Start graph	S	$= \boxed{\text{PN}}$
Insert a new place:	`AddP`	$= \boxed{\text{PN}} \Rightarrow \boxed{\text{PN}} \xrightarrow{\text{place}} \boxed{\text{Pl}}$
Insert a new transition:	`AddT`	$= \boxed{\text{PN}} \Rightarrow \boxed{\text{PN}} \xrightarrow{\text{trans}} \boxed{\text{Tr}}$
Insert a token in a given place:	`AddTok`	$= \boxed{\text{Pl}} \Rightarrow \boxed{\text{Pl}} \xrightarrow{\text{token}} \boxed{\text{tk}}$
Connect a transition to a place:		...
Connect a place to a transition:		...

The net has at least one transition:	tr	$= \exists\,(\boxed{\text{Tr}})$
The net has at least one place:	place	$= \exists\,(\boxed{\text{Pl}})$
There is a place with token:	tok	$= \exists\,(\boxed{\text{Pl}} \xrightarrow{\text{token}} \boxed{\text{tk}})$
There is a place with two tokens:	2tok	$= \exists\,(\boxed{\text{tk}} \xleftarrow{\text{token}} \boxed{\text{Pl}} \xrightarrow{\text{token}} \boxed{\text{tk}})$
There is a place with a token, but every place has at most one token:	rtok	$= \text{tok} \wedge \neg\text{2tok}$

The structure of the paper is as follows. In Sect. 2, we review the definitions of graphs, graph conditions, and graph grammars. In Sect. 3, we sketch some basics on weakest liberal preconditions, guaranteeing rules, containments, and

minimizations. In Sect. 4, we introduce so-called constraint automata, present a backward construction, sketch some closure properties, consider the termination of the backward construction, and derive a goal-oriented graph grammar from the constraint automaton. In Sect. 5, we present some related concepts. In Sect. 6, we give a conclusion and mention some further work.

2 Preliminaries

In the following, we recall the definitions of directed, labelled graphs, graph conditions and graph grammars [EEPT06, HP09] and so-called goal-oriented grammars, essentially equivalent to the ones in [Bec16].

Definition 1 (graphs & morphisms). A *(directed, labeled) graph* (over a label alphabet $\mathcal{L}$) is a system $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_{G,\mathrm{V}}, \mathrm{l}_{G,\mathrm{E}})$ where V_G and E_G are finite sets of *nodes* (or *vertices*) and *edges*. $\mathrm{s}_G, \mathrm{t}_G \colon \mathrm{E}_G \to \mathrm{V}_G$ are total functions assigning *source* and *target* to each edge, $\mathrm{l}_{\mathrm{V},G} \colon \mathrm{V}_G \to \mathcal{L}$, $\mathrm{l}_{\mathrm{E},G} \colon \mathrm{E}_G \to \mathcal{L}$ are labeling functions. If $\mathrm{V}_G = \emptyset$, then G is the *empty graph*, denoted by $\emptyset$. Given graphs G and H, a *(graph) morphism* $g \colon G \to H$ consists of total functions $g_\mathrm{V} \colon \mathrm{V}_G \to \mathrm{V}_H$ and $g_\mathrm{E} \colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve sources, targets and labels, that is, $g_\mathrm{V} \circ \mathrm{s}_G = \mathrm{s}_H \circ g_\mathrm{E}$, $g_\mathrm{V} \circ \mathrm{t}_G = \mathrm{t}_H \circ g_\mathrm{E}$, $\mathrm{l}_{G,\mathrm{V}} = \mathrm{l}_{H,\mathrm{V}} \circ g_\mathrm{V}$, $\mathrm{l}_{G,\mathrm{E}} = \mathrm{l}_{H,\mathrm{E}} \circ g_\mathrm{E}$. The morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is injective and surjective. In the latter case, G and H are *isomorphic*, which is denoted by $G \cong H$.

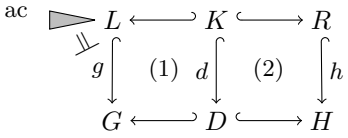
Definition 2 (graph conditions). A *(graph) condition* over a graph P is of the form (a) true or (b) $\exists(a, c)$ where $a \colon P \hookrightarrow C$ is an injective morphism and c is a condition over C. For conditions c, c_i ($i \in I$ for some index set I) over P, $\neg c$ and $\wedge_{i \in I}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

Notation. Graph conditions may be written in a more compact form: $\exists\, a$ abbreviates $\exists\,(a, \mathrm{true})$. $\forall(a, c)$ abbreviates $\nexists\,(a, \neg c)$. The expressions $\vee_{i \in I} c_i$, and $c \to c'$ are defined as usual. For an injective morphism $a \colon P \hookrightarrow C$ in a condition, we just depict the codomain C, if the domain P can be unambiguously inferred, e.g., $\forall(\boxed{\mathrm{Pl}}, \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{preArc}} \boxed{\mathrm{TPArc}}) \vee \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{postArc}} \boxed{\mathrm{PTArc}})$ abbreviates $\forall(\emptyset \hookrightarrow \boxed{\mathrm{Pl}}, \exists\,(\boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{preArc}} \boxed{\mathrm{TPArc}}) \vee \exists\,(\boxed{\mathrm{Pl}} \hookrightarrow \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{postArc}} \boxed{\mathrm{PTArc}})$. Intuitively, the constraint means "There are no isolated places."

Any injective morphism $p \colon P \hookrightarrow G$ *satisfies* true. p *satisfies* $\exists\,(a, c)$ if there exists an injective morphism $q \colon C \hookrightarrow G$ such that $q \circ a = p$ and q satisfies c. p *satisfies* $\neg c$ if p does not satisfy c, and p *satisfies* $\wedge_{i \in I} c_i$ if p satisfies each c_i ($i \in I$). We write $p \models c$ if p satisfies the condition c (over P). A graph G *satisfies* a constraint c, $G \models c$, if the morphism $p \colon \emptyset \hookrightarrow G$ satisfies c. $[\![c]\!]$ denotes the class of all graphs satisfying c.

Two conditions c and c' over P are *equivalent*, denoted by $c \equiv c'$, if for all injective morphisms $p \colon P \hookrightarrow G$, $p \models c$ iff $p \models c'$. A condition c *implies* a condition c', denoted by $c \Rightarrow c'$, if for all injective morphisms $p \colon P \hookrightarrow G$, $p \models c$ implies $p \models c'$.

Definition 3 (rules and transformations). A *rule* $\varrho = \langle p, \mathrm{ac}\rangle$ consists of a *plain rule* $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$ with injective morphisms $K \hookrightarrow L$ and $K \hookrightarrow R$ and an application condition ac over L and is *p-restricting*. A rule $\langle p, \mathrm{true}\rangle$ is abbreviated by p. $\mathrm{Rhs}(\varrho) = R$ denotes the right-hand side of ϱ. A *direct transformation* from a graph G to a graph H applying rule ϱ at an injective morphism g consists of two pushouts[1] (1) and (2) as below where $g \models \mathrm{ac}$. We write $G \Rightarrow_{\varrho,g,h} H$ or $G \Rightarrow_{\varrho,g} H$ if there exists such a direct transformation.



Given graphs G, H and a set $\mathcal{R}$ of rules, G *derives* H by $\mathcal{R}$ ($w \in \mathcal{R}^*$) if $G \cong H$ or there is a sequence of direct transformations $G = G_0 \Rightarrow_{\varrho_1} \ldots \Rightarrow_{\varrho_n} G_n = H$ with $\varrho_1, \ldots, \varrho_n \in \mathcal{R}$ ($w = \varrho_1 \ldots \varrho_n$). In this case we write $G \Rightarrow^*_{\mathcal{R}} H$ or $G \Rightarrow^*_w H$.

Definition 4 (graph grammars). A graph grammar $GG = (\mathcal{N}, \mathcal{R}, S)$ consists of a finite set $\mathcal{N}$ of nonterminal symbols, a finite set $\mathcal{R}$ of rules, and a start graph S. In the case of $\mathcal{N} = \emptyset$, we write $GG = (\mathcal{R}, S)$ instead of $GG = (\emptyset, \mathcal{R}, S)$. The *graph language* generated by GG consists of all graphs derivable from S by $\mathcal{R}$: $L(GG) = \{G \in \mathcal{G} \mid S \Rightarrow^*_{\mathcal{R}} G\}$ where $\mathcal{G}$ denotes the class of all graphs without nonterminal symbols. The grammar GG is *goal-oriented* if there is a terminating[2] rule set $\mathcal{R}_t \subseteq \mathcal{R}$ such that, for all derivable graphs, there is a transformation to a terminal graph using rules form $\mathcal{R}_t$.

3 Weakest Liberal Preconditions

In this section, we sketch the prerequisites for our backward construction in Sect. 4: existential weakest liberal preconditions, similar to (universal) weakest liberal preconditions. Moreover, we introduce the syntactic operations containment and minimization which imply implication and equivalence, respectively.

Definition 5 (guaranteeing rule, weakest liberal precondition). Given a constraint d, a rule ϱ is *d-guaranteeing* if for all direct transformations $G \Rightarrow_\varrho H$, the result H satisfies d. Given a constraint d and a rule ϱ, a condition c is an *(existential) liberal precondition* of a rule ϱ relative to a condition d, if, for all G satisfying c, there exists some $G \Rightarrow_\varrho H$ such that $H \models d$ and an (existential) *weakest* liberal precondition of ϱ relative to d, if any (existential) liberal precondition of ϱ relative to d implies c.

Fact 1 (characterization). A condition c is an existential weakest liberal precondition of ϱ relative to d if, for all G, $G \models c$ iff there exists some $G \Rightarrow_\varrho H$ such that $H \models d$.

[1] For definition & existence of pushouts in the category of graphs see e.g. [EEPT06].
[2] A rule set $\mathcal{R}$ is *terminating* if there is no infinite transformation $G_0 \underset{\mathcal{R}}{\Rightarrow} G_1 \underset{\mathcal{R}}{\Rightarrow} G_2 \ldots$.

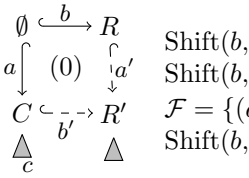
Proof. Analogously to universal weakest liberal preconditions [HP09] □

Proposition 1 (gua and $\mathrm{Wlp}_\exists$ [HP09]). There are constructions gua and $\mathrm{Wlp}_\exists$ such that, for every rule ϱ and every constraint d,

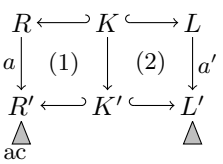
1. $\mathrm{gua}(\varrho, d)$ is a ϱ-restricting[3] and d-guaranteeing rule and
2. $\mathrm{Wlp}_\exists(\varrho, d)$ is an existential liberal weakest precondition of ϱ relative to d.

The guaranteeing rule and the weakest precondition of a rule and a constraint are constructed by the basic transformations from graph constraints to right application conditions (Shift), right to left application conditions (L), and application conditions to constraints ($\mathrm{C}_\exists$).

Construction 1. For a rule $\varrho = \langle p, \mathrm{ac}\rangle$, $\mathrm{gua}(\varrho, d) := \langle p, \mathrm{L}(\varrho, \mathrm{Shift}(b, d)\rangle$ with $b\colon \emptyset \hookrightarrow \mathrm{Rhs}(\varrho)$ and $\mathrm{Wlp}_\exists(\varrho, d) := \mathrm{C}_\exists(\mathrm{gua}(\varrho, d) \wedge \mathrm{Appl}(\varrho))$ where $\mathrm{Shift}, \mathrm{L}, \mathrm{C}_\exists$, and Appl are defined as follows.



$\mathrm{Shift}(b, \mathrm{true}) := \mathrm{true}.$
$\mathrm{Shift}(b, \exists\,(a, c)) := \bigvee_{(a',b')\in\mathcal{F}} \exists\,(a', \mathrm{Shift}(b', c))$ where
$\mathcal{F} = \{(a', b') \mid b' \circ a = a' \circ b,\ a', b' \text{ inj},\ (a', b') \text{ jointly surjective}\}$
$\mathrm{Shift}(b, \neg c) := \neg\mathrm{Shift}(b, c),\ \mathrm{Shift}(b, \wedge_{i\in I} c_i) := \wedge_{i\in I}\mathrm{Shift}(b, c_i).$



$\mathrm{L}(p, \mathrm{true}) := \mathrm{true}.$
$\mathrm{L}(p, \exists\,(a, \mathrm{ac})) := \exists\,(a', \mathrm{L}(p', \mathrm{ac}))$ if p^{-1} is applicable w.r.t. the morphism a, $p' := \langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ is the *derived* rule, and false, otherwise.
$\mathrm{L}(p, \neg\mathrm{ac}) := \neg\mathrm{L}(p, \mathrm{ac}),\ \mathrm{L}(p, \wedge_{i\in I}\mathrm{ac}_i) := \wedge_{i\in I}\mathrm{L}(p, \mathrm{ac}_i).$

For application conditions ac over L, $\mathrm{C}_\exists(\mathrm{ac}) := \exists\,(\emptyset \hookrightarrow L, \mathrm{ac})$[4].

$\mathrm{Appl}(\varrho) = \mathrm{Dang}(p) \wedge \mathrm{ac}$ and $\mathrm{Dang}(p) = \wedge_{a\in A} \nexists\ a$ where A ranges over all minimal morphisms $a\colon L \hookrightarrow L'$ such that $\langle K \hookrightarrow L, a\rangle$ has no pushout complement. The latter condition expresses the *dangling condition* (e.g., see [EEPT06])[5].

Example 2. For the rule `AddTok` and the constraint 2tok in Example 1, the constraint $\mathrm{Wlp}_\exists(\texttt{AddTok}, \mathrm{2tok})$ is constructed as follows: Shift the constraint 2tok over the morphism b from the empty graph to the right-hand side of `AddTok`, shift the obtained right application condition over the rule 2tok to the left (L), and transform the obtained left application condition to a constraint ($\mathrm{C}_\exists$).

$$
\begin{aligned}
(1)\ \mathrm{Shift}(b, \mathrm{2tok}) &= \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{token}} \boxed{\mathrm{tk}}\ \boxed{\mathrm{tk}} \xleftarrow{\mathrm{token}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{token}} \boxed{\mathrm{tk}}) \vee \ldots \\
&\quad \vee \exists\,(\boxed{\mathrm{tk}} \xleftarrow{\mathrm{token}} \boxed{\mathrm{Pl}} \xrightarrow{\mathrm{token}} \boxed{\mathrm{tk}}) = \mathrm{ac}_R \\
(2)\ \mathrm{L}(\texttt{AddTok}, \mathrm{ac}_R) &= \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{token}} \boxed{\mathrm{tk}}) = \mathrm{ac}_L \\
(3)\ \mathrm{C}_\exists(\mathrm{ac}_L) &= \exists\,(\boxed{\mathrm{Pl}}, \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{token}} \boxed{\mathrm{tk}}) \equiv \exists\,(\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{token}} \boxed{\mathrm{tk}}) \\
&= \mathrm{Wlp}_\exists(\texttt{AddTok}, \mathrm{2tok})
\end{aligned}
$$

[3] For a rule $\varrho = \langle p, \mathrm{ac}\rangle$, $\varrho' = \langle \varrho, \mathrm{ac}'\rangle$ denotes the rule $\langle p, \mathrm{ac} \wedge \mathrm{ac}'\rangle$.

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

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

The conditions mean that there exists (1) a place with a token (the comatch of the rule) and a place with two tokens or ... or a place with two tokens, (2) a place (the match of the rule) with one token, and (3) a place with one token, respectively.

Definition 6 (normal form[6]). A condition c is in *normal form* (NF), if every negation symbol is innermost, i.e. every negated subcondition is positive, i.e., of the form $\exists\, C$, every Boolean subcondition is in normal form, and there is no subcondition true or false in c, unless c is true or false. A condition is in *disjunctive* normal form (DNF) if it is a Boolean formula over positive conditions in DNF.

Fact 2. For every condition, an equivalent condition in NF can be effectively constructed. In general, there must not exist an equivalent condition in DNF.

Example 3. The condition $\forall(P, \vee_{i\in I}\exists\, C_i)$ is in NF, but there is no equivalent condition in DNF.

Remark. While a condition in normal form is a disjunction of arbitrary conditions in normal form, a condition in disjunctive normal form is a disjunction of conjunctions of atomic conditions of the form $\exists\, C$ and $\nexists\, C$.

Containment of conditions is based on the existence of an injective morphism.

Definition 7 (containment). The *containment* of conditions d_1 and d_2 over P, denoted $d_1 \sqsubseteq d_2$, is defined as follows. For every condition d_2, true $\sqsubseteq d_2$. For $d_i = \exists\, (P \overset{a_i}{\hookrightarrow} C_i, c_i)$ $(i = 1, 2)$, $d_1 \sqsubseteq d_2$ if there is an injective morphism $b\colon C_1 \hookrightarrow C_2$ such that $b \circ a_1 = a_2$ and $\mathrm{Shift}(b, c_1) \sqsubseteq c_2$. For $d_i = \neg c_i$ $(i = 1, 2)$, $d_1 \sqsubseteq d_2$ if $c_2 \sqsubseteq c_1$. For $d_i = \wedge_{j\in I_i} c_{ij}$ $(i = 1, 2)$, $d_1 \sqsubseteq d_2$ if there is an injective function $f\colon I_1 \hookrightarrow I_2$ such that $c_{1j} \sqsubseteq c_{2f(j)}$ for all $j \in I_1$. A constraint c_2 *subsumes* c_1, $c_2 \sqsupseteq c_1$, if c_1 contains c_2. A constraint c_1 *properly* contains in c_2 (c_2 *properly* subsumes c_1), $c_1 \sqsubset c_2$ ($c_2 \sqsupset c_1$), if the injective morphism is not an isomorphism.

Lemma 1. For conditions d_1, d_2, if $d_1 \sqsubseteq d_2$, then $d_2 \Rightarrow d_1$.

Proof. By induction over the size of nested conditions. □

Definition 8 (minimization). Let $c = \bigvee_{i\in I} c_i$ be a finite disjunction of conditions. Then $\mathrm{Min}(c) = \bigvee_{i\in I'} c_i$ where $I' = \{i \in I \mid \nexists\, c_j.\, c_j \sqsubset c_i \text{ for all } j \in I\}$.

Lemma 2. For disjunctive conditions c, $\mathrm{Min}(c) \equiv c$.

Proof. "⇒". Let $p \models \mathrm{Min}(c) = \vee_{i\in I'} c_i$. Then $p \models \vee_{i\in I} c_i$ since $I' \subseteq I$.
"⇐". Let $p \models \neg\mathrm{Min}(c) = \wedge_{i\in I'} \neg c_i$. By definition of the semantics, $p \models \neg c_i$ for all $i \in I'$. It remains to show $p \models \neg c_i$ for all $i \notin I'$. For $i \notin I'$, there is some index $j \in I'$ such that $c_j \sqsubseteq c_i$. By Lemma 1, we have $c_i \Rightarrow c_j \equiv \neg c_j \Rightarrow \neg c_i$. Now $p \models \neg c_j$ implies $p \models \neg c_i$. Thus, $p \models \neg c_i$ for all $i \in I$. Consequently, $\mathrm{Min}(c) \equiv c$. □

[6] Karl-Heinz Pennemann. Generalized constraints and application conditions for graph transformation systems. Diploma thesis, University of Oldenburg, 2004.

4 Filtering Through Constraints

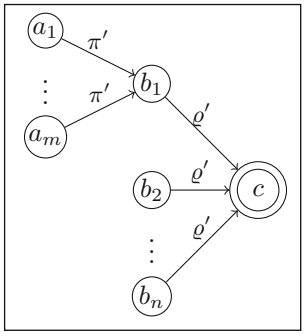
In this section, we investigate the filter problem for graph grammars and constraints. We introduce so-called constraint automata, present a backward construction for constraint automata, sketch some closure properties closely related to the ones in formal-language theory, consider the termination of the backward construction, and derive a goal-oriented graph grammar from the constraint automaton. The section is concluded by a Filter Theorem, summarizing the result.

4.1 Constraint Automata

In the following, we introduce constraint automata based on finite automata having constraints as states. Given a graph grammar, a constraint automaton consists of a finite automaton, with constraints as states and rules as input symbols. The graph language of the automaton is the set of all graphs derived from the start graph by applying the sequences of rules accepted by the finite automaton.

Definition 9 (constraint automaton). The (constraint) automaton for a graph grammar $GG = (\mathcal{R}, S)$ is a tuple $\mathcal{A} = (A, S)$ where $A = (\mathcal{C}, \mathcal{R}', \rightarrow, C_0, F)$ is a (finite) automaton, $\mathcal{C}$ a set of states (constraints), $\mathcal{R}'$ a finite set of $\mathcal{R}$-restricting rules[7], $\rightarrow$ a transition relation, $C_0, F \subseteq \mathcal{C}$ sets of initial and final states, respectively, and S a start graph. The automaton is *goal-oriented* if there is a terminating transition relation $\rightarrow_t \subseteq \rightarrow$ such that, for all initial states and all reachable states, a final state is reachable using transitions from $\rightarrow_t$. The *graph language* of $\mathcal{A}$ is $L(\mathcal{A}) = \{G \mid \exists S \Rightarrow_w G \text{ for some } w \in L(A)\}$ where $L(A)$ is the set of all strings accepted by the finite automaton A.

4.2 Backward Construction



We construct a constraint automaton based on the construction of existential weakest liberal preconditions. Intuitively, the constraint automaton is constructed as follows. Starting with a constraint c, for every rule ϱ, we construct the existential weakest liberal precondition $\mathrm{Wlp}_{\exists}(\varrho, c)$, bring it into normal form (which may be seen as a disjunction of constraints), minimize it, and, for each constraint b in the disjunction, add b to the constraint (and state) set and $b \rightarrow_{\varrho'} c$ with $\varrho' = \mathrm{gua}(\varrho, c)$ to the transition relation. We ignore constraints b subsuming some constraint in $\mathcal{C}$. The constructed constraints become the states of the automaton, the constraint c the final state, and the constraints which are satisfied for the start graph S become the initial states.

[7] A rule ϱ is $\mathcal{R}$-*restricting* if ϱ' is ϱ-restricting for some $\varrho \in \mathcal{R}$.

Assumption. By Fact 2 and Lemma 2, we may assume that the existential weakest liberal preconditions are in normal form and are minimized.

For a constraint set $\mathcal{C}$, $\mathrm{Max}(\mathcal{C})$ denotes the set of constraints b in $\mathcal{C}$ that are maximal with respect to $\rightarrow$, i.e., there is no constraint $b' \in \mathcal{C}$ such that $b \rightarrow b'$.

Construction 2 (backward construction with containment test). Given a grammar GG and a constraint c, we construct constraint automata as follows.

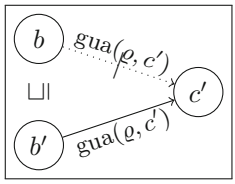
(1) Construct a sequence $\mathcal{C}_0, \mathcal{C}_1, \ldots$ of constraint or state sets by $\mathcal{C}_0 = \{c\}$ and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{b \text{ in } \mathrm{Wlp}_\exists(\varrho, c') \mid \varrho \in \mathcal{R}, c' \in \mathrm{Max}(\mathcal{C}_i) \wedge \nexists\, b' \in \mathcal{C}_i . b \sqsupseteq b'\}$ (For a rule and a maximal constraint of the set $\mathcal{C}_i$, we construct the existential weakest liberal precondition, bring it into normal form, minimize it, and add the components of the disjunction to the set C_{i+1} provided that they do not subsume another constraint of $\mathcal{C}_i$.) For a disjunction $c' = \vee_{i \in I} c_i$, b in c' abbreviates $b \in \{c_i \mid i \in I\}$.

(2a) For $i \geq 0$, let $\mathcal{A}_{c,i} = (A_i, S)$ be the automaton with $A_i = (\mathcal{C}_i, \mathcal{R}', \rightarrow, C_0, \{c\})$, $\mathcal{R}' = \{\mathrm{gua}(\varrho, b) \mid \varrho \in \mathcal{R}, b \in \mathcal{C}_i\}$. For $\varrho \in \mathcal{R}$, $c' \in \mathcal{C}_i$, if b in $\mathrm{Wlp}_\exists(\varrho, c')$ and $b \sqsupseteq b' \in \mathrm{Max}(\mathcal{C}_i)$, then $b' \rightarrow_{\mathrm{gua}(\varrho,c')} c'$ is in $\rightarrow$. (If the constraint b is a component of the existential weakest liberal precondition of a rule ϱ relative to the constraint c' and b subsumes a maximal constraint of $\mathcal{C}_i$, then there is a transition from b' to c' labelled with $\mathrm{gua}(\varrho, c')$.) $C_0 = \mathrm{Max}(\{c_0 \in \mathcal{C} \mid S \models c_0\})$ and $\{c\}$ are the sets of initial and final states, respectively.

(2b) If $\mathcal{C}_i = \mathcal{C}_{i+1}$ for some $i \geq 0$, let $\mathcal{C} = \mathcal{C}_i$ and $\mathcal{A}_c = \mathcal{A}_{c,i}$.

Remark

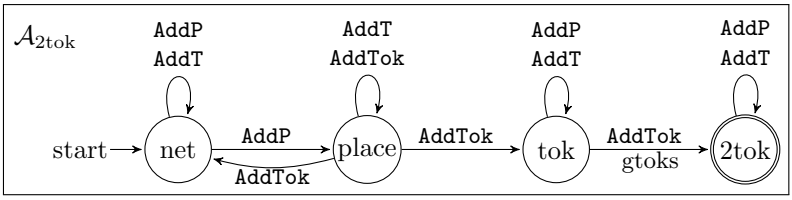
1. A constraint $b \in \mathcal{C}$ represents the class $[\![b]\!]$ of all graphs satisfying b.
2. By the $\mathrm{Wlp}_\exists(\varrho, c)$-construction, we obtain the existence of a direct transformation $G \Rightarrow_\varrho H$ such that $H \models c$. By the $\mathrm{gua}(\varrho, c)$-construction, we filter all those direct transformations that guarantee c: $G \Rightarrow_{\mathrm{gua}(\varrho,c)} H$ such that $H \models c$. In this way, for all direct transformations $G \Rightarrow_{\mathrm{gua}(\varrho,c)} H \wedge H \models c$.
3. An essential step for restricting the state exploration is the following. Whenever a constraint c' is in the constraint set and $b \in \mathrm{Wlp}_\exists(\varrho, c')$ is a constraint such that b subsumes a maximal constraint b' in the constraint set, then the constraint b and an edge from b to c' with label $\mathrm{gua}(\varrho, c')$ is not inserted; instead an edge from b' to c' with label $\mathrm{gua}(\varrho, c')$ is inserted.



4. The set of initial states may be empty; in this case, $L(\mathcal{A}_c)$ is empty.
5. For $i \geq 0$, $\mathcal{A}_{c,i} \subseteq \mathcal{A}_{c,i+1}$ and $L(\mathcal{A}_{c,i}) \subseteq L(\mathcal{A}_{c,i+1})$.
6. In general, the sequence $\mathcal{C}_0, \mathcal{C}_1, \ldots$ need not become stationary, hence the procedure need not terminate, but we can show that it terminates for our specific cases.

Example 4. Consider the grammar with the rules `AddP`, `AddT`, and `AddTok` in Example 1 and the positive constraint 2tok = $\exists$ (tk $\xleftarrow{\text{token}}$ Pl $\xrightarrow{\text{token}}$ tk) meaning

that there exists a place with two tokens. Then the backward construction with containment test yields a finite automaton $\mathcal{A}_{2\text{tok}}$ given below.



The automaton $\mathcal{A}_{2\text{tok}}$ is constructed as follows. Start with the constraint 2tok, construct, for every rule p in the grammar and every constraint b', the weakest liberal precondition $\text{Wlp}_{\exists}(p, b')$, and subsume constraints. Continue with the new constraints in the constraint set. The construction terminates. The components of the weakest liberal preconditions build the set $\mathcal{C} = \{\text{net}, \text{place}, \text{tok}, \text{2tok}\}$ where net = ∃ (PN), place = ∃ (Pl), and tok = ∃ (Pl $\xrightarrow{\text{token}}$ tk). The constraint $\text{Wlp}_{\exists}(\texttt{AddTok}, \text{net})$ = ∃ (PN Pl) is ignored because it is subsumed by place.

The construction of the guaranteeing application condition Gua(AddTok, 2tok) yields gtoks = ∃ (Pl tk $\xleftarrow{\text{token}}$ Pl $\xrightarrow{\text{token}}$ tk) ∨ ∃ (Pl $\xrightarrow{\text{token}}$ tk)[8], meaning that outside of the match of AddTok, there is a place with two tokens or at the place in consideration, there is one token. The initial state is net because PN $\models$ net and the final state is 2tok.

Proposition 2. For arbitrary graph grammars GG and arbitrary constraints c, the constraint automata $\mathcal{A}_{c,i}$ and $\mathcal{A}_c$ are goal-oriented and

1. $L(\mathcal{A}_{c,i}) \subseteq L(GG) \cap [\![c]\!]$ $(i \geq 0)$ and
2. $L(\mathcal{A}_c) = L(GG) \cap [\![c]\!]$ in case of termination.

The proof is based on the following lemma which relates paths in the constraint automaton $\mathcal{A}_{c,i}$ and transformations in the graph grammar GG.

Lemma 3 (correctness & completeness). Let GG be a graph grammar, c a graph constraint, and $\mathcal{A}_{c,i}$ the constructed automaton. Let $w' \in \mathcal{R}'^*$ and $w \in \mathcal{R}^*$ be the sequences of restricted and underlying rules, respectively.

1. For all paths $g \to_{w'} h$ in $\mathcal{A}_{c,i}$ $(i \geq 0)$ and all graphs $G \models g$, there is a graph H and a transformation $G \Rightarrow_w H$ in GG such that $H \models h$.
2. In the terminating case, for all transformations $G \Rightarrow_w H$ in GG such that $H \models h \in \text{Max}(\mathcal{C})$, there is a path $g \to_{w'} h$ in $\mathcal{A}_c$ such that $G \models g \in \text{Max}(\mathcal{C})$.

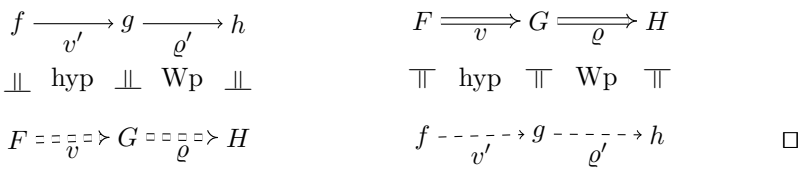
Proof. By induction on the length of the path/transformation.

1. By induction on the structure of w'. Let $f \to_{w'} h$ and $F \models f$.
   Induction basis. For $w' = \varepsilon$, let $h = f$. Then $F \Rightarrow^0 H \models h$.
   Induction step. For $w' = v'\varrho'$, $f \to_{v'\varrho'} h$ can be decomposed into

[8] The match of the rule is marked in a blue color.

$f \to_{v'} g \to_{\varrho'} h$ where $\varrho' = \mathrm{gua}(\varrho, h)$. By induction hypothesis, there is a transformation $F \Rightarrow_v G$ such that $G \models g$. By $g \in \mathrm{Wlp}_\exists(\varrho, h)$, we have $G \models \mathrm{Wlp}_\exists(\varrho, h)$. By Fact 1, there is a direct transformation $G \Rightarrow_\varrho H$ such that $H \models h$. Composing the transformations, we obtain a transformation $F \Rightarrow_{v\varrho} H$ such that $H \models h$.

2. By induction on the structure of w. Let Construction 2 be terminating and $\mathcal{A}_c$ the resulting constraint automaton. Let $F \Rightarrow_w H$ and $H \models h \in \mathrm{Max}(\mathcal{C})$.
Induction basis. For $w = \varepsilon$, $F \cong H$, $F \Rightarrow_\varepsilon H$ and $F \models f = h \in \mathrm{Max}(\mathcal{C})$.
Induction step. For $w = v\varrho$, the transformation $F \Rightarrow_{v\varrho} H$ can be decomposed into $F \Rightarrow_v G \Rightarrow_\varrho H$. By Fact 1 and Lemma 2, $G \Rightarrow_\varrho H.H \models h$ implies $G \models \mathrm{Wlp}_\exists(\varrho, h)$. Consequently, there is some $g' \in \mathrm{Wlp}_\exists(\varrho, h).G \models g'$. By Construction 2, there is some $g \in \mathrm{Max}(\mathcal{C}).g' \sqsupseteq g$ and, by Lemma 1, $g' \Rightarrow g$. By the definition of $\Rightarrow$, $G \models g'$ implies $G \models g$. By Construction 2 and $g \in \mathrm{Max}(\mathcal{C})$, there is a transition $g \to_{\varrho'} h$ in $\mathcal{A}_c$. By $F \Rightarrow_v G.G \models g \in \mathrm{Max}(\mathcal{C})$, the induction hypothesis can be applied yielding a path $f \to_{v'} g$ in $\mathcal{A}_c$ such that $F \models f \in \mathrm{Max}(\mathcal{C})$. Composing the paths, we obtain a path $f \to_{w'} h$ such that $F \models f \in \mathrm{Max}(\mathcal{C})$.



□

Proof (of Proposition 2). The statements follow immediately from Lemma 3:

1. If $G \in L(\mathcal{A}_{c,i})$, then $S \Rightarrow_w G$ for some $w' \in L(A_i)$. Then there is a path $c_0 \to_{w'} c$ from $c_0 \in C_{0,i}$ to c. Since $w' = \varepsilon$ ($S \cong G$ and $c_0 = c$) or the last rule in w is c-guaranteeing, we have $G \models c$. Thus, $G \in L(GG) \cap [\![c]\!]$.
2. In case Construction 2 terminates, the first inclusion follows from the first statement. The second inclusion is as follows. If $G \in L(GG) \cap [\![c]\!]$, then there is a transformation $S \Rightarrow_w G$ in GG such that $G \models c$. By Lemma 3, there is a path $c_0 \to_{w'} c$ in $\mathcal{A}_c$ such that $S \models c_0$. By construction, $c_0 \in C_0$. Thus, $w \in L(A)$ and $G \in L(\mathcal{A}_c)$.
3. The constraint automata are goal-oriented: By the backward construction, the automata are connected and all directed paths end in the final state. Let now $\mathcal{T}$ be a spanning-tree, i.e. a subautomaton which is both a tree and which contains all the states of the automaton. Then the transition relation $\to_t := \to_{\mathcal{T}}$ is terminating and, for all reachable states, there is a path to the final state.

□

4.3 Closure Properties

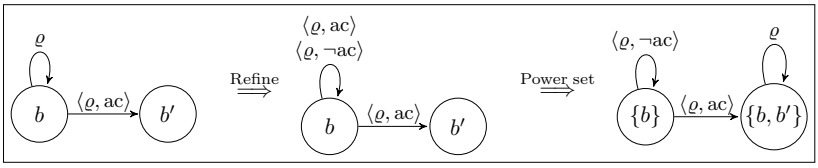
Similar to formal-language theory, we define deterministic constraint automata and transform nondeterministic constraint automata into deterministic ones. Moreover, it can be shown that constraint automata are closed with respect to the Boolean operations complementation and product construction.

Definition 10 (deterministic automata). A constraint automaton $\mathcal{A} = (A, S)$ is *deterministic* if for each constraint $b \in \mathcal{C}$ and each rule ϱ in the automaton, the application conditions in $b \to_{\langle \varrho, \mathrm{ac}_i \rangle} b_i$ are disjoint[9].

Proposition 3. For every constraint automaton $\mathcal{A}$, a deterministic automaton $\mathcal{A}'$ can be effectively constructed such that $L(\mathcal{A}) = L(\mathcal{A}')$.

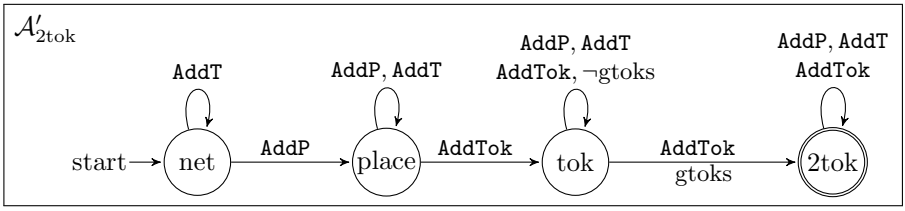
Construction 3. By a refined power-set construction. Let AC be the set of application conditions occurring in the automaton and AC′ the refined set such that all application conditions are disjoint. Then the original application conditions can be seen as a conjunction of non-nested application conditions over AC′. Refine each transition $b \to_{\langle \varrho, \mathrm{ac} \rangle} b'$ by the transitions $b \to_{\langle \varrho, \mathrm{ac}_i \rangle} b'$ with $\mathrm{ac} = \bigwedge \mathrm{ac}_i$, $\mathrm{ac}_i \in \mathrm{AC}'$ and apply the power-set construction.

Remark. In most of our examples, we have an unrestricted rule and a rule restricted by an application condition ac. By the refined power-set construction, we get the following.



Different transitions between the same constraints are drawn by one line.

Example 5 (power-set construction). By the power-set construction, the automaton $\mathcal{A}_{2\mathrm{tok}}$ can be transformed into an equivalent deterministic automaton $\mathcal{A}'_{2\mathrm{tok}}$.



The automation $\mathcal{A}'_{2\mathrm{tok}}$ is deterministic: the restricted rules $\langle \texttt{AddTok}, \mathrm{gtoks} \rangle$ and $\langle \texttt{AddTok}, \neg\mathrm{gtoks} \rangle$ are distinct because the application conditions gtoks and ¬gtoks are distinct.

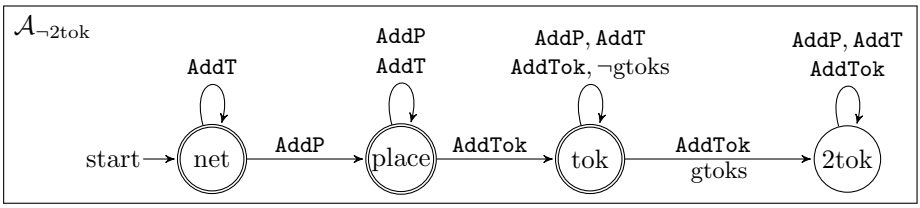
Proof. The refinement step and the power-set construction do not change the semantics of the constraint automaton. By the power-set construction, the automaton is deterministic. □

[9] Two application conditions ac and ac′ are *disjoint* if the sets $[\![\mathrm{ac}]\!]$ and $[\![\mathrm{ac}']\!]$ are disjoint. $[\![\mathrm{ac}]\!] = \{g \mid g \models \mathrm{ac}\}$ denotes the semantics of the application condition ac.

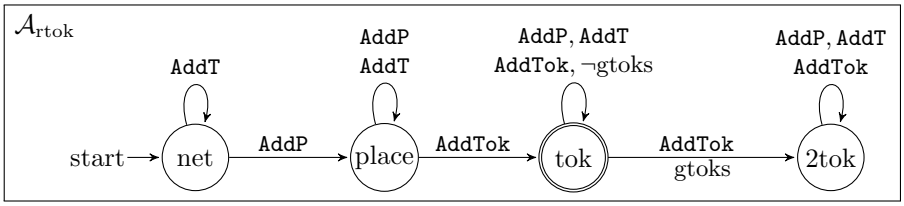
Proposition 4. Given deterministic constraint automata $\mathcal{A}_c$ and $\mathcal{A}_{c,i}$ $(i \in I)$, constraint automata $\mathcal{A}_{\neg c}$ and $\mathcal{A}_{\wedge_{i\in I} c_i}$ can be constructed, respectively.

Proof idea. By the complement and a slightly modified product construction.

Example 6 (complement construction). For the constraint ¬2tok, the constraint automaton $\mathcal{A}_{\neg 2\text{tok}}$ is constructed from the deterministic automaton $\mathcal{A}'_{2\text{tok}}$ according to the complement construction.



Example 7 (product construction). Let $\mathcal{A}'_{\text{tok}}$ be the deterministic automaton obtained from the automaton $\mathcal{A}'_{2\text{tok}}$ by deleting the state 2tok and all incident edges. For the constraint rtok = tok ∧ ¬2tok, the automaton $\mathcal{A}_{\text{rtok}}$ is constructed from the automata $\mathcal{A}'_{\text{tok}}$ and $\mathcal{A}'_{\neg 2\text{tok}}$ according to the product construction.



4.4 Termination

The question remains, under which assumptions the backward construction terminates. Unfortunately, in general, for non-deleting graph grammars and positive constraints, the construction does not terminate. Adding the requirement "n-bounded path", a slightly modified backward construction terminates.

Definition 11. A graph grammar is *non-deleting* if all underlying rules are non-deleting, i.e., for $\langle L \hookleftarrow K \hookrightarrow R\rangle$, $L \cong K$. A graph grammar is *n-bounded path* if all generated graphs are n-bounded path, i.e. all paths have length less than or equal to n. A graph constraint of the form $\exists\, C$ is *positive* and *n-bounded path*, if C is n-bounded path.

Proposition 5 (termination).

1. For non-deleting graph grammars GG and positive constraints c, the backward construction, in general, does not terminate.
2. If, additionally, GG is n-bounded path, there exists a terminating, slightly modified backward construction.

Construction 4 (backward construction with bounded-path test). Modify Construction 2 by replacing step (1) by step (1'): Construct a sequence $\mathcal{C}_0, \mathcal{C}_1, \ldots$ of constraint or state sets by $\mathcal{C}_0 = \{c\}$ and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{b \in \mathrm{Wlp}_{\exists}(\varrho, c') \mid \varrho \in \mathcal{R},\ c' \in \mathcal{C}_i \wedge \nexists b' \in \mathcal{C}_i . b \sqsupseteq b', b \text{ is n-bounded path}\}$.

Example 8 For the 1-bounded path grammar with the rule `AddTrans` = $\bullet\!\to\!\bullet\!\to\!\bullet \Rightarrow \bullet\overset{\frown}{\to\bullet\to}\bullet$ and the constraint $c = \exists$  , Construction 2 does not terminate: Starting with the constraint c, it produces an infinite sequence of positive constraints $\exists\, C_n$ requiring the existence of a path of length n with two loops at the start and end node. Since there does not exist an injective morphism from C_n to C_{n+1}, the constraint $\exists\, C_{n+1}$ is not contained in $\exists\, C_n$. In contrast, Construction 4 terminates.

Proof. 1. The statement follows immediately from the undecidability of the coverability problem for non-deleting graph grammars [BDK+12][10].
The *Coverability Problem* is as follows. Given a graph grammar $GG = (\mathcal{R}, S)$ and a graph C, is there a graph H such that $S \Rightarrow^*_{\mathcal{R}} H$ and $C \sqsubseteq H$, i.e. there is an injective morphism from C to H. Assume the automaton construction terminates for non-deleting graph grammars GG and positive constraints $c = \exists\, C$ with automaton $\mathcal{A}_c$. Then, the coverability problem for GG and final graph C is decidable: If, in $\mathcal{A}_c$, there is a path from an initial state to the final state c, then the output is yes, and no, otherwise. Contradiction [BDK+12, Proposition 13].

2. Correctness and completeness of Construction 4 follows from the correctness and completeness of Construction 2. Termination follows from the well-known fact, that the subgraph relation for n-bounded path graphs is a well-quasi-order[11] for n-bounded path graphs [Din92] and the constraint system is monotone[12] with respect to the subgraph relation [AJ01].
We use the subgraph relation $\sqsubseteq$ on graphs with $C_1 \sqsubseteq C_2$ if there exists an injective morphism from C_1 to C_2. This corresponds directly to the containment relation on positive constraints. Since the subgraph relation is a well-quasi-order for n-bounded path graphs, the containment relation is.
Let G, H be graphs, p be a rule and G is a subgraph of H. Let p be applicable on G, then there exists an injective morphism $g : L \to G$ from the left-hand side of p to G. Since G is a subgraph of H there exists an injective morphism $m : G \to H$. Composing g and m, we obtain the morphism h from L to H. The application of p replaces L by R in both graphs G and H resulting in G'

[10] In [BDK+12], a slight extension of the single-pushout approach is considered, but the simulation of a deterministic Turing machine is done by a non-deleting double-pushout graph transformation system, see, e.g., [EHK+97].

[11] A binary relation $\preceq$ defined on a set Q is a *quasi-ordering* [Din92] if it is reflexive and transitive. A sequence $q_1, q_2, \ldots$ of members of Q is called a *good sequence* (with respect to $\preceq$) if there exist $i < j$ such that $q_i \preceq q_j$. It is a *bad sequence* if otherwise. We call $(Q, \preceq)$ a *well-quasi-ordering* (or a *wqo*) if there is no infinite bad sequence.

[12] Let T be a transition system with a preorder $\preceq$ defined on its states. T is monotone wrt. to $\preceq$ if, for any states c_1, c_2 and c_3, with $c_1 \preceq c_2$ and $c_1 \to c_3$, there exists a state c_4 such that $c_3 \preceq c_4$ and $c_2 \to c_4$.

and H', respectively. Since p is non-deleting, it follows that G' is a subgraph of H', thus the constraint system is monotone wrt. the subgraph relation. □

4.5 Filtering

We construct a graph grammar from a constraint automaton and derive our main theorem: the Filter Theorem for graph grammars and constraints.

Proposition 6 (from constraint automata to graph grammars [Bec16]). For every (goal-oriented) constraint automaton $\mathcal{A}_c$, a (goal-oriented) graph grammar GG_c can be constructed effectively such that $L(GG_c) = L(\mathcal{A}_c)$.

Construction 5. For simplicity, we give the construction for deterministic automata. For non-deterministic automata, the construction is similar.
Let $\mathcal{A}_c = (A, S)$ be a deterministic constraint automaton with the underlying automaton $A = (\mathcal{C}, \mathcal{R}', \delta, C_0, \{c\})$. Then the graph grammar $GG_c = (\mathcal{C}, \mathcal{R}'_c, S)$ is constructed as follows. The nonterminals of the grammar are the constraints in $\mathcal{C}$. The rule set $\mathcal{R}'_c$ is induced by the transition function δ: for instruction $\delta(c_1, \varrho) = c_2$ with rule $\varrho = \langle L \Rightarrow R, \mathrm{ac}\rangle$, we create a new rule $\langle L' \Rightarrow R', \mathrm{ac}'\rangle$ where the states c_1 and c_2 are integrated into the left- and the right-hand side, respectively, and the constraint ac is shifted to L'. In more detail, let $\mathcal{R}'_c = \{S \Rightarrow S + \textcircled{c_0} \mid c_0 \in C_0\} \cup \{\langle c_1, \varrho, c_2\rangle \mid \delta(c_1, \varrho) = c_2\} \cup \{\textcircled{c} \Rightarrow \emptyset\}$ where the start rules add an initial state c_0, the simulating rules $\langle c_1, \varrho, c_2\rangle = \langle L + \textcircled{c_1} \Rightarrow R + \textcircled{c_2}, \mathrm{ac}'\rangle$ with $\mathrm{ac}' = \mathrm{Shift}(L \hookrightarrow L + \textcircled{c_1}, \mathrm{ac})$ simulate the working in the automaton, and the deleting rule allows to terminate.

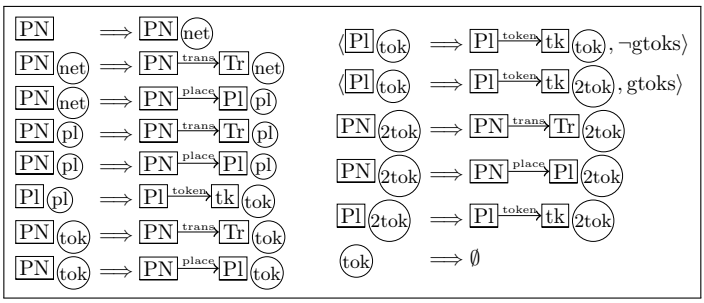
Remark. Since $\mathcal{A}_c$ is a goal-oriented automaton, the resulting grammar is, too.

Theorem 1 (Filter Theorem). For arbitrary graph grammars GG, arbitrary graph constraints c, and $i \geq 0$, goal-oriented graph grammars $GG_{c,i}$ (GG_c) can be constructed such that

1. $L(GG_{c,i}) \subseteq L(GG) \cap [\![c]\!]$ and
2. in case of termination, $L(GG_c) = L(GG) \cap [\![c]\!]$.

Proof. The theorem follows immediately from Propositions 2, 5, and 6. □

Example 9 (from automaton to grammar). The graph grammar GG_{rtok} can easily be derived from $\mathcal{A}_{\mathrm{rtok}}$. The states net, pl, tok, 2tok become the nonterminal symbols of the grammar, pl abbreviates place. The rules are as follows.



General Remark (typed attributed graphs). All results in this paper can be obtained for typed attributed graphs: For the construction of existential weakest liberal preconditions, $\mathcal{E}' - \mathcal{M}$ pair factorization and the existence of $\mathcal{M}$-pushout is used. Typed attributed graphs and morphisms form a category that has these properties where $\mathcal{E}'$ and $\mathcal{M}$ are the classes of all jointly surjective pairs of all type-strict injective morphisms, respectively [RAB+15].

5 Related Concepts

Model generation. Most approaches to instance generation are *logic-oriented*, e.g., [CCR07,KG12]. They translate class models with OCL constraints into logical facts and formulas, such as Alloy [Jac12]. Then, an instance can be generated or it can be shown that no instances exist.

Alternatively, *graph grammars* have been shown to be suitable and natural to specify (domain-specific) visual languages in a constructive way. In [AHRT14, RAB+15], we translate OCL constraints to graph constraints. To formally treat meta-models (without OCL constraints) they are translated to instance and type graphs. Hence, we follow the graph-based approach keeping the graph structure of models as units of abstraction where graph axioms are satisfied by default. Meanwhile, Bergmann [Ber14] has implemented a translator of OCL constraints to graph patterns. which is rather an efficient implementation, than a formal translation.

In [DVH16], instances of meta-models are generated by refining a partial instance model in multiple steps. For that, the meta-model is pruned and the constraints are approximated to evaluate them on partial meta-models. The partial instances for the pruned meta-model are stepwise extended, where each step is a call to an underlying logic solver.

In [SLO17], a parallelizable symbolic model generation algorithm delivering a set of symbolic models is given. For the grammar generating all graphs, the algorithm can be used to generate all graphs satisfying the constraint.

Integration of constraints. In Radke et al. [RAB+15], the integration of a graph constraint c into a graph grammar is done by replacing the rules of the grammar by the corresponding c-guaranteeing rules. In this paper, we refine this approach, construct a set of auxiliary constraints by backward construction and, in the case of termination, obtain a c-filtering grammar.

In Becker [Bec16, Theorem 1], it is shown that the filter problem is solvable for arbitrary graph grammars GG and arbitrary graph constraints: The used method is generate & test: first, generate a graph and, then, test whether the constraint is satisfied. Unfortunately, the grammar is not goal-oriented. In [Bec16, Theorems 2 and 3], it is shown that the filter problem is solvable by a goal-oriented grammar. The statement is based on a backward construction similar to ours. [Bec16, Algorithm 1] assumes the existence of a "disjunctive normal form" for constraints. In general, for arbitrary constraints, there is only a normal form respecting Definition 6. Algorithm 1 makes use of equivalence and

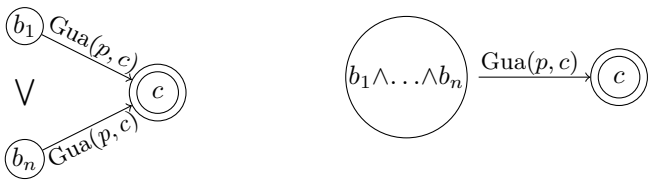
implication, i.e., it is not effective. In this paper, we use of minimization and containment which can be easily constructed and checked and implies equivalence and implication, respectively.

	Grammar	Constraint	Relation	Comments
[RAB+15]	arb	arb	$\subseteq$	Weakest precond
[Bec16, Theorem 1]	arb	arb	$=$	Generate & Test
[Bec16, Theorems 2 and 3]	arb	arb	$\subseteq$ / $=$	Goal-oriented
This paper	arb	arb	$\subseteq$ / $=$	Goal-oriented
This paper	non-del & bp	pos	$=$	Goal-oriented

where arb stands for arbitrary, non-del for plain, non-deleting, bp for bounded path and pos for positive.

6 Conclusion

The backward construction works for arbitrary graph grammars and arbitrary constraints. If the existential weakest liberal precondition $\mathrm{Wlp}_{\exists}(\varrho, c) = \vee_{i \in I} b_i$ is a disjunction of several constraints, there is a proper decomposition into the smaller components b_i which can be handled in the same way. If $\mathrm{Wlp}_{\exists}(\varrho, c)$ is not a disjunction of several constraints, e.g. $\wedge_{i \in I} b_i$, then the complex constraint $\wedge_{i \in I} b_i$ has to be handled.



1. For plain, non-deleting grammars and positive constraints, $\mathrm{Wlp}_{\exists}(p, d)$ is a disjunction of positive constraints and the states in constraint automata are positive constraints.
2. For Boolean formulas over positive conditions, in general, the backward construction yields an automaton with complex constraints. By the closure results in Subsect. 4.3, we obtain automata with positive constraints.

Further topics

We will investigate constraints of the form $\forall(P, \exists\, C)$, e.g. the constraint alltok $= \forall(\boxed{\mathrm{Pl}}, \exists\, (\boxed{\mathrm{Pl}} \xrightarrow{\mathrm{tok}} \boxed{\mathrm{tk}}))$, meaning that all places possess a token. We will generalize constraint automata to program automata (the edges are decorated by graph programs) and the backward construction, we obtain a finite program automaton instead of an "infinite" constraint automaton.

Acknowledgements. We are grateful to Jan Steffen Becker, Berthold Hoffmann, Jens Kosiol, Nebras Nassar, Christoph Peuser, Lina Spiekermann, and Gabriele Taentzer and the anonymous reviewers for their helpful comments to this paper.

References

[AHRT14] Arendt, T., Habel, A., Radke, H., Taentzer, G.: From core OCL invariants to nested graph constraints. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 97–112. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09108-2_7

[AJ01] Abdulla, P.A., Jonsson, B.: Ensuring completeness of symbolic verification methods for infinite-state systems. Theor. Comput. Sci. **256**(1–2), 145–167 (2001)

[BDK+12] Bertrand, N., Delzanno, G., König, B., Sangnier, A., Stückrath, J.: On the decidability status of reachability and coverability in graph transformation systems. In: Rewriting Techniques and Applications (RTA 2012). LIPIcs, vol. 15, pp. 101–116 (2012)

[Bec16] Becker, J.S.: An automata-theoretic approach to instance generation. In: Graph Computation Models (GCM 2016), Electronic Pre-Proceedings (2016)

[Ber14] 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

[CCR07] Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 547–548 (2007)

[Din92] Ding, G.: Subgraphs and well-quasi-ordering. J. Graph Theor. **16**(5), 489–502 (1992)

[DVH16] Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 87–103. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_6

[EEPT06] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2

[EHK+97] Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation. Part II: single-pushout approach and comparison with double pushout approach. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1, pp. 247–312. World Scientific, River Edge (1997)

[HP09] Habel, A., Pennemann, K.-H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**, 245–296 (2009)

[Jac12] Jackson, D.: Alloy Analyzer website (2012). http://alloy.mit.edu/

[KG12] Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 415–431. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_27

[RAB+15] Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential OCL invariants to nested graph constraints focusing on set operations. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 155–170. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_10

[SLO17] Schneider, S., Lambers, L., Orejas, F.: Symbolic model generation for graph properties. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 226–243. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_13

[Tae12] Taentzer, G.: Instance generation from type graphs with arbitrary multiplicities. Electron. Commun. EASST **47** (2012)

Multi-view Consistency in UML: A Survey

Alexander Knapp[1(✉)] and Till Mossakowski[2]

[1] Universität Augsburg, Augsburg, Germany
knapp@informatik.uni-augsburg.de
[2] Otto-von-Guericke Universität Magdeburg, Magdeburg, Germany

Abstract. We study the question of consistency of multi-view models in UML and OCL. We critically survey the large amount of literature that already exists. We find that only limited subsets of the UML/OCL have been covered so far and that consistency checks mostly only cover structural aspects, whereas only few methods also address behaviour. We also give a classification of different techniques for multi-view UML/OCL consistency: consistency rules, the system model approach, dynamic metamodelling, universal logic, and heterogeneous transformation. Finally, we briefly outline a possible comprehensive distributed semantics approach to consistency.

1 Introduction

Hartmut Ehrig was a researcher whose broad scope of interests ranged from category and automata theory through algebraic specifications and graph grammars to models of concurrency, and in all these fields he achieved fundamental results and contributed far-reaching and novel ideas. It is sad that such a great researcher passed away far too early after his retirement.

One of the many themes of Hartmut Ehrig's research has been the multi-viewpoint integration in the specification of complex systems [16,19]. We here address this problem more specifically in the context of the "Unified Modeling Language" (UML [42]). UML is a complex visual language featuring 14 different diagram types which may be complemented by textual annotations in the "Object Constraint Language" (OCL [41]); both languages are standardised by the Object Management Group (OMG). Already for UML 1.1 van Emde Boas observed "that UML is not a single language but a hybrid of several languages" [18] and Cook et al. [10] coined the notion of UML as "a family of languages". The multitude of diagram types and sub-languages offered by the UML/OCL allows the modeller to reduce the complexity of a model by specifying a system from different viewpoints: data, behaviour, interaction, component architecture, etc. Such "multi-view modelling" or "multi-modelling" and the necessity to integrate views devised from different viewpoints has been intensively discussed in the literature in general by Ehrig et al. [16,19] and others [5], in the software architecture community [11,26,27], in the UML community [7], in the SysML community [39,48], and also in other communities [3,22,43].

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 37–60, 2018.
https://doi.org/10.1007/978-3-319-75396-6_3

A central question in multi-view modelling is whether such a family of UML/OCL diagrams and annotations is still consistent, i.e., conjointly realisable such that all views from all viewpoints are satisfied w.r.t. their (well-defined) semantics [c50][1]. This consistency problem has already been stated in the early UML days [7,20,21], and has been addressed quite broadly in the literature. In particular, several categorisations for partitioning the consistency problem have been designed: Engels et al. [c28] suggest to distinguish between horizontal (or intra-model) and vertical (or inter-model) consistency, i.e., whether the views are on the same level of abstraction; as well as syntactic (structural well-formedness of the abstract syntax) and semantic consistency (compatibility of behaviour). Mens et al. [35] focus more on the intention of sub-languages and give a classification into structural vs. behavioural diagrams and their use on the specification vs. instance level. Allaki et al. [2] combine these schemes into a typological frame of mono- vs. multi-diagram, specification vs. instance, and syntactic vs. semantic, and furthermore add a taxonomy of consistency problems in a terminological dimension, mentioning incompleteness, ambiguity, contradiction, incompatibility, and anomaly. From a verification perspective, Hilken et al. [23] present a list of structural and behavioural verification tasks for UML models considering besides consistency the categories of consequence, independence, executability, and reachability. A structural verification task considers a single (integrated) system state only, whereas a behavioural task pertains to a sequence of states. In contrast to [c50], here "[c]onsistency problems are structural problems and do not involve behaviour" [23, p. 122].

The large number of approaches to multi-view consistency in the literature has also been reviewed and summarised [1,4,17,25,34,49–51] from different perspectives. In particular, Torre et al. [49,50] systematically survey existing consistency rules. They find that most rules are syntactic (88.21% in [49] and 81.89% according to the more comprehensive [50]), and that most of the rules are related to class (71.58%), sequence (47.37%), and state machine diagrams (42.11%). Moreover, they deplore that "it appears that researchers tend to discuss very similar consistency rules, over and over again", and conclude that "much more work is needed to develop consistency rules for all 14 UML diagrams, in all dimensions of consistency (e.g., semantic and syntactic on the one hand, horizontal, vertical and evolution on the other hand)" [49].

In this paper, we first give a comprehensive overview over the existing approaches to multi-view UML/OCL consistency in the literature both for UML 1 and UML 2 starting from the surveys mentioned above. We list which diagram types and sub-languages of UML/OCL are covered by each approach, which consistency technique it applies, whether it tackles structural or behavioural consistency, and which formalism and tool it uses. Our main contribution here is to point out and survey the variety of techniques to a grip on consistency for a heterogeneous, multi-view language like UML/OCL, ranging from syntactic consistency rules over an overarching, semantic system model to

[1] References prefixed with a 'c' refer to the multi-view UML/OCL consistency bibliography assembled in a separate list.

heterogeneous transformations. We find that structural consistency is considered more often by far and that either structural, syntactic consistency rules or an encoding into a system model or some universal logic prevails.

Purely syntactic approaches, in fact, do not help in ensuring behavioural consistency. Also for the semantics-based, behavioural schemes, however, we think that not only the system model, but even the heterogeneous approaches still tend to be too monolithic, not properly reflecting the diversity of UML models. We therefore, extending our previous work on a truly heterogeneous approach to UML/OCL semantics [c13, c44, c45], hint at a possible consistency approach based on so-called *distributed heterogeneity*.

2 Approaches to Multi-view Consistency in UML

Tables 1 and 2 contain an overview of existing approaches to multi-view consistency both for UML/OCL 1 (up to 2004) and UML/OCL 2 (starting in 2005), roughly ordered by their date of appearance. While the literature on UML abounds, for our topic this literature review aims at comprehensiveness. Our starting points were the surveys [1,4,17,25,34,50,51], the information of which we aligned, adapted, and extended by search queries for `"UML" "consistency"` on scholar.google.com and dblp.uni-trier.de as well as personal experiences.

From the 14 different UML diagram types (structural: profile, class, composite structure, component, deployment, object, package; behavioural: activity, sequence, communication, interaction overview, timing, use case, state machine [42, p. 681]), we combined, as usual, the sequence, communication, interaction overview, and timing diagram into the single type of interaction diagram for conciseness; and we omitted the profile, deployment, package, and use case diagram. In fact, package diagrams provide a means for namespace modularisation and the package structure may most of the time be resolved statically using fully qualified names without interference with consistency. Still, also the meaning of packages and their relationships has been discussed [12,47]. Use case diagrams, though besides class diagrams the most used UML diagram type [14,32], convey rather little semantics on their own [24,28], and hence their consistency with other diagrams remains a vague and subjective notion. Deployment diagrams, assigning software artefacts to system elements, also show quite limited semantic content such that only structural consistency checks remain [36]. Finally, profile diagrams are used to define a domain-specific UML extension, thus every instance would add a viewpoint of its own, such that consistency cannot be developed in general and a priori.

Our survey thus covers 11 diagram types and sub-languages of the UML/OCL family, where the entry for interactions condenses the information on four diagram sub-types. The sub-language of the UML for a diagram type is given by the representable abstract syntax (meta-model) fragment, though a consistency

Table 1. Overview of UML/OCL 1 consistency approaches. CD means class diagrams, OD object diagrams, SM state machines, ID interaction diagrams (e.g., sequence diagrams), AD activity diagrams (as a special case of state machines), and OCL the Object Constraint Language. A ● means support for at least a substantial subset of the diagram/sub-language type, a ◑ indicates that the diagram/sub-language is supported but only for a limited subset. The consistency technique of the approach is indicated by an S for "system model", D for "dynamic meta-modelling", U for "universal logic", or T for "heterogeneous transformation". An entry "s" in the class(ification) column means that structural, static consistency checks are supported, a "b" that behavioural, dynamic consistency is checked; if the indicator is set into parentheses, the consistency support is quite restricted. The last column shows the used formalisms and tools. An asterisk in front of the reference indicates that more information is given in Sect. 3.

	Reference	CD	OD	SM	ID	AD	OCL	Cons.	Class.	Form./Tool
*	Egyed [c20, c21]	◑	◑	◑	◑			T	s	ViewIntegra
	Krishnan [c47]			◑	◑	◑		U	(b)	PVS
	Tsiolakis [c77]	◑			◑			U	(s/b)	Graph transformation
*	Große-Rhode [c34, c35]	◑		◑	◑			U	b	Transformation systems
	Reggio et al. [c72]	◑		◑				U	(s/b)	CASL-LTL
	McUmber, Cheng [c60]	◑		◑				U	b	SPIN
*	krtUML [c16]	◑	◑	◑				S	s/b	Symbolic transition systems
	Bernardi et al. [c9]			◑	◑			U	(b)	Petri nets
*	xUML [c61]	●		●	◑	◑	◑	S	(s)	Executable UML
*	Küster et al. [c25, c28]	◑	◑	◑	◑			U	b	CSP/FDR
*	Hausmann et al. [c27]			◑	◑			D	b	Graph transformation
	Spanoudakis, Kim [c76]	◑			◑			U	s	Dempster-Shafer
	Litvak et al. [c55]			●	◑			U	b	BVUML
	Rasch, Wehrheim [c70]	◑		◑				U	s/b	Z, CSP/FDR
*	Wirsing, Knapp [c82]	◑		◑	◑			T	s/b	Universal algebra
	Kyas et al. [c51]	◑		◑			◑	U	s/b	PVS
	van der Straeten [c78, c79]	◑		◑	◑			U	s	Description logic
	Amálio et al. [c3]	◑	◑	◑				U	s	Z
	Kim, Carrington [c43]	◑		◑				U	s	Object-Z
	Diethers, Huhn [c18]			◑	◑			U	b	UPPAAL
	Yang et al. [c83]	◑			●			U	s	rCOS
	Yeung [c85]	◑		◑				U	b	CSP, B

approach may not cover it fully. Since we aim at multi-view consistency involving several viewpoints, we do not list approaches here that only consider a single diagram type. In particular, we leave out the consistency of class diagrams, possibly accompanied by object diagrams (see [8] for an overview) or state machines [33,44,45]. Class diagrams have, however, been the first instance of UML consistency investigations [20,21,29]. For (structural) consistency checks for a homogeneous set of views, like a set of class diagrams, model merging ("consistency-checking-by-merging") is a popular technique [46].

We now first review the variety of techniques enabling consistency checking in the listed multi-view approaches, and then report on our general observations and findings.

Table 2. Overview of UML/OCL 2 consistency approaches. The abbreviations are as in Table 1 extended by CMP for component diagrams and CSD for composite structure diagrams. Activity diagrams (AD) have an independent semantics in UML 2. Protocol state machines are not equipped with a diagram type of their own in UML 2; still, [c31] considers them independently.

	Reference	CD	OD	CMP	CSD	SM	ID	AD	OCL	Cons.	Class.	Form./Tool
	Lam, Padget [c53]					◑	◑			U	b	π-calculus
	Long et al. [c58]	◑					◑			U	s	rCOS
	Lucas et al. [c59]	◑					◑			U	s	Maude
	Okalas et al. [c67]	◑				◑				U	s/b	B
	Rasch, Wehrheim [c71]	◑		◑		◑	◑			U	s/b	Z, CSP/FDR
	Wang et al. [c81]					◑	◑			U	b	LTSA
	Bellur, Vallieswaran [c8]	◑		◑		◑	◑			U	s	Meta-model
	Li et al. [c54, c57]	◑	●			◑	●			U	s/(b)	UTP
	O'Keefe [c68]	◑				◑	◑			U	b	Dynamic logic
	Shinkawa [c74]	◑				◑	◑	◑		U	b	CPN
	Yao, Shatz [c84]					◑	◑			U	b	Petri nets
	Zhao et al. [c86]					◑	◑			U	b	SPIN
	Anastasakis et al. [c4]	●							◑	U	s/(b)	Alloy
*	Gogolla et al. [c31]	●	●			◑	◑		●	U	s/(b)	USE
	Knapp, Wuttke [c46]	◑				●	●			U	b	Hugo/RT
	Sapna, Mohanty [c73]	◑				◑	◑	◑		U	s	SQL
	Brændshøi [c10]					◑	◑			U	b	Impl.
	Banerjee et al. [c5, c6]	◑				●				U	b	Rhapsody/LTL
*	Cengarle et al. [c13]	●					◑		◑	T	s/b	Institutions
*	Alanazi [c2]					◑	◑			U	(b)	Impl.
	Hammal [c36]					◑	◑			U	(b)	Petri nets
	Laleau, Polack [c52]	◑				◑	◑			U	s	Meta-model
*	Broy et al. [c11, c12]	●	●			●	●			S	s/b	Set theory
*	Kuske et al. [c48]	◑	◑			◑	◑			U	(s/b)	Graph. transf.
*	Grönniger [c33]	◑	◑			◑	◑		◑	S	s/b	Isabelle/HOL
	Nimiya et al. [c62]					◑	◑			U	b	Alloy
	Abdelhalim et al. [c1]					◑		◑		U	b	CSP
	Khai [c41]	◑					◑			U	s	Prolog
	Ober, Dragomir [c63]			●	●	◑				U	s/b	OMEGA2
	Puczynski [c69]	◑				◑	◑			U	s/b	Impl.
	Baresi et al. [c7]	◑	◑			◑	◑			U	b	TRIO
	Gerlinger et al. [c30]	●			●			●		U	s	Common logic
	El Miloudi et al. [c23, c24]	◑				◑	◑			U	s	Z
	Khan, Porres [c42]	●	●			◑			◑	U	s	Desc. logic
*	fUML [c64, c65, c66]	●			●	●		●		S	s/b	Common logic

3 Consistency Techniques

The most immediate and direct approach to consistency checking of UML diagrams and models uses consistency rules, mostly on the concrete or abstract syntax [15]. These rules extend the well-formedness rules of the UML specification given in OCL [c15]. Another option for such rules is to use other kinds of logics, like description logics [c42, c78, c79]. Many modelling tools incorporate their own rule sets [c19, c22, c56, c80]. Torre et al. [c50] list 116 consistency

rules studied in the literature, where 95 are syntactic (structural), 97 horizontal (intra-model), and 60 heterogeneous (involving several diagram types).

Syntactic checks are indispensable in any approach to consistency, but they do not suffice to uncover the more intricate behavioural consistency problems, e.g., whether a network of state machines admits a trace specified by an interaction. Advanced consistency approaches thus have to develop and rely on a behavioural semantics of the UML/OCL diagrams and sub-languages of discourse. The degree of integration of these semantics varies considerably with the proposed approaches in the literature. Though the borders can not be always drawn with full accuracy, we suggest a categorisation w.r.t. the emphasis which is given to the semantic heterogeneity of UML/OCL. In the "system model" approach a uniform realisation frame is built, into which all sub-language aspects are encoded. "Dynamic meta-modelling" dispenses with the encoding, but enriches the meta-model, i.e., UML's abstract syntax, by semantic information. The "universal logic" approaches still use an encoding, though now to a uniform formalism. Finally, "heterogeneous transformation" approaches aim at employing families of translations for relating sub-languages.

3.1 System Model

The "system model" approach, best exemplified by Broy, Groenniger et al. [c11, c12, c33], builds on a uniform semantic basis for covering all aspects of state and state change present in any UML sub-language to be considered. By representing every facet of a model, expressed in various diagrams, in one and the same instance of the system model, static as well as dynamic checks can be performed. The "Executable UML" (xUML [c61]) as well as the "Foundational Subset of the UML" (fUML [c66]) use such system models for comprehensive and integrated execution. The fUML, based on Common Logic (a standardised dialect of first-order logic), currently serves as basis for an endeavour to arrange an executable, programming language-based precise semantics of composite structures [c64] as well as state machines [c65].

For states, the system model in [c11, c12] contains a data store built from classes, their attributes, and the inheritance relationship as well as the instances; a control store consisting of operations and stacked method calls; and an event store holding also messages. For state changes, it comprises control-flow and event-based state transition systems enriched by time. This system model, though with some modifications, e.g., specialising the event store to a message store, has later on been encoded in Isabelle/HOL and parts of UML class and object diagrams, state machines, and sequence diagrams, as well as a subset of OCL have been represented in this system model [c33]. With the help of the Isabelle prover then both static consistency checks, like whether an inheritance relationship is acyclic, and dynamic consistency checks, like whether a sequence diagrams is realisable by a state machine, can be done.

The manual effort to write down these checks and perform them in an interactive theorem prover seems quite substantial, however. Not to the least part, this is owed to the necessary complexity of the system model. Automation of

various consistency checks has not been the primary goal of the approach. Still, the approach supports a certain degree of variability by exchanging sub-theories of the encoded system model, and other languages, a programming language, for instance, can be integrated [c33] if they can also be represented adequately in the system model.

The "krtUML" approach by Damm et al. [c16] uses symbolic transition systems as its system model for a comprehensive semantics. Their choice of class diagrams and state machines targets real-time systems. Consistency checks are not the main goal but behavioural consistency may be added on the basis of this system model. Damm et al. stress that "[b]ecause all diagrams are only views on one and the same model, the attempts to give semantics for separated UML diagrams fail in producing the right semantics for the entire UML" (p. 94), though this valid observation somewhat neglects the possibility to integrate the relations between the UML diagrams and sub-languages as done, e.g., in the heterogeneous transformation approaches.

3.2 Dynamic Meta-modelling

Inspired by attribute grammars extending the abstract syntax tree of a (textual) language by synthesised and inherited attributes for semantic and contextual analysis, the "dynamic meta-modelling" approach by Hausmann, Engels, et al. [c14, c26, c27, c29, c37, c75] extends the abstract syntax of the UML, i.e., its meta-model, by semantic concepts on this very meta-level. In fact, the UML meta-model already shows several concepts that serve as links between the various sub-languages, like Event originating from, e.g., operation calls, used in state machines and activities for triggering behavioural effects or Message used in interactions for referring to operations and signals. By adding extra semantic concepts, the linkage between the sub-languages can be enhanced and, in particular, lifted to the behavioural, dynamic level. For example, the UML meta-class StateMachine is extended by a new meta-class EventPool for holding instances of the already existing meta-class Event that the instance of StateMachine then can react to; or the meta-class ControlFlow of activities is extended by a new meta-class ControlToken representing the possibility that a control flow activity edge may carry a control token. Using the extensions, an operational semantics based on the extended meta-model and thus covering several UML sub-languages in concert is defined using (typed) graph transformations with negative application conditions, most prominently in the Groove tool [c37, c75] applying state space exploration. The transformations are separated into local small-step rules and transactional big-step rules that call the local rules. (A similar approach, though not targeted specifically at the UML and replacing graph transformations by programming, is used in the GEMOC initiative for "globalising modelling languages" supporting arbitrary domain-specific modelling languages [9].)

This intriguing idea of combining attribute grammars, structural operational semantics, and graph transformations has mainly been applied to activities [c37] and to a limited degree to state machines [c75] and OCL [c14]. For consistency checks proper, the dynamic relation between sequence diagrams and state

machines has been considered as an example, though without tool support [c27]. The overall design of dynamic meta-modelling somewhat resembles the "system model" approach as it builds a single domain of interpretation. By contrast, however, dynamic meta-modelling does not rely on an external semantic domain, but reuses the existing UML concepts and adds those features directly to the meta-model that are missing for behavioural interpretation. The use of a reference model for relating views that are then embedded and integrated into a system model has already been advocated by Ehrig et al. [19]; the use of graph transformations on the meta-model level has also been used by Kuske et al. [c48]. Still, the complexity of the UML meta-model itself, let alone the necessary extensions, and respecting all semantic interconnections in local and global graph transformation rules present a major obstacle for the use of dynamic meta-modelling in comprehensive multi-view consistency checking.

3.3 Universal Logic

The system model approach builds a uniform semantic domain offering the necessary mechanism to interpret the different UML sub-languages and diagrams. By contrast, a "universal logic" approach does not rely on a single domain of interpretation, but just uses a uniform logical technique, like transition systems, for expressing the semantics of all UML diagrams to be checked for consistency. However, as in the system model approach, having to use a single encoding technique may sometimes yield unnecessary and unnatural complexity.

Große-Rhode [c34, c35] uses "transformation systems", i.e., extended labelled transition systems, where both states and transitions are labelled. The (control) states offer observations and synchronisation points, the transitions model the atomic steps of an entity and may be executed synchronously with other system parts. The semantics of class diagrams, state machines as well as their composition, and sequence diagrams are represented as classes of such transformation systems. Consistency can then be expressed by checking that the intersection of model classes (modulo some projections for adapting labels) are not empty. In a similar vein, though not as elaborated, the "super-state analysis" of Alanazi [c2] relies on nets of transition systems which allows to check the consistency of state machines and interactions.

The "Consistency Workbench" by Küster, Engels, et al. [c25, c28, c49] is based on a "partial translation of models into a formal language (called semantic domain) that provides a language and tool support to formulate and verify consistency conditions" [c25, p. 158]. In principle, the employed semantic domain is not fixed for all instalments of the general approach and may vary with the consistency checks and the information extracted from the models by partial translation. The Consistency Workbench itself relies exclusively on the algebraic process language CSP and failure-divergence refinement (FDR). Still, it does not aim to construct an overarching system model, but is parameterised in the consistency problem type. In this sense, it is bordering at an approach using heterogeneous transformation.

In the "film-stripping" approach by Gogolla et al. [c32, c38] a uniform technique for representing behavioural system evolution is used: System behaviour is captured by sequences of snapshots of system states, i.e., object diagrams, linked together by change information in particular recording how the objects evolve. Consistency checks could then be performed, e.g., in the USE tool [c31]. Even for automated analysis, like model checking, however, the general scaling of the technique without appropriate compression or abstraction of the snapshots remains unclear. The approach is complemented by model transformations from full-fledged UML to a simpler "base model" [c39, c40]: Complex modelling constructs, like association classes, are replaced with simpler modelling expressions, though possibly at the expense of having to use OCL. This technique is mainly exemplified by transformations on class diagrams and OCL itself, though, ultimately "[a]ll diagrams conjoined are transformed and combined into a base model" [c40, p. 60]. Thus, there are quite some similarities with the system model approach.

3.4 Heterogeneous Transformation

Approaches based on "heterogeneous transformation" (coined by [c21]) focus on the several sub-languages and diagrams of the UML used in different forms at different development stages, from different viewpoints by different stakeholders [26,27], and their accompanying relations. Such an approach has in particular been advocated by Derrick et al. [c17] for UML from their experiences with "Open Distributed Processing" (ODP) using Z and LOTOS, though not elaborated in detail. In their terminology a set of viewpoint specifications is consistent "if there exists a specification that is a refinement of each of the viewpoint specifications with respect to the identified refinement relations and the correspondences between viewpoints" (p. 35). Here, the refinement and correspondence relations play the role of "transformations".

Egyed's ViewIntegra [c20, c21] defines transformations between the different UML diagram types on the very diagram level. In principle, all eleven diagram types of UML 1 could be covered, but only class diagrams, object diagrams, state machines, and interaction diagrams, i.e., sequence and collaboration diagrams (which, as in UML 2, are mere visual variants), are discussed in [c21]. The transformations are categorised into generalisation, e.g., object to class diagram; structuralisation, e.g., state machine to class diagram; translation, e.g., sequence into collaboration diagram; and abstraction, e.g., class diagram to class diagram. The last class of transformations, abstraction, is employed to relate diagrams at different development and refinement stages. Since the transformations map diagrams to diagrams, the supported consistency checks are structural; neither a static nor a dynamic semantics are provided. The classification of transformations is also used to reduce the number of necessary comparison transformations, which for eleven diagram types would otherwise be 55. However, when employing this design, not all transformations are possible any more, and a common denominator sometimes is needed, e.g., for comparing an object diagram with a state machine both are, perhaps somewhat arbitrarily, transformed into a class diagram.

We have started an effort to formalise the relation of views and viewpoints on a semantic level in [c82], where viewpoints are captured by a formal language category equipped with a model functor to a semantic domain category and views are language expressions in a viewpoint. Consistency is expressed by translations on the syntactic and semantic level; for semantic consistency a "viewpoint of comparison" has to be given. Consistency checks, though only pairwise, are exemplified for class diagrams, state machines, and sequence diagrams. The scheme that we have used in [c13] is similar, but uses the established theory of institutions as its foundation where the linkage between viewpoints, expressed as institutions, is represented by (co-)spans of institution (co-)morphisms. We have developed a vision how this can be extended to almost all UML diagrams in [c45], some initial results can be found in [c44].

Though it has not been directly applied to the UML/OCL family of languages yet, the work of Diskin et al. [13,30,31] on "heterogeneous multi-modelling" and "consistency-checking-by-merging" introduces another rather general and advanced approach to multi-view consistency along the lines of heterogeneous transformations. It is also inspired by the theory of institutions [13, p. 175]. Based on meta-models for expressing viewpoints, overlaps of viewpoints are represented by spans of meta-models, and the overall, global consistency of views can be expressed via a categorical colimit construction. Currently, the main focus lies on structural consistency, though also behavioural consistency is abstractly discussed in [13] using the example of traces of sequence diagrams and state machines.

4 Observations and Results

The 57 approaches listed in Tables 1 and 2 greatly differ in their quality w.r.t. elaboration and comprehensiveness, where this spectrum spreads from a kind of feasibility study [c36, c62] at the one end to detailed accounts involving comprehensive semantics, tools, and larger case studies [c31, c46, c51, c63] on the other. Tool support is similarly diverse and ranges from prototypical proprietary implementation over model checking and model finders to interactive theorem proving. About half of the approaches only support structural or very limited behavioural consistency checks (26), though the inclusion of at least some behavioural properties by 42 approaches demonstrates the necessity to integrate behaviour for true consistency.

In line with the survey results by Torre et al. [49], we also find that the diagram types most often covered are the class diagram (42 out of 57), the state machine diagram (47), and the interaction diagram (in one of its forms, 41); that according to our survey state machines have been considered more frequently than class diagrams may be accounted by our judicious choice of omitting single-view consistency approaches. Activity diagrams have rarely been integrated (7), but this may change with the broader adoption of fUML [c66]. The combination of state machines and interactions is considered most often (34). Both are mainly considered in the context of class diagrams, rarely in combination with component diagrams and composite structure diagrams (4).

Among the approaches listed covering many different diagram types, the top four are the following ones:

- xUML [c61] covers five diagram types. However, for three of them, only a limited subset is covered, and only syntactic, structural checks are provided.
- Gogolla et al. [c31] cover six diagram types, three of them to a substantial portion, and at least partially semantic, behavioural checks are provided.
- Broy et al. [c11, c12] define a comprehensive, though complex system model capturing substantial subsets of four diagram types of UML; however, tool support is not provided.
- Grönniger [c33], based on [c11, c12], integrates a fragment of the UML (partially covering class diagrams, object diagrams, interactions, state machines, and OCL) semantically by an encoding in the interactive theorem prover Isabelle/HOL; however, the degree of coverage is considerably lower than in [c11, c12].

All these either follow the system model or the universal logic approach. In accordance with the survey by Lucas et al. [34] we also find, that these encoding techniques (system model: 5, universal logic: 48) are currently by far prevailing for consistency. This may be explained by the fact that the ultimate question of overall, global consistency is whether there is a single implementing system or "realisation" simultaneously satisfying all views, and that the quest for such a realisation may become simpler when everything is represented or encoded uniformly. The major drawback of the encoding approaches, which may have prevented the further integration of other UML/OCL diagram types and sub-languages into them, is their lack of extensibility: A new viewpoint really extending the realm of expressivity inevitably calls for a considerable amount of work in expanding and adapting the already established semantic or logical domain. Other disadvantages, already pointed out by Derrick et al. [c17], are traceability problems, as reported inconsistencies generally can only be partially translated back into the language of the original models, and the difficulty of incremental consistency checks.

5 Distributed Semantics for Multi-view Consistency

The most comprehensive current UML/OCL multi-view consistency frameworks are based on the system model or the universal logic approach and thus show disadvantages w.r.t. extensibility, traceability, and incrementality. The heterogeneous transformation approach has been introduced to mitigate these problems by taking different viewpoints and views as entities in their own right and relating them by transformations. Still, the integration of behavioural aspects beyond meta-modelling and the resulting problem of obtaining global consistency for such heterogeneous multi-view models have been considered only to a rather limited degree.

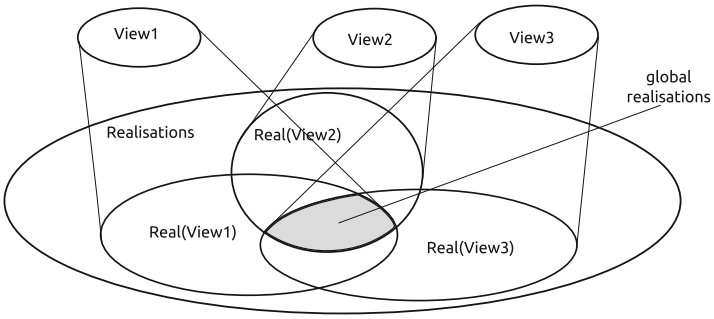
Based on our previous work on formalising several UML/OCL sub-languages and their behavioural semantics as institutions and relating them by institution

(co-)morphisms [c13, c44, c45], we in the following briefly suggest a heterogeneous transformation approach that takes a *distributed* view to heterogeneous consistency: In [38], we have already provided a classification of approaches for handling semantic heterogeneity in the domain of formal specification that is in close correspondence to the present categorisation of multi-view consistency frameworks for UML/OCL:

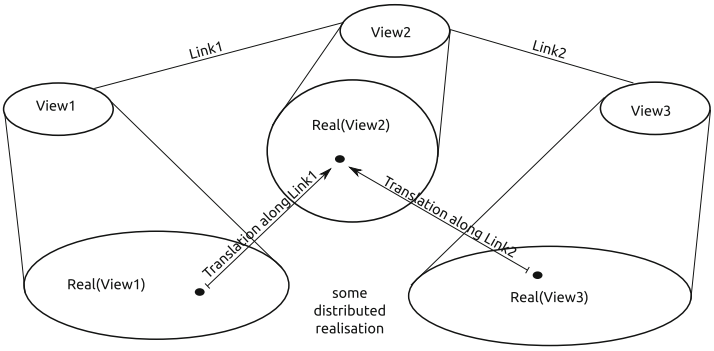
- encoding of heterogeneous languages into some "universal" language. This is applied by the system model, the dynamic meta-modelling, and the universal logic approaches.
- focused heterogeneity, i.e., languages are not per se encoded into some "universal" language, and complex models may involve parts written in different languages. Still, via translations, the end result is formulated in one language. This roughly corresponds to the heterogeneous transformation approach.
- distributed heterogeneity, i.e., truly decentralised networks of models formulated in different languages and linked by transformations and refinements.

The classical approach to multi-view semantics and consistency using focused heterogeneity is illustrated in Fig. 1(a) (taken from [6]). An important feature there is that consistency means the existence of a *single global realisation* that satisfies all the different viewpoints. In particular, if the individual viewpoints are formulated in different formalisms, the existence of a single global realisation assumes their translatability to a single formalism or semantic domain serving as "lingua franca" and hosting the global realisation. In Derrick et al.'s approach [c17] a common refining specification is required, in Diskin et al.'s work [13] a colimit is constructed on which consistency is checked. (Egyed's [c21] and our own previous approach [c82] mainly focus on pairwise consistency.) By contrast, in distributed heterogeneity, a realisation of a distributed network of models is a *family of realisations* of the individual models that is *compatible* with the links, see Fig. 1(b); and a network is consistent if it has such a realisation. Under suitable amalgamation conditions, a single global realisation of the network can be constructed from such a compatible family in a particular viewpoint; however, these conditions are not always satisfied [37]. Moreover, in a *system of systems* that has truly distributed implementations only, a single global realisation may be conceptually unwanted and overly complex.

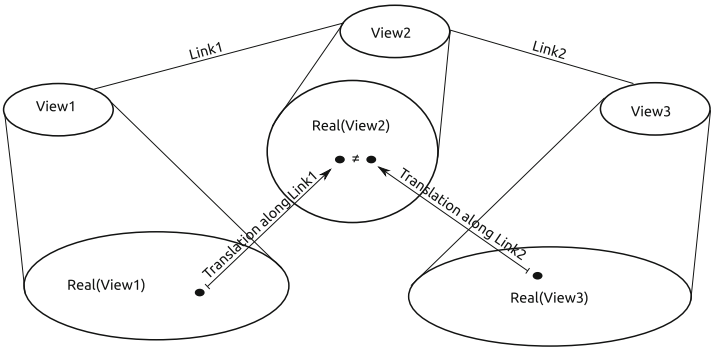
Altogether, the distributed heterogeneity approach that we suggest for addressing UML/OCL multi-view consistency is more flexible than the assumption of a single global realisation, but it still enforces a global compatibility of all constraints, unlike a mere pairwise consistency (cf. Fig. 1(c)). Thus, consistency for distributed heterogeneity could be characterised as "global consistency without the need of a global realisation". The distributed heterogeneity approach in general is supported by the Heterogeneous Tool Set (Hets [37]) and the Distributed Ontology, Model, and Specification Language (DOL [40]), the latter extending the algebraic specification language CASL. However, specific instantiation of this approach to UML/OCL has only started and is subject of future research.



(a) Focused heterogeneity in classical multi-view semantics (see similar illustrations in [6, p. 605] and [c35, p. 11]). The class of global realisations is the intersection of the classes of the (projections of) the realisations of the individual views.



(b) Distributed heterogeneous transformation approach. A realisation of a network (of views and links) is a compatible family of realisations.



(c) Pairwise consistency — a notion that is weaker than distributed consistency: there might be no compatible family of realisations, because the two induced realisations in Real(View2) never match. This can happen if View1 and View3 impose conflicting requirements.

Fig. 1. Multi-view consistency approaches

6 Conclusion and Future Work

UML/OCL is a language for multi-view and multi-viewpoint models, and the detection of view inconsistencies at an early stage of the development is important for avoiding costly redesign. We have classified 57 existing approaches to UML/OCL multi-view consistency. Even the most comprehensive approaches currently cover only five of the 14 UML diagram types, and most of these only partially. Moreover, a "universal logic" approach is predominant, where all UML/OCL diagram types and sub-languages are embedded into one system model or one logic. We have argued that this is not suitable for handling the involved complexity. This would become even more palpable if also extensions of the UML, like SysML or MARTE, or even the integration of domain-specific modelling languages are considered.

We propose a new approach to UML/OCL multi-view consistency, following a "distributed heterogeneous transformation" paradigm. We use institutions for formalising the different UML diagram types and their semantics, and institution (co-)morphisms for formalising transformations between them. Then UML/OCL multi-viewpoint models can be formalised as so-called networks in the OMG-standardised Distributed Ontology, Model, and Specification Language (DOL). This provides a framework where eventually all semantically relevant diagram types and sub-languages can be covered.

In order to use this framework for checking consistency of UML/OCL multi-viewpoint models, there is still a considerable way to go: while some UML diagram types have been formalised as institutions, this needs to be completed to a more comprehensive treatment of both diagram types and their features. Formalisation of transformations as institution (co-)morphisms has just started. In order to make this practically useful in connection with DOL, all the institutions and (co-)morphisms need to be integrated into the Heterogeneous Tool Set (Hets) and interfaced with suitable proof and realisation finding tools. Finally, suitable consistency strategies need to be developed and implemented.

Of course, writing down DOL expressions for large families of UML/OCL diagrams and specifications will be tedious. Hence, we aim at some graphical interface that can generate the needed DOL expressions automatically from a user's selection of those UML/OCL diagrams and specifications that should be interlinked to a network, plus a specification of the involved refinements. Such a specification of both networks and refinements adds the extra information to a given family of UML/OCL diagrams and specifications that is needed when checking multi-view consistency.

Multi-view UML Consistency Approaches

c1. Abdelhalim, I., Schneider, S., Treharne, H.: Towards a practical approach to check UML/fUML models consistency using CSP. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 33–48. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24559-6_5

c2. Alanazi, M.N.: Consistency checking in multiple UML state diagrams using super state analysis. Ph.D. thesis. Kansas State University (2008)
c3. Amálio, N., Stepney, S., Polack, F.: Formal proof from UML models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 418–433. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30482-1_35
c4. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: a challenging model transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_30
c5. Banerjee, A., Ray, S., Dasgupta, P., Chakrabarti, P.P., Ramesh, S., Ganesan, P.V.V.: A dynamic assertion-based verification platform for validation of UML designs. In: Cha, S.S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 222–227. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88387-6_18
c6. Banerjee, A., Ray, S., Dasgupta, P., Chakrabarti, P.P., Ramesh, S., Ganesan, P.V.V.: A dynamic assertion-based verification platform for validation of UML designs. ACM SIGSOFT Softw. Eng. Notes **37**(1), 1–14 (2012)
c7. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: A logic-based semantics for the verification of multi-diagram UML models. ACM SIGSOFT Softw. Eng. Notes **37**(4), 1–8 (2012)
c8. Bellur, U., Vallieswaran, V.: On OO Design Consistency in Iterative Development. In: Proceedings of 3rd International Conference on Information Technology: New Generations (ITNG 2006), pp. 46–51. IEEE (2006)
c9. Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analysable petri net models. In: Proceedings of 3rd International Workshop on Software and Performance (WSOP 2002), pp. 35–45. ACM (2002)
c10. Brændshøi, B.: Consistency checking UML interactions and state machines. Master thesis. Universitetet i Oslo (2008)
c11. Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Considerations and rationale for a UML system model. In: Lano, K. (ed.) UML 2 – Semantics and Applications, chap. 3, pp. 43–60. Wiley (2009)
c12. Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Definition of the system model. In: Lano, K., (ed.) UML 2 – Semantics and Applications, chap. 4, pp. 61–93. Wiley (2009)
c13. Cengarle, M.V., Knapp, A., Tarlecki, A., Wirsing, M.: A heterogeneous approach to UML semantics. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 383–402. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68679-8_23
c14. Chiaradía, J.M., Pons, C.: Improving the OCL semantics definition by applying dynamic meta modeling and design patterns. In: Chiorean, D., Demuth, B., Gogolla, M., Warmer, J., (eds.) Proceedings of 6th OCL Workshop on OCL for (Meta-)Models in Multiple Application Domains (OCL 2006) (2006). Electr. Comm. EASST **5**
c15. Chiorean, D., Paşca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML models consistency using the OCL environment. In: Schmitt, P.H., (ed.) Proceedings of 3rd OCL Workshop on OCL 2.0 (OCL 2003). Elsevier (2004). Electr. Notes Theo. Comp. Sci. **102**, 99–110

c16. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: Understanding UML: a formal semantics of concurrency and communication in real-time UML. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 71–98. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39656-7_3
c17. Derrick, J., Akehurst, D., Boiten, E.: A Framework for UML consistency. In: Proceedings Workshop on Consistency Problems in UML-based Software Development, pp. 30–45 (2002)
c18. Diethers, K., Huhn, M.: Vooduu: verification of object-oriented designs using UPPAAL. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 139–143. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_10
c19. Dubauskaite, R., Vasilecas, O.: Method on specifying consistency rules among different aspect models, expressed in UML. Elektr. Elektrotechn. **19**(3), 77–81 (2013)
c20. Egyed, A.: Heterogenous view integration and its automation. Ph.D. thesis. University of Southern California (2000)
c21. Egyed, A.: Scalable consistency checking between diagrams – the viewintegra approach. In: Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE 2001), pp. 387–390. IEEE (2001)
c22. Egyed, A.: UML/Analyzer: a tool for the instant consistency checking of UML models. In: Proceedings of 29th International Conference on Software Engineering (ICSE 2007), pp. 793–796. IEEE (2007)
c23. El Miloudi, K., Amrani, Y.E., Ettouhami, A.: An automated translation of UML class diagrams into a formal specification to detect UML inconsistencies. In: Proceedings of 6th International Conference on Software Engineering Advances (ICSEA 2011), pp. 432–438 (2011)
c24. El Miloudi, K., Ettouhami, A.: A multi-view approach for formalizing UML state machine diagrams using Z notation. WSEAS Trans. Comp. **14**, 72–78 (2015)
c25. Engels, G., Heckel, R., Küster, J.M.: The consistency workbench: a tool for consistency management in UML-based development. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 356–359. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45221-8_30
c26. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40011-7_23
c27. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Testing the consistency of dynamic UML diagrams. In: Proceedings of 6th World Conference on Integrated Design & Process Technology (IDPT 2002) (2002)
c28. Engels, G., Heckel, R., Küster, J.M.: Rule-based specification of behavioral consistency based on the UML meta-model. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 272–286. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45441-1_21
c29. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML activities using dynamic meta modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72952-5_5

c30. Gerlinger Romero, A., Schneider, K., Gonçalves Vieira Ferreira, M.: Integrating UML composite structures and fUML. In: Geffert, V., Preneel, B., Rovan, B., Štuller, J., Tjoa, A.M. (eds.) SOFSEM 2014. LNCS, vol. 8327, pp. 269–280. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-04298-5_24
c31. Gogolla, M., Büttner, F., Richters, M.: USE: a uml-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**(1–3), 27–34 (2007)
c32. Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., France, R.B.: From application models to filmstrip models: an approach to automatic validation of model dynamics. In: Fill, H.-G., Karagiannis, D., Reimer, U., (eds.) Proceedings of Modellierung 2014. Lecture Notes in Informatics, vol. 225, pp. 273–288. GI (2014)
c33. Grönniger, H.: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten. Ph.D. thesis. RWTH Aachen (2010)
c34. Große-Rhode, M.: Integrating semantics for object—oriented system models. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 40–60. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48224-5_4
c35. Große-Rhode, M.: Semantic Integration of Heterogeneous Software Specifications. Monographs in Theoretical Computer Science. Springer, Heidelberg (2004)
c36. Hammal, Y.: A modular state exploration and compatibility checking of UML dynamic diagrams. In: Proceedings of 6th ACS/IEEE International Conference Computer Systems and Applications (AICCSA 2008), pp. 793–800. IEEE (2008)
c37. Hausmann, J.H.: Dynamic meta modeling. a semantics description technique for visual modeling languages. Ph.D. thesis. Universität Paderborn (2005)
c38. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: Filmstripping and unrolling: a comparison of verification approaches for UML and OCL behavioral models. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 99–116. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09099-3_8
c39. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: From UML/OCL to base models: transformation concepts for generic validation and verification. In: Kolovos, D., Wimmer, M. (eds.) ICMT 2015. LNCS, vol. 9152, pp. 149–165. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21155-8_12
c40. Hilken, F., Niemann, P., Wille, R., Gogolla, M.: Towards a base model for UML and OCL verification. In: Boulanger, F., Famelis, M., Ratiu, D., (eds.) Proceedings of 11th Workshop Model-Driven Engineering, Verification and Validation (MoDEVVA 2014), vol. 1235, pp. 59–68. CEUR-WS Proceedings (2014)
c41. Khai, Z., Nadeem, A., Lee, G.: A prolog based approach to consistency checking of UML class and sequence diagrams. In: Kim, T., Adeli, H., Kim, H., Kang, H., Kim, K.J., Kiumi, A., Kang, B.-H. (eds.) ASEA 2011. CCIS, vol. 257, pp. 85–96. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-27207-3_10
c42. Khan, A.H., Porres, I.: Consistency of UML class, object and statechart diagrams using ontology reasoners. J. Vis. Lang. Comp. **26**, 42–65 (2015)
c43. Kim, S.-K., Carrington, D.: A formal object-oriented approach to defining consistency constraints for UML models. In: Proceedings of 15th Australian Software Engineering Conference (ASWEC 2004), pp. 87–94. IEEE (2004)
c44. Knapp, A., Mossakowski, T.: UML interactions meet state machines - an institutional approach. In: Bonchi, F., König, B., (eds.) Proceedings of 7th International Conference on Algebra and Coalgebra in Computer Science (CALCO 2017). Leibniz International Proceedings in Informatics, pp. 15:1–15:15 (2017)

c45. Knapp, A., Mossakowski, T., Roggenbach, M.: Towards an institutional framework for heterogeneous formal development in UML. In: De Nicola, R., Hennicker, R. (eds.) Software, Services, and Systems. LNCS, vol. 8950, pp. 215–230. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15545-6_15
c46. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Kühne, T. (ed.) MODELS 2006. LNCS, vol. 4364, pp. 42–51. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69489-2_6
c47. Krishnan, P.: Consistency checks for UML. In: Proceedings of 7th Asia Pacific Software Engineering Conference (APSEC 2000), pp. 162–171. IEEE (2000)
c48. Kuske, S., Gogolla, M., Kreowski, H.-J., Ziemann, P.: Towards an integrated graph-based semantics for UML. Softw. Syst. Model. **8**(3), 403–422 (2009)
c49. Küster, J.M.: Consistency management of object-oriented behavioral models. Ph.D. thesis. Universität Paderborn (2004)
c50. Küster, J.M., Engels, G.: Consistency management within model-based object-oriented development of components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 157–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30101-1_7
c51. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. In: Lüttgen, G., Mendler, M., (eds.) Proceedings of Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004). Electronic Notes in Theoretical Computer Science, vol. 115, pp. 39–47. Elsevier (2005)
c52. Laleau, R., Polack, F.: Using formal metamodels to check consistency of functional views in information systems specification. J. Inf. Softw. Techn. **50**(7–8), 797–814 (2008)
c53. Lam, V.S.W., Padget, J.: Consistency checking of sequence diagrams and statechart diagrams using the π-calculus. In: Romijn, J., Smith, G., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 347–365. Springer, Heidelberg (2005). https://doi.org/10.1007/11589976_20
c54. Li, X.: A characterization of UML diagrams and their consistency. In: Proceedings of 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2006), pp. 67–76. IEEE (2006)
c55. Litvak, B., Tyszberowicz, S.S., Yehudai, A.: Behavioral consistency validation of UML diagrams. In: Proceedings of 1st International Conference on Software Engineering and Formal Methods (SEFM 2003), pp. 118–125. IEEE (2003)
c56. Liu, W., Easterbrook, S., Mylopoulos, J.: Rule-based detection of inconsistency in UML models. In: Proceedings Workshop on Consistency Problems in UML-Based Software Development, pp. 106–123 (2002)
c57. Liu, Z., He, J., Li, X.: Towards a rigorous approach to UML-based development. In: Mota, A., Moura, A., (eds.) Proceedings of 7th Brazilian Symposium on Formal Methods (SBMF 2004). Electronic Notes in Theoretical Computer Science, vol. 130, pp. 57–77. Elsevier (2005)
c58. Long, Q., Liu, Z., Li, X., He J.: Consistent code generation from UML models. In: Proceedings of 16th Australian Software Engineering Conference (ASWEC 2005), pp. 23–30. IEEE (2005)
c59. Martínez, F.J.L., Álvarez, A.T.: A precise approach for the analysis of the UML models consistency. In: Akoka, J., Liddle, S.W., Song, I.-Y., Bertolotto, M., Comyn-Wattiau, I., van den Heuvel, W.-J., Kolp, M., Trujillo, J., Kop, C., Mayr, H.C. (eds.) ER 2005. LNCS, vol. 3770, pp. 74–84. Springer, Heidelberg (2005). https://doi.org/10.1007/11568346_9

c60. McUmber, W.E., Cheng, B.H.: A general framework for formalizing UML with formal languages. In: Proceedings of 23rd International Conference on Software Engineering (ICSE 2001), pp. 433–442. IEEE (2001)
c61. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-driven Architecture. Addison-Wesley, Boston (2002)
c62. Nimiya, A., Yokogawa, T., Miyazaki, H., Amasaki, S., Sato, Y., Hayase, M.: Model checking consistency of UML diagrams using Alloy. WASET Intl. J. Comp. Electr. Autom. Contr. Inf. Eng. **4**(11), 1696–1699 (2010)
c63. Ober, I., Dragomir, I.: Unambiguous UML composite structures: the OMEGA2 experience. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Králović, R., Vukolić, M., Wolf, S. (eds.) SOFSEM 2011. LNCS, vol. 6543, pp. 418–430. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18381-2_35
c64. Object Management Group. Precise Semantics of UML Composite Structures (PSCS). Standard formal/2015-10-02. Version 1.0. OMG. http://www.omg.org/spec/PSCS/1.0
c65. Object Management Group. Precise Semantics of UML State Machines (PSSM). Report ptc/2017-04-04. Version 1.0 Beta. OMG. http://www.omg.org/spec/PSSM/1.0/Beta1
c66. Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML). Standard formal/2016-01-05. Version 1.2.1. OMG. http://www.omg.org/spec/FUML/1.2.1
c67. Ossami, D.D.O., Jacquot, J.-P., Souquières, J.: Consistency in UML and B multi-view specifications. In: Romijn, J., Smith, G., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 386–405. Springer, Heidelberg (2005). https://doi.org/10.1007/11589976_22
c68. O'Keefe, G.: Dynamic logic semantics for UML consistency. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 113–127. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_10
c69. Puczynski, P.J.: Checking consistency between interaction diagrams and state machines in UML models. Master thesis. Danmarks Tekniske Universitet (2012)
c70. Rasch, H., Wehrheim, H.: Checking consistency in UML diagrams: classes and state machines. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 229–243. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39958-2_16
c71. Rasch, H., Wehrheim, H.: Checking the validity of scenarios in UML models. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 67–82. Springer, Heidelberg (2005). https://doi.org/10.1007/11494881_5
c72. Reggio, G., Cerioli, M., Astesiano, E.: Towards a rigorous semantics of UML supporting its multiview approach. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 171–186. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45314-8_13
c73. Sapna, P.G., Mohanty, H.: Ensuring consistency in relational repository of UML models. In: Proceedings of 10th International Conference on Information Technology (ICIT 2007), pp. 217–222. IEEE (2007)
c74. Shinkawa, Y.: Inter-model consistency in UML based on CPN formalism. In: Proceedings of 13th Asia Pacific Software Engineering Conference (APSEC 2006), pp. 411–418. IEEE (2006)
c75. Soltenborn, C.: Quality assurance with dynamic meta modeling. Ph.D. thesis. Universität Paderborn (2013)

c76. Spanoudakis, G., Kim, H.: Diagnosis of the significance of inconsistencies in object-oriented designs: a framework and its experimental evaluation. J. Syst. Softw. **64**(1), 3–22 (2002)
c77. Tsiolakis, A.: Consistency analysis of UML class and sequence diagrams based on attributed typed graphs and their transformation. In: Ehrig, H., Taentzer, G., (eds.) Proceedings Workshop on Graph Transformation Systems (GRATRA 2000), pp. 77–86 (2000)
c78. van der Straeten, R.: Inconsistency detection between UML models using RACER and nRQL. In: Bechhofer, S., Haarslev, V., Lutz, C., Moeller, R., (eds.) Proceedings of 3rd International Workshop on Applications of Description Logics (KI 2004), vol. 115. CEUR-WS Proceedings (2004)
c79. van der Straeten, R., Simmonds, J., Mens, T.: Detecting inconsistencies between UML models using description logic. In: Calvanese, D., Giacomo, G.D., Franconi, E., (eds.) Proceedings of International Workshop on Description Logics (DL 2003), vol. 81. CEUR-WS Proceedings (2003)
c80. Wagner, R., Giese, H., Nickel, U.A.: A plug-in for flexible and incremental consistency management. In: Kuzniarz, L., Reggio, G., Sourrouille, J.-L., Huzar, Z., Staron, M., (eds.) Proceedings of 3rd Workshop on Consistency Problems in UML-based Software Development. Blekinge Inst. Techn. Research Report 2003:06 (2003)
c81. Wang, H., Feng, T., Zhang, J., Zhang, K.: Consistency check between behaviour models. In: Proceedings 5th IEEE International Symposium on Communications and Information Technology (ISCIT 2005), pp. 486–489. IEEE (2005)
c82. Wirsing, M., Knapp, A.: View consistency in software development. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 341–357. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24626-8_24
c83. Yang, J., Long, Q., Liu, Z., Li, X.: A predicative semantic model for integrating UML models. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 170–186. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_14
c84. Yao, S., Shatz, S.M.: Consistency checking of UML dynamic models based on petri net techniques. In: Proceedings of 15th International Conference on Computing (CIC 2006). IEEE (2006)
c85. Yeung, W.L.: Checking consistency between UML class and state models based on CSP and B. J. Univ. Comp. Sci. **10**(11), 1540–1559 (2004)
c86. Zhao, X., Long, Q., Qiu, Z.: Model checking dynamic UML consistency. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 440–459. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_24

References

1. Ahmad, M.A., Nadeem, A.: Consistency checking of UML models using description logics: a critical review. In: Proceedings of the 6th International Conference on Emerging Technologies (ICET 2010), pp. 310–315. IEEE (2010)
2. Allaki, D., Dahchour, M., En-Nouaary, A.: A new taxonomy of inconsistencies in UML models with their detection methods for better MDE. Int. J. Comput. Sci. Appl. **12**(1), 48–65 (2015)
3. Amaya, P., Gonzalez, C., Murillo, J.M.: Towards a subject-oriented model-driven framework. In: Aksit, M., Roubtsova, E. (eds.) Proceedings of the 1st Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2005). Electronic Notes in Theoretical Computer Science, pp. 31–44 (2006)

4. Bashir, R.S., Lee, S.P., Khan, S.U.R., Farid, S., Chang, V.: UML models consistency management: guidelines for software quality manager. Int. J. Inf. Manag. Part A **36**(6), 883–899 (2016)
5. Boronat, A., Knapp, A., Meseguer, J., Wirsing, M.: What Is a Multi-modeling Language? In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 71–87. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03429-9_6
6. Braatz, B., Klein, M., Schröter, G.: Semantical Integration of Object-Oriented Viewpoint Specification Techniques. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) Integration of Software Specification Techniques for Applications in Engineering. LNCS, vol. 3147, pp. 602–626. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27863-4_32
7. Breu, R., Grosu, R., Huber, E., Rumpe, B., Schwerin, W.: Systems, views and models of UML. In: Schader, M., Korthaus, A. (eds.) The Unified Modeling Language. Physica-Verlag HD, Heidelberg (1998). https://doi.org/10.1007/978-3-642-48673-9_7
8. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. J. Syst. Softw. **93**, 1–23 (2014)
9. Combemale, B., Deantoni, J., Baudry, B., France, R.B., Jézéquel, J.-M., Gray, J.: Globalizing modeling languages. IEEE Comput. **47**(6), 68–71 (2014)
10. Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A.C.: Defining UML family members using prefaces. In: Mingins, C., Meyer, B. (eds.) Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages (TOOLS 1999), pp. 102–114. IEEE (1999)
11. Dijkman, R.M.: Consistency in multi-viewpoint architectural design. Ph.D. thesis. Universiteit Twente (2006)
12. Dingel, J., Diskin, Z., Zito, A.: Understanding and improving UML package merge. Softw. Syst. Model. **7**(4), 443–467 (2008)
13. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21210-9_16
14. Dobing, B., Parsons, J.: Dimensions of UML diagram use: practitioner survey and research agenda. In: Siau, K., Erickson, J. (eds.) Principle Advancements in Database Management Technologies: New Applications and Frameworks, pp. 271–290. IGI Publishing (2010)
15. Dragomir, I., Graf, S., Karsai, G., Noyrit, F., Ober, I., Torre, D., Labiche, Y., Genero, M., Elaasar, M. (eds.): Joint Proceedings of the 8th International Workshop on Model-Based Architecting of Cyber-physical and Embedded Systems (ACES-MB 2015) and 1st International Workshop on UML Consistency Rules (WUCOR 2015). CEUR WS, vol. 1508 (2015)
16. Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.): Integration of Software Specification Techniques for Applications in Engineering. LNCS, vol. 3147. Springer, Heidelberg (2004). https://doi.org/10.1007/b100778
17. Elaasar, M., Briand, L.C.: An overview of UML consistency management. Technical report SCE-04-18. Carleton University (2004)
18. van Emde Boas, P.: Formalizing UML: mission impossible? In: Andrade, L., Moreira, A., Deshpande, A., Kent, S. (eds.) Proceedings of the OOPSLA 1998 Workshop on Formalizing UML: why? How? (1998)

19. Engels, G., Heckel, R., Taentzer, G., Ehrig, H.: A combined reference model - and view-based approach to system specification. Intl. J. Softw. Eng. Knowl. Eng. **7**(4), 457–477 (1997)
20. Evans, A., Lano, K., France, R., Rumpe, B.: Meta-modeling semantics of UML. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 45–60. Kluver Academic Publisher, Dordrecht (1999). Chapter 4
21. Evans, A., France, R., Lano, K., Rumpe, B.: The UML as a Formal Modeling Notation. In: Bézivin, J., Muller, P.-A. (eds.) UML 1998. LNCS, vol. 1618, pp. 336–348. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-48480-6_26
22. von Hanxleden, R., Lee, E.A., Motika, C., Fuhrmann, H.: Multi-view Modeling and Pragmatics in 2020. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 209–223. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_11
23. Hilken, F., Niemann, P., Gogolla, M., Wille, R.: Towards a catalog of structural and behavioral verification tasks for UML/OCL models. In: Oberweis, A., Reussner, R.H. (eds.) Proceedings of Modellierung 2016. Lecture Notes in Informatics, pp. 117–124. GI, Bonn (2016)
24. Hoffmann, V., Lichter, H., Nyßen, A., Walter, A.: Towards the integration of UML and textual use case modeling. J. Object Technol. **8**(3), 85–100 (2009)
25. Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, J.L.: Consistency problems in UML-based software development. In: Nunes, N.J., Selic, B., da Silva, A.R., Alvarez, A.T. (eds.) UML 2004. LNCS, vol. 3297, pp. 1–12. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31797-5_1
26. IEEE Standards Association: Recommended practice for architectural description for software-intensive systems. Standard 1471–2000. IEEE Computer Society (2000)
27. International Organization for Standardization: Systems and software engineering – architecture description. Standard 42010:2011. ISO/IEC/IEEE (2011)
28. Kholkar, D., Krishna, G.M., Shrotri, U., Venkatesh, R.: Visual specification and analysis of use cases. In: Naps, T.L., Pauw, W.D. (eds.) Proceedings of the ACM Symposium on Software Visualization (SOFTVIS 2005), pp. 77–85. ACM (2005)
29. Kim, S.-K., David, C.: Formalizing the UML Class Diagram Using Object-Z. In: France, R., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 83–98. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-46852-8_7
30. König, H., Diskin, Z.: Advanced Local Checking of Global Consistency in Heterogeneous Multimodeling. In: Wąsowski, A., Lönn, H. (eds.) ECMFA 2016. LNCS, vol. 9764, pp. 19–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_2
31. König, H., Diskin, Z.: Efficient Consistency Checking of Interrelated Models. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 161–178. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_10
32. Langer, P., Mayerhofer, T., Wimmer, M., Kappel, G.: On the usage of UML: initial results of analyzing open UML models. In: Fill, H.-G., Karagiannis, D., Reimer, U. (eds.) Proceedings of Modellierung 2014. Lecture Notes in Informatics, vol. 225, pp. 289–304. GI, Bonn (2014)
33. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Form. Aspects Comput. **11**(6), 637–664 (1999)
34. Lucas, F.J., Molina, F., Toval, A.: A systematic review of UML model consistency management. J. Inf. Softw. Technol. **51**(12), 1631–1645 (2009)

35. Mens, T., van der Straeten, R., Simmonds, J.: A framework for managing consistency of evolving UML models. In: Yang, H. (ed.) Software Evolution with UML and XML, pp. 1–30. Idea Group (2005). Chapter 1
36. Mohammadi, R.G., Barforoush, A.A.: Enforcing component dependency in UML deployment diagram for cloud applications. In: Proceedings of the 7th International Symposium on Telecommunications (IST 2014), pp. 412–417. IEEE (2014)
37. Mossakowski, T.: Heterogeneous specification and the heterogeneous tool set. Habilitation thesis. Universität Bremen (2005)
38. Mossakowski, T., Tarlecki, A.: Heterogeneous Logical Environments for Distributed Specifications. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 266–289. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03429-9_18
39. Munker, F., Albers, A., Wagner, D., Behrendt, M.: Multi-view modeling in SysML: thematic structuring for multiple thematic views. In: Madni, A.M., Boehm, B., Sievers, M., Wheaton, M. (eds.) Proceedings of the Conference on Systems Engineering Research (CSER 2014). Procedia Computer Science, vol. 28, pp. 531–538. Elsevier (2014)
40. Object Management Group. Distributed Ontology, Modeling, and Specification Language (DOL). In Process ptc/2016-02-37. Version 1.0 - Beta1. OMG (2016). http://www.omg.org/spec/DOL/1.0/Beta1
41. Object Management Group. Object Constraint Language: Standard formal/2014-02-03. Version 2.4. OMG (2014). http://www.omg.org/spec/OCL/2.4
42. Mouheb, D., Debbabi, M., Pourzandi, M., Wang, L., Nouh, M., Ziarati, R., Alhadidi, D., Talhi, C., Lima, V.: Unified Modeling Language. Aspect-Oriented Security Hardening of UML Design Models. LNCS, pp. 11–22. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16106-8_2
43. Paige, R.F., Brooke, P.J., Ostroff, J.S.: Metamodel-based model conformance and multiview consistency. ACM Trans. Softw. Eng. Methodol. **16**(3), 11 (2007)
44. Pap, Z., Majzik, I., Pataricza, A., Szegi, A.: Completeness and consistency analysis of UML statechart specifications. In: Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2001), pp. 83–90. IEEE (2001)
45. Pap, Z., Majzik, I., Pataricza, A., Szegi, A.: Methods of checking general safety criteria in UML statechart specifications. Reliab. Eng. Syst. Saf. **87**(1), 89–107 (2005)
46. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: Sutcliffe, A., Jalote, P. (eds.) Proceedings of the 15th IEEE International Conference on Requirements Engineering, pp. 221–230. IEEE (2007)
47. Schürr, A., Winter, A.J.: Formal definition and refinement of UML's module/package concept. In: Bosch, J., Mitchell, S. (eds.) ECOOP 1997. LNCS, vol. 1357, pp. 211–215. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-69687-3_43

48. Shah, A.A., Kerzhner, A.A., Schaefer, D., Paredis, C.J.J.: Multi-view Modeling to Support Embedded Systems Engineering in SysML. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 580–601. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17322-6_25
49. Torre, D., Labiche, Y., Genero, M.: UML consistency rules: a systematic mapping study. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014). ACM (2014)
50. Torre, D., Labiche, Y., Genero, M., Elaasar, M.: A systematic identification of consistency rules for UML diagrams. Technical report SCE-15-01. Carleton University (2016)
51. Usman, M., Nadeem, A., Kim, T.-H., Cho, E.-S.: A survey of consistency checking techniques for UML models. In: Proceedings of the Advanced Software Engineering and Its Applications (ASEA 2008), pp. 57–62. IEEE (2008)

A Simple Notion of Parallel Graph Transformation and Its Perspectives

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

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

Abstract. In this paper, we reconsider an old and simple notion of parallel graph transformation and point out various perspectives concerning the parallel generation of graph languages, the parallelization of graph algorithms, the parallel transformation of infinite graphs, and parallel models of computation.

1 Introduction

In 1976, Hartmut Ehrig and the first author introduced an approach to parallel graph transformation in [1]. Parallel computation is obtained by the application of parallel rules which are composed of component rules by means of disjoint union. This is a particular simple and intuitive operation on graphs (and the graphs within graph transformation rules) because it places graphs – in our case directed edge-labeled graphs – next to each other without overlap and extra connections. The application of parallel rules has some significant properties. The component rules can be applied in arbitrary order yielding the same result as a given application of a parallel rule (sequentialization theorem). Conversely, if the component rules can be applied and their applications are independent of each other in a certain sense, then their parallel rule can also be applied (parallelization theorem). Parallel graph transformation has been one of the major research topics within the area of graph transformation in the last four decades. In Sect. 3, we revisit the starting point by recalling the basic notions and results introduced in 1976. In the rest of the paper, we point out how parallel graph transformation can be used in the context of some significant issues of parallelism: parallel modes of language generation, parallelization of algorithm, infinite graph theory, and transformation of other parallel models into graph transformation models. In all cases, the sequentialization and parallelization theorems play an important role. All explicit examples are new. The parallelization of algorithm and the use of graph transformation in the theory of infinite graphs are novel attempts as far as we know. In more detail, the aspects considered in the Sects. 4 to 7 are the following.

In proper context, parallel generation may provide more generative power than sequential generation. Consider, for example, Lindenmeyer systems with

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 61–82, 2018.
https://doi.org/10.1007/978-3-319-75396-6_4

context-free rules applied with maximum parallelism. If one requires in addition that in each step one set of rules out of several possibilities is used, then these TOL-systems can generate languages that are not context-free. In Sect. 4, we consider hyperedge replacement grammars as a counterpart to context-free grammars and show that they behave in the same way if they are used in the TOL-mode of transformation.

Frequently and in many contexts, parallelism is expected to allow more efficient computations than sequential ones. In Sect. 5, we demonstrate that this idea works also in the framework of graph transformation by analyzing and parallelizing a shortest-path algorithm.

In contrast to the usual approaches, our notion of parallel rules is not restricted to a finite number of component rules. In Sect. 6, we discuss the effect of the application of infinite parallel rules and exemplify that infinite graph transformation may contribute to infinite graph theory.

Parallel graph transformation provides a suitable framework for the modeling and analysis of parallel and concurrent processes. In particular, other approaches to parallel processing can be transformed into and interpreted as graph-transformational parallelism. This is demonstrated for the well-known cellular automata in Sect. 7.

Related work is discussed in the respective sections.

2 Preliminaries

In this section, we recall some basic notions and notations of graphs and graph transformation. In particular, we define the disjoint union of sets and graphs and consider its elementary properties as prerequisites for the introduction of parallel rules.

2.1 Disjoint Union of Sets

Let $F = (X_i)_{i \in I}$ be a family of sets for some index set I. Then a set X together with injective mappings $in_i \colon X_i \to X$ for all $i \in I$ is a *disjoint union* of F if $in_i(X_i) \cap in_j(X_j) = \emptyset$ for all $i \neq j$ and $\bigcup_{i \in I} in_i(X_i) = X$. X may be denoted by $\sum_{i \in I} X_i$. For $I = \{1, 2\}$, one may denote X also by $X_1 + X_2$. A disjoint union can be constructed as $\bigcup_{i \in I} (\{i\} \times X_i)$ with $in_i \colon X_i \to \bigcup_{i \in I} (\{i\} \times X_i)$ defined by $in_i(x) = (i, x)$ for all $i \in I$ and $x \in X_i$.

A disjoint union X with $(in_i)_{i \in I}$ of $F = (X_i)_{i \in I}$ has the following (universal) property: If Y is a set and $(f_i \colon X_i \to Y)_{i \in I}$ is a family of mappings, then there is a unique mapping $f \colon X \to Y$ with $f \circ in_i = f_i$ for all $i \in I$. It is defined by $f(x) = f_i(\overline{x})$ for the unique $\overline{x} \in X_i$ with $in_i(\overline{x}) = x$. It may be denoted by $\langle f_i \rangle_{i \in I}$. The property means that a disjoint union of F is a categorial coproduct in the category of sets. Using the property, one can easily show that Y with $(f_i)_{i \in I}$ is a disjoint union of F if and only if $\langle f_i \rangle_{i \in I}$ is bijective. Given a set Y

and a bijection $f\colon X \to Y$, then Y with the injections $(f \circ in_i)_{i\in I}$ is a disjoint union of F provided that X with $(in_i)_{i\in I}$ is one. In other words, the construction of disjoint unions of sets is unique up to bijection.

We use two further nice properties of disjoint unions. The first property is a (de-)composition property: $\sum_{i\in I} X_i = \sum_{i\in I'} X_i + \sum_{i\in I\setminus I'} X_i$ for $I' \subseteq I$. This means in particular that the $+$-operator is commutative and associative. The second property is that inclusions are preserved. Let $F = (X_i)_{i\in I}$ and $F' = (Y_i)_{i\in I}$ be two families of sets and Y with $(in_i^Y)_{i\in I}$ a disjoint union of F'. Let $(g_i\colon X_i \to Y_i)_{i\in I}$ be a family of mappings. Then $\langle in_i^Y \circ g_i\rangle_{i\in I}$ is also denoted by $\sum_{i\in I} g_i$. It is injective if all g_i are injective. It can be chosen as inclusion of $\sum_{i\in I} X_i$ into $Y = \sum_{i\in I} Y_i$ if the g_i are inclusions, i.e. $X_i \subseteq Y_i$ for all $i \in I$.

2.2 Basic Notions of Graphs

Let Σ be a set of labels. A (directed edge-labeled) *graph* over Σ is a system $G = (V, E, s, t, l)$ where V is a set of *nodes*, E is a set of *edges*, $s, t\colon E \to V$ are mappings assigning a *source* $s(e)$ and a *target* $t(e)$ to every edge in E, and $l\colon E \to \Sigma$ is a mapping assigning a label to every edge in E. An edge e with $s(e) = t(e)$ is also called a *loop*. The components V, E, s, t, and l of G are also denoted by V_G, E_G, s_G, t_G, and l_G, respectively. The set of all graphs over Σ is denoted by $\mathcal{G}_\Sigma$.

This notion of graphs is flexible enough to cover other types of graphs. *Simple graphs* form a subclass consisting of those graphs two edges of which are equal if their sources and their targets are equal respectively. A label of a loop can be interpreted as a label of the node to which the loop is attached so that *node-labeled graphs* are covered. We assume a particular label $*$ which is omitted in drawings of graphs. In this way, graphs where all edges are labeled with $*$ may be seen as *unlabeled graphs*. Moreover, *undirected graphs* can be represented by directed graphs if one replaces each undirected edge by two directed edges attached to the same two nodes, but in opposite directions. Finally, *hypergraphs* can be handled by the introduced type of graphs as done explicitly in Sect. 4.

A graph $G \in \mathcal{G}_\Sigma$ is a *subgraph* of a graph $H \in \mathcal{G}_\Sigma$, denoted by $G \subseteq H$, if $V_G \subseteq V_H$, $E_G \subseteq E_H$, $s_G(e) = s_H(e)$, $t_G(e) = t_H(e)$, and $l_G(e) = l_H(e)$ for all $e \in E_G$. In drawings of graphs and subgraphs, shapes, colors, and names are used to indicate the identical nodes and edges.

Given a graph, a subgraph is obtained by removing some nodes and edges subject to the condition that the removal of a node is accompanied by the removal of all its *incident* edges. More formally, let $G = (V, E, s, t, l)$ be a graph and $X = (V_X, E_X) \subseteq (V, E)$ be a pair of sets of nodes and edges. Then $G \setminus X = (V \setminus V_X, E \setminus E_X, s', t', l')$ with $s'(e) = s(e)$, $t'(e) = t(e)$, and $l'(e) = l(e)$ for all $e \in E \setminus E_X$ is a subgraph of G if and only if there is no $e \in E \setminus E_X$ with $s(e) \in V_X$ or $t(e) \in V_X$. This condition is called *dangling condition* of X in G.

For graphs $G, H \in \mathcal{G}_\Sigma$ a *graph morphism* $g\colon G \to H$ is a pair of mappings $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that are structure-preserving, i.e.

$g_V(s_G(e)) = s_H(g_E(e))$, $g_V(t_G(e)) = t_H(g_E(e))$, and $l_H(g_E(e)) = l_G(e)$ for all $e \in E_G$. We may write $g(v)$ and $g(e)$ for nodes $v \in V_G$ and edges $e \in E_G$ since the indices V and E can be reconstructed easily from the type of the argument. If g_V and g_E of a graph morphism $g\colon G \to H$ are bijective, then g is called a *graph isomorphism*. In this case G and H are *isomorphic*, denoted by $G \simeq H$.

For a graph morphism $g\colon G \to H$, the image of G in H is called a *match* of G in H, i.e. the match of G with respect to the morphism g is the subgraph $g(G) \subseteq H$ which is induced by $(g(V), g(E))$.

Given $F \subseteq G$, then the two inclusions of the sets of nodes and edges define a graph morphism. It is also easy to see that the (componentwise) sequential composition of two graph morphisms $f\colon F \to G$ and $g\colon G \to H$ yields a graph morphism $g \circ f\colon F \to H$. Consequently, if f is the inclusion w.r.t. $F \subseteq G$, then $g(F)$ is the match of F in H w.r.t. g restricted to F.

Instead of removing nodes and edges, one may add some nodes and edges to extend a graph such that the given graph is a subgraph of the extension. The addition of nodes causes no problem at all, whereas the addition of edges requires the specification of their labels, sources, and targets, where the latter two may be given or new nodes. Let $G_1 = (V_1, E_1, s_1, t_1, l_1)$ be a graph and $(V_2, E_2, s_2, t_2, l_2)$ be a structure consisting of two sets V_2 and E_2 and three mappings $s_2\colon E_2 \to V_1 + V_2$, $t_2\colon E_2 \to V_1 + V_2$, and $l_2\colon E_2 \to \Sigma$. Then $H = G_1 + (V_2, E_2, s_2, t_2, l_2) = (V_1 + V_2, E_1 + E_2, \langle \hat{s}_1, s_2 \rangle, \langle \hat{t}_1, t_2 \rangle, \langle l_1, l_2 \rangle)$ is a graph with $G_1 \subseteq H$ where $\hat{s}_1, \hat{t}_1\colon E_1 \to V_1 + V_2$ with $\hat{s}_1(e) = s_1(e), \hat{t}_1(e) = t_1(e)$ for all $e \in E_1$.

Let $G = (G_i)_{i \in I}$ be a family of graphs. Then the *disjoint union* of G is defined by $\sum_{i \in I} G_i = (\sum_{i \in I} V_{G_i}, \sum_{i \in I} E_{G_i}, \sum_{i \in I} s_{G_i}, \sum_{i \in I} t_{G_i}, \langle l_{G_i} \rangle_{i \in I})$. The construction has all the properties summarized in Sect. 2.1 for the disjoint union of sets if one replaces the term *set* by *graph* (with the exception of the index set), *subset* by *subgraph* and *mapping* by *graph morphism*.

2.3 Rule-Based Graph Transformation

Formally, a *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathcal{G}_\Sigma$ such that K is a subgraph of L and R. The components L, K, and R of r are called *left-hand side*, *gluing graph*, and *right-hand side*, respectively.

The application of a graph transformation rule to a graph G consists of replacing a match of the left-hand side in G by the right-hand side such that the match of the gluing graph is kept. Hence, the application of $r = (L \supseteq K \subseteq R)$ to a graph $G = (V, E, s, t, l)$ comprises the following three steps.

First, a graph morphism $g\colon L \to G$ called *matching morphism* is chosen to establish a match of L in G subject to the *gluing condition* consisting of two parts: (a) the dangling condition of $g(L) \setminus g(K) = (g(V_L) \setminus g(V_K), g(E_L) \setminus g(E_K))$ in G; and (b) the identification condition requesting that two nodes or edges of L must be in K if they are identified in the match of L.

Second, the match of L up to $g(K)$ is removed from G, resulting is the intermediate graph $Z = G \setminus (g(L) \setminus g(K))$.

Third, the right-hand side R is added to Z by gluing Z with R in $g(K)$ yielding the graph $H = Z + (R \setminus K, g)$ where $(R \setminus K, g) = (V_R \setminus V_K, E_R \setminus E_K, s', t', l')$ with $s'(e') = s_R(e')$ if $s_R(e') \in V_R \setminus V_K$ and $s'(e') = g(s_R(e'))$ otherwise, $t'(e') = t_R(e')$ if $t_R(e') \in V_R \setminus V_K$ and $t'(e') = g(t_R(e'))$ otherwise, and $l'(e') = l_R(e')$ for all $e' \in E_R \setminus E_K$.

The extension of Z to H is properly defined because s' and t' map the edges of $E_R \setminus E_K$ into nodes of $V_R \setminus V_K$ or $g(V_K)$ which is part of V_Z. As the disjoint union is only unique up to isomorphism, the resulting graph is only unique up to isomorphism. Due to the construction, g can be restricted to $d\colon K \to Z$, and d can be extended to a right matching morphism $h\colon R \to H$ by the identity on $R \setminus K$.

Hence a rule application of r can be depicted by the following diagram.

$$\begin{array}{ccccc} L & \supseteq & K & \subseteq & R \\ \downarrow g & & \downarrow d & & \downarrow h \\ G & \supseteq & Z & \subseteq & H \end{array}$$

It is worth noting that both squares of the diagram are pushouts in the category of graphs if the subgraph relations in the diagram are interpreted as inclusion morphisms. Therefore, an equivalent definition based on double pushouts in category theory can be found in, e.g., [2]. Here the identification condition is significant because the left diagram is not a pushout if g does not obey the identification condition.

The application of a rule r to a graph G is denoted by $G \underset{r}{\Longrightarrow} H$. A rule application is called a *direct derivation*, and an iteration of direct derivations $G \simeq G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n \simeq H$ $(n \in \mathbb{N})$ is called a *derivation* from G to H. As usual, the derivation from G to H can also be denoted by $G \overset{n}{\underset{P}{\Longrightarrow}} H$ where $\{r_1, \ldots, r_n\} \subseteq P$, or by $G \overset{*}{\underset{P}{\Longrightarrow}} H$ if the number of direct derivations is not of interest. The subscript P may be omitted.

As the disjoint union is only uniquely defined up to isomorphism, derived graphs are also only uniquely constructed up to isomorphism. But without loss of generality, one can assume that nodes and edges, which are not removed, keep their identity. We make use of this fact in all our explicit examples.

A *graph class expression* may be any syntactic entity X that specifies a class of graphs $SEM(X) \subseteq \mathcal{G}_\Sigma$ like expressions or formula. A *control condition* may be any syntactic entity that restricts the derivation process. Explicit examples are introduced where they are needed.

A *graph transformation unit* is a system $gtu = (I, P, C, T)$ where I and T are graph class expressions to specify the *initial* and the *terminal* graphs respectively, P is a set of rules, and C is a control condition. Such a transformation unit specifies a binary relation $SEM(gtu) \subseteq \mathcal{G}_\Sigma \times \mathcal{G}_\Sigma$ that contains a pair (G, H) if and only if $(G, H) \in SEM(I) \times SEM(T)$ and there is a derivation $G \overset{*}{\underset{P}{\Longrightarrow}} H$ permitted by C.

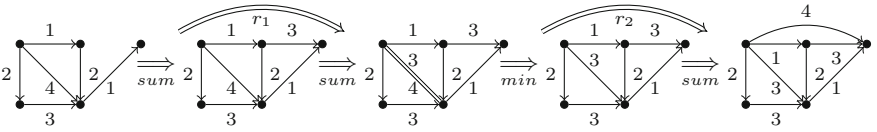
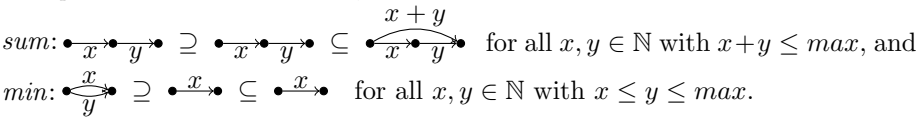


Fig. 1. A derivation based on the *shortest_paths*(*max*) graph transformation unit

Example 1. For some $max \in \mathbb{N}$, consider all rules of the form:



Given a graph with labels in $\mathbb{N}$, the application of a *sum*-rule adds an edge bridging a path of length 2 and summing up the labels of the path. A *min*-rule is applicable to each two parallel edges, keeping the edge with the smaller label or one of the two if the labels are equal. A sample derivation can be seen in Fig. 1. (The derivation applying r_1 and r_2 is explained in Sect. 3.1.) If the two edges of the left-hand side match a single edge, then the identification condition is not satisfied. The dangling condition is always satisfied because nodes are never removed. But if one modifies the *sum*–rule in such a way that the middle node and the edges of the gluing graph are omitted, then the dangling condition is not satisfied whenever the middle node is attached to more than two edges.

The *sum*–rule can be applied to each path of length 2 arbitrarily often. This can be avoided if one requires that there is no bridging edge with a label $z \leq x+y$ in the accessed graph. Such a negative context condition is an example of a control condition. The rules together with this control condition specify a graph transformation unit if one chooses proper initial and terminal graphs in addition. The constant expressions *loop-free*, *strictly-simple* and *max-labeled* denote the classes of graphs without loops; with at most one edge between every two nodes; and with labels in $\mathbb{N}$ whose sum does not exceed *max*, respectively. Combined by &, one gets the intersection of the three classes. Then the expression 0-*looped*(*max-labeled* & *strictly-simple* & *loop-free*) defines the graphs in the intersection with a 0-loop at each node in addition. Starting with these graphs as initial graphs and applying the rules according to the control condition as long as possible, results in graphs where each edge between nodes v and v' is labeled with the shortest distance between v and v' in the respective initial graph, i.e. the minimum label sum of all paths from v to v'. The terminal graphs can be specified by the expressions $\{sum, min\}$-*reduced*. In summary, it is justified to call the unit *shortest_paths*(*max*). It is schematically given in Fig. 2. The components of the unit are preceded with respective keywords, the negative context condition of the *sum*-rule is denoted by the dashed edge.

The example is further discussed in Sect. 5.

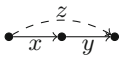
shortest_paths(*max*)
initial: 0-*looped*(*max-labeled* & *strictly-simple* & *loop-free*)
rules: *sum*, *min*
control:  with $z \leq x + y$ wrt *sum*
terminal: {*sum*, *min*}-*reduced*

Fig. 2. The graph transformation unit *shortest_paths*(*max*)

3 Parallel Graph Transformation

In this section, we recall the notion of parallel graph transformation and its fundamental properties of sequentialization and parallelization as introduced in [1] only slightly modified. Our rules consist of two inclusions each while a rule in [1] consists of an injective graph morphism from the gluing graph to the left-hand side graph and an arbitrary graph morphism from the gluing graph into the right-hand side graph. Moreover, the parralel rule in [1] is a disjoint union of two rules while we consider parallel rules composed of an arbitrary family of component rules.

Definition 1. Let $F = (r_i)_{i \in I} = (L_i \supseteq K_i \subseteq R_i)_{i \in I}$ be a family of rules. Then the *parallel rule* of F is defined by $r(F) = \sum_{i \in I} r_i = (\sum_{i \in I} L_i \supseteq \sum_{i \in I} K_i \subseteq \sum_{i \in I} R_i)$.

Due to the properties of disjoint unions, the parallel rule is an ordinary rule so that parallel derivations are just derivations applying parallel rules.

3.1 Sequentialization and Parallelization Theorems

Let $G \underset{r_1+r_2}{\Longrightarrow} X$ be a direct parallel derivation, let $g\colon L_1 + L_2 \to G$ be the corresponding matching morphism, and let $in_1\colon L_1 \to L_1 + L_2$ be the inclusion of L_1 into $L_1 + L_2$. Then $g_1 = g \circ in_1$ defines a matching morphism of L_1 into G. It is easy to see that g_1 satisfies the gluing condition using the satisfaction of the gluing condition of g. This yields a direct derivation $G \underset{r_1}{\Longrightarrow} H_1$. Let Z_1 be its intermediate graph. Then the identification condition satisfied by g yields $g(L_2) \subseteq Z_1$. This allows one to define a matching morphism g_2' of L_2 into H_1 by $g_2'(x) = g(x)$ for all x of L_2. Using again the gluing condition satisfied by g, it turns out that g_2' satisfies the gluing condition and yields a direct derivation $H_1 \underset{r_2}{\Longrightarrow} X_1$. Finally, one can show by the construction of direct derivations and some basic properties of union and difference of sets that X and X_1 are isomorphic. Altogether, the reasoning yields the following result.

Theorem 1 (Sequentialization of parallel derivations). *Let r_1, r_2 be rules and $G \underset{r_1+r_2}{\Longrightarrow} X$ be a direct derivation. Then there is a derivation $G \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} X$.*

The resulting derivation is called the *sequentialization* of $G \underset{r_1+r_2}{\Longrightarrow} X$. We also get $G \underset{r_2}{\Longrightarrow} H_2 \underset{r_1}{\Longrightarrow} X$ as $r_1 + r_2 = r_2 + r_1$. The identification condition satisfied by the given matching morphism $g\colon L_1 + L_2 \to G$ implies for the two matching morphims g_1 and g_2 which restrict g to the components L_1 and L_2 that $g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$, i.e. the two matches may overlap, but only in common gluing elements. A further analysis yields for the right matching morphism $h_1\colon R_1 \to H_1$ of $G \underset{r_1}{\Longrightarrow} H_1$ and the matching morphism $g_2'\colon L_2 \to H_1$ constructed above: $h_1(R_1) \cap g_2'(L_2) \subseteq h_1(K_1) \cap g_2'(K_2)$. The two properties are called parallel and sequential independence respectively. Independence refers to the fact that the application of one of the two rules does not prevent or influence the application of the other one. Nicely enough, independence is sufficient for parallelization.

Definition 2. Let $r_i = (L_i \supseteq K_i \subseteq R_i)$ for $i = 1, 2$ be rules.

1. Two direct derivations $G \underset{r_i}{\Longrightarrow} H_i$ with the matching morphism $g_i\colon L_i \to G$ respectively are *parallel independent* if $g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$.
2. Successive direct derivations $G \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} X$ with the right matching morphism $h_1\colon R_1 \to H_1$ and the (left) matching morphism $g_2'\colon L_2 \to H_1$ are *sequentially independent* if $h_1(R_1) \cap g_2'(L_2) \subseteq h_1(K_1) \cap g_2'(K_2)$.

Theorem 2 (Parallelization of independent direct derivations). *Let $r_i = (L_i \supseteq K_i \subseteq R_i)$ for $i = 1, 2$ be rules.*

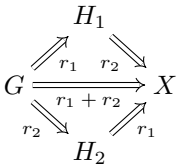
1. *Let $G \underset{r_i}{\Longrightarrow} H_i$ for $i = 1, 2$ be parallel independent direct derivations with matching morphisms $g_i\colon L_i \to G$. Then there is a direct parallel derivation $G \underset{r_1+r_2}{\Longrightarrow} X$ for some $X \in \mathcal{G}_\Sigma$ with matching morphism $\langle g_1, g_2\rangle\colon L_1 + L_2 \to G$.*
2. *Let $G \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} X$ be sequentially independent direct derivations. Then there is a parallel derivation $G \underset{r_1+r_2}{\Longrightarrow} X$.*

The first direct derivations of the two possible sequentializations of the constructed direct parallel derivation in Point 1 coincide with the given direct derivations.

Let $g_1\colon L_1 \to G$ and $g_2'\colon L_2 \to H_1$ be the matching morphisms of the given direct derivation in Point 2. Then the sequential independence and the construction of direct derivations yield $g_2'(L_2) \subseteq G$. This allows to define a matching morphism $g_2\colon L_2 \to G$. Then the direct derivation $G \underset{r_1+r_2}{\Longrightarrow} X$ is given by the matching morphims $\langle g_1, g_2\rangle\colon L_1 + L_2 \to G$. $G \underset{r_1+r_2}{\Longrightarrow} X$ is called *parallelization* of $G \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} X$.

Example 2. Look at the derivation in Fig. 1. The first two steps are sequentially independent as the second step does not match the edge generated by the first step. Moreover, the last two steps are also sequentially independent so that the two possible parallelizations yield the derivation applying $r_1 = sum + sum$ and $r_2 = min + sum$ in Fig. 1.

The sequentialization and parallelization theorems involve the following derivations from G to X where the two direct derivations $G \underset{r_1}{\Longrightarrow} H_1$ and $G \underset{r_2}{\Longrightarrow} H_2$ are parallel independent and both derivations $G \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} X$ and $G \underset{r_2}{\Longrightarrow} H_2 \underset{r_1}{\Longrightarrow} X$ are sequentially independent.



The whole diagram is obtained (a) from the direct parallel derivation $G \underset{r_1+r_2}{\Longrightarrow} X$, (b) from the two parallel independent direct derivations $G \underset{r_1}{\Longrightarrow} H_1$ and $G \underset{r_2}{\Longrightarrow} H_2$, or (c) from each of the sequentially independent derivations from G to X.

As pointed out in, e.g., [3], the diagram reflects the concurrency of two events: One may happen after the other or the other way round or both may happen simultaneously. All three ways to move from G to X are equally possible. There is neither a causal dependence nor any mutual influence.

The results are particularly significant with respect to the construction of matching morphisms which is the most time-consuming part of a rule application. Whether a graph morphism from L to G exists, is a well-known NP-complete problem if L and G are finite input graphs of arbitrary size. Hence, all known algorithms that find graph morphisms for finite, but arbitrary large L and G are exponential. In contrast to that, the search for matching morphisms becomes polynomial in the size of G if L is fixed or the size of L is bounded by a constant. This is the case if one assumes finite sets of finite rules. The number of mappings from a set with k elements to a set with n elements is n^k so that one can check all possible matchings in polynomial time even in an exhaustive search. This applies in particular to the case of finite sets of finite rules. But it does not apply to parallel rules because their left-hand sides may become arbitrary large so that one would have to face the problem of NP-completeness without further knowledge. Fortunately, we know that the matching morphism of a parallel rule is composed of matching morphisms of the atomic component rules so that matching morphism for parallel rules can be found in polynomial time if the number of components is polynomial or the components can be processed in parallel.

3.2 Shifts and Canonical Derivations

The three derivations from G to X in the diagram above may be considered as equivalent from a concurrency point of view. Further, this view can be extended to arbitrary derivations so that the equivalence classes represent concurrent processes. But the equivalence classes may be exponentially large. In order to give an efficient representation a shift operation can be defined as a certain combination of sequentialization and parallelization such that shifting as long as possible yields unique canonical representatives.

Let $s = G_0 \overset{*}{\Longrightarrow} G_i \underset{r_1+r_2}{\Longrightarrow} G_{i+2} \overset{*}{\Longrightarrow} G_n$ and $s' = G_0 \overset{*}{\Longrightarrow} G_i \underset{r_1}{\Longrightarrow} G_{i+1} \underset{r_2}{\Longrightarrow} G_{i+2} \overset{*}{\Longrightarrow} G_n$ be two derivations where $G_i \underset{r_1}{\Longrightarrow} G_{i+1} \underset{r_2}{\Longrightarrow} G_{i+2}$ is a sequentialization of $G_i \underset{r_1+r_2}{\Longrightarrow} G_{i+2}$. Then s is *seq-related* to s' denoted by $s \underset{seq}{\rightarrow} s'$. The equivalence closure is denoted by $\sim$.

Let us restrict the consideration to parallel derivations where only parallel rules with a finite number of component rules are applied. Then the equivalence classes are always finite, but they may have an exponential number of elements. Let s, s' and s'' be three derivations with $s \underset{seq}{\rightarrow} s'$ and $s'' \underset{seq}{\rightarrow} s'$ of the form

$$G_0 \overset{*}{\Longrightarrow} G_{i-1} \underset{r_1}{\Longrightarrow} G_i \underset{r_2}{\Longrightarrow} G_{i+1} \underset{r_3}{\Longrightarrow} G_{i+2} \overset{*}{\Longrightarrow} G_n$$

with $r_1 + r_2$ spanning G_{i-1} to G_{i+1} and $r_2 + r_3$ spanning G_i to G_{i+2}

where s is the derivation with $r_1 + r_2$, s'' is the derivation with $r_2 + r_3$, and s' is the derivation in the middle. Then s'' is *shift-related* to s, denoted by $s'' \underset{shift}{\rightarrow} s$. Moreover, if only s and s' are given with $s \underset{seq}{\rightarrow} s'$, then s' is also *shift-related* to s, denoted by $s' \underset{shift}{\rightarrow} s$. A rule applied in step $i + 1$ can only be shifted if its direct derivation is sequentially independent of the preceding direct derivation, and it can be shifted i times at most. Therefore, *shift*-sequences are never longer than $n(n-1)/2$ if n is the number of applications of atomic rules. In particular, one gets always *shift*-reduced derivations if one shifts as long as possible. These *shift*-reduced derivations are called *canonical* because they are unique representatives of their equivalence classes.

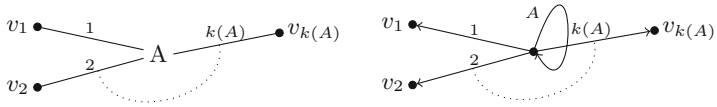
Theorem 3. *Let s and s' be two canonical derivations with $s \sim s'$, then $s = s'$.*

The proof is based on the fact, that the *shift*-relation is locally Church-Rosser: Given $s \underset{shift}{\rightarrow} s'$ and $s \underset{shift}{\rightarrow} s''$, then there is $\overline{s}$ with $s' \overset{*}{\underset{shift}{\rightarrow}} \overline{s}$ and $s \overset{*}{\underset{shift}{\rightarrow}} \overline{s}$.

Example 3. The derivation applying r_1 and r_2 in Fig. 1 is canonical.

3.3 Related Work

In [1] and in the present paper, the proofs of the stated results are only roughly sketched. The full proofs can be found in [2,4]. In the last 40 years, the topic of parallel graph transformation has been further studied by many researchers in various respects modifying and generalizing the approach. As it is impossible to refer to all related publications – there are too many – the reader may consult Volume 3 of the Handbook of Graph Grammars and Computing by Graph Transformation [5] and the two monographs [6,7] where much of the work is systematically presented in the context of the double-pushout approach, and the important references are given in the introductions of the books and of the respective chapters. This covers nicely typed attributed graphs, high-level replacement systems in adhesive categories as well as concurrent and amalgamated rules. Concerning the single- and the sesqui-pushout approaches, the reader is referred to [8,9].



(a) Hyperedge (b) Its graph representation

Fig. 3. The graph representation of a hyperedge

4 Parallelism of Hyperedge Replacement

Hyperedge replacement (see, e.g., [10–13]) is a kind of context-free hypergraph transformation. Hyperedges of hypergraphs may be incident to arbitrary sequences of nodes rather than to two nodes as ordinary edges. But there is a straightforward way to formulate hyperedge replacement within the graph setting introduced in Sect. 2 (cf. [14]). With respect to parallelism, hyperedge replacement is of interest in at least two ways.

First, the context-freeness lemma provides a decomposition of derivations into a set of fibers that corresponds to the decomposition of a direct parallel derivation into direct derivations applying the rule components.

Second, due to the sequentialization theorem, sequential and parallel hyperedge replacement have the same generative power. But if one applies hyperedge replacement rules in the mode of TOL-systems, then one can get quite different languages.

4.1 Hyperedge Replacement and Its Context-Freeness Lemma

We assume some subset $N \subseteq \Sigma$ of *nonterminals* which are typed, i.e. there is an integer $k(A) \in \mathbb{N}$ for each $A \in N$. Moreover, we assume that Σ contains the numbers $1, \ldots, max$ for some $max \in \mathbb{N}$ with $k(A) \leq max$ for all $A \in N$. A *hyperedge* with label $A \in N$ is meant to be an atomic item which is attached to a sequence of nodes $v_1 \cdots v_{k(A)}$. It can be represented by a node with an A-labeled loop and $k(A)$ edges the labels of which are $1, \ldots, k(A)$, respectively, and the targets of which are $v_1, \ldots, v_{k(A)}$, respectively, as depicted in Fig. 3. Accordingly, we call such a node with its incident edges an *A-hyperedge*. A graph is said to be *N-proper* if each occurring nonterminal and each occuring number between 1 and max belong to some hyperedge. Each $A \in N$ induces a particular N-proper graph $A^\bullet$ with the nodes $\{0, \ldots, k(A)\}$ and a single hyperedge where the A-loop is attached to 0 and i is the target of the edge labeled with i for $i = 1, \ldots, k(A)$. Let $[k(A)]$ denote the discrete graph with the nodes $\{1, \ldots, k(A)\}$. Using these notations, a rule of the form $A^\bullet \supseteq [k(A)] \subseteq R$ for some N-proper graph R is a *hyperedge replacement rule*, which can be denoted by $A ::= R$ for short. A *hyperedge replacement grammar* is a system $HRG = (N, T, P, S)$ with $S \in N$, $T \subseteq \Sigma$ with $T \cap N = \emptyset$, and a set of hyperedge replacement rules P with finite right-hand sides. Its generated language contains all terminal graphs that are derivable from $S^\bullet$, i.e. $L(HRG) = \{H \in \mathcal{G}_T \mid S^\bullet \overset{*}{\underset{P}{\Longrightarrow}} H\}$.

In this way, hyperedge replacement is just a special case of graph transformation, but with some very nice properties.

1. Let $r = (A ::= R)$ be a hyperedge replacement rule and G an N-proper graph with an A-hyperedge y. Then there is a unique graph morphism $g\colon A^\bullet \to G$ mapping $A^\bullet$ to the A-hyperedge y such that the gluing condition is satisfied and therefore r is applicable to G.
2. The directly derived graph H is N-proper and is obtained by removing y, i.e. by removing the node with the A-loop and all other incident edges, and by adding R up to the nodes $1, \ldots, k(A)$ where edges of R incident to $1, \ldots, k(A)$ are redirected to $g(1), \ldots, g(k(A))$, respectively. Due to this construction, H may be denoted by $G[y/R]$.
3. Two direct derivations $G \underset{r_1}{\Longrightarrow} H_1$ and $G \underset{r_2}{\Longrightarrow} H_2$ are parallel independent if and only if they replace distinct hyperedges.
4. A parallel rule $r = \sum_{i \in I} r_i$ of hyperedge replacement rules $r_i = (A_i ::= R_i)$ for $i \in I$ is applicable to G if and only if there are pairwise distinct A_i-hyperedges y_i for all $i \in I$. In analogy to the application of a single rule, the resulting graph may be denoted by $G[y_i/R_i \mid i \in I]$.
5. If $I = I_1 + I_2$, then $G[y_i/R_i \mid i \in I] = (G[y_i/R_i \mid i \in I_1])[y_i/R_i \mid i \in I_2]$.
6. Two successive direct derivations $G \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} H$ are sequentially independent if and only if the hyperedge replaced by the second step is not created by the first one.

Altogether, the direct derivations through hyperedge replacement rules can be ordered arbitrarily as long as they deal with different hyperedges. This observation leads to the following result.

Theorem 4 (Context-Freeness Lemma). *Let $HRG = (N, T, P, S)$ be a hyperedge replacement grammar and let $A^\bullet \overset{n+1}{\underset{P}{\Longrightarrow}} H$ be a derivation. Then there are some rule $A ::= R$ and a derivation $A(y)^\bullet \overset{n(y)}{\underset{P}{\Longrightarrow}} H(y)$ for each hyperedge y of R with label $A(y)$ such that $H = R[y/H(y) \mid y \in Y_R]$ and $\sum_{y \in Y_R} n(y) = n$, where Y_R is the set of hyperedges of R.*

A derivation $A^\bullet \overset{n+1}{\Longrightarrow} H$ has $A^\bullet \Longrightarrow R$ as the first step. The tail $R \overset{n}{\Longrightarrow} H$ can be decomposed into fibers $A(y)^\bullet \overset{n_i}{\Longrightarrow} H(y)$ for $y \in Y_R$. The fibres induce rules $A(y) ::= H(y)$ for $y \in Y_R$. They can be applied to R in parallel yielding $R \Longrightarrow H$. In this way, hyperedge replacement allows to generalize the sequentialization and parallelization of direct derivations to derivations.

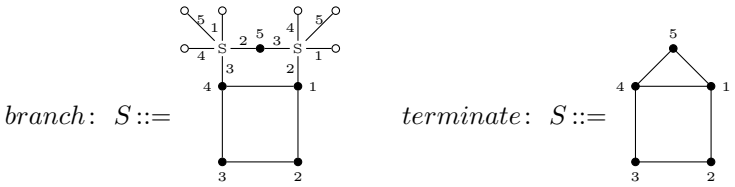
4.2 Maximum Parallel Hyperedge Replacement

Given a hyperedge replacement grammar $HRG = (N, T, P, S)$ and $H \in L(HRG)$. Then there is a derivation $S^\bullet \overset{*}{\Longrightarrow} H$ which can be transformed into a canonical derivation. As the replacements of two hyperedges are parallel independent and

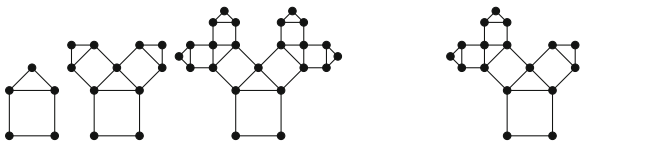
as H is terminal, each direct derivation of the canonical derivation replaces all hyperedges in parallel. This means that canonical derivations are maximum parallel normal forms to generate $L(HRG)$, but maximum parallelism does not extend or vary the generative power.

This changes if the set of rules is partitioned into subsets $P_1, \ldots, P_k$ for some $k \geq 1$ and each direct derivation takes one of them and applies the rules with maximum parallelism, i.e. in the style of TOL- and ETOL-systems (see, e.g., Chap. 5 in [15]). The TOL-mode of hyperedge replacement is a further example of a control condition. It allows to generate all languages that are generated by ordinary hyperedge replacement grammars because one can choose $P_1 = P$. But it also increases the generative power which is proved by a separating example.

Example 4. Consider the hyperedge replacement grammar $KOCHTREE = (\{S\}, \{*\}, P, S)$ where S has type 5 and P contains two rules:



If one decomposes P into $\{branch\}$ and $\{terminate\}$ and applies one or the other with maximum parallelism, one gets very regular finite approximations of the Koch tree (depicted in Fig. 4a). If one applies the rules arbitrarily, then one can also get asymmetric trees (like the one depicted in Fig. 4b). As long as the rule *branch* is used, the number of hyperedges doubles and each hyperedge replacement produces 4 (undirected) edges such that the language of regular Koch trees grows exponentially. On the other hand, it is a well-known fact that the languages generated by ordinary hyperedge replacement grammars have a sublinear growth so that they cannot generate regular Koch trees. Altogether, this shows that hyperedge replacement grammars with a TOL-mode of transformation are more powerful than without.



(a) Regular Koch trees (b) Asymmetric Koch tree

Fig. 4. Approximations of the Koch tree

4.3 Related Work

While hyperedge replacement is a well-studied area of graph transformation, we are not aware of much work on parallel generation of graph languages. But we

would like to mention that we introduced recently fusion grammars in [16] that display quite strong parallelization properties and extend the generative power of hyperedge replacement grammars.

5 Parallelization of Graph Algorithms

A major promise of parallelism is that parallel computation can be faster than sequential computation. Consider, for example, a totally balanced binary tree of height n. It has 2^n leaves, and therefore, a full traversal needs at least this many steps whereas traversing the tree from the root to the leaves with maximum parallelism takes n steps. So it seems worthwhile to look into graph transformation whether the use of parallel derivations can produce similar effects. To demonstrate the potential of this line of research, we look into the well-known search for shortest paths.

5.1 The Case of Shortest Paths

Most shortest-path algorithms like the prominent ones by Floyd/Warshall [17,18] and by Dijkstra [19] are based on two elementary operations: the sequential composition of paths summing up the distances and keeping the path with minimum distance out of some parallel paths (i.e. paths with the same source and target nodes). The algorithms differ from each other by the order in which the two basic operations are applied.

Let us reconsider the graph transformation unit *shortest_paths*(max) in Example 1. To make sure that the unit computes shortest distances, the following correctness properties can be proved. The distance of a path p in a graph G is the sum of the distances of the edges on p and is denoted by $dist_G(p)$.

Proposition 1 (Correctness). Let $G \stackrel{*}{\Longrightarrow} H$ be a derivation where G is initial and H is terminal. Then the following hold:

1. For every shortest path p from v to v' in G, there is some $e \in E_H$ with $s_H(e) = v$, $t_H(e) = v'$, and $l_H(e) = dist_G(p)$.
2. For every $e \in E_H$, there is a shortest path p from $s_H(e)$ to $t_H(e)$ in G with $l_H(e) = dist_G(p)$.

The first statement can be proved by induction on the length of shortest paths, the second one by induction on the length of derivations. The details are omitted for reasons of space limitations.

Now we consider the parallelization of the algorithm. The graph transformation unit *shortest_paths_in_parallel*(max) extends the unit *shortest_paths*(max) by the control condition

$$(sum[double{-}free\ maxpar]; min[largest\ maxpar])^*.$$

It requires that, repeatedly, the *sum*-rule is applied with double-free maximum parallelism followed by the largest maximum parallel application of the *min*-rule.

In a double-free parallel rule application of *sum*, no two matches of left-hand sides may overlap entirely. A largest parallel rule application of *min* must involve as many *min*-rules as possible.

As the left-hand side of the *sum*-rule coincides with the gluing graph, each two applications of *sum* are parallel independent. Therefore, there are at most $n \cdot (n-1) \cdot (n-2)$ double-free applications of *sum* where n is the number of nodes in the initial graph. The following largest maximum parallel *min*-step makes sure that no two parallel edges are left. More precisely, two *min*-applications are parallel independent if they match four different edges or intersect in the edge that is kept. Therefore, whenever there are m parallel edges between two nodes, the largest parallel step removes $m-1$ of them, and this happens if all applications of *min* choose the same edge to be kept.

That the unit computes the shortest distances between each two nodes can be seen as follows. The initial and terminal graphs are the same as in the unit *shortest_paths*(*max*) above. A parallel derivation from an initial to a terminal graph can be sequentialized due to the sequentialization theorem. In this sequential derivation, a *sum*-application may occur that does not obey the negative application condition. But then there is already an edge as good as or better than the edge generated by *sum*. Hence, this step as well as the *min*-step that removes this superfluous edge later on can be omitted without changing the result. If the sequential derivation is modified in this way as long as possible, then we end up with a derivation from an initial to a terminal graph in *shortest_paths*(*max*). Hence the correctness of *shortest_paths_in_parallel*(*max*) follows from the correctness of *shortest_paths*(*max*).

A closer look reveals that the edges after k rounds of a parallel *sum*-step followed by a parallel *min*-step represent the shortest paths of the initial graph of lengths up to 2^k. This implies that after a logarithmic number of parallel steps the terminal graph is reached.

Proposition 2 (Correctness and derivation length). Let $G \overset{2k}{\Longrightarrow} H$ be a derivation in *shortest_paths_in_parallel*(*max*) from an initial graph to a terminal graph with alternating parallel *sum*- and *min*-steps according to the control condition. Then Points 1 and 2 of Proposition 1 hold as well, and the length of the derivation has a logarithmic bound, i.e., $2^k \leq n-1$ where n is the number of nodes in G.

In a similar way, well-known shortest paths algorithms can be parallelized. For example, the parallelization of Mahr's algorithm [20] (which originally is of the order $n^3 \cdot \log n$) yields a logarithmic number of parallel steps and the parallelization of the Floyd/Warshall algorithm (which originally is a cubic algorithm) yields a linear number of parallel steps. But it should be noted that the short parallel derivations do not improve the complexity automatically, but only if the matching of the parallel rules can be found in a time bound that is – multiplied by the logarithmic length of the derivations – still smaller than the complexity of the corresponding sequential algorithms.

5.2 Related Work

There is not much work on parallel and distributed algorithms employing graph transformation. A noteworthy exception is the modeling of distributed algorithms by means of graph relabelling (see, e.g., [21]). Moreover, we would like to mention graph-multiset transformation (see [22]) that can be interpreted as a special case of the parallel graph transformation of Sect. 3 and allows to solve NP-complete graph problems by parallel computations of polynomial lengths. On the other hand, there is a realm of literature on parallel graph algorithm and very much interest in this topic so that the area seems to invite further and deeper studies by means of graph transformation.

6 Infinity

The definition of parallel graph transformation in Sect. 3 includes the case of parallel rules of an infinite family of component rules. In this section, we indicate that such infinite parallel rules may have some potential in the context of infinite graph theory (see, e.g., [23]) but only if one applies them to infinite graphs.

6.1 Application to Finite Graphs

Let $F = (r_i)_{i \in I} = (L_i \supseteq K_i \subseteq R_i)_{i \in I}$ be a family of rules for an infinite index set I. Let G be a finite graph and $G \underset{r(F)}{\Longrightarrow} H$ be an application of the parallel rule of F to G with the matching morphism $g = \langle g_i \rangle_{i \in I} \colon \sum_{i \in I} L_i \to G$. Then the definition of rule application reveals the following facts:

1. Let $I' = \{i \in I \mid K_i \neq L_i\}$ be the set of indices of erasing rules. Then I' is finite because otherwise g would not obey the identification condition. Therefore, $K_i = L_i$ for almost all $i \in I$.
2. Let $I'' = \{i \in I \mid K_i \neq R_i\}$ be the set of indices of adding rules. Then H is finite if and only if I'' is finite.
3. Let I'' be infinite. Then H contains an infinite number of finite subgraphs that are pairwise disjoint or H has a node with infinite degree or both is the case.

Infinite graphs consisting of infinitely many finite disjoint components or with nodes of infinite degree are considered as less interesting in finite graph theory. Therefore, the application of parallel rules of an infinite family of rules can make more sense only if one applies them to infinite graphs.

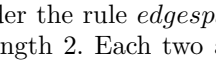
6.2 Application to Infinite Graphs

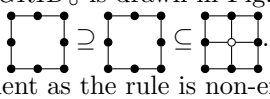
We are not going to study the application of parallel rules of an infinite family of rules to infinite graphs in any depth. But we would like to give an example that displays an interesting property and nourishes the hope that infinite graph transformation may be of interest.

(a) A finite section of *GRID* (b) A finite section of $GRID_\circ$

Fig. 5. Infinite plane grids

Example 5. Consider the infinite plane grid a finite section of which looks like the structure in Fig. 5a. Nodes are the points in the plane with integer coordinates. Each node has four outgoing edges to its Northern, Eastern, Southern, and Western neighbor respectively. As each node is neighbor of its four neighbors, the edges can be drawn as undirected edges. Formally, it can be defined as follows: $GRID = (\mathbb{Z} \times \mathbb{Z}, \mathbb{Z} \times \mathbb{Z} \times \{N, E, S, W\}, s_{GRID}, t_{GRID}, l_{GRID})$ with $s_{GRID}((x, y, D)) = (x, y)$, $t_{GRID}((x, y, N)) = (x, y + 1)$, $t_{GRID}((x, y, E)) = (x + 1, y)$, $t_{GRID}((x, y, S)) = (x, y - 1)$, $t_{GRID}((x, y, W)) = (x - 1, y)$, and $l_{GRID}((x, y, D)) = *$ for all $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ and $D \in \{N, E, S, W\}$.

Consider the rule *edgesplit*:  that splits an edge into a path of length 2. Each two applications of the rule are parallel independent if they match different edges. Therefore, one can apply the rules to all edges in parallel. A finite section of the result $GRID_\circ$ is drawn in Fig. 5b.

Consider now the rule *squaresplit*:  . Each two applications of this rule are parallel independent as the rule is non-erasing. Hence, one can apply the rule to all smallest squares (the cycles of length 8) of $GRID_\circ$ in parallel – one rule per square. Then the result is obviously isomorphic to *GRID*.

This kind of self-replication is remarkable as the applied rules are strictly monotonously growing in that they add more than they erase. Such a property is impossible in the context of finite graphs. Hence it may be worthwhile to study infinite graph transformation in more detail and depth.

7 Parallel Models of Computation

Parallel graph transformation is well-suited for modeling and analyzing parallel processes and, in particular, as a domain into which other visual approaches to parallel processing can be transformed. To demonstrate this, we model the well-known cellular automata as graph transformation units with massive parallelism.

7.1 The Case of Cellular Automata

Cellular automata are computational devices with massive parallelism known for many decades (see, e.g., [24–27]). A cellular automaton is a network of cells where each cell has got certain neighbor cells. A configuration is given by a mapping that associates a local state with each cell. A current configuration can change into a follow-up configuration by the simultaneous changes of all local states. The local transitions are specified by an underlying finite automaton where the local states of the neighbor cells are the inputs. If the network is infinite, one assumes a particular sleeping state that cannot change if all input states of neighbor cells are also sleeping. Consequently, all follow-up configurations have only a finite number of cells that are not sleeping if one starts with such a configuration.

To keep the technicalities simple, we consider 2-dimensional cellular automata the cells of which are the unit squares in the Euclidean plane *GRID* and can be identified by their left lower corner. The neighborhood is defined by a vector $N = (N_1, \ldots, N_k) \in (\mathbb{Z} \times \mathbb{Z})^k$ where the neighbor cells of (i, j) are given by the translations $(i, j)+N_1, \ldots, (i, j)+N_k$. If one chooses the local states as colors, a cell with a local state can be represented by filling the area of the cell with the corresponding color. Accordingly, the underlying finite automaton is specified by a finite set of colors, say *COLOR*, and its transition $d\colon COLOR \times COLOR^k \to COLOR$. Without loss of generality, we assume $white \in COLOR$ and use it as sleeping state, i.e. $d(white, white^k) = white$. Under these assumptions, a configuration is a mapping $S\colon \mathbb{Z} \times \mathbb{Z} \to COLOR$ and the follow-up configuration S' of S is defined by $S'((i, j)) = d(S((i, j)), (S((i, j) + N_1)), \ldots, S((i, j) + N_k)))$.

If one starts with a configuration S_0 which has only a finite number of cells the colors of which are not *white*, then only these cells and those that have them as neighbors may change the colors. Therefore, the follow-up configuration has again only a finite number of cells with other colors than *white*. Consequently, the simultaneous change of colors of all cells can be computed. Moreover there is always a finite area of the Euclidean plane that contains all changing cells. In other words, a sequence of successive follow-up configurations can be depicted as a sequence of pictures by filling the cells with their colors.

Example 6. The following instance of a cellular automaton may illustrate the concept. It is called *SIER*, has two colors, $COLOR = \{white, black\}$, and the neighborhood vector is $N = ((-1, 0), (0, 1))$ meaning that each cell has the cell to its left and the next upper cell as neighbors. The transition of *SIER* changes *white* into *black* if exactly one neighbor is *black*, i.e. $d\colon COLOR \times COLOR^2 \to COLOR$ with $d(white, (black, white)) = d(white, (white, black)) = black$ and $d(c, (c_1, c_2)) = c$ otherwise. Let S_0 be the start configuration with $S_0((0, 0)) = black$ and $S_0((i, j)) = white$ otherwise. Then one gets the configuration S_{30} in Fig. 6 after 30 transitions. The drawing illustrates that *SIER* iterates the Sierpinski gadget (see, e.g., [28]) if one starts with a single black cell.

Cellular automata can be transformed into graph transformation units. Let *CA* be a cellular automaton with the neighborhood vector $N = (N_1, \ldots, N_k) \in (Z \times \mathbb{Z})^k$, the set of colors *COLOR* and the transition function

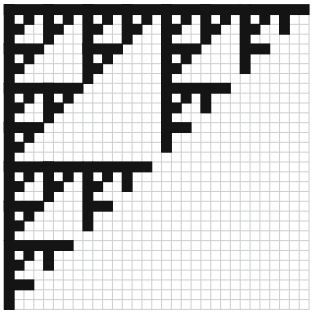
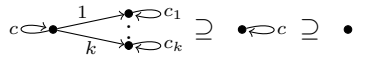


Fig. 6. A pictorial representation of the configuration S_{30} of *SIER*

$d\colon COLOR \times COLOR^k \to COLOR$. Then a configuration $S\colon \mathbb{Z} \times \mathbb{Z} \to COLOR$ can be represented by a graph $gr(N, S)$ with the cells as nodes, with k edges from each cell to its neighbors labeled with $1, \ldots, k$ in the order of the neighborhood, and a loop at each cell labeled with the color of the cell. The set of all these graphs is denoted by $\mathcal{G}(CA)$. If the color of a cell (i, j) changes, i.e. $d(S((i, j)), (S((i, j) + N_1), \ldots, S((i, j) + N_k))) \neq S(i, j)$, then the following rule with positive context  can be applied to the node (i, j) in $gr(N, S)$ provided that $c = S(i, j)$ and $c_p = S((i, j) + N_p)$ for $p = 1, \ldots, k$. Here the rule consists of the two inclusions to the left. The inclusion to the right is the positive context which serves as a control condition: The rule is applicable if the left-hand size is matched and the matching can be extended to the context. The set of all those rules is denoted by $\mathcal{R}(CA)$. A rule application removes a loop so that two rule applications are independent if and only if their matches do not overlap. Consequently, all applicable rules can be applied in parallel yielding $gr(N, S')$ where S' is the follow-up configuration of S. In other words, $gr(N, S) \Longrightarrow gr(N, S')$ with maximum parallelism is a direct derivation in the graph transformation unit $gtu(CA) = (\mathcal{G}(CA), \mathcal{R}(CA), \mathit{maxpar}, \mathcal{G}(CA))$.

Conversely, a derivation step $gr(N, S) \Longrightarrow H$ in $gtu(CA)$ changes a c-loop into a $d(c, (c_1, \ldots, c_k))$-loop at the node (i, j) if and only if, for $l = 1, \ldots, k$, the neighbor $(i, j) + N_l$ has a c_l-loop. All other c-loops are kept. This means that $H = gr(N, S')$. Summarizing, each cellular automaton can be transformed into a graph transformation unit such that the following correctness result holds.

Theorem 5. Let *CA* be a cellular automaton with neighborhood vector N and let *gtu(CA)* be the corresponding graph transformation unit. Then there is a transition from S to S' in *CA* if and only if $gr(N, S)) \Longrightarrow gr(N, S')$ in *gtu(CA)*.

Therefore, cellular automata behave exactly as their corresponding graph transformation units up to the representation of configurations as graphs. We have considered cellular automata over the 2-dimensional space $\mathbb{Z} \times \mathbb{Z}$. It is not difficult to see that all our constructions also work for the d-dimensional space

$\mathbb{Z}^d$ in a similar way. One may even replace the quadratic cells by triangular or hexagonal cells or use completely other networks.

7.2 Related Work

Like cellular automata, other approaches to parallel processing have been transformed into parallel graph transformation like Petri nets (cf. [3,29,30]), production networks (cf. [31]), artificial-ant colonies and particle swarms (cf. [32]). Moreover, parallel graph transformation provides a semantic domain for the graph-transformational modeling of various kinds of parallelism like for autonomous units (cf. [33]) and graph-transformational swarm computing (cf. [32]). Besides these examples that are closely related to the kind of parallel graph transformation considered in this paper, one encounters many further subjects in the literature where parallel rule application on and parallel evaluation of graph-like structures play an important rule like interaction nets, multi-agent systems, parallel and reversible circuits, various kinds of diagrams and networks. It may be worthwhile to look into the diverse research topics from a graph-transformational point of view.

8 Conclusion

In this paper, we have recalled the approach to parallel graph transformation that was introduced in [1], and have discussed some of its perspectives including the parallel generation of graph languages, the parallelization of graph algorithms, the infinite parallel graph transformation, and parallel graph transformation as a framework for the modeling of parallel processes. Further research on these topics can shed more light on their significance.

The theory of graph languages and those obtained by parallel generation in particular is not at all far developed. It may be worthwhile to study decidability and closure properties and to compare the various classes.

Given a specification of a graph algorithm by sequential graph transformation, one can always analyze the independence of rule applications to get a parallelized solution. The use of proper control conditions may lead to further improvement. Alternatively, graph algorithms may be modeled directly by means of parallel graph transformation. So far, not much work is done in this direction, but it may help to prove correctness and to analyze the complexity in a systematic way.

One can handle infinite graphs by the application of parallel rule with infinitely many finite component rules. There is a good chance that this machinery can contribute to infinite graph theory.

Parallel graph transformation has proven to provide a framework for the modeling of parallel processes and a domain into which other approaches to parallel-process modeling can be transformed. Therefore, it may be desirable to develop parallel graph transformation further into a visual modeling languages with suitable tool support.

Acknowledgment. We are grateful to the four reviewers for their helpful comments that lead to various improvements.

References

1. Ehrig, H., Kreowski, H.-J.: Parallelism of manipulations in multidimensional information structures. In: Mazurkiewicz, A. (ed.) MFCS 1976. LNCS, vol. 45, pp. 284–293. Springer, Heidelberg (1976). https://doi.org/10.1007/3-540-07854-1_188
2. 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 [34], pp. 163–245
3. Baldan, P., Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Rossi, F.: Concurrent semantics of algebraic graph transformations. In: Ehrig et al. [5], pp. 107–185
4. Kreowski, H.-J.: Manipulationen von Graphmanipulationen. Ph.D. thesis, Technische Universität Berlin (1978). Fachbereich Informatik
5. 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, Singapore (1999)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. monographs in theoretical computer science. An EATCS Series. 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. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. **109**, 181–224 (1993)
9. 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
10. Habel, A., Kreowski, H.-J.: Some structural aspects of hypergraph languages generated by hyperedge replacement. In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 1987. LNCS, vol. 247, pp. 207–219. Springer, Heidelberg (1987). https://doi.org/10.1007/BFb0039608
11. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Berlin (1992)
12. Drewes, F., Habel, A., Kreowski, H.-J.: Hyperedge replacement graph grammars. In: Rozenberg [34], pp. 95–162
13. Engelfriet, J.: Context-free graph grammars. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 125–213. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59126-6_3
14. Kreowski, H.-J., Klempien-Hinrichs, R., Kuske, S.: Some essentials of graph transformation. In: Esik, Z., Martin-Vide, C., Mitrana, V. (eds.) Recent Advances in Formal Languages and Applications. Studies in Computational Intelligence, vol. 25, pp. 229–254. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-33461-3_9
15. Rozenberg, G., Salomaa, A.: The Mathematical Theory of *L* Systems. Pure and Applied Mathematics: A Series of Monographs and Textbooks, vol. 90. Academic Press, Orlando (1980)

16. 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
17. Floyd, R.W.: Algorithm 97 (shortest path). Commun. ACM **5**(6), 345 (1962)
18. Warshall, S.: A theorem on Boolean matrices. J. ACM **9**(1), 11–12 (1962)
19. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
20. Mahr, B.: Algebraic complexity of path problems. RAIRO Theor. Inf. Appl. **16**(3), 263–292 (1982)
21. Litovski, I., Métivier, Y., Sopena, É.: Graph relabelling systems and distributed algorithms. In: Ehrig et al. [5], pp. 1–56
22. Kreowski, H.-J., Kuske, S.: Graph multiset transformation - a new framework for massively parallel computation inspired by DNA computing. Nat. Comput. **10**(2), 961–986 (2011). https://doi.org/10.1007/s11047-010-9245-6
23. Diestel, R. (ed.): Directions in Infinite Graph Theory and Combinatorics. Topics in Discrete Mathematics, vol. 3. Elsevier, North Holland (1992)
24. Codd, E.F.: Cellular Automata. Academic Press, New York (1968)
25. Kari, J.: Theory of cellular automata: a survey. Theoret. Comput. Sci. **334**, 3–33 (2005)
26. von Neumann, J.: The General and Logical Theory of Automata, pp. 1–41. Wiley, Pasadena (1951)
27. Wolfram, S.: A New Kind of Science. Wolfram Media Inc., Champaign (2002)
28. Peitgen, H.-O., Jürgens, H., Saupe, D.: Chaos and Fractals: New Frontiers of Science. Springer, New York (1992). https://doi.org/10.1007/978-1-4757-4740-9
29. Kreowski, H.-J.: A comparison between Petri nets and graph grammars. In: Noltemeier, H. (ed.) WG 1980. LNCS, vol. 100, pp. 306–317. Springer, Heidelberg (1981). https://doi.org/10.1007/3-540-10291-4_22
30. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundam. Inform. **26**(3/4), 241–265 (1996)
31. Dashkovskiy, S., Kreowski, H.-J., Kuske, S., Mironchenko, A., Naujuk, L., von Totth, C.: Production networks as communities of autonomous units and their stability. Int. Electron. J. Pure Appl. Math. **2**, 17–42 (2010)
32. Abdenebaoui, L., Kreowski, H.-J., Kuske, S.: Graph-transformational swarms. In: Bensch, S., Drewes, F., Freund, R., Otto, F., (eds.) Proceedings of the Fifth Workshop on Non-Classical Models for Automata and Applications (NCMA 2013), pp. 35–50. Österreichische Computer Gesellschaft (2013)
33. Hölscher, K., Kreowski, H.-J., Kuske, S.: Autonomous units to model interacting sequential and parallel processes. Fundam. Inform. **92**, 233–257 (2009)
34. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific, Singapore (1997)

A Tutorial on Graph Transformation

Barbara König[1], Dennis Nolte[1(✉)], Julia Padberg[2], and Arend Rensink[3]

[1] Universität Duisburg-Essen, Duisburg, Germany
dennis.nolte@uni-due.de
[2] Hochschule für Angewandte Wissenschaften Hamburg, Hamburg, Germany
[3] University of Twente, Enschede, Netherlands

Abstract. Graph transformation or graph rewriting has been developed for nearly 50 years and has become a mature and manifold formal technique. Basically, rewrite rules are used to manipulate graphs. These rules are given by a left-hand side and a right-hand side graph and the application comprises matching the left-hand side and replacing it with the right-hand side of the rule.

In this contribution we give a tutorial on graph transformation that explains the so-called double-pushout approach to graph transformation in a rigorous, but non-categorical way, using a gluing construction. We explicate the definitions with several small examples.

We also introduce attributes and attributed graph transformation in a lightweight form. The paper is concluded by a more extensive example on a leader election protocol, the description of tool support and pointers to related work.

1 Introduction

A substantial part of computer science is concerned with the transformation of structures, the most well-known example being the rewriting of words via Chomsky grammars, string rewriting systems [9] or transformations of the tape of a Turing machine. We focus on systems where transformations are rule-based and rules consist of a left-hand side (the structure to be deleted) and a right-hand side (the structure to be added).

If we increase the complexity of the structures being rewritten, wse next encounter trees or terms, leading to term rewriting systems [2]. The next level is concerned with graph rewriting [42], which – as we will see below – differs from string and term rewriting in the sense that we need a notion of interface between left-hand and right-hand side, detailing how the right-hand side is to be glued to the remaining graph.

Graph rewriting is a flexible and intuitive, yet formally rigorous, framework for modelling and reasoning about dynamical structures and networks. Such dynamical structures arise in many contexts, be it object graphs and heaps, UML diagrams (in the context of model transformations [13]), computer networks, the world wide web, distributed systems, etc. They also occur in other domains,

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 83–104, 2018.
https://doi.org/10.1007/978-3-319-75396-6_5

where computer science methods are employed: social networks, as well as chemical and biological structures. Specifically concurrent non-sequential systems are well-suited for modelling via graph transformation, since non-overlapping occurrences of left-hand sides can be replaced in parallel. For a more extensive list of applications see [12].

Graph rewriting has been introduced in the early 1970's, where one of the seminal initial contributions was the paper by Ehrig et al. [17]. Since then, there have been countless articles in the field: many of them are foundational, describing and comparing different graph transformation approaches and working out the (categorical) semantics. Others are more algorithmic in nature, describing for instance methods for analysing and verifying graph transformation. Furthermore, as mentioned above, there have been a large number of contributions on applications, many of them in software engineering [12], but in other areas as well, such as the recent growing interest from the area of biology in connection with the Kappa calculus (see for instance [7]).

Naturally, there are many other formalisms for describing concurrent systems, we just mention a few: Petri nets [38] can be viewed as a special case of graph transformation, missing the ability to model dynamic reconfigurations. There are however extensions such as reconfigurable Petri nets that extend nets with additional rules so that the net structure can be changed. An overview can be found in this collection [35]. Graph transformation is similar in expressiveness to process algebra [19,32,43], but is often more flexible, since a different choice of rules leads to different behaviours. Furthermore, behavioural equivalences, well-known from process algebra, can also be defined for graph transformation [14,24,29].

The aim of this paper is not to give a full overview over all possible approaches to graph transformation and all application scenarios. Instead, we plan to do quite the opposite: in the wealth of papers on graph transformation it is often difficult to discover the essence. Furthermore, readers who are not familiar with the categorical concepts (especially pushouts) used in the field can get easily intimidated. This is true for basic graph rewriting and is even more pronounced for enriched forms, such as attributed graph rewriting [11,18,34].

To solve this, in this paper we give a condensed version that can be easily and concisely defined and explained. For basic graph rewriting, we rely on the double-pushout (DPO) approach [4,17], which is one of the most well-known approaches to graph transformation, although clearly not the only one. In the definition we do not use the notion of pushouts, although we will afterwards explain their role.

For attributed graphs, we chose to give a lightweight, but still expressive version, which captures the spirit of related approaches.

Apart from spelling out the definitions, we will also motivate why they have a specific form. Afterwards, we will give an application example and introduce tool support.

Note that in the context of this paper we use the terms *graph rewriting* and *graph transformation* interchangeably. We will avoid the term *graph grammar*,

since that emphasizes the use of graph transformation to generate a graph language, here the focus is just on the rewriting aspect.

2 A Formal Introduction to Graph Transformation

We start by defining graphs, where we choose to consider directed, edge-labelled graphs where parallel edges are allowed.

Other choices would be to use hypergraphs (where an edge can be connected to any number of nodes) or to add node labels. Both versions can be easily treated by our rewriting approach.[1]

Throughout the paper, we assume the existence of a fixed set Λ from which we take our labels.

Definition 1 (Graph). *A* graph G *is a tuple* $G = (V, E, s, t, \ell)$*, where*

- V *is a set of nodes,*
- E *is a set of edges,*
- $s\colon E \to V$ *is the source function,*
- $t\colon E \to V$ *is the target function* and
- $\ell\colon E \to \Lambda$ *is the labelling function.*

Given a graph G, we denote its components by $V_G, E_G, s_G, t_G, \ell_G$. Given an edge $e \in E_G$, the nodes $s_G(e), t_G(e)$ are called *incident* to e.

A central notion in graph rewriting is a graph morphism. Just as a function is a mapping from a set to another set, a graph morphism is a mapping from a graph to a graph. It maps nodes to nodes and edges to edges, while preserving the structure of a graph. This means that if an edge is mapped to an edge, there must be a mapping between the source and target nodes of the two edges. Furthermore, labels must be preserved.

Graph morphisms are needed to identify the match of a left-hand side of a rule in a (potentially larger) host graph. As we will see below, they are also required for other purposes, such as graph gluing and graph transformation rules.

Definition 2 (Graph morphism). *Let* G*,* H *be two graphs. A* graph morphism $\varphi\colon G \to H$ *is a pair of mappings* $\varphi_V\colon V_G \to V_H$*,* $\varphi_E\colon E_G \to E_H$ *such that for all* $e \in E_G$ *it holds that*

- $s_H(\varphi_E(e)) = \varphi_V(s_G(e))$,
- $t_H(\varphi_E(e)) = \varphi_V(t_G(e))$ and
- $\ell_H(\varphi_E(e)) = \ell_G(e)$.

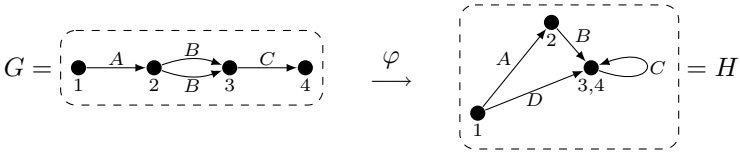
[1] Note that considerable part of graph transformation theory is concerned with making the results independent of the specific graph structure under consideration (see [28, 30]). This however depends on the use of category theory and we will not follow this path here.

A graph morphism φ is called injective (surjective) *if both mappings φ_V, φ_E are injective (surjective). Whenever φ_V and φ_E are bijective, φ is called an* isomorphism. *In this case we say that G_1, G_2 are isomorphic and write $G_1 \cong G_2$.*

Graph morphisms are composed by composing both component mappings. Composition of graph morphisms *is denoted by* $\circ$.

In the following we omit the subscripts in the functions φ_V, φ_E and simply write φ.

Example 1. Consider the following graphs G and H. Note that the numbers written at the nodes are not part of the graph: they are just there to indicate the morphism from G to H.



Here the edges of G are mapped with respect to their corresponding source and target node mappings. Note that the graph morphism φ is not surjective, since the D-labelled edge in H is not targeted. Furthermore, the morphism φ is not injective since the nodes 3 and 4 of the graph G are mapped to the same node in H and the two B-labelled edges in G are mapped to the same edge in H.

Now we come to another central concept that we here call *graph gluing*, but which is more conventionally called *pushout* in the literature. We will stick with the name graph gluing for now and will later explain the relation to categorical pushouts.

An intuitive explanation for the following construction is to think of two graphs G_1, G_2 with an overlap I. Now we glue G_1 and G_2 together over this common interface I, obtaining a new graph $G_1 +_I G_2$. This intuition is adequate in the case where the embeddings of I into the two graphs (called φ_1, φ_2 below) are injective, but not entirely when they are not. In this case one can observe some kind of merging effect that is illustrated in the examples below.

Graph gluing is described via factoring through an equivalence relation. We use the following notation: given a set X and an equivalence $\equiv$ on X, let $X/\equiv$ denote the set of all equivalence classes of $\equiv$. Furthermore $[x]_\equiv$ denotes the equivalence class of $x \in X$.

Definition 3 (Graph gluing). *Let I, G_1, G_2 be graphs with graph morphisms $\varphi_1 \colon I \to G_1$, $\varphi_2 \colon I \to G_2$, where I is called the* interface. *We assume that all node and edge sets are disjoint.*

Let $\equiv$ be the smallest equivalence relation on $V_{G_1} \cup E_{G_1} \cup V_{G_2} \cup E_{G_2}$ which satisfies $\varphi_1(x) \equiv \varphi_2(x)$ for all $x \in V_I \cup E_I$.

The gluing *of G_1, G_2 over I (written as $G = G_1 +_{\varphi_1,\varphi_2} G_2$, or $G = G_1 +_I G_2$ if the φ_i morphisms are clear from the context) is a graph G with:*

$$V_G = (V_{G_1} \cup V_{G_2})/\equiv \qquad E_G = (E_{G_1} \cup E_{G_2})/\equiv$$

$$s_G([e]_\equiv) = \begin{cases} [s_{G_1}(e)]_\equiv \text{ if } e \in E_{G_1} \\ [s_{G_2}(e)]_\equiv \text{ if } e \in E_{G_2} \end{cases} \qquad t_G([e]_\equiv) = \begin{cases} [t_{G_1}(e)]_\equiv \text{ if } e \in E_{G_1} \\ [t_{G_2}(e)]_\equiv \text{ if } e \in E_{G_2} \end{cases}$$

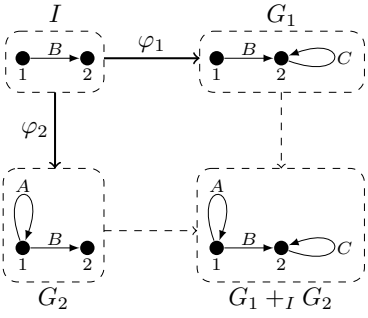
$$\ell_G([e]_\equiv) = \begin{cases} \ell_{G_1}(e) \text{ if } e \in E_{G_1} \\ \ell_{G_2}(e) \text{ if } e \in E_{G_2} \end{cases}$$

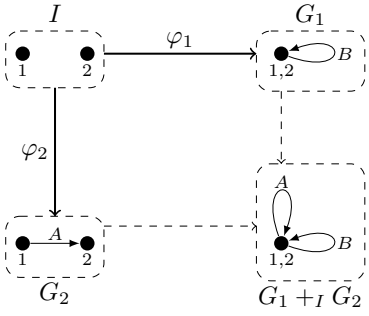
where $e \in E_{G_1} \cup E_{G_2}$.

Note that the gluing is well-defined, which is not immediately obvious since the mappings s_G, t_G, ℓ_G are defined on representatives of equivalence classes. The underlying reason for this is that φ_1, φ_2 are morphisms.

Example 2. We now explain this gluing construction via some examples.

– Let the two graph morphisms $\varphi_1 \colon I \to G_1$ and $\varphi_2 \colon I \to G_2$ to the right be given, where both φ_1 and φ_2 are injective. Since the interface I is present in both graphs G_1 and G_2, we can glue the two graphs together to construct a graph $G_1 +_I G_2$ depicted on the bottom right of the square.





– Now let the graph morphisms $\varphi_1 \colon I \to G_1$ and $\varphi_2 \colon I \to G_2$ to the left be given, where only φ_2 is injective. In the graph G_1, the interface nodes of I are merged via φ_1. The gluing graph $G_1 +_I G_2$ is constructed by merging all nodes in G_1, G_2, resulting in an A-labelled loop, together with the original B-labelled loop. This graph is depicted at the bottom right of the square.

We are now ready to define graph transformation rules, also called productions. Such a rule consists of a left-hand side graph L and a right-hand side graph R. However, as indicated in the introduction, this is not enough. The problem is that, if we simply removed (a match of) L from a host graph, we would typically have dangling edges, i.e., edges where either the source or the target node (or both) have been deleted. Furthermore, there would be no way to specify how the right-hand side R should be attached to the remaining graph.

Hence, there is also an interface graph I related to L and R via graph morphisms, which specify what is preserved by a rule.

Definition 4 (Graph transformation rule). *A* (graph transformation) rule r *consists of three graphs* L, I, R *and two graph morphisms* $L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R$.

Given a rule r, all nodes and edges in L that are not in the image of φ_L are called *obsolete*. Similarly, all nodes and edges in R that are not in the image of φ_R are called *fresh*.

After finding an occurrence of a left-hand side L in a host graph (a so-called match), the effect of applying a rule is to remove all obsolete elements and add all fresh elements. As indicated above, the elements of I are preserved, providing us with well-defined attachment points for R.

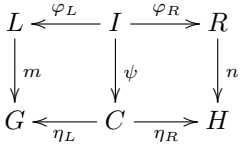
While this explanation is valid for injective matches and rule morphisms, it does not tell the full story in case of non-injective morphisms. Here rules might split or merge graph elements. Using the graph gluing defined earlier, it is easy to give a formal semantics of rewriting.

The intuition is as follows: given a rule as in Definition 4 and a graph G, we ask whether G can be seen as a gluing of L and an (unknown) context C over interface I, i.e., whether there exists C such that $G \cong L +_I C$. If this is the case, G can be transformed into $H \cong R +_I C$.

Definition 5 (Graph transformation). *Let* $r = (L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R)$ *be a rule. We say that a graph* G *is transformed via* r *into a graph* H *(symbolically:* $G \overset{r}{\Rightarrow} H$*) if there is a graph* C *(the so-called* context*) and a graph morphism* $\psi\colon I \to C$ *such that:*

$$G \cong L +_{\varphi_L,\psi} C \qquad H \cong R +_{\varphi_R,\psi} C$$

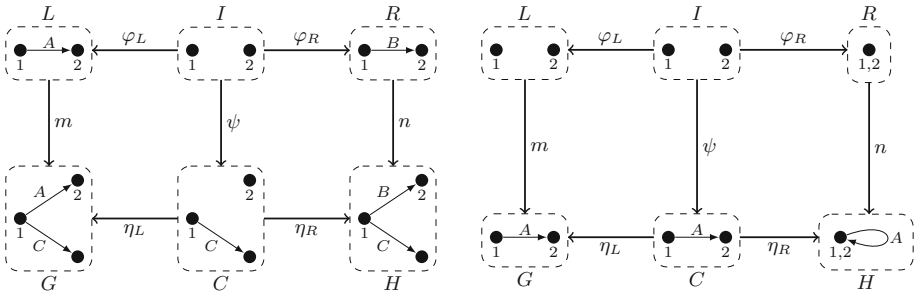
This situation can be depicted by the diagram to the right (also called double-pushout diagram).
The morphism m *is called the* match*,* n *the* co-match.



Depending on the morphisms φ_L and φ_R one can obtain different effects: whenever both φ_L and φ_R are injective, we obtain standard replacement. Whenever φ_L is non-injective we specify splitting, whereas a non-injective φ_R results in merging.

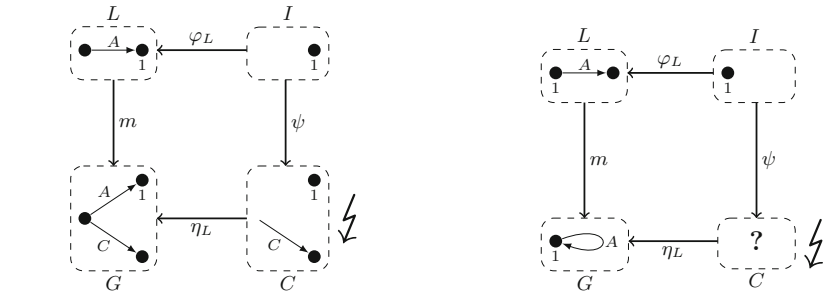
We now consider some examples. First, we illustrate the straightforward case where indeed the obsolete items are removed and the fresh ones are added, see Fig. 1a. Somewhat more elaborate is the case when the right leg φ_R of a rule is non-injective, which causes the merging of nodes, see Fig. 1b.

Different from string or term rewriting, in graph rewriting it may happen that we find a match of the left-hand side, but the rule is not applicable, because no context as required by Definition 5 exists. There are basically two reasons for



(a) Application of a GTS rule (b) Non-injective right leg φ_R

Fig. 1. GTS rule examples



(a) Dangling edge condition example (b) Identification condition example

Fig. 2. Gluing condition examples

this: either the rule removes a node, without removing all edges connected to that node (dangling edge condition, see Fig. 2a), or the match identifies two graph elements which are not preserved (identification condition) (see Fig. 2b).

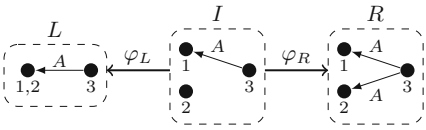
The following proposition [10] states under which circumstances the context exists.

Proposition 1 (Existence of a context, gluing condition). *Let $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ be a graph transformation rule and let $m\colon L \to G$ be a match. Then a context C and a morphism $\psi\colon I \to C$ such that $G \cong L +_{\varphi_L,\psi} C$ exist if and only if the following holds:*

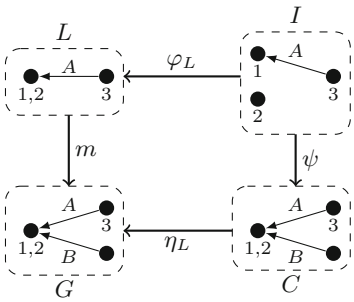
- *Every node $v \in V_L$, whose image $m(v)$ is incident to an edge $e \in E_G$ which is not in the image of m, is not obsolete (i.e. in the image of φ_L).*
- *Whenever two elements $x, y \in V_L \cup E_L$ with $x \neq y$ satisfy $m(x) = m(y)$, then neither of them is obsolete.*

However, even if the context exists, there might be cases where it is non-unique. This happens in cases where φ_L, the left leg of a rule, is non-injective. In this case one can for instance split nodes (see the rule in Fig. 3a) and the

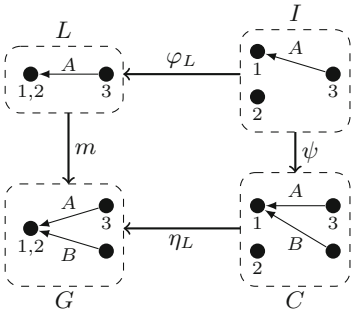
question is what happens to the incident edges. By spelling out the definition above, one determines that this must result in non-determinism. Either, we do not split (Fig. 3b) or we split and each edge can non-deterministically "choose" to stay either with the first or the second node (Fig. 3c and d). Each resulting combination is a valid context and this means that a rule application may be non-deterministic and generate several (non-isomorphic) graphs. In many papers such complications are avoided by requiring the injectivity of φ_L.



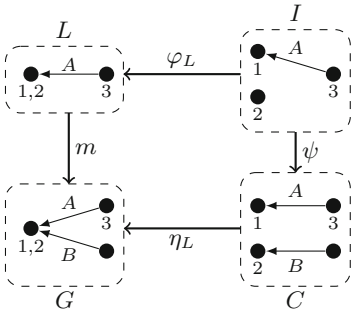
(a) Rule with non-injective left leg φ_L



(b) Valid context (i)



(c) Valid context (ii)



(d) Valid context (iii)

Fig. 3. Non-injective left leg rule with three valid contexts

Note that the specifics such as the dangling edge condition are typical to the double-pushout (or DPO) approach that we are following here. In other approaches (such as SPO, which we only discuss briefly in Sect. 7.2), whenever a node is deleted, all its incident edges are deleted as well (deletion in unknown contexts).

Finally, we can introduce the notion of a graph transformation system that is frequently used. Here we fix a start graph and a set of rules. This enables us to ask questions such as: Which graphs are reachable from the start graph via the given rules?

Definition 6 (Graph transformation system). *A* graph transformation system *is a tuple* $\mathcal{G} = (G_0, \mathcal{R})$ *where*

- G_0 *is an arbitrary graph, the so-called* initial graph *or* start graph, and
- $\mathcal{R}$ *is a set of* graph transformation rules.

3 Attributed Graph Transformation

For many applications one requires more than graphs labelled over a finite alphabet that we considered up to now. For instance, in Sect. 4 we will consider leader election on a ring where edges, representing processes, are labelled with natural numbers as Ids. Hence graphs should be attributed with elements of given data types (e.g. integer, string, boolean) and it should be possible to perform computations (e.g. add two integers) and define guards that restrict the applicability of rules (e.g. apply the rule only if a certain attribute is above some threshold).

In order to achieve this aim we now introduce attributed graph transformation. Choosing data types (also called sorts), carrier sets and operations amounts to defining a signature and a corresponding algebra [16,46] and we will start by introducing these concepts.

Definition 7 (Signature, Algebra). *A* signature Σ *is a pair* $(\mathcal{S}, \mathcal{F})$ *where* $\mathcal{S}$ *is a set of* sorts *and* $\mathcal{F}$ *is a set of function symbols equipped with a mapping* $\sigma\colon \mathcal{F} \to \mathcal{S}^* \times \mathcal{S}$. *Sorts are also called* types. *We require that* $\mathcal{S}$ *contains the sort bool.*

A Σ-algebra $\mathcal{A}$ *consists of carrier sets* $(\mathcal{A}_s)_{s\in\mathcal{S}}$ *for each sort and a function* $f^{\mathcal{A}} : \mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n} \to \mathcal{A}_s$ *for every function symbol* f *with* $\sigma(f) = (s_1 \dots s_n, s)$.

By $T(\Sigma, X)$ *we denote the* Σ-term algebra, *where* X *is a set of variables, each equipped with a fixed sort. That is, the carrier sets of the term algebra consist of all terms of the corresponding sort.*

For an algebra $\mathcal{A}$ *we denote by* $\mathcal{A}_{\mathcal{S}}$ *the set* $\mathcal{A}_{\mathcal{S}} = \biguplus_{s\in\mathcal{S}} \mathcal{A}_s$, *i.e., the union of all carrier sets (under the implicit assumption that they are all disjoint).*

Example 3. As a typical example for an algebra assume that we have two sorts $\mathcal{S} = \{int, bool\}$ and function symbols *add*, *mult*, *eq* with $\sigma(add) = \sigma(mult) = (int\, int, int)$ and $\sigma(eq) = (int\, int, bool)$ (representing addition, multiplication and the equality predicate).

The carrier sets in an algebra $\mathcal{A}$ could be $\mathcal{A}_{int} = \mathbb{Z}$, $\mathcal{A}_{bool} = \{true, false\}$ and functions would be interpreted in the usual way, e.g. $add^{\mathcal{A}}(z_1, z_2) = z_1 + z_2$ and $eq^{\mathcal{A}}(z_1, z_2) = true$ whenever $z_1 = z_2$ and *false* otherwise.

On the other hand, in the term algebra $T(\Sigma, X)$ the carrier sets consist of terms, for instance $T(\Sigma, X)_{int}$ contains $add(mult(x, y), y)$ and $T(\Sigma, X)_{bool}$ contains $eq(add(x, x), y)$, where $x, y \in X$ are variables of sort *int*.

Algebras come equipped with their notion of morphism, so-called algebra homomorphisms. These are mappings between the carrier sets that are compatible with the operations.

Definition 8 (Algebra homomorphism). *Let $\mathcal{A}, \mathcal{B}$ be two Σ-algebras. An* algebra homomorphism *$h : \mathcal{A} \to \mathcal{B}$ is a family of maps $(h_s : \mathcal{A}_s \to \mathcal{B}_s)_{s \in \mathcal{S}}$ such that for each $f \in \mathcal{F}$ with $\sigma(f) = (s_1 \dots s_n, s)$ we have*

$$h_s(f^{\mathcal{A}}(a_1, \dots, a_n)) = f^{\mathcal{B}}(h_{s_1}(a_1), \dots, h_{s_n}(a_n)).$$

The next step is straightforward: add attributes, i.e., elements of a carrier set, to (the nodes or edges of) a graph. In the following we add attributes only to edges, however nothing prevents us from adding attributes also to nodes. In order to have a clean separation, we require that the edge label determines the sort of the corresponding attribute.

Definition 9 (Attributed graph). *Let $\mathcal{A}$ be a Σ-algebra with $\mathcal{A}_{bool} = \{true, false\}$. Let type$\colon \Lambda \to \mathcal{S}$ be a function that assigns a sort to every edge label. An* attributed graph over $\mathcal{A}$ *(G, att) consists of a graph $G = (V, E, s, t, \ell)$, together with a function att$\colon E_G \to \mathcal{A}_{\mathcal{S}}$ such that $att(e) \in \mathcal{A}_{type(\ell(e))}$.*

We first define the notion of attributed graph transformation rule, where left-hand side and right-hand side are attributed over the term algebra. Furthermore there is a guard condition of sort *bool*. We require that in the left-hand side terms are always single unique variables, which can then be used in terms in the right-hand side and in the guard.

Definition 10 (Attributed graph transformation rule). *Let Σ be a signature and let X be a set of variables. An* attributed rule *is a graph transformation rule $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ with two functions $att_L\colon E_L \to T(\Sigma, X)_{\mathcal{S}}$, $att_R\colon E_R \to T(\Sigma, X)_{\mathcal{S}}$ and a guard $g \in T(\Sigma, X)_{bool}$. These attribution functions must respect sorts, i.e., for every $e \in E_L$ it holds that $att_L(e) \in T(\Sigma, X)_{type(\ell_L(e))}$ and analogusly for $e \in E_R$.*

We require that each term $att_L(e)$ for $e \in E_L$ is a single variable and all these variables occurring in the left-hand side are pairwise different. Furthermore, each variable in $att_R(e)$ for $e \in E_R$ and each variable in g occurs in the left-hand side.

Now we are ready to define attributed graph transformation: while the rule graphs L and R are attributed with elements from the term algebra, the graphs to be rewritten are attributed with elements from a carrier set that represents a primitive data type (such as integers or booleans).

Then a match determines the evaluation of the variables in the left-hand side, giving us a corresponding algebra homomorphism. This homomorphism is then used to evaluate the terms in the right hand sides and to generate the corresponding values. All other edges keep their attribute values.

Definition 11 (Graph transformation with attributed rules). *Given an attributed rule $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ with functions att_L, att_R and guard g, it can be applied to a graph (G, att_G) attributed over $\mathcal{A}$ as follows: G is transformed to H as described in Definition 5. The match $m\colon L \to G$ induces an algebra homomorphism $h_m\colon T(\Sigma, X) \to \mathcal{A}$ by defining $h_m(x) = att_G(m(e))$ if $e \in E_L$ and $att_L(e) = x$. For each variable y not occurring in L the value $h_m(y)$ is arbitrary.*

The rule can be applied whenever $h_m(g) = true$*. In this case we define*

$$att_H(e') = \begin{cases} h_m(att_R(e)) \text{ if } e' = n(e), e \in E_R \\ att_G(\eta_L(e)) \text{ otherwise, if } e' = \eta_R(e), e \in E_C \end{cases}$$

where $e' \in E_H$*. Whenever* att_H *is not well-defined, the rule can not be applied.*[2]

Note that the algebra homomorphism h_m above is well-defined due to the requirement that each occurrence of a variable in the left-hand side is unique.

We start with a straightforward case where we apply an attributed graph transformation rule, see Fig. 4. The given rule shifts a B-labelled loop (which has an attribute y) over an A-labelled edge with corresponding attribute x. After the rule application the edge is attributed with the sum $add(x, y)$ and the loop inherits the former attribute x. (Note that in order to have a more compact notation we slightly abuse notation and write $x + y$ instead of $add(x, y)$.)

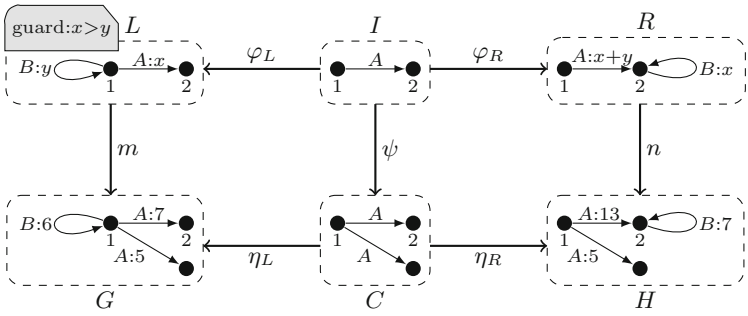


Fig. 4. Attributed graph transformation rule application example

Note that in the definition above, we have to require that the attribution function att_H of H is well-defined, here is an example that illustrates why: imagine a rule with two (equally labelled) edges in the left-hand side, which have attributes x, y. The edges are preserved and in the right-hand side they are attributed with the sum $add(x, y)$ and the product $mult(x, y)$ (see Fig. 5). Now, since we allow non-injective matches, such a rule can be applied to a single edge with attribute value 1 in the host graph.

The (preserved) edge has two different preimages under the co-match n. The first preimage would require to set the value $att_H(e)$ to $add^{\mathcal{A}}(1, 1) = 2$, the second to $mult^{\mathcal{A}}(1, 1) = 1$.

Here, the straightforward solution is to say that the rule is not applicable, since it would create an inconsistent situation. Such issues can be avoided by requiring that all morphisms (rule morphisms, match, etc.) are injective.

[2] The morphism n need not be injective, hence an edge e might have several preimages under n. In this case, it is possible that the new attribute of an edge cannot be uniquely determined.

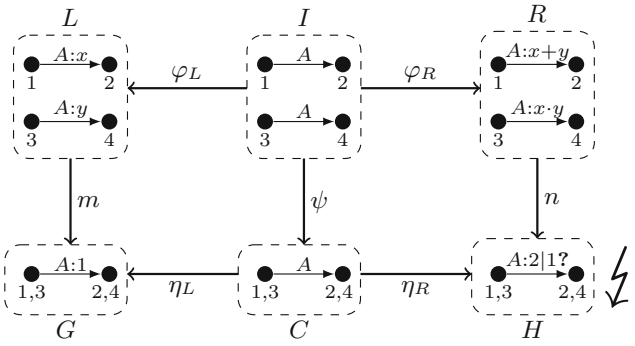


Fig. 5. Example for a non-applicable rule

4 Example: Leader Election

We now demonstrate the modelling power of attributed graph transformation systems, by modelling a variation of the leader election protocol. The protocol (according to Chang and Roberts [3]) works as follows: there is a set of processes arranged in a ring, i.e., every process has a unique predecessor and a unique successor. Furthermore, each process has a unique Id and there exists a total order on the Ids (which is easily achieved by assuming that Ids are natural numbers).

The leader will be the process with the smallest Id, however no process knows what is the smallest Id at the start of the protocol. Hence every process generates a message with its own Id and sends it to its successor. A received message with content *MId* is treated as follows by a process with Id *PId*:

- if $MId < PId$, then the message is forwarded to the successor
- if $MId = PId$, then the process declares itself the leader
- If $MId > PId$, then the message is not passed on (or alternatively discarded).

Whenever the Ids are unique, it can be shown that the protocol terminates with the election of a unique leader.

In this example we additionally assume that the topology of the ring changes. We extend the protocol allowing processes to enter and to leave the ring. Processes entering the ring obtain a unique Id via a central counter, larger than all other Ids existing so far in the ring. This additional feature does not interfere with the election of the leader, since we elect the process with the minimal Id. As any process might be deleted, we can not resort to the simplistic (and non-distributed) solution of choosing the first process that is created and has thus the lowest Id.

We now model this protocol as a graph transformation system (see Fig. 6). The start graph S consists of a single node and a loop labelled `count` which represents the counter, the current counter value `0` is stored in the attribute (where all attributes are natural numbers). In a first step, modelled by rule

first proc, if the counter attribute `count` satisfies the rule's guard `i=0`, then the first process labelled `proc:i+1` is created and the counter is set to `count:i+1`. Remember, only the edges are equipped with labels and corresponding terms. As required in Definition 10 the left-hand side is attributed only by variables. Again, the numbers in the nodes denote the morphisms from the interface to the left-hand and right-hand side. Then, in subsequent steps (rule *add proc*) other processes are created (incrementing the current counter and using it as the Id of the process) and are inserted after an arbitrary existing process. Processes may leave the ring, provided at least one other process is present, see rule *del proc*.

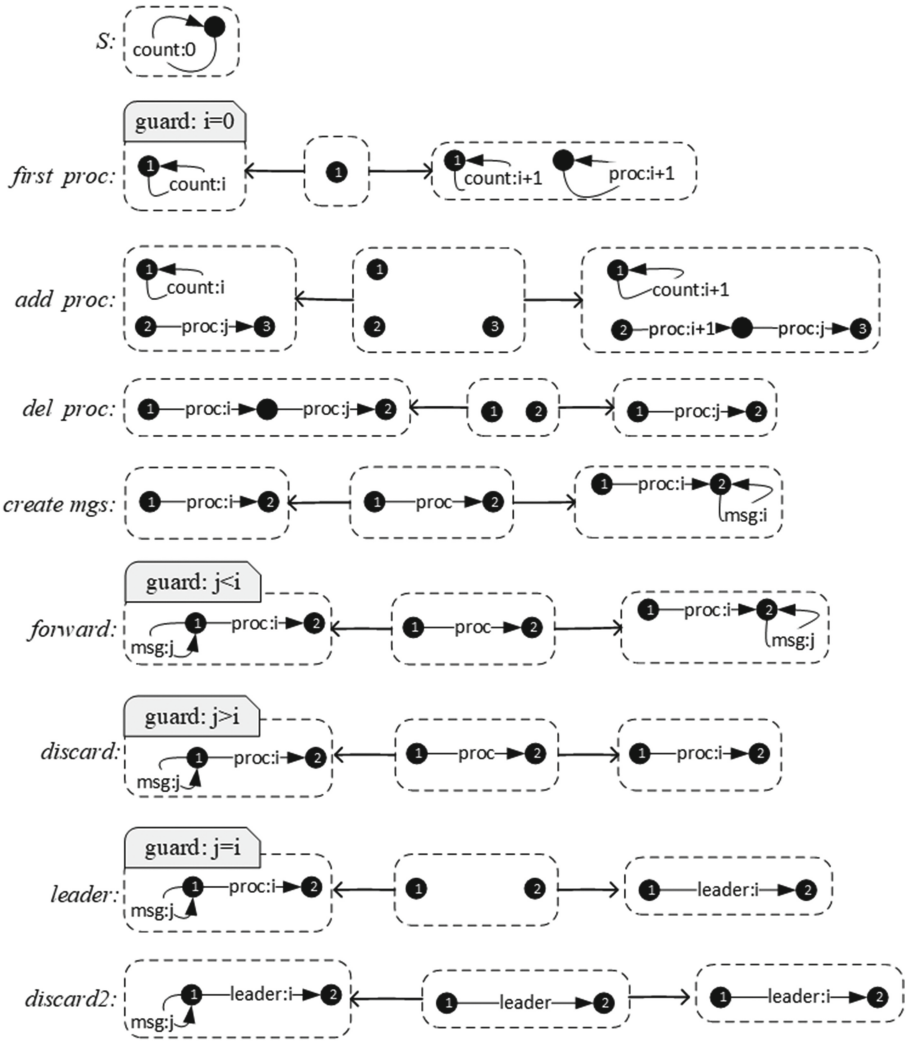


Fig. 6. Leader election in dynamic ring

Processes can create messages as described above, represented by `msg`-labelled loops (rule *create msg*). The attribute of the `msg`-loop is the Id of the sending process. The message is forwarded if its Id is less than the Id of the receiving process (rule *forward*) and if it is greater the message is discarded (rule *discard*). If a process receives a message with its own Id, it declares itself the leader (rule *leader*). Once the leader has been chosen, it cannot be deleted any longer since the rule *del proc* requires the label `proc`. Moreover, all subsequent messages arriving at the leader are discarded as well (rule *discard2*).

The application of rules may yield a graph G as given in Fig. 7. The match morphism m induces the algebra homomorphism $h_m \colon T(\Sigma, X) \to \mathcal{A}$ with $h_m(i) = 2$ and $h_m(j) = 2$. Obviously the guard is satisfied, so the edges with label `msg:2` and `prc:2` are deleted, yielding the graph C. The graph H is obtained by gluing an edge with label `leader:2` between the two nodes.

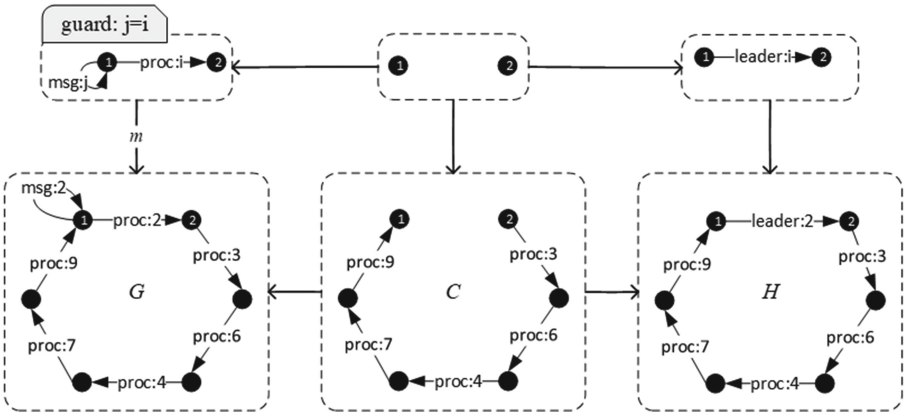


Fig. 7. Transformation step: declaring the leader process

The state space of this system is infinite, due to the capability of creating new processes with ever higher numbers, as well as unbounded numbers of messages. There are techniques for verifying infinite-state graph transformation systems (see for instance [26] where a similar system is analysed), but those are out of scope for this tutorial. Alternatively, if we restrict the number of processes to a fixed bound, then model checking becomes available as a technique, for instance as implemented in the tool GROOVE (see Sect. 5.1). Here are some example desirable properties, expressed in Computation Tree Logic (CTL) respectively Linear Temporal Logic (LTL):

1. Safety (CTL): AG ! twoLeaders
2. Reachability (CTL): AG EF hasLeader
3. Termination (LTL): G F hasLeader

Here, twoLeaders and hasLeader are propositions that are fulfilled by a given state graph if it contains, respectively, at least two leader-edges or at least one leader-edge. Those propositions can themselves be formulated as transformation rules that do not actually change the graph, but only check for applicability. Property 1 expresses the crucial safety property that there can never be two or more leaders elected; property 2 expresses that from every state, a future state is reachable in which a leader has been elected. Finally, property 3 expresses that all paths will eventually actually reach a point where a leader has been elected.

The protocol encoded in the rule system at hand satisfies properties 1 and 2, but not 3. Two reasons why termination fails to hold are that the process with the lowest number may forever be deleted *before* it receives back its own message and so is elected leader, or one process creates an infinite number of messages which are never forwarded.

The protocol satisfies the reachability property 2 even in the case of changes in topology. This can be reasoned out as follows: Since the added processes have higher Ids than already inserted processes, the message which the new process sends, will be discarded by the subsequent process. Another consequence of the changing topology could be that the process `proc:min1` with the minimal Id `min1` sends its message and is deleted before becoming leader. Then its message is forwarded along the ring as it always satisfies the condition of rule *forward*. But at some point the next minimal Id `min2` is sent by some other process and leads to the leader with Id `min2`. Then the message with `min1` will be eventually discarded with *discard2*.

5 Tools

Here, we merely hint at some of graph transformation tools that are available for many different purposes and describe two of them in more detail. We introduce AGG and GROOVE, since both of them can be considered to be general purpose graph transformation tools. Other graph transformation tools that are actively maintained are, ATOM[3] [8], VIATRA [6], FUJABA [33] and AUGUR [25].

5.1 GROOVE: Graphs for Object-Oriented Verification

The tool GROOVE[3] was originally created to support the use of graphs for modelling the design-time, compile-time, and run-time structure of object-oriented systems [39], but since then has grown to be a full-fledged general-purpose graph transformation tool. The emphasis lies on the efficient exploration of the state space, given a particular graph transformation system; see, for instance, [20]. While doing so, GROOVE recognises previously visited graphs modulo (graph) isomorphism, avoiding duplication during their exploration. GROOVE has a built-in model checker that can run temporal logic queries (LTL or CTL) over the resulting state space.

[3] http://groove.cs.utwente.nl/.

GROOVE has a very rich set of features to enable the powerful and flexible specification of transformation rules, including quantified rules [41] (so that a single rule may be applied in one shot to all subgraphs that satisfy a given property) and an extensive language to schedule rule applications (so that the default strategy of applying every rule to all reachable graphs can be modified).

GROOVE graphs do not conform precisely to the definition in this paper. Some important differences are:

- GROOVE graphs are typed; that is, all nodes have one of a set of types from a user-defined type graph (and so do all edges, but an edge type essentially corresponds to its label).
- More importantly, in GROOVE attributes are associated with nodes rather than edges; moreover, they are always named. Thus, rather than an edge `proc:1` between two untyped process nodes, one would have an unnumbered edge `next` (say) between two `Proc`-type nodes, in combination with a named attribute `nr = 1` on its target node.

However, the graphs of this paper can be easily mimicked. A rule system for leader election that corresponds to Sect. 4 is provided together with this paper.[4]

The results reported in the previous section on the safety, reachability and termination properties 1–3 can easily be checked on this rule system by disabling the rule *add proc* and starting with a graph that already has a given number of processes (so that the state space is finite), and then invoking the LTL or CTL model checker with the formulas given above. As stated before, the outcome is that properties 1 and 2 are satisfied, whereas 3 is violated.

5.2 AGG: The Attributed Graph Grammar System

The Attributed Graph Grammar System (AGG) [45] is a development environment for attributed graph transformation systems and aims at specifying and rapidly prototyping applications with complex, graph structured data. AGG[5] supports the editing of graphs and rules that can be attributed by Java objects and types. Basic data types as well as object classes already available in Java class libraries may be used. The graph rules may be attributed by Java expressions which are evaluated during rule applications. Additionally, rules may have attribute conditions that are boolean Java expressions. AGG provides simulation and analysis techniques, namely critical pair analysis and consistency checking. The application of rules can be manipulated using control structures such as negative application conditions to express requirements for non-existence of substructures. Further control over the rules is given by rule layers that fix the order in which rules are applied. The interpretation process applies rules of lower layers first, which means applying the rules of a layer as long as possible before applying those of the next layer. These rule layers allow the specification of a simple control flow.

[4] http://groove.cs.utwente.nl/wp-content/uploads/leader-electiongps.zip.
[5] http://www.user.tu-berlin.de/o.runge/agg/.

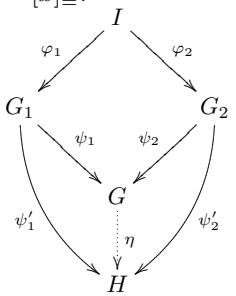
6 Some Remarks on the Categorical Background

In this section, we explain the name *double-pushout approach*. It gives some background information that is useful for understanding papers on the topic, but is not required for the formal definition of graph transformation given earlier.

Graph gluing as in Definition 3 can alternatively be characterized via the categorical notion of pushout. Category theory relies on so-called universal properties where, given some objects, one defines another object that is in some relation to the given object and is – in some sense – the most general object which is in this relation. The prototypical example is the supremum or join, where – given two elements x, y of a partially ordered set $(X, \leq)$ – we ask for a third element z with $x \leq z$, $y \leq z$ and such that z is the *smallest* element which satisfies both inequalities. There is at most one such z, namely $z = x \vee y$, the join of x, y.

In the case of graphs, the order relation $\leq$ is replaced by graph morphisms.

Proposition 2. *Let I, G_1, G_2 be graphs with graph morphisms $\varphi_1 \colon I \to G_1$, $\varphi_2 \colon I \to G_2$ as in Definition 3. Assume further that $G = G_1 +_{\varphi_1,\varphi_2} G_2$ is the gluing and that $\psi_i \colon G_1 \to G$, $i = 1, 2$, are graph morphisms that map each element $x \in V_{G_i} \cup E_{G_i}$ to its equivalence class $\psi_i(x) = [x]_{\equiv}$.*
The gluing diagram consisting of the morphisms $\varphi_1, \varphi_2, \psi_1, \psi_2$ commutes, i.e., $\psi_1 \circ \varphi_1 = \psi_2 \circ \varphi_2$ and it has the following universal property*: for any two morphisms $\psi_1' \colon G_1 \to H$, $\psi_2' \colon G_2 \to H$ satisfying $\psi_1' \circ \varphi_1 = \psi_2' \circ \varphi_2$, there exists a unique morphism $\eta \colon G_1 +_I G_2 \to H$ such that $\eta \circ \psi_1 = \psi_1'$ and $\eta \circ \psi_2 = \psi_2'$.*
Squares which commute and satisfy the universal property spelled out above are called pushouts*. The graph G is unique up to isomorphism.*



Intuitively, the pushout characterization says that G should be a graph where the "common" parts of G_1, G_2 must be merged (since the square commutes), but it should be obtained in the most general way by merging only what is absolutely necessary and adding nothing superfluous. This corresponds to saying that for any other merge H, G is more general and H can be obtained from G by further merging or addition of graph elements (expressed by a morphism from G to H).

7 Literature Overview

7.1 Introductory Papers

This paper is by no means the first introductory paper to graph transformation. It was our aim to write a paper that fills a niche and gives precise formal definitions, but does not rely on category theory. At the same time, we wanted to treat general rules and not restrict to injective matches or rule spans, which is often done in tutorial papers.

Since not all introductory papers are well-known, it is worth to make a meta-survey and to provide a list.

Of the original papers, the survey paper by Ehrig [10] is in any case worth a read, however the notation has evolved since the seventies.

A standard reference is of course the "Handbook of Graph Grammars and Computing by Graph Transformation", which appeared in three volumes (foundations [42] – applications, languages and tools [12] – concurrency, parallelism and distribution [15]). Strongly related to our exposition is the chapter on DPO rewriting by Corradini et al. [4], which is based on categorical definitions.

The well-known book by Ehrig et al. [11] revisits the theory of graph rewriting from the point of view of adhesive categories, a general categorical framework for abstract transformations. In an introductory section it defines the construction of pushouts via factorization, equivalent to our notion of graph gluing.

Nicely written overview papers that however do not give formal definitions are by Heckel [23] and by Andries et al. [1]. The paper by Giese et al. [21] is aimed towards software engineers and gives a detailed railcab example.

The habilitation thesis by Plump [36] and the introductory paper by Kreowski et al. [27] give very clear formal, but non-categorical, introductions. Both make injectivity requirements, either on the rule spans or on the match.

The paper by Löwe and Müller [31] makes a non-categorical, but slightly non-standard, introduction to graph transformation, while the introduction by Schürr and Westfechtel [44] is very detailed and treats set-theoretical, categorical and logical approaches. Both articles are however written in German.

7.2 Further Issues

Deletion in unknown contexts: In the DPO approach that was treated in this note, it is forbidden to remove a node that is still attached to an edge, which is not deleted. In this case, the rule is not applicable. In the single-pushout (or SPO) approach [30] however, the deletion of a node to which an undeleted edge is attached, is possible. In this case all incident edges are deleted as well. In contrast to DPO, SPO is based on partial graph morphisms.

Attributed graph rewriting: Our way of defining attributed graph rewriting was inspired by [26,37]. We provide some remarks on alternative approaches to attributed graph transformation: in an ideal world one would like to extend all the notions that we introduced previously (graph morphisms, gluing, rules, etc.) to this new setting. This would mean to extend graph morphisms to attributed graph morphisms by requiring algebra homomorphisms on the attributes.

This has been done [11,18,34], but unfortunately there are some complications. The first problem is that, as explained above, we want to work with two different algebras: the term algebra and an algebra representing primitive data types. This means that in the double-pushout diagrams, we would need algebra homomorphisms between different algebras on the vertical axis and identity algebra homomorphisms on the horizontal axis.

But this causes another problem: nodes or edges that are preserved by a rule, i.e., items that are in the interface usually should *not* keep their attribute value. Otherwise it would be impossible to specify attribute changes. (Note that it is possible to delete and recreate an edge, but not a node, since it is usually connected to unknown edges and the dangling condition would forbid this.) But this is contrary to the idea of having identity homomorphisms horizontally.

Hence, as announced above, we here opted for a lightweight approach where we do not define a new notion of attributed graph morphism, but only add algebra homomorphisms as required (for the match and co-match). Other options, which we do not pursue here, is to add the carrier sets to the graphs and add pointers from edges to the attribute values [18]. However, this formally turns graphs into infinite objects.

As a side remark, we would also like to mention that graphs themselves are two-sorted algebras with sorts node and edge and two function symbols (source and target). This view has been exploited in order to generalize the structures that can be transformed [30].

Application conditions: As we saw in the section on attributed graph transformation (Sect. 3) and in the example (Sect. 4), it is often useful to specify guards that restrict the applicability of rules. So far our guards talked about attributes, but it is also very useful to consider guards that refer to the structural properties of a graph, so-called application conditions [11,22].

A special case are negative applications conditions that inhibit the application of a rule whenever a certain structure is present in the vicinity of the left-hand side. This can for instance be used to specify rules for computing the transitive closure of a graph: whenever there exists an edge from node s to v and from v to t, add a direct edge from s to t, but only if such an edge is not already present. In order to gain expressiveness, application conditions can be nested and it has been shown that such conditions are equal in expressiveness to first-order logic [40].

8 Conclusion

Naturally, there are many topics related to graph transformation that we did not treat in this short tutorial. For instance, there exists a substantial amount of work on theory, generalizing graph transformation by means of category theory [11,28]. Furthermore, confluence or Church-Rosser theorems, in connection with critical pair analysis have been extensively studied. Other verification and analysis techniques have been studied as well (termination analysis, reachability analysis, model checking, etc.). Work on graph transformation is also closely connected to work on specification languages on graphs, such as nested application conditions [22] and monadic second-order logic [5].

Acknowledgements. We would like to thank all the participants of the North German GraTra Day in February 2017 in Hamburg for the discussion about this paper. Especially, we would like to acknowledge Berthold Hoffmann, Leen Lambers and Hans-Jörg Kreowski who contributed by commenting on our paper and giving valuable suggestions and hints.

References

1. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A., Taentzer, G.: Graph transformation for specification and programming. Sci. Comput. Program. **34**(1), 1–54 (1999)
2. Bezem, M., Klop, J.W., de Vrijer, R. (eds.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
3. Chang, E.J.H., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (1979)
4. 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: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific (1997). Chapter 3
5. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic, A Language-Theoretic Approach. Cambridge University Press, New York (2012)
6. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - visual automated transformations for formal verification and validation of UML models. In: 17th IEEE International Conference on Automated Software Engineering, pp. 267–270. IEEE Computer Society (2002)
7. Danos, V., Feret, J., Fontana, W., Harmer, R., Hayman, J., Krivine, J., Thompson-Walsh, C.D., Winskel, G.: Graphs, rewriting and pathway reconstruction for rule-based models. In: Proceedings of the FSTTCS 2012. LIPIcs, vol. 18. Schloss Dagstuhl - Leibniz Center for Informatics (2012)
8. Lara, J., Vangheluwe, H.: AToM^3: a tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45923-5_12
9. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (eds.) Formal Models and Semantics. Handbook of Theoretical Computer Science, vol. B, pp. 243–320. Elsevier (1990). Chapter 6
10. 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
11. 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
12. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications Languages and Tools. World Scientific, Singapore (1999)
13. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation - General Framework and Applications. Monographs in Theoretical Computer Science. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47980-3
14. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Math. Struct. Comput. Sci. **16**(6), 1133–1163 (2006)
15. Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallellism, and Distribution. World Scientific, Singapore (1999)

16. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1, Equations and Initial Semantics. Monographs in Theoretical Computer Science. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69962-7
17. Ehrig, H., Pfender, M., Schneider, H.: Graph grammars: an algebraic approach. In: Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, pp. 167–180 (1973)
18. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_13
19. Fokkink, W.: Introduction to Process Algebra. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04293-9
20. Ghamarian, A.H., de Mol, M.J., Rensink, A., Zambon, E., Zimakova, M.V.: Modelling and analysis using groove. Int. J. Soft. Tools Technol. Transf. **14**(1), 15–40 (2012)
21. Giese, H., Lambers, L., Becker, B., Hildebrandt, S., Neumann, S., Vogel, T., Wätzoldt, S.: Graph transformations for MDE, adaptation, and models at runtime. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 137–191. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30982-3_5
22. Habel, A., Pennemann, K.-H.: Nested constraints and application conditions for high-level structures. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 293–308. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31847-7_17
23. Heckel, R.: Graph transformation in a nutshell. In: Bezivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings (2005)
24. Høgh Jensen, O., Milner, R.: Bigraphs and mobile processes (revised). Technical report UCAM-CL-TR-580, University of Cambridge (2004)
25. König, B., Kozioura, V.: Augur - a tool for the analysis of graph transformation systems. Bull. EATCS **87**, 126–137 (2005)
26. König, B., Kozioura, V.: Towards the verification of attributed graph transformation systems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 305–320. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_21
27. Kreowski, H.-J., Klempien-Hinrichs, R., Kuske, S.: Some essentials of graph transformation. In: Ésik, Z., Martin-Vide, C., Mitrana, V. (eds.) Recent Advances in Formal Languages and Applications, pp. 229–254. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-33461-3_9
28. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO - Theor. Inf. Appl. **39**(3), 511–545 (2005)
29. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_19
30. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. **109**, 181–224 (1993)
31. Löwe, M., Müller, J.: Algebraische Graphersetzung: mathematische Modellierung und Konfluenz. Forschungsbericht des Fachbereichs Informatik, TU Berlin, Berlin (1993)

32. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
33. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) Proceedings of the 22nd International Conference on on Software Engineering, pp. 742–745. ACM (2000)
34. Orejas, F.: Symbolic graphs for attributed graph constraints. J. Symbolic Comput. **46**(3), 294–315 (2011)
35. Padberg, J., Kahloul, L.: Overview of reconfigurable Petri nets. In: Heckel, R., Taentzer, G. (eds.) Ehrig Festschrift. LNCS, vol. 10800, pp. 201–222. Springer, Cham (2018)
36. Plump, D.: Computing by Graph Rewriting. Habilitation thesis, Universität Bremen (1999)
37. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 128–143. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_11
38. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69968-9
39. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_40
40. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_23
41. Rensink, A., Kuperus, J.-H.: Repotting the geraniums: on nested graph transformation rules. In: Boronat, A., Heckel, R. (eds.) Graph Transformation and Visual Modelling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 18 (2009)
42. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific, Singapore (1997)
43. Sangiorgi, D., Walker, D.: The π-calculus-A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
44. Schürr, A., Westfechtel, B.: Graph grammars and graph rewriting systems (in German). Technical report AIB 92–15, RWTH Aachen (1992)
45. Taentzer, G.: AGG: a tool environment for algebraic graph transformation. In: Nagl, M., Schürr, A., Münch, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 481–488. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45104-8_41
46. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Formal Models and Semantics. Handbook of Theoretical Computer Science, vol. B, pp. 675–788. Elsevier (1990). Chapter 13

Initial Conflicts and Dependencies: Critical Pairs Revisited

Leen Lambers[1(✉)], Kristopher Born[2], Fernando Orejas[3], Daniel Strüber[4], and Gabriele Taentzer[2]

[1] Hasso-Plattner-Institut, Potsdam, Germany
leen.lambers@hpi.de
[2] Philipps-Universität Marburg, Marburg, Germany
{born,taentzer}@informatik.uni-marburg.de
[3] Technical University of Catalunia, Barcelona, Spain
orejas@lsi.upc.edu
[4] Universität Koblenz-Landau, Koblenz, Germany
strueber@uni-koblenz.de

Abstract. Considering a graph transformation system, a critical pair represents a pair of conflicting transformations in a minimal context. A conflict between two direct transformations of the same structure occurs if one of the transformations cannot be performed in the same way after the other one has taken place. Critical pairs allow for static conflict and dependency detection since there exists a critical pair for each conflict representing this conflict in a minimal context. Moreover it is sufficient to check each critical pair for strict confluence to conclude that the whole transformation system is locally confluent. Since these results were shown in the general categorical framework of M-adhesive systems, they can be instantiated for a variety of systems transforming e.g. (typed attributed) graphs, hypergraphs, and Petri nets.

In this paper, we take a more declarative view on the minimality of conflicts and dependencies leading to the notions of *initial conflicts* and *initial dependencies*. Initial conflicts have the important new characteristic that for each given conflict a unique initial conflict exists representing it. We introduce initial conflicts for M-adhesive systems and show that the Completeness Theorem and the Local Confluence Theorem still hold. Moreover, we characterize initial conflicts for typed graph transformation systems and show that the set of initial conflicts is indeed smaller than the set of essential critical pairs (a first approach to reduce the set of critical pairs to the important ones). Dual results hold for initial dependencies.

1 Introduction

Graph transformations are often affected by conflicts and dependencies between the included rules. To improve their transformation specifications, users may require a list of all potential conflicts and dependencies occurring between the contained rules. Critical pair analysis (CPA) is a static analysis to enable

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 105–123, 2018.
https://doi.org/10.1007/978-3-319-75396-6_6

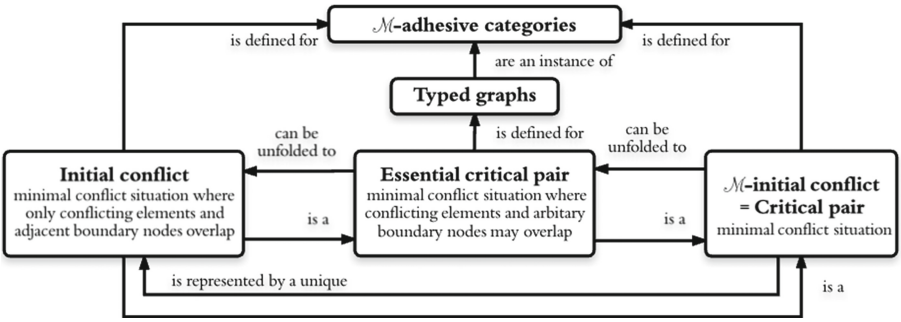


Fig. 1. Overview of critical pair kinds with their formal foundations. Characterizations are given in the category of typed graphs.

the automated computation of such a list. The notion of critical pair was coined in the domain of mathematical logic, where it was first introduced for term rewriting systems [1]. More recently, it turned out to be useful for graph transformation systems as well. Plump [2] and Heckel et al. [3] introduced critical pair notions for term graph rewriting and typed attributed graphs, respectively. It was Ehrig who, together with his colleagues, came up with a generalized theory of critical pairs for adhesive high-level replacement systems [4]. A remarkable feature that CPA inherits from graph transformations is its versatility. CPA has been used in many scenarios, including conflict detection in functional system requirements [5], detection of product-line feature interactions [6], verification of model transformations [7], and numerous other software engineering scenarios. In these settings, CPA was used to show the correctness of a specification, to improve a rule set by fostering the independent application of its rules, and to support developers during design decisions.

The original critical-pair notion was focused on *delete-use* conflicts, i.e., situations where a rule deletes an element required by the second one, and the dual counterpart of *produce-use* dependencies. To consider *produce-forbid* conflicts as well, the notion of critical pair was extended to rules with negative application conditions in [8]. Each critical pair represents one such conflict situation in a *minimal context*: It comprises a graph specifying an overlap of the two considered rules, together with two jointly surjective match morphisms embedding the rules' left-hand sides into this graph.

Essential critical pairs [9] were introduced to optimize static conflict detection and local confluence analysis. They specify a well-defined subset of the set of critical pairs between a pair of rules to support a more "compact" representation of potential conflicts and dependencies, while providing the same main benefits as regular critical pairs: *completeness*, i.e. each potential conflict or dependency is represented by a critical pair in a minimal context, and *analyzibility of local confluence*, i.e., strict confluence of each critical pair implies local confluence. However, we shall see that essential critical pairs do not provide the *most* compact representation of potential conflicts and dependencies.

In this paper, we consider the following question: *Can the set of essential critical pairs be reduced even further without losing completeness and local confluence?* To answer this question, we introduce the notion of *initial conflicts*. As shown in Fig. 1, initial conflicts further reduce the set of critical pairs, in the sense that the same initial conflict represents multiple essential critical pairs. More precisely, the initial conflict is obtained from these essential critical pairs by "unfolding" them, i.e., reducing the overlap of the conflicting rules. A similar relationship between essential and regular critical pairs was shown in [9]. In contrast to essential critical pairs, initial critical pairs are defined declaratively and generically in a categorical way, rather than constructively and restricted to the category of typed graphs. In sum, we make the following contributions:

- We define the notion of initial conflicts in a purely category-theoretical way, using the framework of $\mathcal{M}$-adhesive categories.
- We provide results to show that the set of initial conflicts still enjoys the completeness property and the local confluence theorem. Moreover, we introduce $\mathcal{M}$-initial conflicts and show that they are equivalent to critical pairs.
- We characterize initial conflicts for typed graph transformation systems and show that the set of initial conflicts is effectively smaller than the set of essential critical pairs.
- Dually to initial conflicts, we introduce initial dependencies.

The rest of this paper is structured as follows. Section 2 introduces a running example, whereas Sect. 3 revisits the necessary preliminaries. Section 4 introduces the notion of initial conflicts for $\mathcal{M}$-adhesive systems and its relationship with critical pairs. Readers mainly interested in initial conflicts for graph transformation systems may skip this section. Section 5 formally characterizes initial conflicts in the category of typed graphs. Section 6 outlines how new results can be transferred to dependencies. Section 7 discusses related work and concludes our work.

2 Motivating Example

Throughout this paper, we illustrate the new concepts with an example, which specifies the operational semantics of finite automata by graph transformation. Finite automata are mainly used to recognize words that conform to regular expressions. A finite automaton consists of a set of states, a set of labeled transitions running between states, a start state, and a set of end states. If the whole word can be read by the automaton such that it finally reaches an end state, the word is in the language of this automaton. In the literature, deterministic automata are distinguished from non-deterministic ones. An automaton is *deterministic* if, for every state and symbol, there is at most one transition starting in that state and being labeled with that symbol. We will see that the specification of non-deterministic automata shows conflicts.

In the upper left corner of Fig. 2, a simple type graph for finite automata and input words is shown. A *Transition* has a *(s)ource* and a *(t)arget* edge to

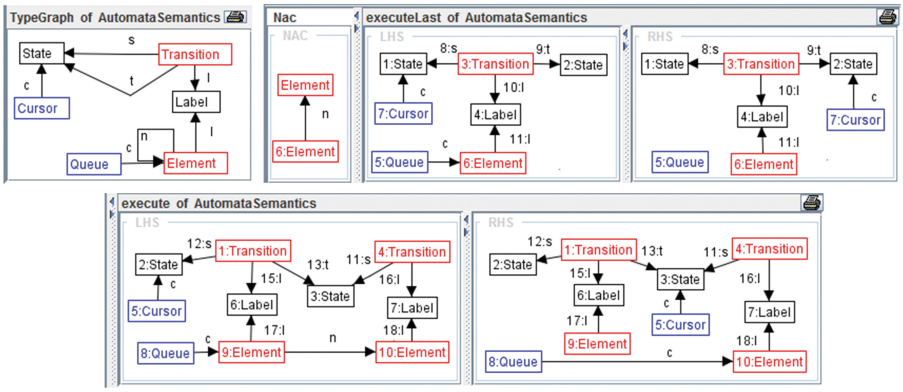


Fig. 2. Type graph and rules for executing transitions in finite automata

two *States* and has a *Label*. The *Cursor* points to the *(c)urrent* state. An input word is given by a *Queue* of *Elements* corresponding to labels. The queue points to the *(n)ext* symbol to be recognized.

Additionally, Fig. 2 depicts two rules specifying the execution of automata. Rule *execute* executes a transition which is not a loop. The cursor is set to the next state and the input queue cursor points to the next element. For the last symbol we use rule *executeLast* which just consumes the last symbol and sets the cursor to the next state. Finally, all queue elements may be deleted.

Figure 3 shows an example automaton A in concrete and abstract syntax. This automaton recognizes the language $L(A) = \{ab^n c | n \geq 0\}$. An example input word is *abbc*. The abstract syntax graph in Fig. 3 shows an instance graph conforming to the type graph in Fig. 2. It contains the abstract syntax information for both the example automaton and the input word, glued at all *Label*-nodes. Note that n-typed edges define the order of symbols in the input word.

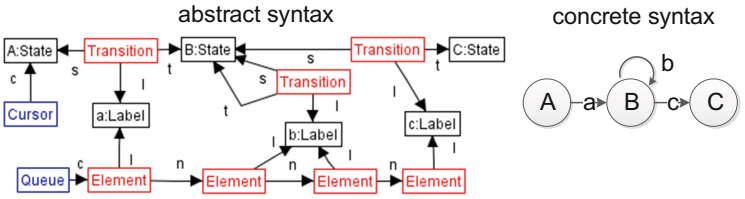


Fig. 3. An example automaton with an example input word

To recognize label a, rule *execute* is applied at its only possible match. As the result, the cursor points to *B:State*, the first n-edge is deleted, and the queue points to the first element containing label b. To recognize the whole word three further rule applications are needed.

The execute-rules cause many potential conflicts; for example, the pair *(execute, execute)* has 49 essential critical pairs. It will turn out that most of them

just show variants of basically the same conflicts. Their set of initial conflicts, however, contains just 7 pairs. (By the way, AGG [10] runs out of memory when computing all critical pairs after checking over 12,000 rule overlaps. Verigraph [11] found 51,602 overlaps with monomorphic matches, where 21,478 of them represent critical pairs.)

3 Preliminaries

In this section, we give a short introduction to $\mathcal{M}$-adhesive categories [4,12] and $\mathcal{M}$-adhesive systems to define the setting for the categorical results in Sect. 4. Moreover, we recall the classical notions of conflict and critical pair as well as the corresponding results Completeness Theorem and Local Confluence Theorem within this categorical framework [4,13].

By considering $\mathcal{M}$-adhesive categories it is possible to avoid similar investigations for different instantiations like e.g. attributed graphs, Petri nets, hypergraphs, and algebraic specifications. An *$\mathcal{M}$-adhesive category* $\langle \mathbf{C}, \mathcal{M} \rangle$ is a category $\mathbf{C}$ with a distinguished class $\mathcal{M}$ of monomorphisms satisfying certain properties. The most important one is the van Kampen (VK) property stating a certain kind of compatibility of pushouts and pullbacks along $\mathcal{M}$-morphisms. In [13] it is proven that the category of typed graphs $\langle \mathbf{Graphs_{TG}}, \mathcal{M} \rangle$ with the class $\mathcal{M}$ of all injective typed graph morphisms is $\mathcal{M}$-adhesive. In Sect. 5 we will instantiate the idea of initial conflicts to this category.

Within this categorical framework we introduce our notion of rule, direct transformation, and $\mathcal{M}$-adhesive system following the so-called DPO approach [13]. Note that these definitions can be instantiated to the case of typed graph transformation by replacing each object with a typed graph and each morphism with a typed graph morphism. In the category $\mathbf{Graphs}_{TG}$, $\mathcal{M}$-adhesive systems are then called typed graph transformation systems.

Definition 1 (Rule, direct transformation, $\mathcal{M}$-adhesive system). *Given an $\mathcal{M}$-adhesive category $\langle \mathbf{C}, \mathcal{M} \rangle$, then we define the following:*

- *A* rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ *is a span of morphisms $l, r \in \mathcal{M}$. We call L (resp. R), the left-hand side (LHS) (resp. right-hand side (RHS)) of rule p.*
- *A* direct transformation $G \overset{p,m}{\Rightarrow} H$ *from G to H via a rule $p : L \leftarrow K \rightarrow R$ and a morphism $m : L \rightarrow G$, called* match*, consists of the double pushout (DPO) [14] as depicted in Fig. 4. Since pushouts along $\mathcal{M}$-morphisms in an $\mathcal{M}$-adhesive category always exist, the DPO can be constructed if the pushout complement of $m \circ l$ exists. Then, the match m satisfies the* gluing condition *of rule p.*
- *A* transformation*, denoted as $G_0 \overset{*}{\Rightarrow} G_n$, is a sequence $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ of direct transformations. For $n = 0$, we have the identical transformation $G_0 \Rightarrow G_0$. Moreover, for $n = 0$ we also allow isomorphisms $G_0 \cong G'_0$, because pushouts, and hence also direct transformations, are only unique up to isomorphism.*
- *Given a set of rules $\mathcal{R}$, triple $(\mathcal{C}, \mathcal{M}, \mathcal{R})$ is an* $\mathcal{M}$-adhesive system*.*

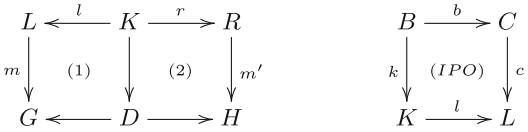
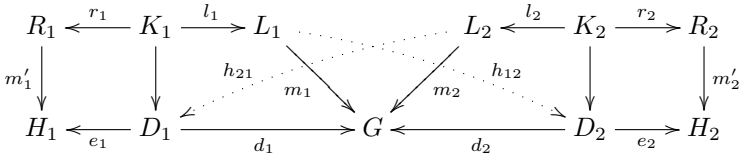


Fig. 4. Direct transformation as DPO, deletion graph constructed by initial pushout

The classical definitions for transformation pairs in conflict and critical pairs are recalled in [13]. The latter represent conflicts in a minimal context materialized by a pair of matches being jointly epimorphic. In particular, for the critical pair definition it is assumed that the $\mathcal{M}-$adhesive category comes with a so-called $\mathcal{E}'$-$\mathcal{M}$ pair factorization, generalizing the classical epi-mono factorization to a pair of morphisms with the same codomain. It is proven in [13] that the category $\mathbf{Graphs_{TG}}$ of typed graphs has a unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization, where $\mathcal{E}'$ is the class of jointly surjective typed graph morphism pairs. Note that we stick to the notation $\mathcal{E}'$ for jointly epimorphic morphisms as in [13], where $\mathcal{E}$ on the other hand is used to denote a class of epimorphisms.

Definition 2 (conflict, critical pair). *A pair of direct transformations* $t_1 : G \overset{p_1,m_1}{\Rightarrow} H_1$ *and* $t_2 : G \overset{p_2,m_2}{\Rightarrow} H_2$ *is in* conflict *if* $\nexists h_{12} : L_1 \to D_2 : d_2 \circ h_{12} = m_1$ *or* $\nexists h_{21} : L_2 \to D_1 : d_1 \circ h_{21} = m_2$.



Given an $\mathcal{E}'$-$\mathcal{M}$ *pair factorization, a* critical pair *is a pair of direct transformations* $K \overset{p_1,m_1}{\Rightarrow} P_1$ *and* $K \overset{p_2,m_2}{\Rightarrow} P_2$ *in conflict with* (m_1, m_2) *in* $\mathcal{E}'$.

Now, we recall the Completeness Theorem for critical pairs, where we need the notion of extension morphism and extension diagram as presented in [4,13].

Definition 3 (Extension diagram). *An* extension diagram *is a diagram (1) as shown on the left of Fig. 5 where* $f : G' \to G$ *is a morphism, called* extension morphism, *and* $t : G \overset{p}{\Longrightarrow} H$ *as well as* $t' : G' \overset{p}{\Longrightarrow} H'$ *are two direct transformations via the same rule* p *with matches* m' *and* $f \circ m'$ *respectively, defined by the four pushouts in the middle of Fig. 5.*

Transformations are actually extended by extending their context D' to D. Morphisms $f : G' \to G$ and $f' : H' \to H$ are the resulting pushout morphisms. In the category $\mathbf{Graphs}_{TG}$, this means that the context graph D' may be embedded into a larger one and/or elements of it may be glued together. Corresponding actions are reflected in f and f' but no additional actions may happen.

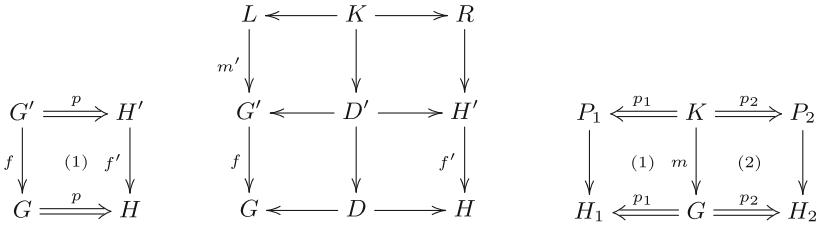


Fig. 5. Extension diagram (overview and more detailed), extension diagram for transformation pair

The Completeness Theorem [4,13] for critical pairs states that each potential conflict can be represented in a minimal context by some critical pair. For conciseness reasons in the following we sometimes write that the $\mathcal{M}$-adhesive system comes with an $\mathcal{E}'$-$\mathcal{M}$ pair factorization (or some other additional requirement) if the corresponding $\mathcal{M}$ adhesive category does.

Theorem 4 (Completeness Theorem). *Let an $\mathcal{M}$-adhesive system with an $\mathcal{E}'$-$\mathcal{M}$ pair factorization be given. For each pair of direct transformations $H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ in conflict, there is a critical pair $P_1 \overset{p_1}{\Longleftarrow} K \overset{p_2}{\Longrightarrow} P_2$ with extension diagrams (1) and (2) and $m \in \mathcal{M}$ as depicted on the right of Fig. 5.*

The Local Confluence Theorem [4,13] states that, by checking each critical pair for *strict confluence*, one can conclude local confluence of the overall transformation system. Strict confluence ensures that the largest subobject of K preserved by both t_1 and t_2 is preserved by the transformations establishing local confluence. Note that for this result the $\mathcal{M}$-adhesive category needs to fulfill an additional requirement: The category needs so-called initial pushouts describing the existence of a "smallest" pushout over a morphism [13]. It is proven in [13] that the category $\mathbf{Graphs_{TG}}$ of typed graphs has initial pushouts.

Theorem 5 (Local Confluence Theorem). *Given an $\mathcal{M}$-adhesive system with an $\mathcal{E}'$-$\mathcal{M}$ pair factorization and initial pushouts over $\mathcal{M}$-morphisms, it is locally confluent if all its critical pairs are strictly confluent.*

For a closer look at conflicts we have to identify the following two rule parts: the deletion object comprising the part to be deleted and its boundary specifying how the deletion object is connected to the preserved rule part.

Definition 6 (Boundary and deletion objects). *Let an $\mathcal{M}$-adhesive system with initial POs [13] over $\mathcal{M}$ and a rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ as well as an initial pushout (IPO) (see Fig. 4) over morphism l be given. Then we say that B is the* boundary object *for rule p and the context object C in this IPO is the* deletion object *for rule p.*

4 Initial Conflicts

The original idea of critical pairs consists of considering all possible conflicting transformations in a minimal context. In the classical critical pair definition this minimal context is materialized by a pair of jointly epimorphic matches from a special set $\mathcal{E}'$ arising from the $\mathcal{E}'$-$\mathcal{M}$ pair factorization as additional requirement to the $\mathcal{M}$-adhesive category. We propose here a more *declarative* view on a pair of direct transformations in conflict to be minimal resulting in the subsequent definition of *initial conflicts*. In categorical terms, one can use actually the notion of *initiality of transformation pairs* to obtain this new view on critical pairs. Interestingly, it will turn out that each initial conflict is a critical pair but not the other way round. We will show however at the end of this section that all initial conflicts still satisfy the Completeness Theorem as well as the Local Confluence Theorem. Consequently, we have found an important subset within the set of classical critical pairs for performing static conflict detection as well as local confluence analysis for $\mathcal{M}$-adhesive systems. Finally, we will see also that the notion of $\mathcal{M}$-initiality allowing merely $\mathcal{M}$-morphisms as extension morphisms leads to the notion of $\mathcal{M}$-*initial conflicts*, representing an equivalent characterization of critical pairs provided that the $\mathcal{E}'$-$\mathcal{M}$ pair factorization for building them is unique. We will see that by definition ($\mathcal{M}$-)initial conflicts have the important new characteristic that for each given pair of conflicting transformations there exists a *unique* ($\mathcal{M}$-)initial conflict representing it.

Definition 7 (($\mathcal{M}$-)Initial transformation pair). *Given a pair of direct transformations* $(t_1, t_2) : H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$, *then* $(t_1^I, t_2^I) : H_1^I \overset{p_1,m_1^I}{\Longleftarrow} G^I \overset{p_2,m_2^I}{\Longrightarrow} H_2^I$ *is an* initial transformation pair *(resp. $\mathcal{M}$-initial transformation pair) for* (t_1, t_2) *if it can be embedded into* (t_1, t_2) *via extension diagrams (1) and (2) and extension morphism* f^I *(resp.* $f^I \in \mathcal{M}$*) as in Fig. 6 such that for each transformation pair* $(t_1', t_2') : H_1' \overset{p_1,m'_1}{\Longleftarrow} G' \overset{p_2,m'_2}{\Longrightarrow} H_2'$ *that can be embedded into* (t_1, t_2) *via extension diagrams (3) and (4) and extension morphism* f *(resp.* $f \in \mathcal{M}$*) as in Fig. 6 it holds that* (t_1^I, t_2^I) *can be embedded into* (t_1', t_2') *via unique extension diagrams (5) and (6) and unique vertical morphism* f'^I *(resp.* $f'^I \in \mathcal{M}$*) s.t.* $f \circ f'^I = f^I$.

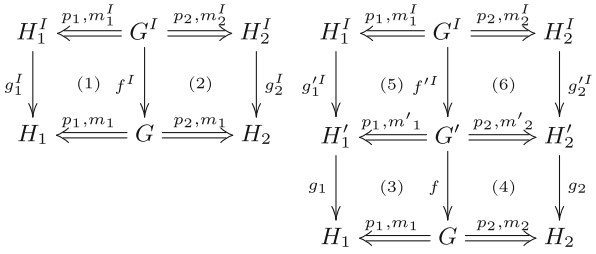


Fig. 6. ($\mathcal{M}$-)initial transformation pair $H_1^I \overset{p_1,m_1^I}{\Longleftarrow} G^I \overset{p_2,m_2^I}{\Longrightarrow} H_2^I$ for $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$

Lemma 8 (Uniqueness of ($\mathcal{M}$-)initial transformation pair). *Given a pair of direct transformations* $(t_1, t_2) : H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ *then, if* $(t_1^I, t_2^I) : H_1^I \overset{p_1,m_1^I}{\Longleftarrow} G^I \overset{p_2,m_2^I}{\Longrightarrow} H_2^I$ *is an initial pair of transformations (resp. $\mathcal{M}$-initial pair of transformations) for* (t_1, t_2)*, any other initial transformation pair (resp. $\mathcal{M}$-initial transformation pair) for* (t_1, t_2) *is isomorphic to* (t_1^I, t_2^I)*.*

Proof. Consider some other initial pair $(t_1'^I, t_2'^I) : H_1'^I \overset{p_1,m_1^I}{\Longleftarrow} G'^I \overset{p_2,m_2^I}{\Longrightarrow} H_2'^I$ for (t_1, t_2). Then the extension diagrams in Fig. 7 can be built by definition of ($\mathcal{M}$)-initial pairs. Now consider for (t_1^I, t_2^I) trivial extension diagrams via the identity extension morphism $id : G^I \to G^I$. The extension morphism of the extension diagrams $(7) + (5)$ and $(8) + (6)$ w.r.t. (t_1, t_2) needs to be equal to the identity extension morphism by definition. Analogously, one can argue for $(5) + (7)$ and $(6) + (8)$. Therefore both initial pairs are isomorphic. □

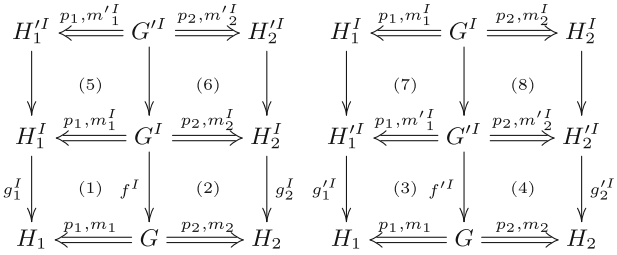


Fig. 7. Uniqueness of ($\mathcal{M}$-)initial transformation pair

Our key notion of initial conflicts is based on the *existence of initial transformation pairs* for *conflicting* transformation pairs. It describes the "smallest" conflict that can be embedded into a given conflict. It is an open issue to come up with a constructive categorical characterization in the context of $\mathcal{M}$-adhesive systems, which is the reason for having it as an additional requirement (formulated in Definition 9) for now. It is possible, however, to constructively characterize $\mathcal{M}$-initial transformation pairs for conflicts provided that a unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization is given (see Lemma 10). The key difference between initiality and $\mathcal{M}$-initiality is that the extension morphism used to embed the "smallest" conflict into a given conflict is general or needs to be in $\mathcal{M}$, respectively.

Definition 9 (Existence of initial transformation pair for conflict). *An $\mathcal{M}$-adhesive system has* initial transformation pairs for conflicts *if, for each transformation pair in conflict* (t_1, t_2)*, the initial transformation pair* (t_1^I, t_2^I) *exists.*

Lemma 10 (Existence of $\mathcal{M}$-initial transformation pair for conflict). *In an $\mathcal{M}$-adhesive system with unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization, for each pair of transformations* (t_1, t_2) *in conflict, there exists an $\mathcal{M}$-initial transformation pair* (t_1^I, t_2^I)*. In particular, it corresponds to the classical critical pair as constructed in Theorem 4.*

Proof. Consider the critical pair (t_1^I, t_2^I) as given by Theorem 4. We show that this is indeed an $\mathcal{M}$-initial transformation pair for (t_1, t_2). Given matches (m_1, m_2) of transformation pair (t_1, t_2) and matches (m_1^I, m_2^I) for the pair (t_1^I, t_2^I) built via the pair factorization (as on the left of Fig. 6). Then $(m_1^I, m_2^I) \in \mathcal{E}'$ and the extension morphism f^I from (t_1^I, t_2^I) to (t_1, t_2) is in $\mathcal{M}$ and $f^I \circ m_1^I = m_1$ and $f^I \circ m_2^I = m_2$. Consider some other pair (t_1', t_2') that can be embedded via some extension morphism $f : G' \to G \in \mathcal{M}$ into (t_1, t_2) (as on the right of Fig. 6). According to Theorem 4 we again have a critical pair with matches in $\mathcal{E}'$ that can be embedded into (t_1', t_2') via some extension morphism f'^I in $\mathcal{M}$. Since the $\mathcal{E}'$-$\mathcal{M}$ pair factorization is unique and $\mathcal{M}$-morphisms are closed under composition, this will actually be indeed the same critical pair (t_1^I, t_2^I) as for (t_1, t_2). □

Now we are ready to introduce our notion of ($\mathcal{M}$-)initial conflicts representing the set of all possible "smallest" conflicts. Like for classical critical pairs they are defined for a given $\mathcal{M}$-adhesive system allowing for static conflict detection.

Definition 11 (($\mathcal{M}$-)Initial conflict). *Given an $\mathcal{M}$-adhesive system with initial transformation pairs for conflicts, a pair of direct transformations in conflict $(t_1, t_2) : H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ is an* initial conflict *if it is isomorphic to the initial transformation pair for (t_1, t_2).*

Given an $\mathcal{M}$-adhesive system with unique $\mathcal{E}'$-$\mathcal{M}$-pair factorization, a pair of direct transformations in conflict $(t_1, t_2) : H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ is an $\mathcal{M}$-initial conflict *if it is isomorphic to the $\mathcal{M}$-initial transformation pair for (t_1, t_2).*

It follows quite straightforwardly that the set of $\mathcal{M}$-initial conflicts corresponds to the classical set of critical pairs for an $\mathcal{M}$-adhesive system with unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization.[1] Moreover, it follows that each initial conflict is an $\mathcal{M}$-initial conflict (or critical pair), in particular. A counterexample for the reverse direction will be given in the next section.

Theorem 12 ($\mathcal{M}$-Initial conflict = critical pair). *In an $\mathcal{M}$-adhesive system with unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization, each $\mathcal{M}$-initial conflict is a critical pair and vice versa.*

Proof. Given some $\mathcal{M}$-initial conflict $(t_1^I, t_2^I) : H_1^I \overset{p_1}{\Longleftarrow} G^I \overset{p_2}{\Longrightarrow} H_2^I$. Then it follows directly from Definitions 2, 11 and Lemma 10 that (t_1^I, t_2^I) is a critical pair because it is in conflict and its matches are in $\mathcal{E}'$.

Given a critical pair $(t_1^I, t_2^I) : H_1^I \overset{p_1}{\Longleftarrow} G^I \overset{p_2}{\Longrightarrow} H_2^I$, we need to show that it is an $\mathcal{M}$-initial conflict. When constructing the initial transformation pair for (t_1^I, t_2^I) according to Lemma 10, a pair of isomorphic transformations w.r.t. (t_1^I, t_2^I) would be constructed because of the $\mathcal{E}'$-$\mathcal{M}$ pair factorization being unique and the fact that one could choose alternatively as extension morphism the identity morphism on G^I (being in $\mathcal{M}$), since the matches are already in $\mathcal{E}'$. □

[1] Classical critical pairs are slightly more general since they do not require uniqueness of the $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

Theorem 13 (Initial conflict is $\mathcal{M}$-Initial conflict). *In an $\mathcal{M}$-adhesive system with initial transformation pairs for conflicts and a unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization, each initial conflict is an $\mathcal{M}$-initial conflict.*

Proof. Given some initial conflict $(t_1^I, t_2^I) : H_1^I \overset{p_1}{\Longleftarrow} G^I \overset{p_2}{\Longrightarrow} H_2^I$, then because of Lemma 10 we can construct an $\mathcal{M}$-initial transformation pair for it. By definition, each $\mathcal{M}$-initial transformation pair is also an initial transformation pair since each morphism in $\mathcal{M}$ is a regular morphism. Because of Lemma 8, such an initial pair is unique and, for an initial conflict, isomorphic to (t_1^I, t_2^I) in particular, such that the initial transformation pair is indeed an $\mathcal{M}$-initial pair. □

To decide if initial conflicts can replace critical pairs for detecting conflicts and analyzing local confluence statically, we investigate now if the Completeness Theorem and Local Confluence Theorem hold. The Completeness Theorem for initial conflicts can indeed be formulated in a slightly modified way w.r.t. Theorem 4. This is because the extension morphism is not necessarily in $\mathcal{M}$ anymore. Informally speaking, we are able to represent several critical pairs by one initial conflict by unfolding elements that were overlapped unnecessarily (i.e. without having importance for the described conflict). Note also that, instead of requiring an $\mathcal{E}'$-$\mathcal{M}$ pair factorization as in the classical Completeness Theorem for critical pairs, we assume the existence of initial transformation pairs for conflicts.

Lemma 14 (Conflict inheritance). *Given a pair of direct transformations $(t_1, t_2) : H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ in conflict and another pair of direct transformations $(t_1', t_2') : H_1' \overset{p_1}{\Longleftarrow} G' \overset{p_2}{\Longrightarrow} H_2'$ that can be embedded into (t_1, t_2) via extension morphism f and corresponding extension diagrams as depicted in Fig. 8, then (t_1', t_2') is also in conflict.*

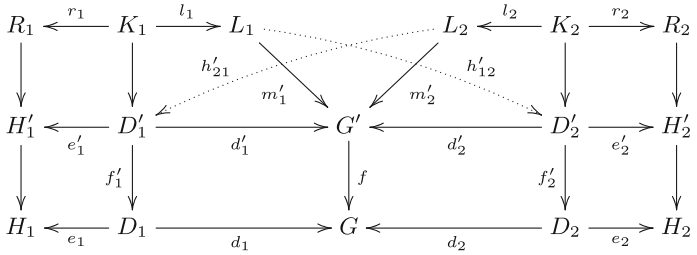


Fig. 8. Conflict inheritance

Proof. Assume that $(t_1', t_2') : H_1' \overset{p_1, m_1'}{\Longleftarrow} G' \overset{p_2, m_2'}{\Longrightarrow} H_2'$ are parallel independent. This means that some morphism h_{12}' (and h_{21}') exists such that $d_1' \circ h_{12}' = m_2'$ (and $d_2' \circ h_{21}' = m_1'$). Then $(t_1, t_2) : H_1 \overset{p_1, m_1}{\Longleftarrow} G \overset{p_2, m_2}{\Longrightarrow} H_2$ with $f \circ m_1' = m_1$ and $f \circ m_2' = m_2$ would be parallel independent as well, which is a contradiction. This

is because a morphism $h_{12} = f'_1 \circ h'_{12}$ would exist such that $d_1 \circ h_{12} = d_1 \circ f'_1 \circ h'_{12} = f \circ d'_1 \circ h'_{12} = f \circ m'_2 = m_2$ and similarly, a morphism $h_{21} = f'_2 \circ h'_{21}$ would exist such that $d_2 \circ h_{21} = m_1$. □

Theorem 15 (Completeness theorem for initial conflicts). *Consider an $\mathcal{M}$-adhesive system with initial transformation pairs for conflicts. For each pair of direct transformations $(t_1, t_2) : H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ in conflict, there is an initial conflict $(t_1^I, t_2^I) : P_1 \overset{p_1}{\Longleftarrow} K \overset{p_2}{\Longrightarrow} P_2$ with extension diagrams (1) and (2).*

Proof. We can assume the existence of the initial transformation pair (t_1^I, t_2^I) for the given pair (t_1, t_2) in conflict. It remains to show that the initial transformation pair (t_1^I, t_2^I) for (t_1, t_2) is indeed an initial conflict according to Definition 11. Firstly, the transformation pair (t_1^I, t_2^I) is in conflict according to Lemma 14. Secondly, each initial conflict for (t_1^I, t_2^I) needs to be isomorphic to (t_1^I, t_2^I) since we would have found a non-isomorphic initial transformation pair for (t_1, t_2) by composition of extension diagrams otherwise. This contradicts Lemma 8. □

The Local Confluence Theorem can be formulated for initial conflicts similarly to the one for classical critical pairs because its proof actually does not need the requirement that extension morphisms should be in $\mathcal{M}$.

Theorem 16 (Local confluence theorem for initial conflicts). *Given an $\mathcal{M}$-adhesive system with initial pushouts and initial transformation pairs for conflicts, an $\mathcal{M}$-adhesive system is locally confluent if all its initial conflicts are strictly confluent.*

Proof. The proof runs completely analogously to the proof of the regular Local Confluence Theorem (Theorem 5 in [13]). The only difference is that for this proof, we need initial pushouts over general morphisms whereas in Theorem 5 initial pushouts over $\mathcal{M}$-morphisms are sufficient. The proof requires initial pushouts over the extension morphism m embedding a critical pair (or initial conflict) into a pair of conflicting transformations. This extension morphism belongs to the special subset $\mathcal{M}$ of monomorphisms for classical critical pairs, but it is a general morphism in the case of initial conflicts. □

In summary, given an $\mathcal{M}$-adhesive system, we obtain the Completeness and Local Confluence Theorem in slightly different flavors. For Completeness of $\mathcal{M}$-initial conflicts (or classical critical pairs) we assume to have a unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization and for Local Confluence we in addition require initial POs over $\mathcal{M}$. For Completeness of initial conflicts we assume the existence of initial transformation pairs for conflicts (*) and for Local Confluence we in addition require initial POs. For the category of typed graphs it is shown in [13] that all these requirements hold apart from requirement (*) proven in the next section.

5 Initial Conflicts for Typed Graph Transformation

In this section, we discuss how initial conflicts look like in graph transformation systems, i.e., in the category $\mathbf{Graphs}_{TG}$. Moreover, we clarify how they are

related to essential critical pairs which were introduced in [9] as a first optimization of critical pairs in graph transformation systems. Essential critical pairs form a subset of critical pairs for which the Completeness Theorem as well as the Local Confluence Lemma still hold. Therefore, an obvious question is the following: Does the set of initial conflicts correspond to the set of essential critical pairs in the case of typed graph transformation systems? It turns out that, in general, the set of initial conflicts is a proper subset of the set of essential critical pairs here. First, we show an initial conflict occurring in our running example.

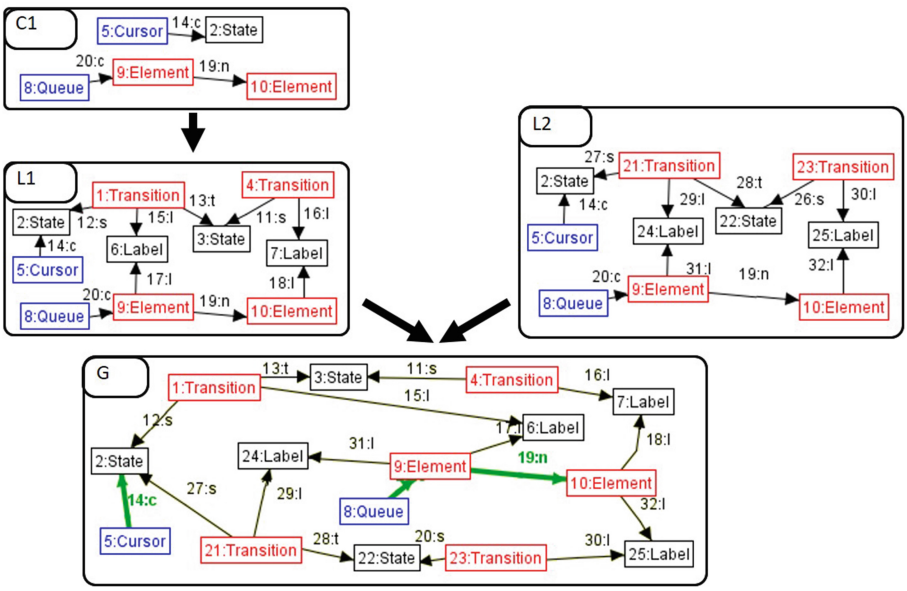


Fig. 9. Example for an initial conflict (Color figure online)

Example 17 (Initial conflict). In a non-deterministic automaton there may be a state with two subsequent transitions, both triggered by the same label. This situation is described symbolically by the (excerpt of the) initial conflict in Fig. 9.[2] Both transitions can be executed, i.e., the rule *execute* is applicable with different results at two different matches. These matches lead to transformations in conflict since they are both triggered and therefore change the current queue pointer as well as the current cursor position. The corresponding edges are highlighted in the overlap graph (green) at the bottom of the figure. Together with their

[2] Note that this situation is somewhat unrealistic, since in principle it allows a symbol to be connected to two different labels. However, graph G is supposed to be embedded into realistic situations to check if a pair of transformations is conflicting. It is part of future work to integrate the notion of constraints into our theory of initial conflicts, leading – if possible – to realistic situations already in initial conflicts.

adjacent nodes they form the actual overlap of both matches. Note that applying rule *execute* at these matches leads to an initial conflict since the overlap is in deleted elements and their adjacent boundary nodes only. (See also Lemma 21 below.) If the overlap is so small, no other transformation pair is embeddable since unfoldings can occur in preserved elements only.

In the category $\mathbf{Graphs}_{TG}$, a critical pair is essential if two injective matches overlap in deleted elements and boundary nodes only [9]. The following example illustrates that indeed not each essential critical pair is an initial conflict.

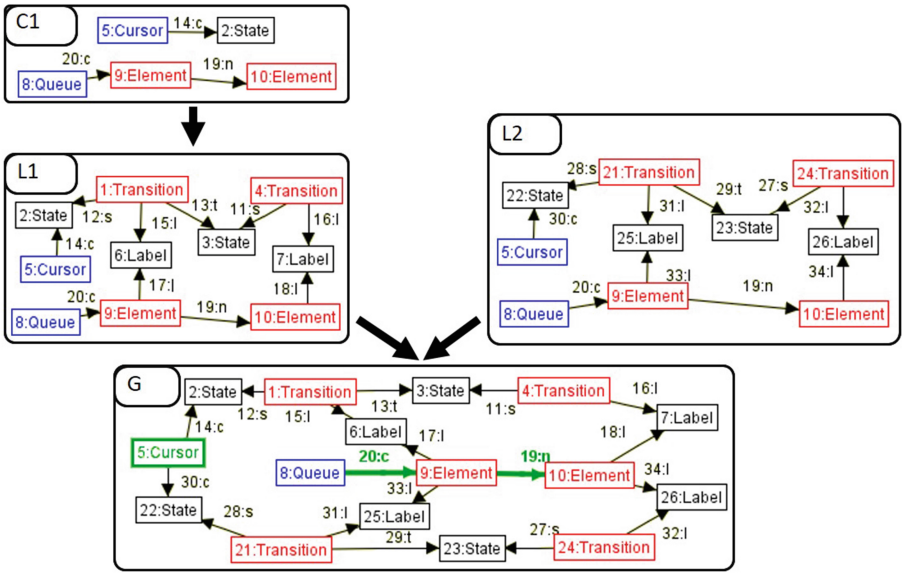


Fig. 10. Example for an essential critical pair not being an initial conflict

Example 18 (Essential critical pair not being an initial conflict). A *parallel* automaton may have several transitions that can be executed in the current states. Such a situation is described by the (excerpt of the) critical pair in Fig. 10. There exist two different current states with outgoing transitions both recognizing the same label. In this case, both transitions could be executed, i.e., the rule *execute* can be applied at two different matches. Since the matches overlap in deleted elements and isolated boundary nodes (as highlighted in green) only, the critical pair shown in Fig. 10 is essential. In particular, the isolated boundary node *5:Cursor* occurs in the overlap (and not the adjacent edges that are to be deleted). The same conflict would be specified if the cursor nodes of both LHSs were not overlapped. Hence, this essential critical pair is not an initial conflict. A similar critical pair with two cursors is embeddable into the depicted one. Since that cannot be further unfolded, it represents the corresponding initial conflict.

The following lemma shows that the category $\mathbf{Graphs}_{TG}$ has initial transformation pairs for conflicts and hence, initial conflicts. As a preparatory work,

we define matches that do not overlap in isolated boundary nodes. If they would, then it would be possible to unfold the matches at these isolated boundary nodes.

Definition 19 (No isolated boundary node). *Given two rules p_1 and p_2 with LHSs L_1 and L_2, boundary graphs B_1 and B_2 as well as deletion graphs C_1 and C_2 as in Definition 6. Morphisms $m_1 : L_1 \to G$ and $m_2 : L_2 \to G$* do not overlap in isolated boundary nodes *if* $\forall x \in m_1(c_1(b_1(B_1))) \cap m_2(L_2)$:

$\exists e \in m_1(c_1(C_1)) \cap m_2(L_2) : x = src(e) \vee x = tgt(e)$ *and*
$\forall x \in m_2(c_2(b_2(B_2))) \cap m_1(L_1)$:
$\exists e \in m_2(c_2(C_2)) \cap m_1(L_1) : x = src(e) \vee x = tgt(e)$

Lemma 20 (Existence of initial transformation pairs in Graphs$_{TG}$). *Given a pair of direct transformations (t_1, t_2) in conflict, there is an initial transformation pair for (t_1, t_2), in the category* **Graphs**$_{TG}$.

Proof. Due to the Completeness Theorem for critical pairs [13] there is a critical pair $cp : H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ for (t_1, t_2). By the critical pair definition the matches m_1 and m_2 are jointly surjective. If $O = m_1(L_1) \cap m_2(L_2)$ contained some graph elements preserved by both rules, cp is tried to be unfolded at these nodes and edges, i.e., a critical pair cp' is searched which does not map these elements to the same one in O. This is always possible for edges. It is also possible for nodes if they do not have incident edges to be deleted, also being in O. The dangling edge condition for unfolded nodes cannot be violated after unfolding if it was not violated before since the same amount or fewer incident edges per unfolded node arise. The identification condition is also fulfilled after unfolding if it was fulfilled before since fewer elements are identified afterwards. Unfolding a critical pair as much as possible in this way yields the transformation pair $itp : H_1^I \overset{p_1,m_1^I}{\Longleftarrow} G^I \overset{p_2,m_2^I}{\Longrightarrow} H_2^I$ where the only preserved elements in $m_1^I(L_1) \cap m_2^I(L_2)$ are boundary nodes with incident edges to be deleted. A further unfolding is not possible since we would not find a corresponding extension diagram. Remember that an extension morphism can only unfold elements that are commonly preserved by both transformations. Preserved nodes with at least one incident edge to be deleted being overlapped as well cannot be unfolded since this edge would have to be unfolded as well.

We have to show now that itp is an initial transformation pair for (t_1, t_2). It is obvious that itp can be embedded into cp, which can be embedded into (t_1, t_2) via extension diagrams and extension morphisms. Given any other transformation pair tp that can be embedded into (t_1, t_2), tp may differ from (t_1, t_2) just by having fewer commonly preserved elements or by unfolding of preserved elements. itp can be embedded into tp since it contains the minimal number of preserved elements and the minimal overlap of preserved elements. The uniqueness of the corresponding extension diagrams and morphism follows from the construction of itp, i.e., the construction of critical pairs uses a unique $\mathcal{E}'$-$\mathcal{M}$ pair factorization and the unfolding is canonical. □

As Lemma 20 suggests an initial conflict is a transformation pair in conflict with minimal context and maximal unfolding of preserved elements.

Theorem 21 (Initial conflict in Graphs$_{TG}$). *In the category* **Graphs**$_{TG}$*, a transformation pair* $ic : H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ *is an initial conflict iff* ic *has the following properties:*

1. *Minimal context:* m_1 *and* m_2 *are jointly surjective.*
2. *At least one element in delete-use conflict:* $m_1(L_1) \cap m_2(L_2) \not\subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$.
3. *Overlap in deletion graphs only:* $m_1(L_1) \cap m_2(L_2) \subseteq (m_1(c_1(C_1) \cap m_2(L_2)) \cup (m_1(L_1) \cap m_2(c_2(C_2)))$ *with* $c_1 : C_1 \to L_1$ *and* $c_2 : C_2 \to L_2$ *being defined as in Definition 6.*
4. *No isolated boundary node in overlap graph:* m_1, m_2 *as given in Definition 19.*

Proof. Given the initial conflict ic, we show that it fulfills items 1. to 4.: According to Definition 11, ic is isomorphic to the initial transformation pair for ic. This transformation pair can be constructed as in Lemma 20 and it is unique due to Lemma 8. Hence, we follow this construction and deduce the properties ic has to satisfy. The first step yields a critical pair which fulfills items 1. and 2. as shown in e.g. [13]. After the maximal unfolding of this critical pair, items 1. and 2. are still fulfilled since unfolding does not add context (item 1.) and does not unfold elements to be deleted (item 2.). In addition, items 3. and 4. are fulfilled.

Given the transformation pair ic fulfilling items 1. to 4., we show that ic is an initial conflict. When constructing the initial transformation pair for ic according to Lemma 20, a pair of isomorphic transformations to ic would be constructed since items 1. and 2. lead to an isomorphic critical pair and items 3. and 4. ensure that no more unfoldings can be made. □

The theorem above shows in particular that each initial conflict is an essential critical pair satisfying properties 1. to 3. Example 18 shows, however, that not each essential critical pair is an initial conflict.

6 Initial Dependencies

To reason about initial dependencies for a rule pair (p_1, p_2), we consider the dual concepts and results that we get when inverting the left transformation of a conflicting pair. This means that we check if $G \overset{p_1^{-1},m'_1}{\Longleftarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$ is parallel dependent, which is equivalent to the sequence $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$ being sequentially dependent. Rule p^{-1} is the inverse of rule p obtained by exchanging morphisms l and r (Definition 1). This exchange is possible since a transformation is symmetrically defined by two pushouts. They ensure in particular that morphisms $m : L \to G$ as well as $m' : R \to H$ fulfill the gluing condition.

Initial transformation sequences and dependencies can then be defined analogously to Definitions 7 and 11. Initial dependencies show dependencies in such a way that there is no other dependency that can be extended to it. In the category **Graphs**$_{TG}$ this means that each initial dependency is characterized by a jointly surjective pair of morphisms, consisting of the co-match of p_1 and match

of p_2, which overlap in at least one graph element produced by p_1 and used by p_2, the overlap consists of produced elements and boundary nodes only, and none of these boundary nodes is isolated. Results presented for conflicts above can be formulated and proven for dependencies in an analogous way.

7 Related Work and Conclusion

The critical pair analysis (CPA) has developed into the standard technique for detecting potential conflicts and dependencies in graph transformation systems [3] and more generally, of $\mathcal{M}$-adhesive systems [4,12]. We introduced the notions of initial conflict and dependency as a new yardstick to present potential conflicts and dependencies in graph transformation systems in a minimal way. These notions are defined in a purely category-theoretical way within the framework of $\mathcal{M}$-adhesive systems. While each initial conflict is a critical pair, it turns out that this is not true vice versa. Actually, our running example shows that, given a rule pair, the set of initial conflicts can be considerably smaller than the set of critical pairs and even than the set of essential critical pairs. We characterized initial conflicts in graph transformation systems as transformation pairs with minimal context and maximal unfolding of preserved graph elements.

The CPA is offered by the graph transformation tools AGG [10] and Verigraph [11] and the graph-based model transformation tool Henshin [15]. All of them provide the user with a set of (essential) critical pairs for each pair of rules as analysis result at design time. Since initial conflicts turned out to be a real subset of essential critical pairs, we intend to optimize the conflict and dependency analysis (CDA) in AGG and Henshin by prioritizing the initial ones. We also intend to investigate how far we can speed up this analysis by our new results. It would be interesting to come up with some results on the amount of reduction of critical pairs, maybe w.r.t. a particular characterization of the rules.

Novel conflict and dependency concepts at several granularity levels are presented in [16]. It is up to future work to investigate the relation of this work with initial conflicts and dependencies. The CPA is not only available for plain rules but also for rules with application conditions (ACs) [17]. Due to their definition in a purely category-theoretical form, we are quite confident that the theory for initial conflicts and dependencies can be extended to rules with ACs.

Acknowledgements. Many thanks to Leila Ribeiro and Jonas Santos Bezerra for providing us with support to CPA of our running example in Verigraph [11].

This work was partially funded by the German Research Foundation, Priority Program SPP 1593 “Design for Future – Managed Software Evolution”. This research was partially supported by the research project Visual Privacy Management in User Centric Open Environments (supported by the EU’s Horizon 2020 programme, Proposal number: 653642).

References

1. Huet, G.: Confluent reductions: abstract properties and applications to term rewriting systems: abstract properties and applications to term rewriting systems. J. ACM (JACM) **27**(4), 797–821 (1980)
2. Plump, D.: Critical pairs in term graph rewriting. In: Prívara, I., Rovan, B., Ruzička, P. (eds.) MFCS 1994. LNCS, vol. 841, pp. 556–566. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58338-6_102
3. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45832-8_14
4. 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
5. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: 22rd International Conference on Software Engineering (ICSE), pp. 105–115. ACM (2002)
6. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_11
7. Baresi, L., Ehrig, K., Heckel, R.: Verification of model transformations: a case study with BPEL. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 183–199. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75336-0_12
8. Lambers, L.: Certifying rule-based models using graph transformation. Ph.D thesis. Berlin Institute of Technology (2010)
9. Lambers, L., Ehrig, H., Orejas, F.: Efficient conflict detection in graph transformation systems by essential critical pairs. Electr. Notes Theor. Comput. Sci. **211**, 17–26 (2008)
10. 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
11. Verigraph: Verigraph. https://github.com/Verites/verigraph
12. Ehrig, H., Golas, U., Hermann, F.: Categorical frameworks for graph transformation and HLR systems based on the DPO approach. Bull. EATCS **102**, 111–121 (2010)
13. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
14. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation I: basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations, pp. 163–245. World Scientific, Singapore (1997)

15. 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. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9. http://www.eclipse.org/henshin/
16. 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
17. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: $\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 2: embedding, critical pairs and local confluence. Fundam. Inform. **118**(1–2), 35–63 (2012)

Towards a Navigational Logic for Graphical Structures

Leen Lambers[1], Marisa Navarro[2], Fernando Orejas[3], and Elvira Pino[3(✉)]

[1] Hasso Plattner Institut, University of Potsdam, Potsdam, Germany
Leen.Lambers@hpi.de
[2] Universidad del País Vasco (UPV/EHU), San Sebastián, Spain
marisa.navarro@ehu.es
[3] Universitat Politècnica de Catalunya, Barcelona, Spain
{orejas,pino}@cs.upc.edu

Abstract. One of the main advantages of the Logic of Nested Conditions, defined by Habel and Pennemann, for reasoning about graphs, is its generality: this logic can be used in the framework of many classes of graphs and graphical structures. It is enough that the category of these structures satisfies certain basic conditions.

In a previous paper [14], we extended this logic to be able to deal with graph properties including paths, but this extension was only defined for the category of untyped directed graphs. In addition it seemed difficult to talk about paths abstractly, that is, independently of the given category of graphical structures. In this paper we approach this problem. In particular, given an arbitrary category of graphical structures, we assume that for every object of this category there is an associated edge relation that can be used to define a path relation. Moreover, we consider that edges have some kind of labels and paths can be specified by associating them to a set of label sequences. Then, after the presentation of that general framework, we show how it can be applied to several classes of graphs. Moreover, we present a set of sound inference rules for reasoning in the logic.

1 Introduction

Graphs and graphical structures play a very important role in most areas of computer science. For instance, they are used for modeling problems or systems (as done, e.g., with the UML or with other modeling formalisms). Or they are also used as structures to store data in many computer science areas. In particular, in the last few years, in the database area, graph databases are becoming relevant in practice and partially motivate our work. A consequence of this graph

This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R, TIN2013-46181-C2-2-R) and from the Basque Project GIU15/30, and grant UFI11/45.

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 124–141, 2018.
https://doi.org/10.1007/978-3-319-75396-6_7

ubiquity is that being able to express properties about graphical structures may be interesting in many areas of computer science.

We can use two kinds of approaches to describe graph properties. Obviously, we may use some standard logic, after encoding some graph concepts in the logic. For instance, this is the approach of Courcelle (e.g., [3]), who studied a graph logic defined in terms of first-order (or monadic second-order) logic. The second kind of approach is based on expressing graph properties in terms of formulas that include graphs (and graph morphisms). The most important example of this kind of approach is the *logic of nested graph conditions* (LNGC), introduced by Habel and Pennemann [9] proven to be equivalent to the first-order logic of graphs of Courcelle. A main advantage of LNGC is its genericity, since it can be used for any category of graphical structures, provided that this category enjoys certain properties. This is not the case of approaches like [3] where, for each class of graphical structures, we would need to define a different encoding.

A main problem of (first-order) graph logics is that it is not possible to express relevant properties like "there is a path from node n to n'", because they are not first-order. As a consequence, there have been a number of proposals that try to overcome this limitation by extending existing logics (like [7,10,20]). Along similar lines, in [14] we extended the work presented in [12], allowing us to state properties about paths in graphs and to reason about them. Unfortunately, the work in [14] applies only to untyped unattributed directed graphs. As a continuation, in this paper we show how to overcome this limitation, extending some of the ideas in [14] to deal with arbitrary categories of graphical structures. Moreover, we allow for a more expressive specification of paths, assuming that edges have some kinds of labels and specifying paths using language expressions over these labels. Since this new generic logic allows one to describe properties of paths in graphical structures, we have called it a *navigational logic*.

The paper is organized as follows. In Sect. 2 we present some examples for motivation. In Sect. 3 we introduce the basic elements to define our logic and in the Sect. 4 we see how these elements can be defined in some categories of graphs, implicitly showing that our logic can be used in these categories. In Sect. 5 we introduce the syntax and semantics of our logic, including some proof rules that are shown to be sound. Completeness is not studied, because in our framework we implicitly assume that paths are finite, which means that our inference rules can not be complete [22]. However we conjecture that our rules will be complete in a more complex framework, where graphs may be infinite. Finally, in Sect. 6 we present some related and future work.

2 Motivation

In this section, we present and motivate the basic concepts required to introduce our navigational logic, that is, patterns with paths, and graph properties. In order to give some intuition and motivation, in Subsect. 2.1, we consider a toy example consisting of a network of airports connected by airline companies that operate between them, that is, a graph where nodes are airports and edges are direct

flights from an airport to another. The example follows the framework presented in [14]. Then, in Subsect. 2.2 an example of a social network is introduced to motivate the extension of that framework, including labels in paths and edges, allowing us to specify the form of paths.

2.1 A First Navigational Logic Example

The graph in Fig. 1 represents a network with four airports: Barcelona (BCN), Paris (CDG), New York (JFK) and Los Angeles (LAX) and the six directed edges represent the existing direct flights between these airports. In this scenario, a path (i.e. a concatenation of one or more edges) represents a connection from an airport to another by a sequence of, at least, one direct flight. For instance, BCN is connected to CDG and JFK by direct flights, whereas it is required to concatenate at least two flights to arrive to LAX from BCN. To express basic properties we use patterns, which are graphs extended with a kind of arrows that represent "paths" between nodes. For instance, in Fig. 2, we have two patterns that are present in the airport network in Fig. 1: The first pattern represents a connection from BCN to LAX, and, the second one a direct flight from BCN to CDG followed by a connection from CDG to LAX.

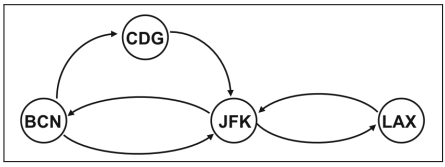


Fig. 1. A graph of connected airports

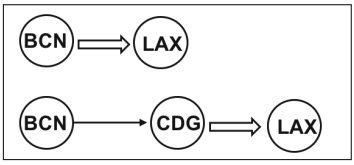


Fig. 2. Two connection patterns

Imagine that we want to state that there should be a connection from BCN to LAX with a stopover either in Paris or in New York and, moreover, that, in the former case, there should be a direct flight from BCN to CDG, whereas in the latter, the flight from JFK to LAX must be direct. The first graph condition in Fig. 3 states those requirements.

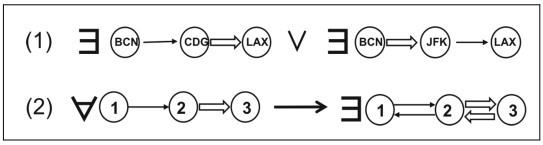


Fig. 3. Properties on airports networks

The second graph condition in Fig. 3 states that if there is a connection from an airport 1 to an airport 3 with a first stopover at an airport 2, it must be

possible to go back from 3 to 1 with a similar flight plan that also stops at 2 but as the last stopover. Our network in Fig. 1 does not satisfy this requirement, since there is a connection from BCN to LAX with a first stopover in CDG but there are no direct flights from CDG to BCN.

2.2 Path Expressions

In the framework described in the previous subsection [14], we can specify the existence of a path between two nodes, but we cannot provide any description of such path. For instance, suppose that edges are labelled with the name of the airline company operating that flight. In the described framework it is impossible to specify that there is a connection between BCN and LAX consisting of flights from the same company.

A simple way of dealing with this situation is to label paths with language expressions over an alphabet of edge labels. For instance, in most approaches (e.g., [1,2,4,13,23]), paths are labeled by regular expressions. The idea is that, if a pattern includes a path from node 1 to node 2 labelled with a language expression denoting a language L, then if a graph G includes that pattern, it must include some sequence of edges labelled with $l_1, \ldots, l_k$, such that $l_1 \cdot l_2 \cdot \ldots \cdot l_k \in L$.

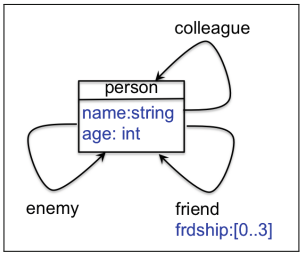


Fig. 4. A social network type graph

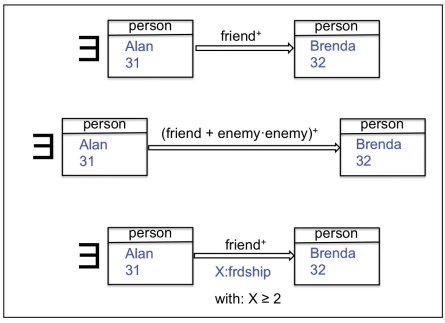


Fig. 5. Patterns of labeled connections

For example, in Fig. 4 we depict the type graph of a social network including nodes of type person and edges of type friend, enemy, and colleague. Then, in Fig. 5 we depict some conditions over this type graph. In the first two conditions we (implicitly) assume that edges are labeled with the name of their types. So, the first condition describes the existence of a path, consisting only of a sequence of edges of type friend between nodes having Alan and Brenda as their name attributes, while the second condition describes the existence of a path between Alan and Brenda, consisting of edges of type friend or two consecutive edges of type enemy. In the third condition, we implicitly assume that edges are labeled not only with types, but also with the values of their attributes and it describes the existence of a path between Alan and Brenda consisting of edges of type friend, whose friendship attribute is greater or equal to 2.

3 Patterns with Paths for Arbitrary Graphical Structures

In this paper our aim is to define a general framework that will allow us to express properties about arbitrary graphical structures and their paths, and to reason about them. A main problem is how to cope with this level of generality. In particular, given a specific class of graphs, like directed graphs (as in [14]) the notion of a path is clear. However, when working with an arbitrary category, that is supposed to represent any kind of graphical structure (e.g. graphs, Petri Nets or automata), we need some abstract notion of path that can accommodate the notion of path that we would have in each of these categories.

In principle, a path is a sequence of edges, but not all kinds of categories that we may consider have a proper notion of edge, although they may have something that we may consider to be similar. For instance, in a Petri Net we may consider that transitions play the role of edges[1]. So our first step is to consider that we can associate to every category of graphical structures an associated category including an explicit edge relation. Then, it will be simple to define paths in these categories.

We assume that edges are labelled, so that we can use these labels to describe paths, as seen above. Moreover, we will assume that all edges in our graphical structures are defined over a universal set of nodes and a universal set of labels.

Definition 1 (Edges). *Given a set of labels Σ and a set of nodes V, the set of all possible Σ-labeled edges over V is* $\mathtt{Edges}_{\Sigma,V} = V \times \Sigma \times V$.

Definition 2 (Edge-Labelled Structures). *Given a set of labels Σ and a set of nodes V, and given a category of graphical structures* $\underline{\mathtt{Struct}}$ *with pushouts and initial objects, we say that* $\underline{\mathtt{StructEdges}}_{\Sigma,V}$ *is its associated category of edge-labeled structures over V and Σ, EL-structures in short, if the following conditions hold:*

1. *The objects in* $\underline{\mathtt{StructEdges}}_{\Sigma,V}$ *are pairs (S, E), where S is an object in* $\underline{\mathtt{Struct}}$ *and E is a set of Σ-labeled edges over V.*
2. *A morphism $f : (S, E) \to (S', E')$ in* $\underline{\mathtt{StructEdges}}_{\Sigma,V}$*, consists of functions $f = (f_s, f_v, f_e)$ such that*
   - *$f_s : S \to S'$ is a morphism in* $\underline{\mathtt{Struct}}$*, and*
   - *$f_v : V \to V$ and $f_e : E \to E'$ satisfy for every $\langle n, l, n' \rangle \in E$ that $f_e(\langle n, l, n' \rangle) = \langle f_v(n), l, f_v(n') \rangle$.*
3. $\underline{\mathtt{Struct}}$ *and* $\underline{\mathtt{StructEdges}}_{\Sigma,V}$ *are isomorphic. Specifically, there must exist an isomorphism $\psi : \underline{\mathtt{Struct}} \to \underline{\mathtt{StructEdges}}_{\Sigma,V}$.*

We will write just $\mathtt{Edges}$ *and* $\underline{\mathtt{StructEdges}}$ *whenever Σ and V are clear.*

[1] But in Petri Nets we may also consider that both places and transitions play the role of the nodes in a graph and that the edges in a Petri Net are the arrows in the graphical representation of the net going from places to transitions or from transitions to places.

Intuitively, the idea is that for each object S of an arbitrary category of graphical structures, we can associate a set of labelled edges that we assume that are implicit in S. Notice that if S is a graph this does not mean that it is a labelled graph. It only means that we can associate some kind of labels to its edges. For instance, if S is a typed graph, then we may consider that edges are labeled by their types. Similarly, as said above, if S is not a graph, like in the case of Petri Nets, this does not mean that S must include a proper notion of edge, but that we may consider that its edges are some of its elements.

Now, before defining the notion of pattern, we must first define the notion of path expressions, that is, the specification of a set of paths between two nodes. Moreover, we also define the notion of closure of a set of path expressions under composition and decomposition. Intuitively, a path is in the closure of a set of path expressions if its existence is a consequence of these expressions.

Definition 3 (Paths, Path Expressions and their Closure). *We define the set of path expressions over* Σ, V, $\texttt{PathExpr}_{\Sigma,V} = V \times 2^{\Sigma^*} \times V$.[2]

A path *specified by a path expression* $pe = \langle n, L, m\rangle$, *is any triple* $\langle n, s, m\rangle \in V \times \Sigma^+ \times V$ *such that* $s \in L$. *Then,* $paths(pe)$ *denotes the set of paths specified by* pe.

If $R \subseteq \texttt{PathExpr}_{\Sigma,V}$ *is a set of path expressions, then the* closure *of* R, *written* R^+, *is the set of path expressions defined inductively:*

1. $R \subseteq R^+$.
2. Empty paths*: For every node* n, $\langle n, \epsilon, n\rangle \in R^+$.
3. Composition*: If* $\langle n, L_1, m\rangle, \langle m, L_2, n'\rangle \in R^+$ *then* $\langle n, L_1L_2, n'\rangle \in R^+$.

Now, we can define what patterns are:

Definition 4 (Patterns). *Given a category* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *of EL-structures, its associated category of patterns,* $\underline{\texttt{StructPatterns}}_{\Sigma,V}$, *is defined as follows:*

1. *Objects are triples* $P = (S, E, PE)$ *where*
   - (S, E) *is in* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *and*
   - $PE \subseteq \texttt{PathExpr}_{\Sigma,V}$.
2. *A* pattern morphism $f : (S, E, PE) \to (S', E', PE')$, *is a morphism* $f : (S, E) \to (S', E')$ *in* $\underline{\texttt{StructEdges}}_{\Sigma,V}$ *such that, for every* $\langle n, L, m\rangle \in PE$, *there is a path expression* $\langle f_v(n), L', f_v(m)\rangle \in (E' \cup PE')^+$ *with* $L' \subseteq L$.

We will write just PathExpr *and* $\underline{\texttt{StructPatterns}}$ *whenever* Σ, *and* V *are clear.*

Notice that a Σ-labeled edge $\langle n, l, n'\rangle \in \texttt{Edges}$ can be considered a special kind of *unit* path expression $\langle n, \{l\}, n'\rangle$. As we may see in the definition above, even if there is an abuse of notation, given a set of edges E we will consider that E also denotes its associated set of unit path expressions.

[2] Even if we may consider that empty paths are not really paths, assuming that every node is connected to itself through an empty path provides some technical simplifications.

Intuitively, a structure S can be considered a trivial pattern that is always present in S itself. However, technically, following the above definition S is not a pattern, but we can define a pattern, $Pattern(S)$, that intuitively represents S. Given S and its associated set of edges E, $Pattern(S) = (S, E, E^+)$, i.e. the path expressions in $Pattern(S)$ are precisely the paths defined by the edges in E. Conversely, any pattern (S, E, PE) where $PE = E^+$ can be considered equivalent to the structure S[3]. As a consequence, even if it is an abuse of notation, we will identify structures with their associated patterns. For instance, if we write that there is a pattern morphism $f : P \to S$, we really mean $f : P \to (S, E, E^+)$.

As we will see in Sect. 5.3, an (important) inference rule for reasoning in our logic is the unfolding rule that roughly says that if a pattern in a given condition includes the path expression $\langle n, L, n' \rangle$, then we may replace this pattern by another one that includes some of its possible decompositions. For instance, if $\langle n, a(c^*), n' \rangle$ is a path expression in P, we should be able to infer that the structures that satisfy the pattern either have an edge $\langle n, a, n' \rangle$, or an edge $\langle n, a, n_0 \rangle$ followed by a path from n_0 to n' consisting of edges labelled by c. More precisely, from the condition $\exists P$ we should be able to infer $\exists P_1 \vee \exists P_2$, with $P_1 = P + \{\langle n, a, n' \rangle\}$ and $P_2 = P + \{\langle n, a, n_0 \rangle, \langle n_0, c^+, n' \rangle\}$, where n_0 is a node that is not present in P and $P + s$ denotes the pattern obtained adding to P the paths and edges in the set s. The problem is how can we define formally P_1 and P_2. If $P = (S, E, PE)$, it would be wrong to define $P_1 = (S, E \cup \{\langle n, a, n' \rangle\}, PE)$, because E is the set of edges (implicitly) included in S, and $E \cup \{\langle n, a, n' \rangle\}$ can not also be the set of edges of S (unless S already included $\langle n, a, n' \rangle$, which in general will not be the case). Instead, we will assume that every specific framework is equipped with a procedure to define the structure S' that includes S and whose set of edges is $E \cup \{\langle n, a, n' \rangle\}$. This procedure is the mapping called `Unfold` in Definition 6, which actually does not return S', but the morphism $u : S \to S'$ that represents the inclusion of S in S'.

Before defining the unfolding construction, we define the decompositions that are associated to these unfoldings. First, a subdecomposition sd of a path expression $pe = \langle n, L, n' \rangle$ can be seen as a refinement of pe, in the sense that we may consider that sd defines a path expression $\langle n, L', n' \rangle$, where $L' \subseteq L$. For instance, in the example above $\{\langle n, a, n' \rangle\}$ and $\{\langle n, a, n_0 \rangle, \langle n_0, c^+, n' \rangle\}$ are subdecompositions of $\langle n, L, n' \rangle$. In the former case $L' = \{a\}$ and in the later case $L' = \{ac, acc, accc, \dots\}$. Then, a decomposition of $\langle n, L, n' \rangle$ is a set of subdecompositions such that L coincides with the union of the languages associated to its subdecompositions. For, instance $\{\langle n, a, n' \rangle\}$ and $\{\langle n, a, n_0 \rangle, \langle n_0, c^+, n' \rangle\}$ are a decomposition of $\langle n, L, n' \rangle$, since $ac^* \equiv a|ac^+$.

Definition 5 (Path Expression Decomposition). *If $pe = \langle n, L, n' \rangle$ is a path expression, a* subdecomposition *sd of pe is a pair (L', s), where $L' \subseteq L$ and s is a set of edges and path expressions such that one of the following conditions holds:*

1. $s = \{\langle n, l, n' \rangle\}$ and $L' = \{l\}$.

[3] That is `Struct` is embedded in `Patterns` via the functor *Pattern*.

2. $s = \{\langle n, L_1, n_1\rangle, \langle n_1, l, n_2\rangle, \langle n_2, L_2, n'\rangle\}$ *and* $L' = L_1\{l\}L_2$*, with* L_1,$L_2 \subseteq \Sigma^*$[4].

A decomposition d *of* $pe = \langle n, L, n'\rangle$ *is a finite set of subdecompositions,* $d = \{sd_1, \ldots, sd_k\}$*, with* $sd_i = (L_i, s_i)$*, for* $i \in \{1, \ldots, k\}$*, such that* $(L_1 \cup \cdots \cup L_k) = L$*.*

Definition 6 (Unfolding Morphisms). *We say that the category* StructPatterns *has* unfolding morphisms *if it is equipped with a function* `Unfold` *that given a pattern* $P = (S, E, PE)$*, a path expression* $pe = \langle n, L, n'\rangle \in PE$*, and a subdecomposition* $sd = (L', s)$ *of* pe*, it returns a morphism* $u : P \to P'$*, where* $P' = (S', E', PE')$*, such that:*

1. $E' = E \cup \{\langle n_1, l, n_2\rangle\}$ *if* $\langle n_1, l, n_2\rangle \in s$*, with* $l \in \Sigma$*, and*
2. $PE' = PE \cup \{\langle n_1, L_i, n_2\rangle | \langle n_1, L_i, n_2\rangle \in s\}$*.*
3. *For every morphism* $f : P \to P_0$*, with* $P_0 = (S_0, E_0, PE_0)$*, if there is a path expression* $\langle f(n), L_0, f(n')\rangle \in PE_0$*, with* $E' \subseteq E_0$ *and* $PE' \subseteq PE_0$*, then there is a morphism* $h : P' \to P_0$*, such that* $f = h \circ u$*.*

That is, if $u : P \to P' = \texttt{Unfold}(P, \langle n, L, n'\rangle, sd)$ then P' contains an unfolding of $\langle n, L, n'\rangle$ in P, built by adding new edges and paths to P. Notice that the component $u_s : S \to S'$ of every $u = \texttt{Unfold}(P, \langle n, L, n'\rangle, sd)$ must build the proper unfolded version S' of S so that the isomorphism ψ : Struct $\to$ StructEdges is preserved.

From now on, we assume that our categories of patterns have unfolding morphisms.

4 Instantiation to Different Classes of Graphs

In this section we present how our general framework works in the context of some classes of graphs. We assume that the reader knows the (more or less) standard definitions in the literature of these classes of graphs (see, e.g., [5]). For simplicity, we assume that the languages used to label paths are defined by means of a regular expression. It should be clear that the three categories of graphs have pushouts and an initial object (the empty graph). Moreover, it is trivial to define `Unfold` for the three classes of graphs. In particular, given a pattern $P = (G, E, PE)$ where G is a graph of any of the three classes considered below, $\texttt{Unfold}(P, pe, sd)$ would return the inclusion $(G, E, PE) \hookrightarrow (G', E', PE')$, where G' is the graph obtained after adding to G the edges and new nodes in sd, E' is E plus these edges and PE' is PE plus the path expressions in sd. In all cases, V will be the class of all nodes of the given class of graphs.

[4] Notice that L_1 or L_2 may just consist of the empty string, in which case $n = n_1$ or $n' = n_2$, respectively.

4.1 Untyped Directed Graphs

The category of untyped directed graphs can be seen as an instance of our general framework, where:

- Σ is a set with a single label l.
- The isomorphism ψ between this category and `StructEdges` that defines how a graph G is seen as an object in `StructEdges` is defined as follows:
  - For every graph $G = (V_G, E_G, s_G, t_G)$, $\psi(G)$ is the EL-graph (G, E) where E is the set implicitly defined by E_G, that is, $E = \{\langle n, l, m\rangle \mid n, m \in V_G$, such that there exists $e \in E_G$ with $s_G(e) = n$ and $t_G(e) = m\}$.
  - For every morphism $f_s : G \to G'$, the corresponding EL-morphism $\psi(f_s)$ is defined as (f_s, f_v, f_e) with $f_e(\langle n, l, m\rangle) = \langle f_v(n), l, f_v(m)\rangle$ for each $\langle n, l, m\rangle \in E$.

In this context, the only path expressions $PE \subseteq$ `PathExpr` are of the form $\langle n, L, m\rangle$, where L is a regular expression over the single label l. In particular, $\langle n, l^+, m\rangle$ would mean that there is a path from node n to node m formed by a non specified number of edges. For instance, the patterns in Fig. 2 could be seen as patterns in our framework if we consider that the paths depicted in those patterns are labeled with l^+.

4.2 Typed Graphs

To see that the category of typed graphs over a given type graph TG is an instance of our general framework, we assume that types have unique names.

- $\Sigma = \{t_1, t_2, \dots\}$ is a set of names for the types in TG.
- The isomorphism ψ between this category and `StructEdges`, that defines how a typed graph $(G, type_G)$ is seen as an object in `StructEdges`, is defined as follows:
  - For every typed graph $(G, type_G)$, with $G = (V_G, E_G, s_G, t_G)$, $\psi((G, type_G))$ is the EL-graph (G, E) where $E = \{\langle n, t, m\rangle \mid n, m \in V_G$, and $t = type_G(e)$ for some edge $e \in E_G$ with $s_G(e) = n$ and $t_G(e) = m\}$.
  - For every morphism f, the corresponding EL-morphism $\psi(f)$ is defined as (f_s, f_v, f_e) with $f_e(\langle n, t, m\rangle) = \langle f_v(n), t, f_v(m)\rangle$ for each $\langle n, t, m\rangle$ in E.

For instance, the first two conditions in Fig. 5 include examples of patterns for the given type graph.

We may notice that a different category `StructEdges` (and consequently a different category `StructPatterns`) can be associated to typed graphs, if we consider that an edge e is labeled not by the name of its type, but by the pair (t_1, t_2) where t_1 and t_2 are the names of the types of the source and target nodes of e, respectively.

4.3 Attributed Graphs

Roughly, an attributed graph can be seen as some kind of labelled graph whose labels (the values of attributes) consist of values from a given data domain. There are several approaches to formalize this kind of graphs. In this paper we use the notion of *symbolic graph* ([15,16]), because it is the most adequate approach to define patterns that include conditions on the attribute values. Symbolic graphs are defined using the notion of E-graphs, introduced in [5] as a first step to define attributed graphs. Intuitively, an E-graph is a kind of labelled graph, where both nodes and edges may be decorated with labels from a given set E. Being more precise, a symbolic graph G consists of an E-graph EG_G whose labels are seen as variables that represent the values of the given attributes, together with a formula Φ_G over these variables, used to constrain the possible values of the associated attributes. In general, a symbolic graph G can be considered a specification of a class of attributed graphs, since every model of Φ_G can be considered a graph specified by G. However, we can identify attributed graphs with *grounded symbolic graphs*, i.e. symbolic graphs G, where Φ_G is satisfied by just one graph (up to isomorphism).[5]

Then the category of attributed graphs (grounded symbolic graphs) can be seen as an instance of our general framework, where:

- Labels in Σ consist of the types of the edges together with their attributes and the variables associated to these attributes, i.e. labels are tuples $\langle t, x_1 \colon att_1, \ldots, x_k \colon att_k \rangle$, where $att_1, \ldots, att_k$ are the attributes of type t and $x_1, \ldots, x_k$ are their associated variables.
- The isomorphism Ψ between this category and `StructEdges` is defined as follows.
  - For every symbolic graph G, $\Psi(G)$ is the EL-graph (G, E) where $E = \{\langle n, \langle t, x_1 \colon att_1, \ldots, x_k \colon att_k \rangle, m \rangle \mid$ there is an edge from n to m of type t with attributes $att_1, \ldots, att_k$ and $x_1, \ldots, x_k$ are their associated variables$\}$.
  - For every attributed morphism $f_s : G \to G'$, the corresponding EL-morphism $\Psi(f_s)$ is defined as (f_s, f_v, f_e) with $f_e(\langle n, l, m \rangle) = \langle f_v(n), l, f_v(m) \rangle$ for each label l and each $\langle n, l, m \rangle$ in E.

In this case, if we want to put conditions on paths, as in the third pattern in Fig. 5, a path expression pe could be a triple $pe = \langle n, (exp, \Phi_{pe}), m \rangle$, where exp could be a regular expression over labels of the form $\langle t, x_1 \colon att_1, \ldots, x_k \colon att_k \rangle$ and Φ_{pe} would be a formula on the variables in exp. In this case, the third pattern in Fig. 5, the path from Alan to Brenda would be labelled with the regular expression $\langle friend, X \colon friendship \rangle^+$ together with the condition $X > 2$.

[5] In particular, we may consider that in a grounded symbolic graph G we have $\Phi_G \equiv (x_1 = v_1 \wedge \cdots \wedge x_k = v_k)$, for some values $v_1, \ldots, v_k$.

5 Reasoning About Navigational Properties

In this section we introduce in detail our logic. In the first subsection, we define its syntax and semantics. In the next one we show some properties that are used in the third subsection to define our inference rules and to show their soundness.

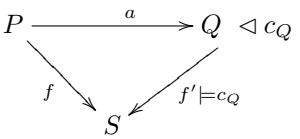
5.1 Nested Pattern Conditions, Models and Satisfaction

For our convenience, we express our properties using a nested notation [9] and avoiding the use of universal quantifiers.

Definition 7 (Conditions over Patterns, Satisfaction of Conditions). *Given a pattern P in* `StructPatterns`*, a* condition *over P is defined inductively as follows:*

- `true` *is a condition over P. We say that* `true` *has nesting level* 0.
- *For every morphism $a : P \to Q$ in* `StructPatterns`*, and every condition c_Q over Q with nesting level $j \geq 0$, $\exists(a, c_Q)$ is a condition over P, called literal, with nesting level $j + 1$.*
- *If c_P is a condition over P with nesting level j, then $\neg c_P$ is a condition over P with nesting level j.*
- *If c_P and c'_P are conditions over P with nesting level j and j', respectively, then $c_P \wedge c'_P$ is a condition over P with nesting level $max(j, j')$.*

Given a structure S, we inductively define when the pattern morphism $f : P \to S$ satisfies *a condition c_P over P, denoted $f \models c_P$:*



- $f \models$ `true`.
- $f \models \exists(a, c_Q)$ *if there exists $f' : Q \to S$ such that $f' \circ a = f$ and $f' \models c_Q$.*
- $f \models \neg c_P$ *if* $f \not\models c_P$
- $f \models c_P \wedge c'_P$ *if* $f \models c_P$ *and* $f \models c'_P$.

If c_P is a condition over P, we also say that P is the context *of c_P.*

Definition 8 (Navigational Logic: Syntax and Semantics). *The language of our Navigational Logic (NL) consists of all conditions over the initial pattern, $\emptyset$, in the category of patterns. Given a literal $\exists(a : \emptyset \to P, c_P)$ of NL, we also denote it by $\exists(P, c_P)$. A structure S satisfies a property c of NL if the unique morphism $i : \emptyset \to S$ satisfies c.*

5.2 Transformation by Lift and Unfolding

In this section we introduce some constructions that are used in our inference rules. The first one is the shift construction (introduced in [18,19]) that allows us to translate conditions along morphisms.

Lemma 1 (Shift of Conditions over Morphisms). *Let Shift be a transformation of conditions inductively defined as follows:*

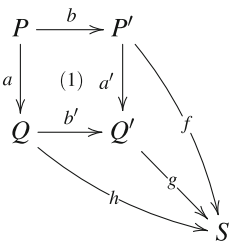
$$\begin{array}{ccc} P & \xrightarrow{b} & P' \\ {\scriptstyle a}\downarrow & (1) & \downarrow{\scriptstyle a'} \\ Q & \xrightarrow[b']{} & Q' \\ \triangle & & \triangle \\ c_Q & & c_{Q'} \end{array}$$

- $Shift(b, \texttt{true}) = \texttt{true}$.
- $Shift(b, \exists(a, c_Q)) = \exists(a', c_{Q'})$ *with* $c_{Q'} = Shift(b', c_Q)$ *such that (1) is a pushout.*
- $Shift(b, \neg c_P) = \neg Shift(b, c_P)$
- $Shift(b, c_P \wedge c'_P) = Shift(b, c_P) \wedge Shift(b, c'_P)$.

Then, for each condition c_P over P and each morphism $b : P \to P'$, $c_{P'} = Shift(b, c_P)$ is a condition over P' with smaller or equal nesting level, such that for each morphism $f : P' \to S$ we have that $f \models Shift(b, c_P) \Leftrightarrow f \circ b \models c_P$.

Proof. The proof uses double induction on the structure and the nesting level of conditions. The base case is trivial since $Shift(b, \texttt{true}) = \texttt{true}$, so they have the same nesting level $j = 0$, and every morphism satisfies `true`.

If c_P is not `true`, we proceed by induction on the nesting level of conditions. The base case is proven. Let c_P be $\exists(a, c_Q)$ of nesting level $j + 1$ and suppose there is a morphism $f : P' \to S$ such that $f \models Shift(b, \exists(a, c_Q))$. That is, $f \models \exists(a', Shift(b', c_Q))$, according to the definition and diagram (1) below. This means there exists a morphism $g : Q' \to S$ such that $g \models Shift(b', c_Q)$ and $f = g \circ a'$. Then, since (1) is a pushout, we know that $f \circ b = g \circ a' \circ b = g \circ b' \circ a$ and, by induction, we have that $g \circ b' \models c_Q$. Therefore, $f \circ b \models c_P$.



Conversely, if $f \circ b \models c_P$ there exists $h : Q \to S$ such that $f \circ b = h \circ a$ and $h \models c_Q$. By the universal property of pushouts, there exists $g : Q' \to S$ such that $f = g \circ a'$ and $h = g \circ b'$ and, by induction, $g \models Shift(b', c_Q)$. Hence, $f \models Shift(b, \exists(a, c_Q))$. In addition, $\exists(a', Shift(b', c_Q))$ has nesting level smaller or equal to $j+1$ since, again as a consequence of the induction hypothesis $Shift(b', c_Q)$ has nesting level smaller or equal to j.

The rest of the cases easily follow from the induction hypothesis and the satisfaction and nesting level definitions. ■

In [18,19], it is proved that, given two literals ℓ_1 and ℓ_2, a new literal ℓ_3 can be built (pushing ℓ_2 inside ℓ_1) that is equivalent to the conjunction of ℓ_1 and ℓ_2. Again, the following lemma is our version of that result:

Lemma 2 (Lift of Literals). *Let $\ell_1 = \exists(a_1, c_1)$ and ℓ_2 be literals with morphisms $a_i : P \to Q_i$, for $i = 1, 2$. We define the lift of literals as follows:*

$$Lift(\exists(a_1, c_1), \ell_2) = \exists(a_1, c_1 \wedge Shift(a_1, \ell_2))$$

Then, $f \models \ell_1 \wedge \ell_2$ if, and only if, $f \models Lift(\ell_1, \ell_2)$.

Proof. Assume $f : P \to S$ such that $f \models \exists(a_1, c_1 \wedge \mathit{Shift}(a_1, \ell_2))$. That is, there exists a morphism $g : Q_1 \to S$ such that $f = g \circ a_1$ and $g \models c_1 \wedge \mathit{Shift}(a_1, \ell_2)$. Then, this is equivalent to $f \models \ell_1$ and $f \models \ell_2$, since by Lemma 1 we have that $g \circ a_1 \models \ell_2$. ■

Note that when pushing ℓ_2 inside ℓ_1, the literal ℓ_2 can be positive or negative. But we will also need a special way of pushing a negative literal ℓ_2 inside a positive one ℓ_1 under some conditions, as shown in next lemma. In this case, the literal resulting from the lifting is just a consequence of the conjunction of ℓ_1 and ℓ_2.

Lemma 3 (Partial Lift of Literals). *Let $\ell_1 = \exists(a_1 : P \to Q_1, c_1)$ and $\ell_2 = \neg\exists(a_2 : P \to Q_2, c_2)$ such that there exists a morphism $g : Q_2 \to Q_1$ satisfying $a_1 = g \circ a_2$. We define the partial lift of literals as follows:*

$$\mathit{PLift}(\exists(a_1, c_1), \ell_2) = \exists(a_1, c_1 \wedge \mathit{Shift}(g, \neg c_2))$$

Then, $f \models \ell_1 \wedge \ell_2$ implies $f \models \mathit{PLift}(\ell_1, \ell_2)$.

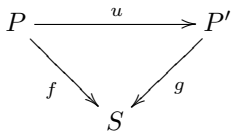
Proof. On the one hand, since $f \models \ell_1$, there exists a morphism $h_1 : Q_1 \to S$ such that $f = h_1 \circ a_1 = h_1 \circ g \circ a_2$, and $h_1 \models c_1$. On the other hand, since $f \models \ell_2$, it cannot exist a morphism $h_2 : Q_2 \to S$ satisfying both conditions $f = h_2 \circ a_2$, and $h_2 \models c_2$. Now, we consider the morphism $h_1 \circ g : Q_2 \to S$, which satisfies the first condition. Then necessarily $h_1 \circ g \models \neg c_2$ which implies $h_1 \models \mathit{Shift}(g, \neg c_2)$ by Lemma 1. Since $h_1 \models c_1 \wedge \mathit{Shift}(g, \neg c_2)$ we conclude that $f \models \mathit{PLift}(\ell_1, \ell_2)$. ■

Moreover, in addition to the lifting and partial lifting rules based on the *Shift* operation, we also need a rule that allows us to unfold the paths occurring in the contexts of conditions. For this purpose, in the rest of this subsection, we formalize the unfolding mechanism that we will use in the rest of the paper.

The following proposition establishes a key tautology in our logic with paths:

Proposition 1 (Unfolding Tautology). *Given a pattern $P = (S, E, PE)$, a pattern expression $pe = \langle n, L, n' \rangle \in PE$, and a decomposition d of pe, we have that the condition $\bigvee_{sd \in d} \exists(\mathtt{Unfold}(P, \langle n, L, n' \rangle, sd), \mathtt{true})$ is a tautology over P.*

Proof. We have to prove that every $f : P \to S \models \bigvee_{sd \in d} \exists(\texttt{Unfold}(P, \langle n, L, n' \rangle, sd), \texttt{true})$, where S is a structure. That is, we have to prove that there exist $sd \in d$ and $g : P' \to S$ such that $g \circ u = f$, where $u : P \to P' = \texttt{Unfold}(P, \langle n, L, n' \rangle, sd)$.



Since f is a pattern morphism and S is a structure, we have that $\langle f_v(n), L_0, f_v(n') \rangle \in E_S^+$, where L_0 includes only the sequence of labels of the path from $f_v(n)$ to $f_v(n)$, i.e., $L_0 = \{l_1 \dots l_k\} \subseteq L$. Then, since d is a decomposition of $\langle n, L, n' \rangle$, there is a subdecomposition $sd \in d$, with $sd = (L', s)$ such that $L_0 \subseteq L'$. This means there exists $u = \texttt{Unfold}(P, \langle n, L, n' \rangle, sd)$ and, as a consequence of (3) in Definition 6, we have that the morphism g exists, such that $g \circ u = f$. ■

5.3 Inference Rules

We consider the following set of rules, where ℓ_2 means any (positive or negative) literal condition, c_P is any condition over $P = (S, E, PE)$, and $d \in D(pe)$ denotes that d is a decomposition of pe. Without loss of generality[6], we will assume that our conditions are in clausal form, that is, they are sets of disjunctions of *literals*, where a literal is either `true` or a condition $\exists(a, c_Q)$ or $\neg\exists(a, c_Q)$, where c_Q is again in clausal form.[7]

$$(\texttt{Lift}) \quad \frac{\exists(a_1, c_1) \quad \ell_2}{\exists(a_1, c_1 \wedge \mathit{Shift}(a_1, \ell_2))}$$

$$(\texttt{Partial Lift}) \quad \frac{\exists(a_1, c_1) \quad \neg\exists(a_2, c_2)}{\exists(a_1, c_1 \wedge \mathit{Shift}(g, \neg c_2))} \text{ if } a_1 = g \circ a_2$$

$$(\texttt{Unfolding}) \quad \frac{c_P}{\bigvee_{sd \in d} \exists(\texttt{Unfold}(P, pe, sd), \texttt{true})} \text{ if } d \in D(pe) \text{ for } pe = \langle n, L, n' \rangle \in PE$$

$$(\texttt{Split Introduction}) \quad \frac{\neg\exists(a, c)}{\exists(a, \texttt{true})} \text{ if } a \text{ is a split mono}$$

$$(\texttt{False}) \quad \frac{\exists(a_1, \texttt{false})}{\texttt{false}}$$

Let us prove the soundness of the inference rules.

[6] In [18,19] it is proved that we can transform any condition into a clausal form.

[7] Split Introduction rule may seem not very useful, however in [14] it was needed to achieve completeness. A morphism $a : P \to Q$ is a split mono if it has a left inverse, that is, if there is a morphism a^{-1} such that $a^{-1} \circ a = id_P$.

Theorem 1 (Soundness of Rules). *The above rules are sound.*

Proof. Let S be a structure and $f : P \to S$ be a pattern morphism. We need to prove that whenever f is a model of the premise(s) of a rule, it is also a model of the conclusion.

Lemmas 2 and 3 respectively prove the soundness of the Lift and Partial Lift rules, whereas soundness of the Unfolding rule is obtained from Proposition 1.

Soundness of Split Introduction is a consequence of the following property: If $a : P \to Q$ is a *split mono* then $\exists(a, \texttt{true})$ is equivalent to `true`. The reason is that every morphism $h : P \to S$ satisfies $\exists(a : P \to Q, \texttt{true})$, because the morphism $h \circ a^{-1} : Q \to S$ satisfies $(h \circ a^{-1}) \circ a = h \circ (a^{-1} \circ a) = h$, and $h \circ a^{-1}$ trivially satisfies `true`.

Finally, soundness of False is trivial, because there is no structure that satisfies $\exists(a_1, \texttt{false})$. ■

As a very simple example, let us now show that the set of three conditions in Fig. 6 is unsatisfiable.

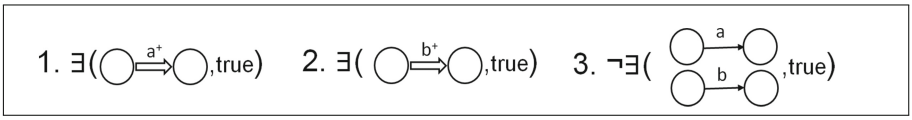


Fig. 6. Example of insatisfiable properties

Applying the Lift rule to conditions 1 and 2 we get condition 4 in Fig. 7, and applying again Lift to condition 4 and condition 3 we get condition 5 also in Fig. 7. Now, let us consider the inner conditions in condition 5, i.e. conditions 6 and 7 in Fig. 7. Applying Unfolding to the path expression labelled with a^+ in condition 6, we get condition 8, and applying unfolding to the path expression labelled with b^+ in condition 8, we get condition 9. Now, applying Partial Lift to condition 7 and, successively, to the four conditions in the disjunction in condition 9, we get condition 10. Then applying four times the rule False to condition 10, we get `false`, which means that if we replace the inner conditions of condition 5, we get condition 11. Finally, if we apply the rule False to that condition we get `false`.

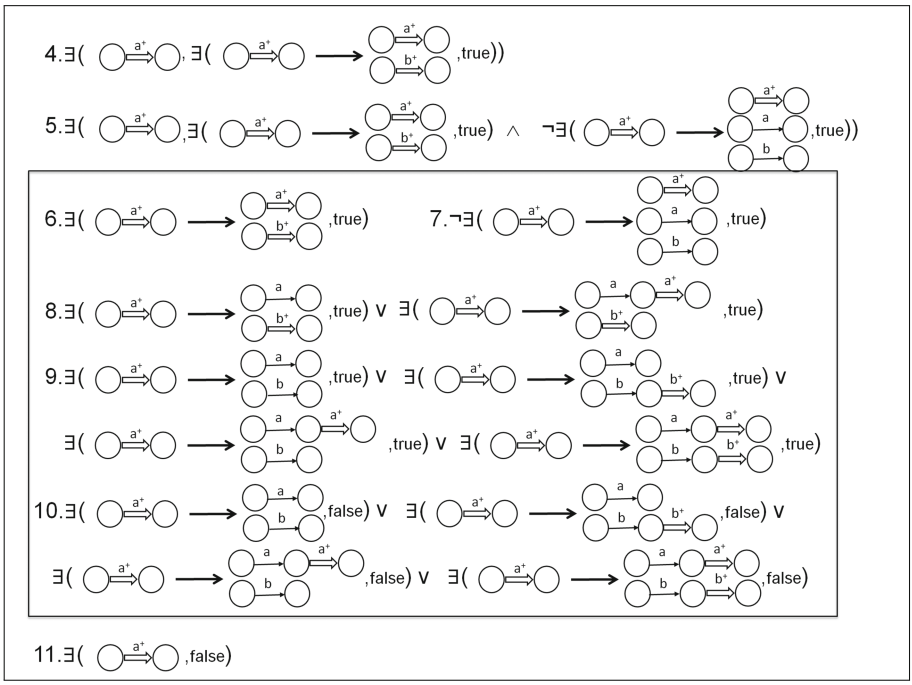


Fig. 7. Example of inferences

6 Related Work, Conclusion and Future Work

The idea of expressing graph properties by means of graphs and graph morphisms has its origins in the notions of graph constraints and application conditions [6,8,11]. In [21], Rensink presented a logic for expressing graph properties, closely related with the Logic of Nested Graph Conditions (LNGC) defined by Habel and Penneman [9]. First approaches to provide deductive methods to this kind of logics were presented in [17] for a fragment of LNGC, and by Pennemann [18,19] for the whole logic. Among the extensions allowing us to state path properties, in [10], Habel and Radke presented a notion of HR^+ conditions with variables that allowed them to express properties about paths, but no deduction method was presented. Also, in [20], Poskitt and Plump proposed an extension of nested conditions with monadic second-order (MSO) properties over nodes and edges. Within this extension, they can define path predicates that allow for the direct expression of properties about paths between nodes, but without defining any deduction method. Finally, in [7], Flick extended the LNGC with recursive definitions using a μ notation and presented a proof calculus showing its partial correctness.

In [14] we presented an extension of LNGC, restricted to the case of directed graphs, including the possibility of specifying the existence of paths between nodes, together with a sound and complete tableau proof method for this logic.

The specification of paths by means of language expressions (in particular, regular expressions) is a usual technique in query languages for graph databases (e.g., [1,2,4,13,23]), but no associated logic is defined.

In this paper we have shown how to generalize the approach presented in [14] to arbitrary categories of graphical structures, including attributed typed graphs. In this sense, the results presented in this paper can be seen as a first step to define a logic underlying graph databases. The next obvious step will be showing the completeness of our inference rules.

Acknowledgements. We are grateful to the anonymous reviewers for their comments that have contributed to improve the paper.

References

1. Barceló, P.: Querying graph databases. In: Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, 22–27 June 2013, pp. 175–188 (2013), https://doi.org/10.1145/2463664.2465216
2. Barceló, P., Libkin, L., Reutter, J.L.: Querying graph patterns. In: Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, 12–16 June 2011, Athens, pp. 199–210 (2011). https://doi.org/10.1145/1989284.1989307
3. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg, G. (ed.) Handbook of Graph Grammars, pp. 313–400. World Scientific, River Edge (1997). https://dl.acm.org/citation.cfm?id=278918.278932
4. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. In: Proceedings SIGMOD 1987 Annual Conference, San Francisco, 27–29 May 1987, pp. 323–330 (1987). https://doi.org/10.1145/38713.38749
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
6. Ehrig, H., Habel, A.: Graph grammars with application conditions. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 87–100. Springer, Heidelberg (1986)
7. Flick, N.E.: On correctness of graph programs relative to recursively nested conditions. In: Workshop on Graph Computation Models (GCM 2015), vol. 1403, pp. 97–112 (2015). http://ceur-ws.org/Vol-1403/
8. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26**, 287–313 (1996). https://doi.org/10.3233/FI-1996-263404
9. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009). https://doi.org/10.1017/S0960129508007202
10. Habel, A., Radke, H.: Expressiveness of graph conditions with variables. ECEASST, 30 (2010) https://doi.org/10.14279/tuj.eceasst.30.404
11. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph rewriting - a constructive approach. Electr. Notes Theor. Comput. Sci. **2**, 118–126 (1995). https://doi.org/10.1016/S1571-0661(05)80188-4

12. Lambers, L., Orejas, F.: Tableau-Based Reasoning for Graph Properties. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 17–32. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09108-2_2
13. Libkin, L., Vrgoc, D.: Regular path queries on graphs with data. In: 15th International Conference on Database Theory, ICDT 2012, Berlin, 26–29 March 2012, pp. 74–85 (2012). https://doi.org/10.1145/2274576.2274585
14. Navarro, M., Orejas, F., Pino, E., Lambers, L.: A logic of graph conditions extended with paths. In: Workshop on Graph Computation Models (GCM 2016), Vienna (2016)
15. Orejas, F.: Attributed graph constraints. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 274–288. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_19
16. Orejas, F.: Symbolic graphs for attributed graph constraints. J. Symb. Comput. **46**(3), 294–315 (2011). https://doi.org/10.1016/j.jsc.2010.09.009
17. Orejas, F., Ehrig, H., Prange, U.: Reasoning with graph constraints. Formal Asp. Comput. **22**(3–4), 385–422 (2010). https://doi.org/10.1007/s00165-009-0116-9
18. Pennemann, K.-H.: Resolution-like theorem proving for high-level conditions. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 289–304. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_20
19. Pennemann, K.H.: Development of Correct Graph Transformation Systems, PhD Thesis. Department of Computing Science, University of Oldenburg (2009)
20. 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
21. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_23
22. Trakhtenbrot, B.A.: The impossibility of an algorithm for the decision problem on finite classes (In Russian). Doklady Akademii Nauk SSSR, 70:569–572, 1950. English translation. In: Nine Papers on Logic and Quantum Electrodynamics, AMS Transl. Ser. **2**(23), 1–5 (1963)
23. Wood, P.T.: Query languages for graph databases. SIGMOD Rec. **41**(1), 50–60 (2012). https://doi.org/10.1145/2206869.2206879

Model Transformations as Free Constructions

Michael Löwe(✉)

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

Abstract. Hartmut Ehrig was an active researcher in Algebraic Specifications on the one hand and Graph and Model Transformations on the other hand. We demonstrate that these two research fields are closely connected, if we consider *generating* graph transformations only and use *partial algebras* instead of total algebras as the underlying category.

1 Introduction

In the last two decades, algebraic graph transformations became popular as a visual specification method for model transformations [8,9]. There are two major reasons for that: Graph transformations use a graphical notation and are rule-based. A graphical notation is adequate, since many modern modelling techniques are graphical themselves, like for example use case, class, state, or activity diagrams in the Unified Modelling Language UML [15] or process specifications in Business Process Model and Notation BPMN [14]. And a rule-based mechanism avoids explicit control structures. Thus, it opens up the chance for massively parallel transformations.

Model transformations define the mapping between artefacts in possibly different modelling languages. Examples are the mapping of Petri-Nets [20] to State Charts [17], entity relationship diagrams [4] to relational database schemata, XML-schemata [6] to UML class diagrams [15], or arbitrary UML class diagrams to UML class diagrams without symmetric associations and multiple inheritance[1]. In all these situations, model transformations are *generators*: given a model in the source language, the transformation process step by step generates a model in the target language together with a mapping that records the correspondence between the original items in the source to the generated items. Every model generation of this type must satisfy the following general requirements:

Termination. The generation process terminates for every finite source model.
Uniqueness. It produces a uniquely determined target for every source model.[2]
Persistence. The target generation does not change the source model.

These three requirements suggest that model transformations can be understood as some sort of (persistent) free construction from a suitable category of

[1] This transformation can serve as a prerequisite for the "compilation" of UML class diagrams to object-oriented programming languages like Java.

[2] Up to isomorphism, if the semantics of transformations is using category theory.

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 142–159, 2018.
https://doi.org/10.1007/978-3-319-75396-6_8

source models to a suitable category of target models.[3] In this paper, we provide such an interpretation. For this purpose, we do not have to introduce any new definitions or results in terms of theorems and mathematical proofs. Instead, we interpret the available results in a slightly adjusted environment:

1. We restrict the well-known algebraic graph transformations to generating rules, i.e. to rules that do not delete or copy any items.
2. For the underlying category, we pass from total unary algebras[4] to arbitrary partial algebras.

The first adjustment leads to a special case of graph transformations, in which the three different algebraic approaches to graph transformation, namely the double-pushout [12], the single-pushout [18], and the sesqui-pushout approach [7] coincide. This is due to the fact that generating rules are simple morphisms $L \xrightarrow{r} R$. For rules of this format, the direct derivation at a match $m : L \to G$ is a simple pushout construction of r and m in *all three* algebraic approaches.

The second adjustment provides a simple encoding of a "growing" correspondence between items in the source model and generated items in the target model, namely by partial mappings that are getting more and more defined within the transformation process.[5] And, in partial algebras, the generation of elements, the creation of definedness for predicates, and the production of equivalences can be controlled by the *same* simple mechanism, namely Horn-type formulae.[6]

The rest of the paper is structured as follows. Section 2 introduces a typical example of a model transformation scenario, namely the generation of relational database schemata for object-oriented class diagrams. This example has also been used as a running example in [9].[7] We show by the example, that even complex model transformation tasks can be modelled by purely generating rules. Section 3 presents the rich theory of generating transformations in a very general categorical framework. Especially, we show in Sect. 3.2 that special generating transformation systems can be interpreted as specifications defining epi-reflective sub-categories of the underlying category. Section 4 presents the basic theory for algebraic specifications of partial algebras as a concrete syntax for transformation rules. In this framework, the sample transformation rules in Sect. 2 obtain a formal semantics, which allows a detailed review of all samples in Sect. 5. The analysis results in substantial improvements that, on the one hand, simplify the transformations from the practical point of view and, on the other hand, turn all of them into free constructions.

[3] For persistent free construction between abstract data types compare [5,10,11].
[4] Like graphs, hypergraphs, graph structures [18], or general functor categories from a finite category to the category of sets and mappings.
[5] Possibly without becoming total at last.
[6] Alfred Horn, American mathematician, 1918 – 2011.
[7] Compare [9], Example 3.6 on page 54.

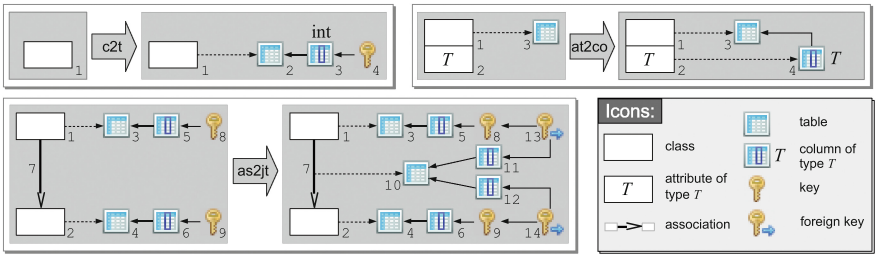


Fig. 1. Handling of classes, attributes, and associations in ST and CT

2 Sample Transformations – from Classes to Relations

A typical model transformation scenario is the generation of relational schemata for object-oriented class diagrams. In this section, we present transformation rules for all three patterns that are useful to cope with inheritance, namely Single Table Inheritance (ST), Class Table Inheritance (CT), and Concrete Class Table Inheritance (CCT) [13].[8] The presentation in this section stays on the intuitive level and trusts in the suggestive power of the rule visualisations as graphs with significant icons. Section 4 presents the necessary formal underpinning for a precise semantics which is described in Sect. 5.

In this section, we use the same mapping from class diagrams to relational schemata which has been used in Example 3.6 of [9]: Classes are mapped to tables, attributes are mapped to columns, and associations are mapped to junction tables. The corresponding three rules for Single Table Inheritance and Class Table Inheritance are depicted in Fig. 1.[9] For each class, the rule `c2t` generates a new table together with a column of numeric type (`int`) that is marked as key column[10]. That a class has been mapped is stored by a (partial) map indicated by the dotted arrow. The rule that generates columns in tables for class attributes is `at2co`. It is applicable to each attribute the class of which possesses an assigned table, possibly generated by an application of rule `c2t`. In order to simplify the presentation in this paper, we suppose that the set of base types (int, bool, string etc.) is identical in class models and relational schemata. Finally, associations between classes are mapped to junction tables by rule `as2jt`. This rule presupposes that the two classes at the ends of the association possess an

[8] This section uses material from [21].

[9] The correspondence of the items in the left hand side of the rule to items in the right-hand side is always indicated by the visual correspondence in the layouts. In Fig. 1, we give an additional explicit definition of the mapping from left to right by index numbers. In most cases, this explicit indication is superfluous. Since mappings must be type-conform, i.e. classes can only be mapped to classes, attributes can only be mapped to attributes etc., and the mapping must respect the graphical structure, the mapping is uniquely determined for all rules that are depicted below. Thus, we do not use the index numbers in the following.

[10] Standard key columns are `unique` and `not null`.

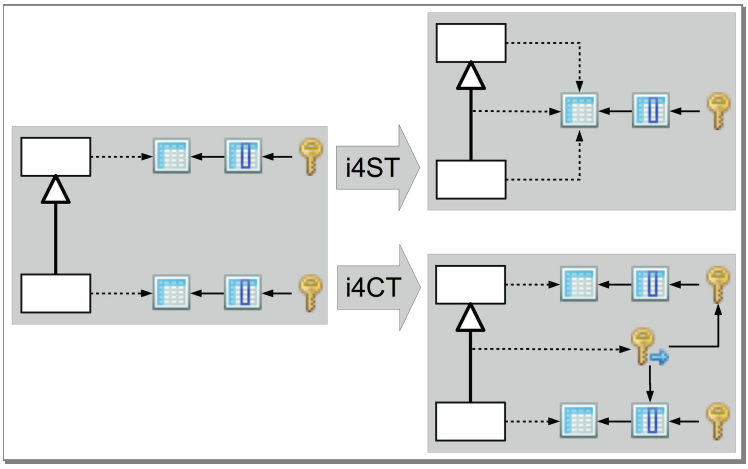


Fig. 2. Handling of inheritance in ST and CT

assigned table with a primary key column each, which have possibly been generated by two applications of rule `c2t`. To each association, it assigns (dotted arrow) a new table with two columns that both are marked as foreign keys.[11] The foreign keys reference the keys that have been found by the match. For the sake of simplicity, we suppose that the type of a foreign key column is implicitly set to the type of the referenced key column.

The handling of inheritance relations between classes is different in ST and CT. Figure 2 depicts the corresponding rules `i4ST` respectively `i4CT` which have the same left-hand side. The ST-pattern puts a complete inheritance hierarchy into a single table. This effect is implemented by the rule `i4ST` which merges the tables and keys for the super- and the sub-class of an inheritance relation and maps the relation itself to the same table, compare upper part of Fig. 2.[12,13]

Figure 3 depicts a sample transformation sequence for a small composite class diagram in the single-table scenario. In the first step, rule `c2t` is applied three times and produces a table with primary key column for each class in the start object. The second step ($2 \times$ `at4co`) handles the column generation for the two given attributes. The third step ($2 \times$ `i4ST`) merges the three tables that have been generated before and ensures that there is only one table for the complete inheritance hierarchy. Finally, the fourth step applies the rule `as2jt` once which adds the junction table for the given association.

The CT-pattern realises inheritance relations by foreign key references. Therefore, the rule `i4CT` maps an inheritance relation to a foreign key, compare lower part of Fig. 2. The key column of the table for the sub-class is

[11] Standard foreign key columns are `not null`.
[12] Merging of objects is expressed by non-injective rules.
[13] Due to the merging, some of the generated junction tables for association may no longer accurately model the association's semantics.

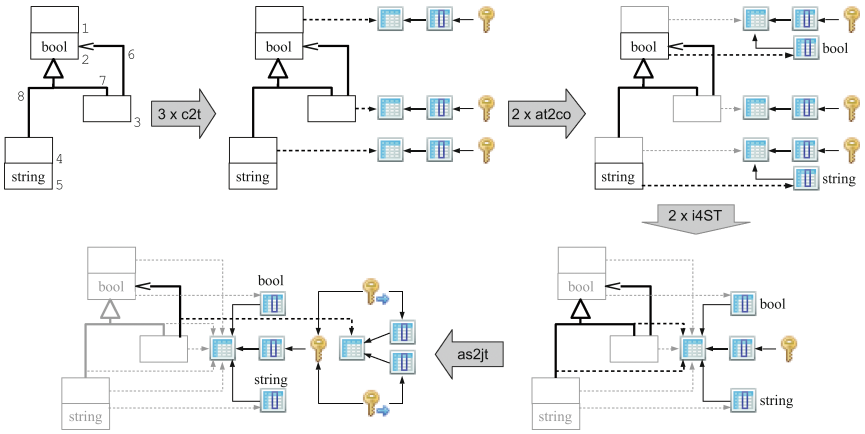


Fig. 3. Sample transformation sequence

simultaneously used as a foreign key reference to the key of the table for the super-class. This strategy requires coordinated key value generation for all tables "in an inheritance hierarchy" but provides unique object identity for all "parts" of the same object in different tables.

The CCT-pattern generates tables for all concrete classes only and "copies" all inherited attributes and associations to these tables. For the control of this copying process, we need a relation that provides all direct and indirect (transitive) sub-classes for a super-class. Figure 4 shows the rules, that "compute" the reflexive ($\mathtt{t}^0$) and transitive ($\mathtt{t}^*$) closure of the given inheritance relation ($\mathtt{t}^1$).

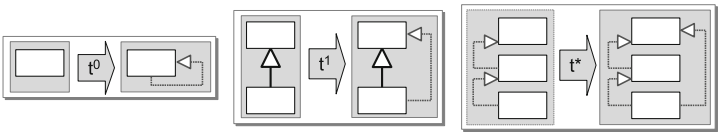


Fig. 4. Reflexive/transitive closure of inheritance

Since tables are generated for concrete classes only, we distinguish between abstract and concrete classes. Concrete classes are visualised by the annotation {c}. The rule cc2t in Fig. 5 generates tables and keys in the CCT-pattern.[14]

A single attribute can result in several columns, compare rule `cat2co` in Fig. 5: For an attribute a of type T in class c, a column of type T is generated into every table for a concrete class c' that is a sub-class of c. Thus, the attribute mapping gets indexed by the concrete sub-classes of the owner class

[14] The rule cc2t is a simple modification of rule `c2t` in Fig. 1. Here, the class-to-table mapping is partial, since abstract classes are never mapped in CCT.

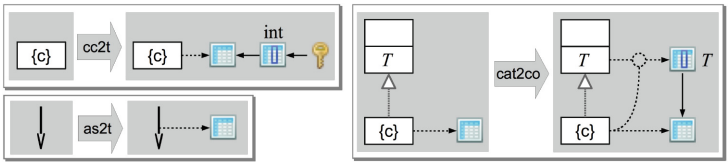


Fig. 5. Handling of classes, attributes, and associations in CCT

of the attribute, compare dotted circle in the visualisation of rule `cat2co` in Fig. 5.[15]

The handling of associations in CCT is even more complex. They cannot be mapped to *one* foreign key pair, since rows from several "unconnected" tables can be linked by instances of an association in this pattern. For an accurate mapping of the class model semantics, we need orthogonal combinations of foreign keys into the tables for all concrete sub-classes of the association's source class with foreign keys into the tables for all concrete sub-classes of the association's target class. The generation of these foreign keys is prepared by rule `as2t` in Fig. 5. It provides the table for all foreign key columns that are generated for an association.

The rules `s2co` and `t2co` in Fig. 6 generate these columns together with the foreign key references to the corresponding tables. These two rules are almost identical; `s2co` handles all concrete sub-classes of the association's source and `t2co` all concrete sub-classes of the association's target class. As in the case of the attribute mapping, the two mappings that store the correspondence between items in the class model and items in the relational schema are indexed by the concrete sub-class either on the source or the target side, compare the two dotted circles in Fig. 6.

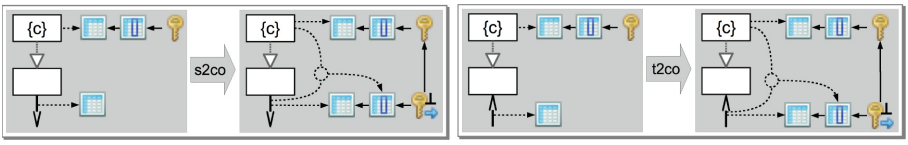


Fig. 6. Foreign keys for associations in CCT

For an accurate representation of the semantics of the class model, in each row of the "multi-junction table" exactly one foreign key on the source side and exactly one foreign key on the target side must be `not null`.[16] Thus, we need

[15] Rule `cat2co` also works for the own attributes of a concrete class due to the reflexivity rule $\mathtt{t}^0$ in Fig. 4. This situation requires non-injective matches!

[16] For the sake of simplicity of the presentation we do not generate a suitable `check` constraint in the relational schema.

foreign key columns that allow `null` values. This sort of foreign keys is depicted by a foreign key icon that is decorated by a $\perp$-symbol in Fig. 6.

3 Generating Transformations and Epi-Reflections

The samples in the preceding section show that model transformations can be specified by generating rules. This special case of transformation rules and transformation systems is presented in this section. We show that there is a rich theory especially wrt. parallel and sequential independence and parallel rule application. We assume an underlying category $\mathcal{C}$ with all small co-limits.[17,18]

3.1 Generating Transformation Systems

A *rule* is a morphism $r : L \to R$, a *match* for rule $r : L \to R$ in object G is a morphism $m : L \to G$, and a *direct transformation* with rule r at match m is defined by the pushout $(r\langle m\rangle : G \to r@m, m\langle r\rangle : R \to r@m)$ of the pair (r, m).[19] The derivation result is denoted by $r@m$, the morphisms $r\langle m\rangle$ is called the *trace* of the direct derivation, and the morphism $m\langle r\rangle$ the *co-match*.[20] The result $r@m$ is unique up to isomorphism, since pushouts are.

A *transformation system* $\mathbb{R}$ is a set of rules. The class $\mathbb{R}^{\rightarrow}$ of $\mathbb{R}$-*transformations* is the least class of morphisms which (i) contains all isomorphisms, (ii) contains all traces $r\langle m\rangle$ for all rules in $r \in \mathbb{R}$ and all suitable matches m, and (iii) is closed under composition. By $\mathbb{R}^{\rightarrow}_G = \{h \in \mathbb{R}^{\rightarrow} \mid \text{domain}(h) = G\}$, we denote the $\mathbb{R}$-transformations starting at object G. An object G is *final* wrt. a transformation system $\mathbb{R}$, if all $h \in \mathbb{R}^{\rightarrow}_G$ are isomorphisms.[21] A system $\mathbb{R}$ is

terminating if, for any infinite sequence $(t_i : G_i \to G_{i+1} \in \mathbb{R}^{\rightarrow})_{i\in\mathbb{N}}$, there is $n \in \mathbb{N}$, such that G_n is final,
confluent if, for any two $\mathbb{R}$-transformation $t_1 : G \to H_1$ and $t_2 : G \to H_2$, there are $\mathbb{R}$-transformations $t'_1 : H_2 \to K$ and $t'_2 : H_1 \to K$, and
functional if it is terminating and confluent.

Every generating transformation system is *strongly* confluent as the following argument demonstrates. Consider Fig. 7 which depicts two arbitrary direct

[17] A category has all small co-limits, if it has all co-limits for small diagram categories. A category is small if its collection of objects is a sets. As in [1], the family of morphisms in a category is a family of sets anyway.
[18] Examples for such categories are all total or partial Σ-algebras for a given signature Σ or every epi-reflective sub-category of such categories of Σ-algebras, see below.
[19] Neither rules nor matches are required to be monomorphisms. Rules and matches can be arbitrary morphisms.
[20] Sub-diagram (1) in Fig. 7 denotes a direct derivation with rule r_1 at match m_1.
[21] Object G being final wrt. system $\mathbb{R}$ does not mean that there are no matches for rules in $\mathbb{R}$ into G. But all these matches produce traces that are isomorphisms, i.e. do not have any effect.

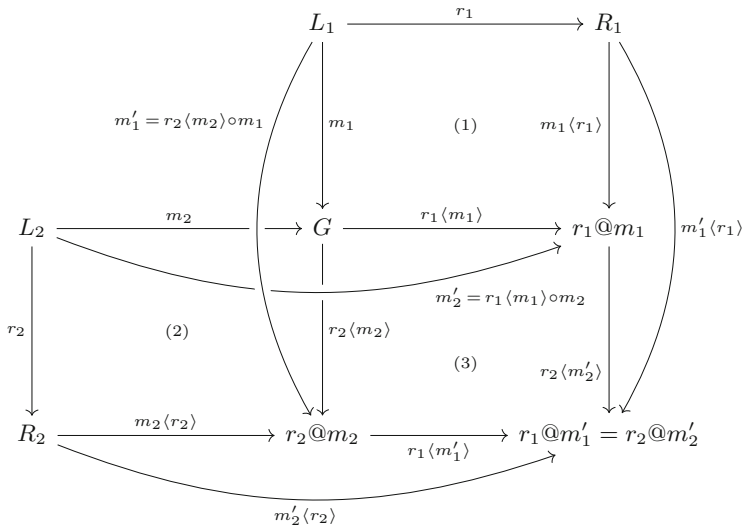


Fig. 7. Strong confluence

derivations with rules r_1 and r_2 at matches m_1 and m_2 respectively, i.e. sub-diagrams (1) and (2) are pushouts. Composing the original match of one rule with the trace induced by the other rule provides the two *residual* matches $m_1' = r_2\langle m_2\rangle \circ m_1$ and $m_2' = r_1\langle m_1\rangle \circ m_2$. Applying rule r_1 at that residual match m_1' leads to the pushout $(r_1\langle m_1'\rangle : r_2@m_2 \to r_1@m_1', m_1'\langle r_1\rangle : R_1 \to r_1@m_1')$. Since sub-diagram (1) is a pushout and $(r_1\langle m_1'\rangle \circ r_2\langle m_2\rangle) \circ m_1 = r_1\langle m_1'\rangle \circ m_1' = m_1'\langle r_1\rangle \circ r_1$, we obtain the unique morphism from $r_1@m_1$ to $r_1@m_1'$, which we call $r_2\langle m_2'\rangle$, satisfying $r_2\langle m_2'\rangle \circ m_1\langle r_1\rangle = m_1'\langle r_1\rangle$ and $r_2\langle m_2'\rangle \circ r_1\langle m_1\rangle = r_1\langle m_1'\rangle \circ r_2\langle m_2\rangle$. Since pushouts decompose, the sub-diagram (3) in Fig. 7 is a pushout, i.e. the pair $(r_2\langle m_2'\rangle, r_1\langle m_1'\rangle)$ is pushout of $(r_1\langle m_1\rangle, r_2\langle m_2\rangle)$. Since pushouts compose, the sub-diagrams (2) and (3) together constitute a pushout, i.e. the pair $(r_1\langle m_1'\rangle \circ m_2\langle r_2\rangle, r_2\langle m_2'\rangle)$ is pushout of r_2 and $r_1\langle m_1\rangle \circ m_2 = m_2'$. Therefore $r_2\langle m_2'\rangle$ is the trace of the direct derivation with rule r_2 at match m_2' and $r_1\langle m_1'\rangle \circ m_2\langle r_2\rangle$ is the co-match $m_2'\langle r_2\rangle$. Combining these results, we obtain:

$$r_1@(r_2\langle m_2\rangle \circ m_1) = r_2@(r_1\langle m_1\rangle \circ m_2) \text{ and}$$
$$r_2\langle m_2'\rangle \circ r_1\langle m_1\rangle = r_1\langle m_1'\rangle \circ r_2\langle m_2\rangle.$$

Thus, the system is strongly confluent, which implies that it is confluent and, furthermore, that *it is functional, iff it is terminating*. That every generating transformation system is strongly confluent has some further direct positive consequences, especially wrt. sequential independence and parallel transformations.

Two composable traces $r_2\langle m_2\rangle \circ r_1\langle m_1\rangle$ are *sequentially independent*, if there is a match m_2' for the rule r_2 such that $m_2 = r_1\langle m_1\rangle \circ m_2'$. In the case of sequential independence, the order of rule application can be interchanged,

Algorithm 1. Calculation of final objects

(1) Set the current object c to the start object o.
(2) Find all matches for all $\mathbb{R}$-rules in c
(3) Produce trace $t : c \to c'$ for corresponding parallel rule at induced parallel match.
(4) If t is an isomorphism stop and return c.
(5) Otherwise set the current object c to c' and continue at (2).

i.e. $r_2\langle m_2\rangle \circ r_1\langle m_1\rangle = r_1\langle r_2\langle m_2'\rangle \circ m_1\rangle \circ r_2\langle m_2'\rangle$. The proof for this result is a simple reduction to strong confluence, compare Fig. 7.

Given two rules $r_1 : L_1 \to R_1$ and $r_2 : L_2 \to R_2$, we can construct the *parallel rule* $r_1{+}r_2 : L_1{+}L_2 \to R_1{+}R_2$ as the unique co-product morphism, where $(i_{L_1} : L_1 \to L_1{+}L_2,\ i_{L_2} : L_2 \to L_1{+}L_2)$ is the co-product of the rule's left-hand sides, $(i_{R_1} : R_1 \to R_1{+}R_2,\ i_{R_2} : R_2 \to R_1{+}R_2)$ is the co-product of the rule's right-hand sides, and $r_1{+}r_2$ satisfies $i_{R_1} \circ (r_1 + r_2) = r_1 \circ i_{L_1}$ and $i_{R_2} \circ (r_1 + r_2) = r_2 \circ i_{L_2}$. Having two matches m_1 and m_2 in the same object for r_1 and r_2 respectively, the parallel match $m_1{+}m_2$ is uniquely determined by $(m_1 + m_2) \circ i_{L_1} = m_1$ and $(m_1 + m_2) \circ i_{L_2} = m_2$. Since any co-limit construction of the same situation in any order results in the same object (up to isomorphism), we immediately obtain: $r_1{+}r_2\langle m_1 + m_2\rangle = r_2\langle r_1\langle m_1\rangle \circ m_2\rangle \circ r_1\langle m_1\rangle = r_1\langle r_2\langle m_2\rangle \circ m_1\rangle \circ r_2\langle m_2\rangle$.

This result provides a good (maximal parallel) strategy for the search of a final object for a start object o in a system $\mathbb{R}$, compare Algorithm 1. If the system is terminating, the algorithm finds *the* final object for all start objects. Unfortunately, most systems are not terminating as the following argument shows.

The application of a rule r at a match m is idempotent, i.e. $r\langle r\langle m\rangle \circ m\rangle$ is an isomorphism, if and only if the trace $r\langle m\rangle$ is an epimorphism. For the proof, consider Fig. 7 again and let $r_1 = r_2$ as well as $m_1 = m_2$. For the if-part, let $r_1\langle m_1\rangle$ as well as $r_2\langle m_2\rangle$ be epic. Then $r_1\langle m_1'\rangle$, and $r_2\langle m_2'\rangle$ are epic, since pushouts preserve epimorphisms. Since pushouts are unique up to isomorphism, there is an isomorphism $i : r_1@m_1 \to r_2@m_2$ such that $i \circ r_1\langle m_1\rangle = r_2\langle m_2\rangle$. Since (3) is pushout, we obtain $j : r_1@m_1' \to r_2@m_2$ with $j \circ r_1\langle m_1'\rangle = \mathrm{id}$ and $j \circ r_2\langle m_2'\rangle = i$. Thus, $r_1\langle m_1'\rangle$ is section and epic, which means that it is isomorphism. For the only-if-part, suppose the application is idempotent, i.e. $r_1\langle m_1'\rangle$ is isomorphism and $r_2\langle m_2'\rangle$ is isomorphism with inverse morphism j. Then $r_1\langle m_1\rangle$ is epic: $h \circ r_1\langle m_1\rangle = k \circ r_1\langle m_1\rangle$ implies $h \circ r_1\langle m_1\rangle = k \circ r_1\langle m_1\rangle = k \circ j \circ r_2\langle m_2'\rangle \circ r_1\langle m_1\rangle = (k \circ j \circ r_1\langle m_1'\rangle) \circ r_2\langle m_2\rangle$. Since $(r_1\langle m_1'\rangle, r_2\langle m_2'\rangle)$ is pushout, there is unique morphism u such that $u \circ r_1\langle m_1'\rangle = k \circ j \circ r_1\langle m_1'\rangle$ and, since $r_1\langle m_1'\rangle$ is isomorphism, (i) $u = k \circ j$ as well as $u \circ r_2\langle m_2'\rangle = h$ and, since j is inverse of $r_2\langle m_2'\rangle$, (ii) $h \circ j = u \circ r_2\langle m_2'\rangle \circ j = u$. Now (i) and (ii) imply that $h \circ j = k \circ j$ and $h = k$ since j is isomorphism.

Therefore, a single non-epic trace $r\langle m\rangle$ in a system prevents termination, since the rule r can be applied over and over again at residuals of m with "new results". Thus, a necessary condition for termination is that all traces in the system are epic. Since pushouts preserve epimorphisms, this property can be guaranteed, if we restrict rules to epimorphisms. Such systems consisting of epic rules only possess another important property as the next sub-section demonstrates.

3.2 Transformation Systems as Epi-Reflections

Every generating transformation system $\mathbb{R}$ in the sense of Sect. 3.1 which consists of epic rules only, can be interpreted as a specification of an epi-reflective sub-category of the underlying category $\mathcal{C}$. This section recapitulates the construction of this epi-reflection.[22]

Every epic transformation rule $r : L \twoheadrightarrow R$ can be interpreted as a *constructive axiom.*[23] It is *finite* or of *Horn-type*, if it satisfies the following condition: For every chain of morphisms $(m_i : O_i \to O_{i+1})_{i\in\mathbb{N}}$ with the co-limit $\big(C, (c_i : O_i \to C)_{i\in\mathbb{N}}\big)$, every morphism $p : L \to C$ into the co-limit object factors through an object in the chain, i.e. there is $i \in \mathbb{N}$ and a morphism $p_i : L \to O_i$ with $c_i \circ p_i = p$.

A morphism $m : L \to A$ *solves* axiom $r : L \twoheadrightarrow R$ in object A, written $m \models r$, if there is morphism $m^r : R \to A$ such that $m^r \circ r = m$.[24] An object A *satisfies* axiom r, written $A \models r$, if every morphism $m : L \to A$ solves r. An object A *satisfies* a transformation system $\mathbb{R}$ of epic rules, written $A \models \mathbb{R}$, if $A \models r$ for all $r \in \mathbb{R}$. The full sub-category of $\mathcal{C}$ which contains all objects satisfying $\mathbb{R}$ is denoted by $\mathcal{C}_{\mathbb{R}}$. Every such sub-category turns out to be an epi-reflection of $\mathcal{C}$.

Given an object A and a transformation system $\mathbb{R}$ of epic rules,

$$A_{\mathbb{R}} = \left\{ A \xleftarrow{m} L_m \xrightarrow{r_m} R_m \mid (r : L \twoheadrightarrow R) \in \mathbb{R}, m : L \to A \right\}$$

denotes the diagram of all occurrences of the left-hand sides of all rules in the transformation system in A.[25] In that diagram, for every morphism $m : L \to A$, the morphism $r_m : L_m \to R_m$ is a "copy" of $r : L \twoheadrightarrow R$. The co-limit of $A_{\mathbb{R}}$ is denoted by $A^{\mathbb{R}} = \left(A^{\mathbb{R}}, r^*_A : A \to A^{\mathbb{R}}, \big(m^{r_m} : R_m \to A^{\mathbb{R}}\big)_{\left(A \xleftarrow{m} L_m \xrightarrow{r_m} R_m\right) \in A_{\mathbb{R}}}\right)$ and we have $r^*_A \circ m = m^{r_m} \circ r_m$ for all $\left(A \xleftarrow{m} L_m \xrightarrow{r_m} R_m\right) \in A_{\mathbb{R}}$. The morphism r^*_A is an epimorphism, since the equation (i) $h \circ r^*_A = k \circ r^*_A$ implies, for all $\left(A \xleftarrow{m} L_m \xrightarrow{r_m} R_m\right) \in A_{\mathbb{R}}$, $h \circ m^{r_m} \circ r_m = h \circ r^*_A \circ m = k \circ r^*_A \circ m = k \circ m^{r_m} \circ r_m$ and, since rules are epimorphisms, (ii) $h \circ m^{r_m} = k \circ m^{r_m}$. Since $A^{\mathbb{R}}$ is co-limit, (i) and (ii) imply $h = k$.

For given object A and transformation system $\mathbb{R}$, consider $(r^*_{A_i} : A_i \twoheadrightarrow A_{i+1})_{i\in\mathbb{N}}$ as the chain of epimorphisms starting at $A_1 = A$ and having $A_{i+1} = A_i^{\mathbb{R}}$. We denote the co-limit of this chain by $\big(\mathbb{R}(A), (a_i : A_i \twoheadrightarrow \mathbb{R}(A))_{i\in\mathbb{N}}\big)$ and obtain $\big(a_{i+1} \circ r^*_{A_i} = a_i\big)_{i\in\mathbb{N}}$. All these co-limit morphism are epimorphisms, since all morphisms in the chain are epic.

Furthermore $\mathbb{R}(A) \in \mathcal{C}_{\mathbb{R}}$: Let $(r : L \twoheadrightarrow R) \in \mathbb{R}$ and let $m : L \to \mathbb{R}(A)$. Since r is finite, there is $i \in \mathbb{N}$ and $m_i : L \to A_i$ with $a_i \circ m_i = m$. Since $A_{i+1} = A_i^{\mathbb{R}}$, we obtain morphism $m^r_i : R \to A_{i+1}$ with $m^r_i \circ r = r^*_{A_i} \circ m_i$. Putting all parts

[22] The principle set-up follows [1], page 278 ff.

[23] In this context, the objects L and R are called *premise* and *conclusion* respectively.

[24] The morphism m^r is unique, if it exists, since r is epic.

[25] $A_{\mathbb{R}}$ is a small diagram, since the family of morphisms in a category is a family of *sets*, compare definition of categories in [1].

together provides: $(a_{i+1} \circ m_i^r) \circ r = a_{i+1} \circ r^*_{A_i} \circ m_i = a_i \circ m_i = m$, such that we found $a_{i+1} \circ m_i^r$ as the desired morphism $m^r : R \to \mathbb{R}(A)$ with $m^r \circ r = m$.

Finally, we get the following result*: For a transformation system* $\mathbb{R}$ *and object* A, $a_1 : A \twoheadrightarrow \mathbb{R}(A)$ *is the epi-reflector for* A *into* $\mathcal{C}_\mathbb{R}$. The proof is straightforward: If $X \in \mathcal{C}_\mathbb{R}$ and $f : A \to X$ are given, we show by induction on i that there are morphisms $(f_i : A_i \to X)_{i\in\mathbb{N}}$ for all $(A_i)_{i\in\mathbb{N}}$ in the chain $\left(r^*_{A_i} : A_i \twoheadrightarrow A_i^\mathbb{R}\right)_{i\in\mathbb{N}}$ constructed above. Since $A_1 = A$, the induction can start with $f_1 : A_1 \to X := f : A \to X$. Now let, as induction hypothesis, $f_i : A_i \to X$ be given. By construction $A_{i+1} = A_i^\mathbb{R}$ and $A_i^\mathbb{R}$ is a co-limit object. Let $(r : L \twoheadrightarrow R) \in \mathbb{R}$ and $m : L \to A_i$ be a morphism to A_i, then $f_i \circ m : L \to X$ is a morphism to X. Since $X \models \mathbb{R}$, there is $(f_i \circ m)^r : R \to X$ with $(f_i \circ m)^r \circ r = f_i \circ m$. Thus, f_i together with the family of these morphisms are a co-cone for the diagram that has been used to construct $A_{i+1} = A_i^\mathbb{R}$ from A_i. Since $A_i^\mathbb{R}$ is the co-limit of this diagram, we obtain the unique morphism $f_{i+1} : A_i^\mathbb{R} \to X$ that satisfies $f_{i+1} \circ r^*_{A_i} = f_i$. This completes the induction. Now $\left(X, (f_i : A_i \to X)_{i\in\mathbb{N}}\right)$ is a co-cone for the chain $\left(r^*_{A_i} : A_i \twoheadrightarrow A_i^\mathbb{R}\right)_{i\in\mathbb{N}}$. Since $\mathbb{R}(A)$ is the limit of this chain, we get a morphism $f^* : \mathbb{R}(A) \to X$ that satisfies $(f^* \circ a_i = f_i)_{i\in\mathbb{N}}$ and, especially, $f^* \circ a_1 = f_1 = f$. Uniqueness of f^* follows from a_1 being epic.

The co-limit construction for the chain $\left(r^*_{A_i} : A_i \twoheadrightarrow A_{i+1}\right)_{i\in\mathbb{N}}$ is superfluous, if $r^*_{A_k}$ is an isomorphism for some $k \in \mathbb{N}$. In this case, (i) $A_k \in \mathcal{C}_\mathbb{R}$, (ii) r^*_{k+j} is isomorphism for all $j \in \mathbb{N}$, (iii) A_{k+1} is the limit of the chain, and (iv) the reflector for A is given by $r^*_{A_K} \circ \cdots \circ r^*_{A_1} : A \twoheadrightarrow A_{k+1}$.

The presented approximation of the epi-reflector for a transformation system $\mathbb{R}$ with epic rules is an instance of Algorithm 1. Each approximation step $r^*_A : A \twoheadrightarrow A^\mathbb{R}$ is the trace of an application of a maximal parallel rule, compare diagram $A_\mathbb{R}$ above. And the approximation stops after finitely many steps, if and only if the computed object is final wrt. $\mathbb{R}$, i.e. admits isomorphic $\mathbb{R}$-transformations only. Thus, Algorithm 1 calculates the epi-reflector in $\mathcal{C}_\mathbb{R}$ for every start object o in a terminating transformation system $\mathbb{R}$ with epic rules.

4 Partial Algebras

Section 3.2 shows that all generating transformation systems in which all rules are epimorphisms induce free constructions and possess useful properties like idempotent rule applications that facilitates the analysis with respect to termination. Unfortunately, all epimorphisms in categories of *total* algebras, which usually constitute the underlying category for most (graph) transformation frameworks, are surjective, i.e. are just able to generate new equalities. Therefore, frameworks based on total algebras must live with non-epic rules and need an additional machinery for termination, like *negative application conditions* [16], *source consistence derivations* in triple graph grammars [2], or artificial *translation attributes* as in [9], Sect. 7.4. The situation stays simple, if we pass from total to *partial* algebras [3,19] as we recapitulate in this section.

A *signature* $\Sigma = (S, O)$ consists of a set of sorts S and a family of operations $O = (O_{d,c})_{d,c\in S^*}$. For $d, c \in S^*$ and $f \in O_{d,c}$, d is called the *domain*

specification of f and c is called the *co-domain specification*. An (*algebraic*) *system* $A = \left(A_S, O^A\right)$ for a given signature $\Sigma = (S, O)$ consists of a family $A_S = (A_s)_{s \in S}$ of *carrier sets*, and a family $O^A = \left(f^A : A^d \to A^c\right)_{f \in O_{d,c}, d,c \in S^*}$ of *partial functions*.[26]

This set-up allows functions that provide several results simultaneously, since co-domain specifications are taken from the free monoid over the sort set. Especially, operations with an empty co-domain specification are possible. They are interpreted as *predicates* in algebraic systems: If $p \in O_{d,\varepsilon}$ for $d \in S^*$, the function $p^A : A^d \to \{*\}$ maps to the one-element-set in every system A. Thus p^A singles out the elements in A^d for which it is defined, i.e. for which it is "true".

Given two systems A and B with respect to the same signature $\Sigma = (S, O)$, a *homomorphism* $h : A \to B$, is given by a family of *total* mappings $h = (h_s : A_s \to B_s)_{s \in S}$ such that the following condition is satisfied:[27]

$$\forall d, c \in S^*, f \in O_{d,c}, x \in A^d \colon \text{if } f^A(x) \text{ is defined, } f^B\left(h^d(x)\right) = h^c\left(f^A(x)\right). \quad (1)$$

The condition (1) means for the special case where $f \in O_{d,\epsilon}$, that f^B must be defined for $h^d(x)$, if f^A is defined for x. The concrete value of these functions is irrelevant, since there is a single value in $\{*\}$ and $h^\varepsilon = \mathrm{id}_{\{*\}}$. By $\mathsf{Sys}(\Sigma)$, we denote the category of all Σ-systems and all Σ-homomorphisms. For every signature Σ, $\mathsf{Sys}(\Sigma)$ has all small limits and co-limits for each signature Σ.[28]

The most important property of $\mathsf{Sys}(\Sigma)$ for the purposes of this paper is, that epimorphisms are not necessarily surjective. This fact can be demonstrated by a simple example using the signature with one sort $\mathtt{N}$ and one unary operation $\mathtt{s} \in O_{\mathtt{N},\mathtt{N}}$, the system $K = \left(K_\mathtt{N} = \{x\}, \mathtt{s}^K = \emptyset\right)$, the system of natural numbers $N = \left(N_\mathtt{N} = \mathbb{N}_0, \mathtt{s}^N :: i \mapsto i + 1\right)$, and the morphism $k : K \to N$ defined by $x \mapsto 0$. This morphism is epic, since N is generated by the function $\mathtt{s}^N$ starting at value $0 = k(x)$.[29] However, the morphism $k' : K \to N$ with $x \mapsto 1$ is not epic, since the value 0 is not reachable by function calls of $\mathtt{s}^N$ starting at 1.[30]

For a general characterisation of epimorphisms in $\mathsf{Sys}(\Sigma = (S, O))$, we need the following closure operations for a family of subsets $(B_s \subseteq A_s)_{s \in S}$ of a system A. For all sorts $s \in S$:

$$B_s^0 = B_s \cup \left\{ y \in A_s \;::\; f^A(*) = (p, y, q),\ f \in O_{\epsilon,v} \right\}$$

$$B_s^{i+1} = B_s^i \cup \left\{ y \in A_s \;::\; f^A(x) = (p, y, q),\ f \in O_{w,v},\ x \in \left(B^i\right)^w,\ |w| \geq 1 \right\}$$

$$B_s^* = \bigcup_{i \in \mathbb{N}_0} B_s^i$$

[26] Given a family of sets $A = (A_s)_{s \in S}$, $k \geq 0$, and $s_1 \ldots s_k \in S^*$, $A^{s_1 \cdots s_k} = A_{s_1} \times \cdots \times A_{s_k}$.

[27] For a family of mappings $f = (f_s : A_s \to B_s)_{s \in S}$, $k \geq 0$, and $w = s_1 \ldots s_k \in S^*$, $f^w : A^w \to B^w$ is defined by $f^w(x_1, \ldots, x_k) = (f_{s_1}(x_1), \ldots, f_{s_k}(x_k))$ for all $(x_1, \ldots x_k) \in A^w$.

[28] Compare [19].

[29] Any two morphisms $p, q : N \to M$ with $p(k(x)) = q(k(x))$ must coincide on all natural numbers due to the homomorphism condition (1).

[30] It is easy to construct $p, q : N \to M$ with $p(k'(x)) = q(k'(x))$ that differ at value 0.

A Σ -morphism $h : A \to B$ is epic, if and only if $h(A)^*_s = B_s$ for all sorts $s \in S$, i.e. if and only if the system B is function-generated starting at the h -image of A in B.[31] Therefore, a constructive axiom in the sense of Sect. 3.2 in a category $\mathsf{Sys}(\Sigma)$ of algebraic systems can generate new elements in the carriers (as long as they are operation generated), can generate new "truths" by defining new predicate instances, and can generate new equalities if it is not injective.

Constructive axioms are usually presented as finite implications, the elementary building blocks of which are formulae. Given signature $\Sigma = (S, O)$ and variable set $X = (X_s)_{s\in S}$, the set of formulae $\mathcal{F}^{\Sigma,X} = \left(T_s^{\Sigma,X}\right)_{s\in S} \cup F^{\Sigma,X}$ is defined by:

$$x \in T_s^{\Sigma,X} \text{ if } x \in X_s \tag{2}$$
$$f_i(t) \in T_{s_i}^{\Sigma,X} \text{ if } f \in O_{w,s_1\dots s_k},\, t \in \left(T^{\Sigma,X}\right)^w, 1 \le i \le k, k \ge 1 \tag{3}$$
$$f(t) \in F^{\Sigma,X} \text{ if } f \in O_{w,\epsilon},\, t \in \left(T^{\Sigma,X}\right)^w \tag{4}$$
$$l = r \in F^{\Sigma,X} \text{ if } l, r \in T_s^{\Sigma,X}, s \in S \tag{5}$$

A syntactical presentation $P_X = \left(X, P \subseteq \mathcal{F}^{\Sigma,X}\right)$ of a Σ-system consists of a variable set X and a set of formulae P; it is *finite*, if X and P are finite sets. The *presented system* $\mathbf{A}^{P_X} = \mathbf{T}^{P_X}/_{\equiv}$ is constructed as follows. The *carriers* $\left(\mathbf{T}_s^{P_X}\right)_{s\in S}$ are inductively defined by:

$$x \in \mathbf{T}_s^{P_X} \text{ if } x \in X_s \text{ or } \left(x \in P \text{ and } x \in T_s^{\Sigma,X}\right) \tag{6}$$
$$f_j(x) \in \mathbf{T}_{s_j}^{P_X} \text{ if } f_i(x) \in \mathbf{T}_{s_i}^{P_X}, f \in O_{w,s_1\dots s_k}, 1 \le i, j \le k \tag{7}$$
$$t_j \in \mathbf{T}_{s_j}^{P_X} \text{ if } f_i(t_1, \dots t_m) \in \mathbf{T}_{s'_i}^{P_X}, f \in O_{s_1\dots s_m, s'_1\dots s'_n}, 1 \le j \le m, i \le n \tag{8}$$
$$t_j \in \mathbf{T}_{s_j}^{P_X} \text{ if } p(t_1, \dots t_k) \in P, p \in O_{s_1\dots s_k,\epsilon}, 1 \le j \le k, k \ge 1 \tag{9}$$
$$l, r \in \mathbf{T}_s^{P_X} \text{ if } (l = r) \in P \text{ and } l, r \in T_s^{\Sigma,X} \tag{10}$$

For an *operation* $f \in O_{w,s_1\dots s_k}$ $(k \ge 1)$ and a possible argument $x \in \left(\mathbf{T}^{P_X}\right)^w$, $f^{\mathbf{T}^{P_X}}(x) = (f_1(x), \dots, f_k(x))$, if $f_j(x) \in \mathbf{T}_{s_j}^{P_X}$ for all $1 \le j \le k$. For a *predicate* $p \in O_{w,\epsilon}$ and a possible argument $x \in \left(\mathbf{T}^{P_X}\right)^w$, $p^{\mathbf{T}^{P_X}}(x)$ is defined, if $p(x) \in P$. And the *quotient* relation $\equiv\, \subseteq \mathbf{T}^{P_X} \times \mathbf{T}^{P_X}$ is the smallest congruence containing $\{(l, r) \mid (l = r) \in P\}$.[32]

Since $\mathbf{T}^{P_X}$ is closed wrt. sub-terms, compare Eq. (8), and $\equiv$: $\mathbf{T}^{P_X} \to \mathbf{A}^{P_X}$ is surjective, $\mathbf{A}^{P_X}$ is generated by X which means, that there is an epimorphism $x^{P_X} : X \twoheadrightarrow \mathbf{A}^{P_X}$ mapping x to $[x]_{\equiv}$.[33] If we have two syntactical presentations $P_X = \left(X, P \subseteq \mathcal{F}^{\Sigma,X}\right)$ and $C_X = \left(X, C \subseteq \mathcal{F}^{\Sigma,X}\right)$ with the same variable set X

[31] Here, $h(A) = (h_s(A_s))_{s\in S}$ and $h_s(A_s) = \{y \in B_s \mid y = h_s(x), x \in A_s\}$. For a proof of the proposition, compare [19].

[32] An equivalence $\equiv$ on a Σ-System A is a congruence, if $f^A(x_1, \dots x_m) = (y_1, \dots y_n)$, $f^A(x'_1, \dots x'_m) = (y'_1, \dots y'_n)$ and $x_i \equiv x'_i$ for all $1 \le i \le m$ implies $y_j \equiv y'_j$ for all $1 \le j \le n$ for all operations $f \in O$.

[33] Every variable set is a Σ-system with completely undefined operations!

Specification 1. Formalisation of class models

```
CM:= sorts C(lass), B(asetype), At(tribute), As(sociation), I(nheritance)
     opns     int:     ⟶B,      conc:C
           o(wner):At⟶C, t(arget):At⟶B
           o(wner):As⟶C, t(arget):As⟶C
               sub: I⟶C,  sup(er): I⟶C,
                                <=: C,C
     axms       x:C::==> <=(x,x)                                        (a1)
           x:I;l,u:C::sub(x)=l,sup(x)=u ==> <=(l,u)                     (a2)
             x,y,z:C::    <=(x,y),<=(y,z) ==> <=(x,z)                   (a3)
               x,y:C::    <=(x,y),<=(y,x) ==> x=y                       (a4)
```

such that $P \subseteq C$, then the kernel of x^{Px} is contained in the kernel of x^{Cx} and we obtain an epimorphism $\overrightarrow{\overrightarrow{P_XC}} : \mathbf{A}^{Px} \twoheadrightarrow \mathbf{A}^{Cx}$ with $\overrightarrow{\overrightarrow{P_XC}} \circ x^{Px} = x^{Cx}$.[34]

A *Horn-type presentation* $H = \left(X, P \subseteq C \subseteq \mathcal{F}^{\Sigma,X}\right)$ of an axiom consists of a finite variable set X, a finite syntactical presentation $C \subseteq \mathcal{F}^{\Sigma,X}$, called the *conclusion*, and a sub-presentation P of C, called the *premise*. The *presented axiom* is the uniquely determined epimorphism $\overrightarrow{\overrightarrow{P_XC}}$. A system *satisfies* H, if it satisfies $\overrightarrow{\overrightarrow{P_XC}}$, compare Sect. 3.2.

Specification 1 presents a formalisation `CM` of the class diagrams that we used in Sect. 2, i. e. class diagrams are `CM`-algebras. As an example, consider the class diagram which is the start object in Fig. 3. It is modelled by the following `CM`-algebra S: $S_{\mathtt{C}} = \{1,3,4\}$; $S_{\mathtt{B}} = \{\mathtt{int},\mathtt{bool},\mathtt{string}\}$; $S_{\mathtt{At}} = \{2,5\}$; $S_{\mathtt{As}} = \{6\}$; $S_{\mathtt{I}} = \{7,8\}$; $\mathtt{int}^S :: * \mapsto \mathtt{int}$; $\mathtt{conc}^S = \{1,3,4\}$; $\mathtt{o}^S_{\mathtt{At}} :: 2 \mapsto 1, 5 \mapsto 4$; $\mathtt{t}^S_{\mathtt{At}} :: 2 \mapsto \mathtt{bool}, 5 \mapsto \mathtt{string}$; $\mathtt{o}^S_{\mathtt{As}} :: 6 \mapsto 3$; $\mathtt{t}^S_{\mathtt{As}} :: 6 \mapsto 1$; $\mathtt{sub}^S :: 7 \mapsto 3, 8 \mapsto 4$; $\mathtt{sup}^S :: 7 \mapsto 1, 8 \mapsto 1$; $\mathtt{<=}^S = \{(1,1),(3,3),(4,4),(3,1),(4,1)\}$.

All underlined operation are implicitly required to be total. This requirement can be explicitly specified by very simple axioms. For example, the axiom for the operation `o(wner):At⟶C` is: `x:At :: ==> o(x)`. The axioms `(a1)` – `(a4)` specify that inheritance is hierarchical, i.e. induces a partial order `<=`. The rules $\mathtt{t}^0$, $\mathtt{t}^1$, and $\mathtt{t}^*$ in Fig. 4 are picturesque visualisations of the epimorphisms presented by axioms `(a1)`, `(a2)`, and `(a3)` respectively.

Specification 2. Formalisation of relational schemata

```
RS:= sorts T(able), B(asetype), Co(lumn), K(ey), F(oreign)K(ey)
     opns       int:   ⟶B
           ta(ble):Co⟶T,    t(ype):Co⟶B,   n(ullable):Co
           c(olumn):K ⟶Co, c(olumn):FK⟶C,   r(efers):FK⟶K
```

[34] Compare for example Theorem 99 (Homomorphism Theorem 1) in [19].

Specification 3. Formal basis for Single Table Inheritance (ST)

ST:= CM +Basetype RS +	ST' := CM +Basetype RS +
opns C2T: Class ⟶ Table	opns C2K: Class ⟶ Key
At2Co: Attribute ⟶ Column	At2Co: Attribute ⟶ Column
As2JT: Association ⟶ Table	As2FKP: Association ⟶ FK,FK
I2T: Inheritance ⟶ Table	I2K: Inheritance ⟶ Key

5 Sample Transformations – Revisited

The informally introduced model transformation rules in Sect. 2 can be precisely formalised on the basis of the definitions in Sect. 4. Specifications 1 and 2 specify class models and relational schemata respectively. For the model transformation pattern "Single Table Inheritance (ST)", we devise the partial operations depicted in the specification ST in the left part of Specification 3.[35] With this interpretation all rules in Fig. 1 and the rule i4ST in Fig. 2 depict morphisms in Sys(ST). As an example, consider the rule c2t in Fig. 1. Its left-hand side L is the following ST-algebra: $L_{\mathtt{C}} = \{1\}$; $L_{\mathtt{B}} = \{\mathtt{int},\mathtt{bool},\mathtt{string}\}$; $\mathtt{int}^L :: * \mapsto \mathtt{int}$; $\mathtt{conc}^L = \{1\}$; $\mathtt{<=}^L = \{(1,1)\}$; and all other components of L are empty. Its right-hand side is represented by the ST-algebra R: $R_{\mathtt{C}} = \{1\}$; $R_{\mathtt{B}} = \{\mathtt{int},\mathtt{bool},\mathtt{string}\}$; $\mathtt{int}^R :: * \mapsto \mathtt{int}$; $\mathtt{conc}^R = \{1\}$; $\mathtt{<=}^R = \{(1,1)\}$; $R_{\mathtt{T}} = \{2\}$; $R_{\mathtt{Co}} = \{3\}$; $R_{\mathtt{K}} = \{4\}$; $\mathtt{ta}^L :: 3 \mapsto 2$; $\mathtt{t}^L_{\mathtt{Co}} :: 3 \mapsto \mathtt{int}$; $\mathtt{c}^L_{\mathtt{K}} :: 4 \mapsto 3$; $\mathtt{C2T}^L :: 1 \mapsto 2$; and all other components of R are empty. The rule morphism $r : L \to R$ maps as follows: $r_{\mathtt{C}} :: 1 \mapsto 1$; $r_{\mathtt{B}} :: \mathtt{int} \mapsto \mathtt{int}, \mathtt{bool} \mapsto \mathtt{bool}, \mathtt{string} \mapsto \mathtt{string}$; and all other components of the morphism are empty.

Unfortunately, the corresponding transformation system is not terminating for almost all class models. This is due to the fact, that the rules c2t and as2jt are not epic and induce non-epic traces. Therefore, they are not idempotent.

This defect can be avoided by a simple reengineering of ST to the specification ST' in the right part of Specification 3 together with the adapted rules in Fig. 8 and the adapted rule i4ST' in Fig. 9 which are epimorphism and guarantee termination of the transformation system for finite class models, since all rules are idempotent and the specification ST' does not contain any recursive function.

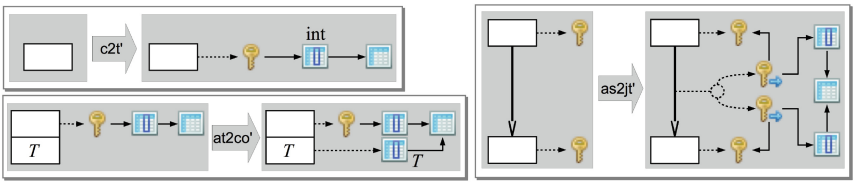


Fig. 8. Epic handling of classes, attributes, and associations in ST (and CT)

[35] The notation indicates that we assume the same carrier for Basetype in CM and RS.

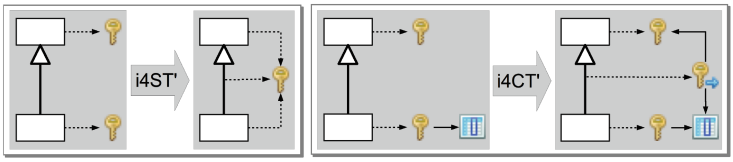


Fig. 9. Epic handling of inheritance in ST and CT

From the practical point of view, `ST'` allows more compact rules (compare Figs. 2 and 9.) and is a more precise description for the mapping of class model items to elements in a relational schema than `ST`: Classes are mapped to keys, since the only feature classes provide for their objects is *object identity*. Attributes are mapped to columns as before. And associations are mapped to foreign key pairs, since it is the key pair that enforces *type conformance* of the association's links in the relational model.

From the theoretical point of view, `ST'` and the corrected, now epic rules are better than the rules in Sect. 2, since they induce an epi-reflection, are idempotent, and, therefore, can easily be analysed wrt. termination.

The two other patterns `CT` and `CCT`, discussed in Sect. 2, can also be turned into epi-reflections. The pattern `CT` differs from `ST` just by the handling of inheritance. The corresponding transformation rule is `i4CT'` in Fig. 9. The necessary mapping of inheritance relations to relational schemata can be provided by a partial operation `I2FK : Inheritance ⟶ ForeignKey`.

Specification 4. CCT as parametric specification

```
Source := CM;   Target := Source +Basetype RS +
opns          C2K:              Class ⟶ Key
           At2Co:   Attribute,Class ⟶ Column
            As2T:        Association ⟶ Table          [x:As::==> As2T(x)]
     AS2FK,AT2FK: Association,Class ⟶ ForeignKey
axms           x:C::conc(c) ==> t(c(C2K(x))) = int                    (a5)
     x:At;y:C;z:T::z = ta(c(C2K(y))), <=(y,o(x)) ==>
                              ta(At2Co(x,y)) = z, t(At2Co(x,y)) = t(x)  (a6)
     x:As,y:C,z:K::<=(y,o(x)), z = C2K(y) ==> r(AS2FK(x,y)) = z,
                        n(c(AS2FK(x,y))), ta(c(AS2FK(x,y))) = As2T(x)  (a7)
     x:As,y:C,z:K::<=(y,t(x)), z = C2K(y) ==> r(AT2FK(x,y)) = z,
                        n(c(AT2FK(x,y))), ta(c(AT2FK(x,y))) = As2T(x)  (a8)
```

Specification 4 presents `CCT` as a parametric specification in the sense of [10].[36] The five rules for the transformation of class models into relation schemata are specified by the total operation `As2T` and the axioms `(a5)` – `(a8)`. The semantics of such a specification is the free construction from `Sys(Source)` to

[36] Again, `Source` and `Target` share sort `Basetype` with defined constant `int`.

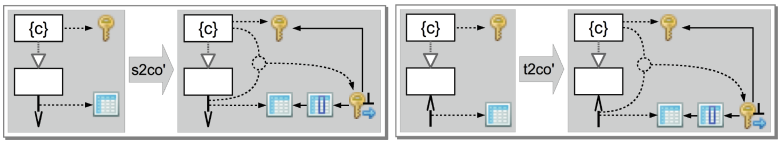


Fig. 10. Epic generation of foreign keys for associations in CCT

Sys(`Target`) wrt. the obvious forgetful functor in the opposite direction. The point-wise construction of these free objects is described in Sect. 3.2.

It is obvious that the presentation of constructive axioms as Horn-formulae is not as suggestive as the presentation as visual transformation rules, compare for example Fig. 10 which graphically depicts the axioms `(a7)` and `(a8)` by rules `s2co'` and `t2co'` respectively. But as we have shown in this paper both variants are semantically equivalent, if the presented morphisms are epic.

6 Summary

In this paper, we discussed the close connection between generating graph rules and the point-wise construction of epi-reflectors for finite constructive axioms. Constructive axioms turned out to be special cases of arbitrary generating rules. And a finite transformation from an arbitrary object o to a final one can be interpreted as the calculation of the epi-reflector of o, if all rules are epimorphisms.

In categories of partial algebras, constructive axioms can (operation-) generate new elements (e.g. rules in Fig. 10), can add new predicate definitions (e.g. rules in Fig. 4) and are able to identify items (e.g. rule `i4ST'` in Fig. 9). Therefore, constructive axioms in categories of partial algebras are suitable for the application area of model transformations for two reasons.

First of all, model transformation rules are typically generating rules, compare for example Triple Graph Grammars (TGG) [2,9], which have been proposed as a standard framework for model transformation. All rules in TGG are generating, especially all sets of derived rules that can be used for model transformation, i.e. forward, backward, source/target, and integration rules. Future research will investigate the connection between TGG and the framework proposed here.

Secondly, a model transformation produces *the derived* target model for a given source model, i.e. the target shall be *uniquely determined* (possibly up to isomorphism) for each source model and it shall be computable in a *finite* number of steps. If the computation is a calculation of an epi-reflector, uniqueness is for free. And, as we showed above, constructive axioms show better termination behaviour than arbitrary rules. We demonstrated these features in this paper by some typical examples. Future research, especially the elaboration of more and bigger transformation examples, will show, if epimorphisms are sufficient for model transformation.

References

1. Adámek, J., Herrlich, H., Strecker, G.E.: Abstract and Concrete Categories - The Joy of Cats (2004). http://katmat.math.uni-bremen.de/acc
2. Anjorin, A., Leblebici, E., Schürr, A.: 20 years of triple graph grammars: a roadmap for future research. ECEASST, 73 (2015)
3. Burmeister, P.: Introduction to theory and application of partial algebras - Part I. Mathematical Research, vol. 32. Akademie-Verlag, Berlin (1986)
4. Chen, P.P.: The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst. **1**(1), 9–36 (1976)
5. Claßen, I., Ehrig, H., Wolz, D.: Algebraic specification techniques and tools for software development: the act approach. AMAST Series in Computing, vol. 1. World Scientific, River Edge (1993)
6. World Wide Web Consortium. Xml Schema, W3C (2012). https://www.w3.org/standards/techs/xmlschema
7. 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
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
9. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation - General Framework and Applications. Springer, Monographs in Theoretical Computer Science. An EATCS Series (2015)
10. Ehrig, H., Mahr, B.: Fundamentals of algebraic specification 1: equations and initial semantics. EATCS Monographs on Theoretical Computer Science., vol. 6. Springer, Heidelberg (1985)
11. Ehrig, H., Mahr, B.: Fundamentals of algebraic specification 2. EATCS Monographs on Theoretical Computer Science, vol. 21. Springer, Heidelberg (1990)
12. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: an algebraic approach. In: FOCS, pp. 167–180. IEEE (1973)
13. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, Boston (2003)
14. Object Management Group: Business Process Model and Notation 2.0.2. OMG (2013). http://www.omg.org/spec/BPMN/index.htm
15. Object Management Group: Unified Modeling Language 2.5. OMG (2015). http://www.omg.org/spec/UML/
16. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. **26**(3/4), 287–313 (1996)
17. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
18. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. **109**(1&2), 181–224 (1993)
19. Löwe, M., Schulz, C.: Algebraic Systems. May 2017. http://ux-02.ha.bib.de/daten/Löwe/Master/TheorieInformationssystem/Algebra20170523.pdf
20. Petri, C.A., Reisig, W.: Petri net. Scholarpedia **3**(4), 6477 (2008)
21. Tempelmeier, M., Löwe, M.: Single-Pushout Transformation partieller Algebren. Technical Report 2015/1, FHDW-Hannover (2015) (in German)

The Verigraph System for Graph Transformation

Guilherme Grochau Azzi, Jonas Santos Bezerra, Leila Ribeiro, Andrei Costa, Leonardo Marques Rodrigues, and Rodrigo Machado(✉)

Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil
{ggazzi,jsbezerra,leila,acosta,lmrodrigues,rma}@inf.ufrgs.br

Abstract. Graph transformation (GT) is a rule-based framework, suitable for modelling both static and dynamic aspects of complex systems in an intuitive yet formal manner. The algebraic approach to GT is based on category theory, allowing the instantiation of theoretical results to multiple graph-like structures (e.g. labelled or attributed graphs, Petri nets, even transformation rules themselves). There exists a rich theory of algebraic GT which underlies verification techniques such as static analysis. Current tools based on GT are implemented in a very concrete way, unlike the theory, making them hard to extend with novel theoretical results. Thus a new software system called *Verigraph* was created, with the goal of implementing the theory as closely as possible, while maintaining a reasonable execution time. In this paper we present the architecture of Verigraph, which enabled an almost direct implementation of the theory. We also provide a step-by-step guide to implementing a new graph model within the system, using second-order graph transformation as an example. Finally, we compare the performance of Verigraph and AGG.

Keywords: Graph transformation · Software system · Static analysis

1 Introduction

Graph transformation is a rule-based framework, suitable for modelling both static and dynamic aspects of complex systems in an intuitive yet formal manner [8,24]. The main idea is to use graphs to specify the states of a system, describing existing entities and their relations at each time of execution, and to model the transitions between such states as *graph rewriting rules*, also called *productions*. These rules describe precisely how the states can be modified. Besides having an intuitive and visual representation, graph transformation has a solid formal background, which enables several analysis techniques.

There are several approaches to describe Graph Transformation (GT) [24], differing on the kinds of graphs that are used and how rules and their application are defined. In some approaches, these notions are defined using set theory. The algebraic approach to graph transformation [8] uses notions of category theory to describe graph transformation rules and rule application. Category theory

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 160–178, 2018.
https://doi.org/10.1007/978-3-319-75396-6_9

provides a language to describe and reason about complex situations at a high level of abstraction. An advantage of this approach is that great part of the rich algebraic GT theory is applicable not only to a particular kind of graph, but also to several different structures such as labelled graphs, typed graphs, attributed graphs [8] and even transformation rules themselves [16]. This is possible since most of the theory is developed at the categorial level, as high-level replacement systems [11] or $\mathcal{M}$-adhesive categories [10]. The main idea is that theory is developed at a very abstract level assuming that the concrete category to which the theory should hold has particular properties. Then, by showing that specific graph categories have these properties, the theory immediately applies to them.

Although most of the algebraic GT theory is done at this abstract level, existing GT tools (AGG [26], Groove [22], among others), are implemented at a rather concrete level: each tool supports only a fixed set of concrete graph models, operations and analysis techniques. Furthermore, their implementation is very far from the formal definitions, hindering arguments about correctness and the construction of extensions to deal with novel approaches.

This led to the creation of the Verigraph System [7], which is open source[1] and implemented in Haskell. It has a current focus on static analysis techniques and the following design goals:

G1. Quick prototyping and experimentation of novel theory
G2. Easy integration of different graph models
G3. Direct implementation of formal concepts at a high level of abstraction, making it easier to reason about correctness
G4. Reasonable execution time.

In this paper we detail the architecture of Verigraph, explaining how the separation of applications (e.g. simulation, static analysis techniques) from concrete graph models (e.g. simple directed graphs, typed graphs, attributed graphs) allows us to achieve goals G1–G3. We also provide a step-by-step guide to implementing custom graph models within the framework, using second-order graph transformation as an example, and provide further evaluation of Verigraph's performance as evidence that goal G4 was achieved.

This paper is organized as follows. Section 2 reviews the theory of algebraic graph transformation. Section 3 presents Verigraph's architecture. To illustrate the flexibility of the system, Sect. 4 provides a step-by-step guide to implementing a graph model within Verigraph. Section 5 provides an overview of currently implemented applications and graph models. Section 6 lists related tools. Section 7 compares the performance of static analysis techniques in Verigraph and AGG. Section 8 provides final remarks and discusses features that are currently under development.

2 Algebraic Graph Transformation

In this section we briefly review the basic definitions of algebraic graph transformation, according to the Double-Pushout (DPO) approach [9]. We follow the

[1] Source code available at https://github.com/Verites/verigraph/.

generalization of DPO to work with objects of any $\mathcal{M}$-adhesive category [10], which include variations of graphs (typed, labelled, attributed), Petri nets and algebraic specifications. The reader is assumed to be familiar with basic concepts of Category Theory. A more detailed introduction to algebraic graph transformation is available in [8].

We begin by reviewing the notion of $\mathcal{M}$-adhesive category, which underlies the other definitions.

Definition 1 ($\mathcal{M}$-adhesive Category). *A category* $\mathbf{C}$ *is called* $\mathcal{M}$*-adhesive, where* $\mathcal{M}$ *is a suitable [10] class of monomorphisms, if*

(i) $\mathbf{C}$ *has pushouts along* $\mathcal{M}$*-morphisms;*
(ii) $\mathbf{C}$ *has pullbacks along* $\mathcal{M}$*-morphisms;*
(iii) pushouts along monomorphisms are van Kampen (VK) squares [14].

These properties ensure that $\mathbf{C}$ has well-behaved pushouts. This equips the category with natural notions for union and intersection of $\mathcal{M}$-subobjects, since it ensures for all objects of $\mathbf{C}$ that their $\mathcal{M}$-subobjects form a distributive lattice. Furthermore, it ensures uniqueness of pushout complements, as described below.

Definition 2 (Pushout Complement)

Given the morphisms $(A \xrightarrow{f} B \xrightarrow{g} C)$ *of a category* $\mathbf{C}$*, a pushout complement of* (f, g) *is a pair of morphisms* $(A \xrightarrow{f'} B' \xrightarrow{g'} C)$ *making the square on the right a pushout.*

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g'}\downarrow & & \downarrow{\scriptstyle g} \\ B' & \xrightarrow[f']{} & C \end{array}$$

Fact 1 (Uniqueness of Pushout Complements). *In an* $\mathcal{M}$*-adhesive category* $\mathbf{C}$*, pushout complements along* $\mathcal{M}$*-morphisms (i.e. when* $f \in \mathcal{M}$ *in the square above) are unique up to isomorphism, when they exist.*

We proceed by reviewing the basic concepts of DPO rewriting for any $\mathcal{M}$-adhesive category $\mathbf{C}$.

Definition 3 (Negative Condition)
A **negative condition** *has the form* $\mathrm{NC}(a)$*, where* $a : P \to C$ *is an arbitrary morphism. We say that a morphism* $m : P \to G$ satisfies the condition *when there is no monomorphism* $q : C \rightarrowtail G$ *with* $q \circ a = m$*, i.e. factoring* m *through* a*. We denote by* $\mathrm{NC}(A)$ *a set of negative conditions, where* A *is a set of morphisms.*

$$\begin{array}{ccc} & & C \\ & {\scriptstyle q}\swarrow & \uparrow{\scriptstyle a} \\ G & \xleftarrow[m]{} & P \end{array}$$

Definition 4 (Double-Pushout Rule). *A* **rule***, also called* production*,* $\rho = (L \xleftarrow{l} K \xrightarrow{r} R, \mathrm{NC}(N))$ *contains a span in* $\mathbf{C}$ *with* $l, r \in \mathcal{M}$*, as well as a set of negative conditions* $N = \{n_i : L \to N_i\}_{i \in I}$*. We call* L *and* R *the* left- *and* right-hand sides*, respectively, while* K *is called the* interface*. The conditions* $\mathrm{NC}(N)$ *are referred to as* Negative Application Conditions *(*NACs*).*

Definition 5 (Match and Transformation Step). *A* **match** *for the rule* $\rho = (L \overset{l}{\hookleftarrow} K \overset{r}{\rightarrowtail} R,\ \text{NC}(N))$ *in the object* G *is any morphism* $m : L \to G$*. A match is* applicable *if it satisfies all* NACs *and* (l, m) *has a pushout complement* $(K \overset{k}{\to} D \overset{l'}{\to} G)$*.*

Given an applicable match $m : L \to G$ *for rule* ρ*, we obtain the* **transformation step** *or* derivation $G \overset{\rho,m}{\Longrightarrow} H$ *by the diagram on the right, where both squares are pushouts.*

$$\begin{array}{ccccc} L & \overset{l}{\hookleftarrow} & K & \overset{r}{\rightarrowtail} & R \\ {\scriptstyle m}\downarrow & & {\scriptstyle k}\downarrow & & {\scriptstyle m'}\downarrow \\ G & \underset{l'}{\hookleftarrow} & D & \underset{r'}{\rightarrowtail} & H \end{array}$$

Definition 6 (Double-Pushout Transformation System). *A* DPO ***transformation system*** *(TS) in the* $\mathcal{M}$*-adhesive category* $\mathbf{C}$ *is a tuple* $\mathcal{G} = (G_0, P)$ *where* G_0 *is a* $\mathbf{C}$*-object, representing the initial state, and* P *a set of rewriting rules.*

The categorial foundation for DPO has enabled the definition of multiple analyses that are also applicable to any $\mathcal{M}$-adhesive category. In particular, critical pair analysis helps understand the control flow that emerges from interacting rules [11]. It is based on parallel independence.

Parallel independence captures the notion that two transformation steps do not interfere with each other, being applicable in any order while still reaching the same result. When they are not parallel-independent, at least one of the steps disables the application of the other, which is called a conflict. In the following discussion, we omit the treatment of NACs due to limited space.

Definition 7 (Parallel Independence). *Given two transformation steps* $G \overset{\rho_1,m_1}{\Longrightarrow} H_1$ *and* $G \overset{\rho_2,m_2}{\Longrightarrow} H_2$*, they are* **parallel-independent** *if there exist morphisms* $q_{12} : L_1 \to D_2$ *and* $q_{21} : L_2 \to D_1$ *making the following diagram commute. The rules are said to be in* **conflict** *when they are not parallel-independent.*

$$\begin{array}{ccccccccccc} R_1 & \overset{r_1}{\leftarrow} & K_1 & \overset{l_1}{\rightarrow} & L_1 & & & & L_2 & \overset{l_2}{\leftarrow} & K_2 & \overset{r_2}{\rightarrow} & R_2 \\ \downarrow & & \downarrow & & & {\scriptstyle q_{21}}\searrow & {\scriptstyle m_1}\searrow & {\scriptstyle m_2}\swarrow & & & \downarrow & & \downarrow \\ H_1 & \leftarrow & D_1 & & \underset{l_1'}{\longrightarrow} & & G & & \underset{l_2'}{\longleftarrow} & & D_2 & \rightarrow & H_2 \end{array}$$

Remark 1. The conflicts characterized by the definition above are called **delete-use** conflicts, because in the context of graphs they detect elements that are deleted by one rule and used by the other. The complete notion of parallel independence in the presence of NACs is more involved, including other kinds of conflicts. A thorough treatment of these notions may be found in [15].

The notion of critical pair captures conflicts in a minimal context. By enumerating all critical pairs, we get an account of all possible interference between two rules. Their formal definition and a thorough discussion is provided in [11].

The categories of graphs and of typed graphs, which are now introduced, are $\mathcal{M}$-adhesive. Thus, the generalized theory of DPO transformation applies to those categories.

Definition 8 (Graph, Graph Morphism). *A* **graph** $G = (V, E, s, t)$ *contains a set* V *of nodes, a set* E *of edges and two functions* $s, t : E \to V$ *mapping each edge into its source and target node, respectively. Given graphs* $G_1 = (V_1, E_1, s_1, t_1)$ *and* $G_2 = (V_2, E_2, s_2, t_2)$*, a* **graph morphism** $f : G_1 \to G_2$ *is a pair of functions* $f = (f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ *that preserve incidence, that is,* $f_V \circ s_1 = s_2 \circ f_E$ *and* $f_V \circ t_1 = t_2 \circ f_E$*.*

Definition 9 (T-typed Graph, T-typed Graph Morphism). *Given a graph* T*, called the* type graph*, a* T**-typed graph** *is a pair* (G, t) *where* G *is the* instance graph *and* $t : G \to T$ *the* typing morphism*. Given two* T*-typed graphs* (G_1, t_1) *and* (G_2, t_2)*, any graph morphism* $f : G_1 \to G_2$ *that preserves typing, i.e. with* $t_2 \circ f = t_1$ *is a* T**-typed graph morphism**.

Definition 10 (Categories of Graphs). *Graphs along with graph morphisms form the category* **Graph**. T*-typed graphs along with their morphisms form the category* $\mathbf{Graph}_T$*. Note that* $\mathbf{Graph}_T$ *is the slice category* $\mathbf{Graph} \downarrow T$*.*

Fact 2. *The categories* **Graph** *and* $\mathbf{Graph}_T$ *are* $\mathcal{M}$*-adhesive, taking as* $\mathcal{M}$ *the class of all monomorphisms [10].*

2.1 Example: Pacman

In this paper we will use the Pacman game as a running example, adapted from [24], especially to discuss second-order graph transformation in Sect. 4. The example is depicted in Fig. 1. We use a typed graph transformation system having 4 types of nodes and 5 types of edges (graph T). Rules describe how Pacman and the ghosts may move (rules movePacman and moveGhost); how a ghost may kill Pacman (rule killPacman, that has a NAC – graph with gray background – stating that Pacman may only be killed if it does not carry a berry); how Pacman may kill a ghost (rule killGhost); and how Pacman may get and drop berries (rules getBerry and dropBerry). Note that only the left- and right-hand sides of rules are shown since in these examples all items that are shown in both sides are preserved (thus graph K is their intersection and rule morphisms are obvious).

3 Architecture

Verigraph was implemented in Haskell [19], a purely functional programming language with mathematical foundations. We exploit its abstraction mechanisms and functional style to reduce the mismatch between theory and code as much as possible.

The core of the system is organized into three layers: The *Abstract* layer is the central part of the architecture, providing high-level categorial and rewriting APIs. The *Application* layer provides a series of analysis techniques mainly implemented over the abstract APIs. Finally, the *Concrete* layer encapsulates the realisation of those APIs for specific categories, such as **Graph** and $\mathbf{Graph}_T$.

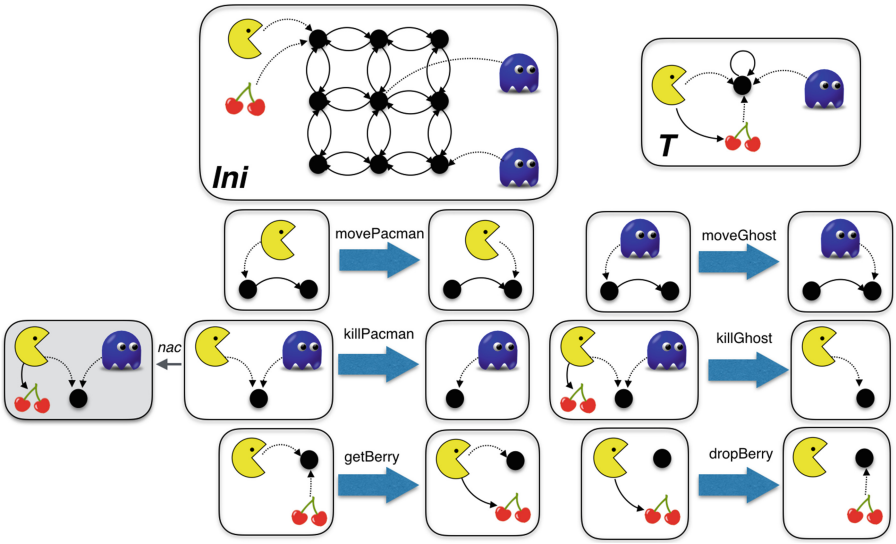


Fig. 1. Pacman transformation system

Figure 2 summarizes the architecture of Verigraph. Folders represent Haskell modules, while simple boxes represent data types or, when text is in italic, type classes. Dashed arrows indicate dependencies between modules. A thorough documentation of Verigraph's modules is available online[2].

In the remainder of this section, we explain the design of each layer in detail, and how they are used to achieve a generic and extensible architecture that is also very close to theory.

3.1 The Abstract Layer

The abstract layer is responsible for defining basic constructions and operations for category theory and rewriting systems. This is mainly accomplished by a series of contracts, in the form of Haskell type classes, that specify abstract categorial operations. The application layer can then be largely generic with respect to the category, depending only on these contracts. The actual implementation of most operations is left for the concrete layer, though some operations have a default implementation in terms of other categorial constructs (e.g. pushout as coproduct and coequalizer).

Being the layer that directly expresses categorial concepts, its operations closely reflect categorial definitions. In Verigraph, a category is described by the type class `Category`, shown in Fig. 3. It defines the basic structure and operations that any category implemented in Verigraph must provide. We omit some details of the type classes due to space restrictions.

[2] API documentation available at https://verites.github.io/verigraph-docs/.

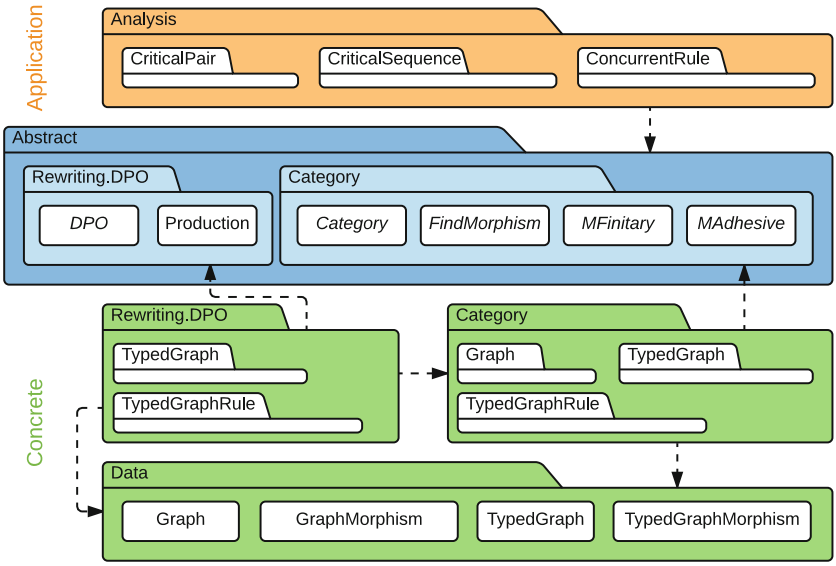


Fig. 2. Verigraph architecture

Essentially, a category must have an *object type*[3], a *morphism composition* operation (`<&>`), a function that returns the *identity* morphism of an object, together with functions that return the *domain* and *codomain* objects of a morphism.

However, not all categories are suitable for all purposes. In general, the theory of rewriting assumes that categories have enough structure to enable particular constructions. This is also specified by type classes, such one for $\mathcal{M}$-adhesive categories, which have pushouts and pullbacks along monomorphisms, as shown in Fig. 3.

The categorial portion of this module provides further type classes. For example, `FindMorphism` defines the operations for finding all morphisms between a pair of objects, possibly restricting to a specific class or satisfying restrictions (e.g. make a span or a cospan commute). Similarly, `E'PairCofinitary` deals with enumerating jointly epic pairs of morphisms with given domains.

Besides the high-level categorial framework, the abstract layer also contains APIs for each rewriting approach. This is kept in a different submodule to decouple categorial operations from the rewriting approaches.

Currently, the only rewriting approach implemented in the stable version of Verigraph is DPO, although new approaches are being studied and have ongoing implementation, particularly Sesqui-Pushout [6] and AGREE [5].

[3] We use an extension of Haskell's type system enabling *type families* to associate a type of object to each type of morphism.

```
class Eq morph => Category morph where
    type Obj morph :: *
    (<&>)    :: morph -> morph -> morph
    identity :: Obj morph -> morph
    domain   :: morph -> Obj morph
    codomain :: morph -> Obj morph

class Category morph => MAdhesive morph where
  calculatePushoutAlongM           :: morph -> morph -> (morph, morph)
  calculatePullbackAlongM          :: morph -> morph -> (morph, morph)
  hasPushoutComplementAlongM       :: morph -> morph -> Bool
  calculatePushoutComplementAlongM :: morph -> morph -> (morph, morph)
```

Fig. 3. Type classes for categories and $\mathcal{M}$-adhesive categories.

```
data Production morph = Production
  { left  :: morph
  , right :: morph
  , nacs  :: [morph] }
```

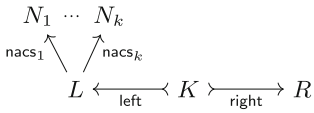


Fig. 4. Data type for DPO productions (see Definition 4)

```
calculateDPO :: MAdhesive morph => morph -> Production morph
                -> (morph, morph, morph, morph)
calculateDPO m (Production l r _) =
  let (k, l') = calculatePushoutComplementAlongM l m
      (r', m') = calculatePushoutAlongM r k
  in (k, m', l', r')
```

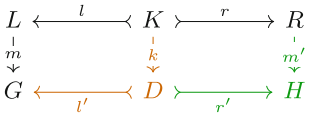


Fig. 5. Implementation of DPO rewriting steps (see Definition 5)

Similar to the categorial APIs, those for rewriting provide basic data structures and operations of their underlying approaches, often defined in terms of categorial operations. Examples are data types for rules and transformation systems, along with functions for checking if matches are applicable, performing rewritings, etc. Figure 4 shows the data definition for productions in the DPO approach and Fig. 5 shows a function that, given a production and an applicable match, calculates the transformation step.

3.2 The Application Layer

The application layer provides a collection of analysis techniques, mainly based on the categorial and rewriting APIs. An example is *critical pair analysis*,

described in Sect. 2. It is implemented at a high level of abstraction, directly depending only on the `DPO` and `E'PairCofinitary` type classes (which in turn depend on `FindMorphism`).

Thus, critical pair analysis is available for any category that conforms to the aforementioned type classes. Figure 6 shows the main functions that implement critical pair analysis. Once again, we omit the treatment of NACs due to limited space. The code also has been slightly simplified for readability and space.

```
isDeleteUse :: (DPO morph, E'PairCofinitary morph) =>
    Production morph -> (morph, morph) -> Bool
isDeleteUse p1 (m1,m2) =
  let (_,l1') = calculatePushoutComplementAlong (leftMorphism p1) m1
  in null (findCospanCommuters m2 l1')

findAllDeleteUse :: (DPO morph, E'PairCofinitary morph) =>
    Production morph -> Production morph -> [(morph, morph)]
findAllDeleteUse p1 p2 =
  let
    allPairs = createJointSurjections (leftMorphism p1) (leftMorphism p2)
    satisfyingPairs = filter (satisfyRewritingConditions p1 p2) allPairs
  in filter (isDeleteUse p1) satisfyingPairs
```

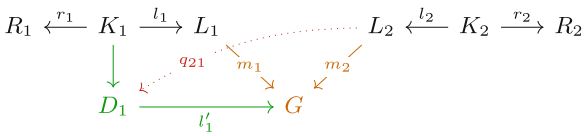


Fig. 6. Functions that calculate critical pairs for delete-use (see Definition 7 and Remark 1)

In the code snippet shown in Fig. 6, the function `findJointSurjections` enumerates all jointly epic pairs of morphisms that have the left-hand sides of either rule as domain. Function `satisfyRewritingConditions` tests the existence of pushout complements for a pair of matches. Function `findCospanCommuter` finds all morphisms $q_{21} : L_2 \to D_1$ such that $g_1 \circ q_{21} = m_2$.

Although the main focus of this layer is to be generic and based on the abstract layer, there exist analyses that are tied to the internal structure of a particular category and/or its objects. For such situations, Verigraph allows the application layer to directly access features of the concrete layer, at the cost that these particular analyses will not be available to other categories. Some examples are given in Sects. 4 and 5.

3.3 The Concrete Layer

The concrete layer deals with the implementation of particular categories, i.e. the data structures and category-specific operations for objects and morphisms (e.g.

applying a morphism to a node or edge), as well as the realisation of contracts from the abstract layer, such as operations for finding and composing morphisms, pushouts and pullbacks.

This layer is split into three modules, as shown in Fig. 2. The `Data` module provides basic data structures, while the `Category` and `Rewriting` modules provide instantiations of the appropriate type classes for those data structures.

Currently, there are three main categories implemented in Verigraph: **Graph**, $\mathbf{Graph}_T$ (see Definition 10) and $\mathbf{GraphRule}_T$ (which will be described in Sect. 4). A category for typed attributed graphs is under development. Figure 7 shows data structures representing objects and morphisms of the first two categories.

In order to use the existing generic analyses with a different category, this is the only layer that needs to be changed, as explained in Sect. 4. One of the main advantages is that new programmers do not have to worry about too many categorial details, as long as they implement the basic operations defined in the upper layers. Figure 8 shows the instantiation of category $\mathbf{Graph}_T$.

Another advantage of this architecture is that optimizations dependent on the internal representation of objects and morphisms can be done here without

```
data Graph n e = Graph
    { nodeMap :: [(NodeId, Node n)]
    , edgeMap :: [(EdgeId, Edge e)] }

data GraphMorphism a b = GraphMorphism
  { domainGraph   :: Graph a b
  , codomainGraph :: Graph a b
  , nodeRelation  :: Relation NodeId
  , edgeRelation  :: Relation EdgeId }

type TypedGraph a b = GraphMorphism a b

data TypedGraphMorphism a b = TypedGraphMorphism
  { domainGraph   :: TypedGraph a b
  , codomainGraph :: TypedGraph a b
  , mapping       :: GraphMorphism (Maybe a) (Maybe b) }
```

Fig. 7. Implementation of typed graphs and their morphisms

```
instance Category (TypedGraphMorphism a b) where
  type Obj (TypedGraphMorphism a b) = TypedGraph a b
  domain     = domainGraph
  codomain   = codomainGraph
  t2 <&> t1  = TypedGraphMorphism (domainGraph t1) (codomainGraph t2)
                                  (mapping t2 <&> mapping t1)
  identity t = TypedGraphMorphism t t (identity (domain t))
```

Fig. 8. `Category` instance for $\mathbf{Graph}_T$

compromising the clarity of the abstract operations or tying them to a particular category. An example are optimized search procedures for morphisms that satisfy particular restrictions, such as commuting with a particular span or cospan.

Finally, despite being focused on graphs, Verigraph is not necessarily limited to them. Therefore, any category with the necessary properties can be implemented in the system, such as sets, Petri nets, algebraic specifications, etc.

4 Implementing a Graph Model in Verigraph

In this section, we provide a step-by-step guide to implementing a new graph model in Verigraph. Note that all implementations shown in this section belong to the concrete layer. Each step will be illustrated by describing the implementation of Second-Order Graph Grammars (SOGGs), which allow the transformation of graph rewriting rules using DPO transformation [17]. In the context of Model-Driven Engineering, SOGGs are well-suited to analyse changes introduced during the evolution or maintenance phase of development.

4.1 Step 1: Define the Graph Model as an $\mathcal{M}$-adhesive Category

In order to integrate a graph model into Verigraph's architecture, it must be defined as an $\mathcal{M}$-adhesive category. The notions of object and morphism must be clear, as well as constructions such as (co)limits, initial pushouts, $\mathcal{E}$-$\mathcal{M}$ factorization and $\mathcal{E}'$-$\mathcal{M}$ pair factorization.

Note that most of the implemented categories are finitary, that is, each object has a finite number of subobjects. In this case, $\mathcal{M}$-adhesiveness guarantees $\mathcal{E}$-$\mathcal{M}$ factorization and the existence of initial pushouts, and a strict initial object additionally guarantees finite coproducts and $\mathcal{E}'$-$\mathcal{M}$ pair factorization [4].

Second-order graph transformations are defined in the category of typed graph spans, which is $\mathcal{M}$-adhesive under certain assumptions [16].

Graph rewriting rules, in the DPO approach, are spans $L \hookleftarrow K \rightarrowtail R$ in $\mathbf{Graph}_T$. Thus, the following category is appropriate to model the rewriting of graph rules.

Definition 11. *The category* $\mathbf{GraphRule}_T$ *has, as objects, monic spans* $L \hookleftarrow K \rightarrowtail R$ *of* $\mathbf{Graph}_T$*. A* **rule morphism** $f : \alpha \to \beta$ *is then a triple of graph morphisms* (f_L, f_K, f_R) *between the graphs of both rules making the diagram on the right commute.*

$$\begin{array}{ccccc} L_\alpha & \xleftarrow{l_\alpha} & K_\alpha & \xrightarrow{r_\alpha} & R_\alpha \\ \downarrow{\scriptstyle f_L} & & \downarrow{\scriptstyle f_K} & & \downarrow{\scriptstyle f_R} \\ L_\beta & \xleftarrow[l_\beta]{} & K_\beta & \xrightarrow[r_\beta]{} & R_\beta \end{array}$$

Fact 3. $\mathbf{GraphRule}_T$ *is* $\mathcal{M}$*-adhesive [16]. Its limits, colimits and pushout complements can be constructed componentwise in* $\mathbf{Graph}_T$*.*

Since $\mathbf{GraphRule}_T$ is $\mathcal{M}$-adhesive, the framework of DPO rewriting can be instantiated for it. Thus, ***second-order rules***, also called *2-rules*, are spans of

rule morphisms. Figure 9 shows two examples of 2-rules for the Pacman transformation system. Note that the interfaces are omitted from first- and second-order rules, since all items shown in both sides are preserved (thus the interfaces are the intersection of the left- and right-hand sides). The 2-rule *noViolence* transforms a deletion of *Pacman* into its preservation, maintaining the deletion of the edge linking *Pacman* to a *block*. The 2-rule *fastGhost* adds a new preserved *block* in a rule that moves a *ghost*, allowing it to move two blocks at a time instead of just one.

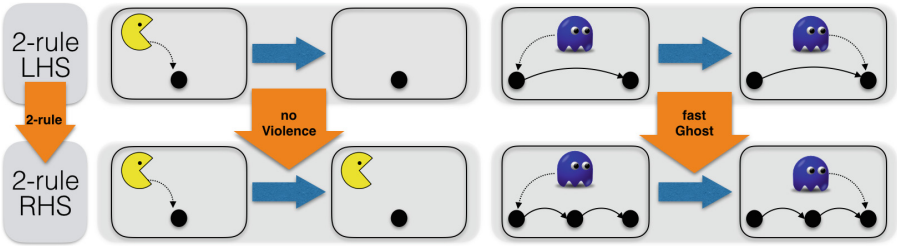


Fig. 9. Pacman second-order rules

4.2 Step 2: Implement Data Structures for Objects and Morphisms

The building blocks for DPO transformation are the objects and morphisms of the category. Thus, the design of data structures that represent them is crucial to ensure reasonable runtime and memory consumption.

The category $\mathbf{GraphRule}_T$ is particularly simple, since it can reuse the data structures that implement $\mathbf{Graph}_T$. The types of graph rules and their morphisms are show in Fig. 10, and they directly reflect Definition 11.

```
type TypedGraphRule n e = Production (TypedGraphMorphism n e)

data RuleMorphism n e = RuleMorphism
  { rmDomain          :: TypedGraphRule n e
  , rmCodomain        :: TypedGraphRule n e
  , mappingLeft       :: TypedGraphMorphism n e
  , mappingInterface  :: TypedGraphMorphism n e
  , mappingRight      :: TypedGraphMorphism n e }
```

Fig. 10. `RuleMorphism` implementation

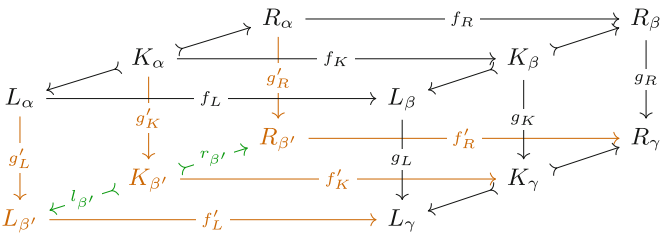
4.3 Step 3: Instantiate the Appropriate Type Classes

According to the constructions available in the category, the type classes of the abstract layer should be instantiated, that is, their operations should be

implemented for the objects and morphisms of this category. The instantiated type classes will define which applications are available to the implemented graph model. The correspondence between each type class and the categorial concepts it embodies are explained in the API documentation[4].

Many of the type classes provide default implementations for some of the operations, which are correct but inefficient. These are useful for early prototypes, but careful implementation of all operations is important to ensure reasonable runtime and memory consumption.

In the case of $\mathbf{GraphRule}_T$ all of the aforementioned type classes are instantiated. Indeed, most of the algorithms are constructed over $\mathbf{Graph}_T$ primitives since limits, colimits and other constructions on $\mathbf{GraphRule}_T$ can be calculated componentwise. An example is the implementation of pushout complements, shown in Fig. 11. The function receives two `RuleMorphisms` $f : \alpha \to \beta$




```
instance MAdhesive (RuleMorphism a b) where
  calculatePushoutComplementAlongM
    g@(RuleMorphism _ ruleC fL fK fR)
    f@(RuleMorphism ruleA ruleB gL gK gR) =
    let
      (gL', fL') = calculatePushoutComplementAlongM fL gL
      (gK', fK') = calculatePushoutComplementAlongM fK gK
      (gR', fR') = calculatePushoutComplementAlongM fR gR

      leftB'  = commutingMorphismSameCodomain
                (leftMorphism ruleC <&> fK') fL'
                 gK' (gL' <&> leftMorphism ruleA)
      rightB' = commutingMorphismSameCodomain
                (rightMorphism ruleC <&> fK') fR'
                 gK' (gR' <&> rightMorphism ruleA)

      ruleB' = Production leftB' rightB' []
    in
      ( RuleMorphism ruleA ruleB' gL' gK' gR',
        RuleMorphism ruleB' ruleC fL' fK' fR' )
```

Fig. 11. Implementation of *Pushout Complement* for `RuleMorphism`

[4] API documentation available at https://verites.github.io/verigraph-docs/.

and $g : \beta \to \gamma$, returning its pushout complement ($f' : \alpha \to \beta'$, $g' : \beta' \to \gamma$) (see Definition 2). The first part of the function performs pushout complements in $\mathbf{Graph}_T$, shown in orange in the previous diagram. This implicitly constructs the typed graphs of rule β' as codomains of g'_L, g'_K and g'_R. The second part searches for graph morphisms $l_{\beta'}$ and $r_{\beta'}$ that make the diagram commute. Finally, the rule β' and the rule morphisms f' and g' are assembled from their components.

The complete realisation of `MAdhesive`, `DPO` and `E'PairCofinitary` type classes for `RuleMorphism` enables the usage of many generic applications of Verigraph with 2-rules, including critical pair analysis and the calculation of concurrent rules.

4.4 Step 4 (Optional): Implement Category-Specific Applications

There are often applications that depend on details of the graph model, and cannot or were not generalized categorially. These may be still be implemented within Verigraph, although they may not profit from the separation of the abstract layer.

In the case of SOGGs, inter-level critical pair analysis detects situations where the applicability of first-order rules is changed after applying a second-order rule. This operation is inherent to the transformation of graph rules, and was implemented in Verigraph.

4.5 Step 5: Adapt the Command-Line Interface

In order for end users to execute applications over the new graph model, some functionality must still be implemented, such as: reading files that define transformation systems of the graph model, writing files that describe the results, interpreting configuration options. This needs to be implemented separately for each graph model, since the syntax of input/output files and the available options vary greatly. It also needs to be integrated into the executable providing the command-line interface. This step is unrelated to the theory of algebraic graph transformation, so it is beyond the scope of this paper.

5 Overview of Implemented Techniques

Verigraph already provides several applications for end users. It allows the execution of first- and second-order rewriting rules over typed graphs (i.e. transformation of typed graphs and of graph rules). Its current main focus, however, are static analysis techniques. Attributed graphs are not yet supported. At the present time, Verigraph provides a Command Line Interface, using AGG [26] `.ggx` and `.cpx` files as input/output formats.

Many of the implemented features are not specific to first- and second-order graph transformation, being implemented generically with respect to the transformed structure. This includes the execution of second-order rules, which may be applied to first-order transformation of any structure. Other generic applications are the following static analysis techniques described in [8].

Critical Pair Analysis: Captures all possibilites of conflicts between rules in a minimal context.
Critical Sequence Analysis: Similar to critical pair analysis, captures all possibilites of dependencies between rules in a minimal context.
Concurrent Rules Calculation: Generates rules that summarize the application of several rules in a single step.

As for the applications available to a single kind of structure, Verigraph provides:

Inter-level Critical Pairs Analysis (GraphRule$_T$) [17]: Detects conflicts between first- and second-order graph rules, i.e. whether and how the application of second-order rules affects the applicability of first-order rules. This is useful for analysing software evolution.
Occurrence Grammar Calculation (Graph$_T$) [23]: Generates doubly-typed graph grammars that describe the semantics of typed graph grammars and their application history. These can be used for the generation of test cases.

Another important aspect of DPO transformation systems are Graph Constraints, which are also supported by Verigraph. They are not described in this paper due to lack of space.

6 Related Work

Existing tools based on graph transformation (GT) vary according to the supported graph models, transformation approaches, execution models and analysis techniques. Unlike Verigraph, most of them are domain-specific, but other domain-neutral tools based on the algebraic approach include the following.

AGG [26] supports typed attributed graphs, allowing execution of transformation rules, critical pair analysis and calculation of concurrent rules.
GROOVE [22] supports a model of labelled graphs with types and attributes, allowing execution of transformation rules, state space exploration and model checking.
GrGEN.NET [12] supports typed attributed graphs, allowing the compilation of transformation rules into C# for efficient execution. It also provides a domain-specific language for controlling the application of rewrite rules.
Graph Programs [18] is a programming language containing transformation rules of labeled graphs as primitive statements. A compiler is available, generating bytecode for the York abstract machine. Although a reasoning system based on Hoare logic was proposed for verifying Graph Programs [21], tool support is not yet available.

A particular application domain where many transformation tools are available is Model-Driving Engineering (MDE). In this setting, transformation rules generally manipulate graph-based models such as those from the Unified Modeling Language (UML) or the Eclipse Modeling Framework (EMF). Besides DPO

rewriting, Triple Graph Grammars (TGG) [25] are often used to define bidirectional model transformations. **EMorF** [13] supports in-place model modification rules (based on DPO), as well as model transformation and synchronization (based on TGG) for EMF models, allowing execution of transformation rules. **eMoflon** [1] supports Story Driven Modeling (a combination of UML Activity Diagrams and DPO) as well as TGG for EMF models, allowing compilation into Java code. **Henshin** [2] supports transformation rules for EMF models, along with control-flow constructs to guide their execution. It allows execution of rules, compilation into Java, state space exploration integrated with model-checking tools and critical pair analysis (using AGG as a component).

7 Performance Evaluation

One of the design goals of Verigraph was a reasonable execution time. In order to validate the satisfaction of this goal, we have performed some experiments comparing the execution time of static analysis techniques on Verigraph and AGG [26].

We have selected five transformation systems (TS) to use as inputs of the experiment. PACMAN is the TS presented in Fig. 1. ELEV models the behaviour of an elevator system with 9 rules, adapted from [15]. MED1, MED2 and MED3 model guidelines for a medical procedure, containing 36 rules in total [3]. PACMAN and MED1 contain only relatively small graphs, with at most 5 nodes, and at most 2 of the same type. ELEV contains slightly larger graphs, with up to 7 nodes and 3 of the same type. MED2 and MED3 have even larger graphs, with up to 8 nodes and 2 of the same type.

The experiment consisted of running critical pair and sequence analysis over the TSs, that is, calculating all critical pairs and sequences for each pair of rules in each TS. The analysis was executed 10 times for each TS, with each tool, in an Intel I5-3330 processor running at 3 GHz with 16 GiB of RAM. The total time and peak memory usage was measured for each execution using GNU `time`. Although the measurements include reading and writing XML files, this overhead should be similar for both tools, and both execution time and memory usage should be dominated by the analysis itself. Furthermore, the results directly reflect the experience of end users. It is also important to note that both AGG and Verigraph parallelize the analysis. All input files and the scripts used for running the tests are available at https://github.com/Verites/verigraph/tree/critical-pairs-benchmarks.

Table 1 presents the number of critical pairs and sequences found for each TS, and Table 2 presents the measured execution times. For the TSs PACMAN, MED1 and ELEV, Verigraph's performance was slightly better than AGG. On MED2 and MED3 AGG's performance degraded substantially, and it was significantly outperformed by Verigraph. This indicates that AGG is more sensitive to the size of the graphs contained in rewriting rules, since the last two TSs contained the largest graphs. Memory usage is shown in Table 3, and followed a similar pattern.

Table 1. Number of critical pairs and sequences for each transformation system.

	PACMAN	ELEV	MED1	MED2	MED3
Rules	6	9	9	11	12
Critical pairs	135	328	85	858	1462
Critical sequences	146	242	16	229	272

Table 2. Average and standard deviation for execution times of critical pair and sequence analysis, in seconds.

Tool	PACMAN		ELEV		MED1		MED2		MED3	
	avg	dev	avg	dev	avg	dev	avg	dev	avg	dev
Verigraph	3.82	0.07	13.21	0.09	3.55	0.06	14.94	0.18	35.84	0.30
AGG	4.90	0.05	15.16	0.12	5.49	0.05	1233.10	12.91	1375.62	16.43

Table 3. Average and standard deviation for peak memory usage of critical pair and sequence analysis, in MiB.

Tool	PACMAN		ELEV		MED1		MED2		MED3	
	avg	dev	avg	dev	avg	dev	avg	dev	avg	dev
Verigraph	65.50	2.27	157.88	4.12	47.37	2.02	434.34	15.70	764.60	23.52
AGG	180.70	7.66	287.00	4.74	112.65	1.10	8660.15	299.06	8739.41	229.26

8 Conclusion

Verigraph is a new system for Graph Transformation (GT) that exploits the use of category theory to promote flexibility and extensibility. The use of category theory as a basis allows not only for flexibility, but also provides a framework in which formal definitions of algebraic GT can be implemented in a rather straightforward way. Conceptually, this idea is similar to the use of institutions as a basis for the HETS tool [20].

The flexibility of Verigraph was demonstrated by the implementation of second-order graph transformation: by instantiating a categorial API for $\mathbf{GraphRule}_T$, existing applications like critical pair analysis are automatically available for second-order graph rewriting. Moreover, we have shown that it is also possible to develop category-specific applications, like Interlevel Conflict Analysis (which requires at least two levels of rewriting rules).

Despite flexibility and extensibility, a reasonable execution time can be achieved by an efficient implementation of the categorial API. This was demonstrated by comparing execution time of critical pair analysis on Verigraph and AGG, having Verigraph outperform AGG in realistic test cases.

Currently Verigraph implements rewriting under the DPO approach using the categories $\mathbf{Graph}_T$ and $\mathbf{GraphRule}_T$, as well as critical pair/sequence

analysis and calculation of concurrent rules. We are working on support for (typed) attributed graphs, as well as the Sesqui-Pushout and AGREE transformation approaches. Research is also being done on applying Verigraph to generate test cases for software that is modeled with graph transformation.

The existing version of Verigraph uses AGG files for input and output via a Command Line Interface. Nonetheless, Verigraph's own Graphical User Interface is under development, following a web-based approach completely decoupled from the system's code.

A flexible architecture makes Verigraph suitable as a platform for testing new ideas in Graph Transformation. The system is free and open source, currently available online at GitHub[5], which allows collaborative development and discussion. Extensive automated testing helps maintain its correctness, and its API is thoroughly documented. All these aspects could enable the use and development of Verigraph by the community of researchers.

References

1. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon: leveraging EMF and professional CASE tools. Informatik **192**, 281 (2011)
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. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_9
3. Bezerra, J.S., Costa, A., Ribeiro, L., Cota, É.F.: Formal verification of health assessment tools: a case study. Electron. Notes Theor. Comput. Sci. **324**, 31–50 (2016)
4. Braatz, B., Ehrig, H., Gabriel, K., Golas, U.: Finitary M-adhesive categories. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 234–249. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15928-2_16
5. 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
6. 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
7. Costa, A., Bezerra, J., Azzi, G., Rodrigues, L., Becker, T.R., Herdt, R.G., Machado, R.: Verigraph: a system for specification and analysis of graph grammars. In: Ribeiro, L., Lecomte, T. (eds.) SBMF 2016. LNCS, vol. 10090, pp. 78–94. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49815-7_5
8. 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

[5] https://github.com/Verites/verigraph/.

9. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: an algebraic approach. In: Switching and Automata Theory, pp. 167–180 (1973)
10. Ehrig, H., Golas, U., Hermann, F., et al.: Categorical frameworks for graph transformation and HLR systems based on the DPO approach. Bull. EATCS **102**, 111–121 (2010)
11. 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
12. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: a fast SPO-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006). https://doi.org/10.1007/11841883_27
13. Klassen, L., Wagner, R.: EMorF - a tool for model transformations. Electron. Commun. EASST **54**, 1–6 (2012)
14. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO - Theor. Inf. Appl. **39**(3), 511–545 (2005)
15. Lambers, L.: Certifying rule-based models using graph transformation. Ph.D. thesis, Elektrotechnik und Informatik der Technischen Universität Berlin (2010)
16. Machado, R.: Higher-order graph rewriting systems. Ph.D. thesis, Instituto de Informática - Universidade Federal do Rio Grande do Sul (2012)
17. Machado, R., Ribeiro, L., Heckel, R.: Rule-based transformation of graph rewriting rules: towards higher-order graph grammars. Theor. Comput. Sci. **594**, 1–23 (2015)
18. Manning, G., Plump, D.: The GP programming system. Electron. Commun. EASST **10**, 1–13 (2008)
19. Marlow, S.: Haskell 2010 language report (2010). https://www.haskell.org/onlinereport/haskell2010/
20. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, Hets. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_40
21. Poskitt, C.M.: Verification of graph programs. Ph.D. thesis, University of York (2013)
22. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_40
23. Ribeiro, L.: Parallel composition and unfolding semantics of graph grammars. Ph.D. thesis, Technical University of Berlin (1996)
24. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1, Foundations. World Scientific Publishing Co., Inc., River Edge (1997)
25. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59071-4_45
26. Taentzer, G.: AGG: a tool environment for algebraic graph transformation. In: Nagl, M., Schürr, A., Münch, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 481–488. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45104-8_41

Decomposition Structures for Soft Constraint Evaluation Problems: An Algebraic Approach

Ugo Montanari[1], Matteo Sammartino[2(✉)], and Alain Tcheukam[3]

[1] University of Pisa, Pisa, Italy
[2] University College London, London, UK
m.sammartino@ucl.ac.uk
[3] New York University, Abu Dhabi, United Arab Emirates

Abstract. (Soft) Constraint Satisfaction Problems (SCSPs) are expressive and well-studied formalisms to represent and solve constraint-satisfaction and optimization problems. A variety of algorithms to tackle them have been studied in the last 45 years, many of them based on dynamic programming. A limit of SCSPs is its lack of compositionality and, consequently, it is not possible to represent problem decompositions in the formalism itself. In this paper we introduce *Soft Constraint Evaluation Problems* (SCEPs), an algebraic framework, generalizing SCSPs, which allows for the compositional specification and resolution of (soft) constraint-based problems. This enables the systematic derivation of efficient dynamic programming algorithms for any such problem.

1 Introduction

(Soft) Constraint Satisfaction Problems (SCSPs) are expressive and well-studied formalisms [20,24] to represent and solve constraint-satisfaction and optimization problems [4]. A CSP consists of a network of hyperedges, interpreted as predicates on (variables associated to) the adjacent vertices. A *solution* is a variable assignment satisfying all the predicates (or providing a "best" level of satisfaction, in the soft version).

Finding a solution for an SCSP is in general an NP-complete problem. A variety of algorithms have been studied in the last 45 years, many of them based on *dynamic programming* [2]. Dynamic programming is a well-known method for solving optimization problems. It consists in: (a) decomposing repeatedly the problem into smaller subproblems; (b) solving subproblems in a bottom-up order, by combining solutions of smaller problems into those of bigger problems.

Key to the approach is the fact that repeated subproblems are only solved once. Different decompositions can have substantially different computational costs, and choosing a best one is known as *secondary optimization problem* of dynamic programming [3]. This is also an NP-complete problem. When the problem has a graphical representation, as in the case of CSPs, a class of tree-shaped structures, called *tree decompositions* [19,22], have been used to represent dynamic programming hierarchies. The solution process corresponds to a

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 179–200, 2018.
https://doi.org/10.1007/978-3-319-75396-6_10

bottom-up visit of the tree decomposition (see e.g. [12] for algorithms for CSPs based on tree decompositions).

A limit of SCSPs is the lack of compositionality and, consequently, of mechanisms to represent problem decompositions for dynamic programming in the formalism itself. In this paper we introduce a new, compositional framework for a wide class of constraint-based problems, which we call *Soft Constraint Evaluation Problems* (SCEPs), generalizing SCSPs. In this framework, both the *structure* and the *solution process* can be represented at the same time, with a formal connection between the two. This provides a correct-by-construction mechanism to decompose and solve SCEPs via dynamic programming.

SCEPs are specified via a simple syntax inspired by process algebras, with a natural interpretation in terms of constraints. As an example, the term:

$$p = (y)((x)A(x, y) \parallel (z)B(y, z))$$

represents a problem made of two constraints A and B, over x, y and y, z respectively, where $(\_)$ precedes $\_ \parallel \_$. Notice that y is shared.

The syntax is expressive enough to represent both the structure of the problem and a *decomposition* into subproblems. For instance, $A(x, y)$ being in the scope of (x) means that it must be solved w.r.t. x, which will produce a solution parametric in y. A fundamental role is played by the *axiom of scope extension*

$$(x)(p \parallel q) = (x)p \parallel q \qquad (x \text{ not free in } q)$$

which allows for the manipulation of the subproblem structure of terms.

Given an SCEP, represented as the term p defined above, its *solution* is just the evaluation of p in a given SCEP algebra, i.e., an algebra providing an interpretation of basic constraints and operations. In other words, the solution can be computed via structural recursion on terms, using the interpreted operations. For instance, in a typical optimization problem, $\_ \parallel \_$ is interpreted as summing up each subproblem's contribution, e.g., cost, and $(x)\_$ as minimizing w.r.t. the variable x.

A key challenge here is achieving structural recursion in the presence of variable binding, such as the restriction operator (x) described above. In fact, if treated naively, variable binding leads to possibly ill-defined recursive definitions, where notions such as "free/bound variable" and "variable capture" need to be consistently taken into account. To tackle this, SCEP algebras are *permutation algebras* [15], including explicit variable permutations that enable a proper treatment of free and bound variables. This approach is equivalent to abstract syntax with binding via nominal sets (see, e.g., [21]).

The main contributions of this paper are as follows:

– In Sect. 3 we propose a *strong* axiomatization of SCEPs, and we present one of the main results of the paper: soundness and completeness of constraint networks w.r.t. our strong specification, namely networks form its initial algebra. Then we introduce a *weak* specification, where each term describes a specific decomposition. This enables decomposing and solving SCEPs, and in particular traditional constraint networks, in a unified framework.

- In Sect. 4 we show how SCSPs are an instance of SCEPs.
- In Sect. 5 we introduce the notion of *complexity* of term evaluation, and we characterize terms that are local optima w.r.t. complexity.
- In Sect. 6 we give a formal translation from tree decompositions to weak terms, which enables applying algebraic techniques to the former, and improving their complexity via the results of Sect. 5.
- In Sect. 7 we give a simple algorithm, inspired by *bucket elimination* [23, Sect. 5.2.4]. We show that our algorithm can achieve better decompositions than the latter one.
- Finally, in Sect. 8 we give a non-trivial example of a problem which can be represented and solved as an SCEP, but not as an SCSP.

2 Background

We recall some basic notions. A *ranked alphabet* $\mathcal{E}$ is a set equipped with an *arity* function $ar\colon \mathcal{E} \to \mathbb{N}$. A *labelled hypergraph* over a ranked alphabet $\mathcal{E}$ is a tuple $G = (V_G, E_G, a_G, lab_G)$, where: V_G is the set of vertices; E_G is the set of (hyper)edges; $a_G\colon E_G \to V_G^\star$ assigns to each hyperedge e the tuple of vertices attached to it ($V_G^\star$ is the set of tuples over V_G); $lab_G\colon E_G \to \mathcal{E}$ is a labeling function, assigning a label to each hyperedge e such that $|a_G(e)| = ar(lab_G(e))$.

Given two hypergraphs G_1 and G_2 over $\mathcal{E}$, a *homomorphism* between them is a pair of functions $h = (h_V\colon V_{G_1} \to V_{G_2}, h_E\colon E_{G_1} \to E_{G_2})$ preserving connectivity and labels, namely: $h_V \circ a_{G_1} = a_{G_2} \circ h_E$ and $lab_{G_2} \circ h_E = lab_{G_1}$. It is an *isomorphism* whenever h_V and h_E are bijections. We write $G_1 \uplus G_2$ for the component-wise disjoint union of G_1 and G_2.

2.1 Soft Constraint Satisfaction Problems

Let $\mathbb{V}$ be a denumerable set of variables and let $\mathcal{E}_C$ be a ranked alphabet of *soft constraints* (or just constraints). We assume that $\mathcal{E}_C$ also has a function $var\colon \mathcal{E}_C \to \mathbb{V}^\star$ (with $ar(A) = |var(A)|$, for all $A \in \mathcal{E}_C$), assigning a tuple of *distinct* canonical variables to each constraint. Canonical variables are such that $var(A) \cap var(B) = \varnothing$ if $A \neq B$. The structure of soft constraint problems can be described as a particular kind of hypergraphs labelled over $\mathcal{E}_C$.

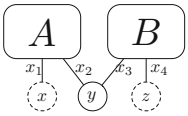
Definition 1 (Concrete network). *A* concrete network *(of constraints) is a pair $I \blacktriangleright N$, where:*

- *$N = (V_N, E_N, a_N, lab_N)$ is a labelled hypergraph over $\mathcal{E}_C$ such that $V_N \subseteq \mathbb{V}$ and there are no isolated vertices, i.e., vertices v such that $v \notin a_N(e)$, for all $e \in E_N$;*
- *$I \subseteq V_N$ is a finite set of* interface variables.

In a concrete network, for every edge $e \in E_N$ we define a substitution of variables σ_e mapping component-wise the tuple of canonical variables $var(lab_N(e))$ to the actual variables $a_N(e)$ e is connected to. Hyperedges can be understood as

instances of constraints, where canonical variables are replaced by concrete ones, describing how subproblems are connected. Interface variables are "external", in the sense that they allow networks to interact when composed.

Example 1. Let A and B be two constraints with $ar(A) = ar(B) = 2$ and $var(A) = \langle x_1, x_2 \rangle$, $var(B) = \langle x_3, x_4 \rangle$. Consider the labelled hypergraph N, with $V_N = \{x, y, z\}$, $E_N = \{e_1, e_2\}$, $a_N(e_1) = \langle x, y \rangle$, $a_N(e_2) = \langle y, z \rangle$, $lab_N(e_1) = A$, $lab_N(e_2) = B$. The concrete network $\{y\} \blacktriangleright N$ is depicted below:



Labels are placed inside the corresponding edge and connections to vertices are labelled with the corresponding canonical variable. Canonical variables will be often omitted in pictures of networks. Interface vertices, namely y, have solid outline, and non-interface ones, namely x and z, have dashed outline. As instantiations of the canonical to the concrete variables, we have $\sigma_{e_1} = \{x_1 \mapsto x, x_2 \mapsto y\}$, $\sigma_{e_2} = \{x_3 \mapsto y, x_4 \mapsto z\}$.

We now introduce *Soft Constraint Satisfaction Problems* (SCSPs in short) [4]. They are based on *c-semirings*, which are semirings $(S, +, \times, 0, 1)$ such that the additive operation $+$ is idempotent, 1 is its absorbing element and the multiplicative operation $\times$ is commutative.

Definition 2 (SCSP). *An SCSP is a tuple $(I \blacktriangleright N, \mathbb{D}, S, val)$ of a concrete network $I \blacktriangleright N$, a finite set $\mathbb{D}$, a c-semiring S and a set of functions $val_A \colon (var(A) \to \mathbb{D}) \to S$, one for each constraint A occurring in the network.*

In an SCSP, every constraint A is assigned a *value* val_A, that is a function giving a cost in S to every assignment in $\mathbb{D}$ of canonical variables of A. As a shorthand, for $e \in E_N$ and $A = lab_N(e)$, we write $val_e \colon (a_N(e) \to \mathbb{D}) \to S$ for the function $val_e = val_A(- \circ \sigma_e)$, giving a cost to every assignment to variables e is attached to, according to $var(A)$. Variables I are those of interest, i.e., those of which we want to know the possible assignments compatible with all the constraints. Values for each constraint are used to compute the solution for the SCSP, using the semiring operations, plus an operation of *projection* over variable assignments: given $\rho \colon X \to \mathbb{D}$ and $Y \subseteq X$, $\rho \downarrow_Y$ is the restriction of ρ to Y.

The solution is a function $sol \colon (I \to \mathbb{D}) \to S$: for each $\rho \colon I \to \mathbb{D}$

$$sol(\rho) = \sum_{\{\rho' \colon V_N \to \mathbb{D} \mid \rho' \downarrow_I = \rho\}} \left(val_{e_1}(\rho' \downarrow_{a_N(e_1)}) \times \cdots \times val_{e_n}(\rho' \downarrow_{a_N(e_n)}) \right)$$

where $E_N = \{e_1, \ldots, e_n\}$. Notice that the function sol is computed via the point-wise application of semiring operations: each value function is applied to the (relevant part of the) variable assignment ρ, and then $\times$ is used on the results.

In other words, $\times$ can be lifted to value functions, giving a natural interpretation of composition of two constraint networks N_1 and N_2:

$$val_{N_1} \otimes val_{N_2} = (\lambda\rho\colon (V_{N_1} \cup V_{N_2} \to \mathbb{D}) \to S).val_{N_1}(\rho \downarrow_{V_{N_1}}) \times val_{N_2}(\rho \downarrow_{V_{N_2}})$$

Example 2. SCSPs can be used to model and solve optimization problems where the goal is to minimize the total cost. Suppose we have two cost functions $A, B\colon \mathbb{D}^2 \to \mathbb{R}^+_\infty$, assigning a (possibly infinite) cost to pairs of values from a finite set $\mathbb{D}$. We want to find the minimum of $A(x, y) + B(y, z)$. This problem can be represented as a SCSP as follows. We introduce a constraint for each function, and we connect constraints to form the concrete network $\emptyset \blacktriangleright N$, where N is the hypergraph of Example 1. The interface is empty because we want to minimize w.r.t. all variables. In order to capture sums and minimization of constraints, we use the *weighted c-semiring* $S_W = (\mathbb{R}^+, \min, +, +\infty, 0)$. Then, the problem corresponds to the SCSP $(\emptyset \blacktriangleright N, \mathbb{D}, S_W, val)$, where $val(A)$ and $val(B)$ act as the functions A and B. The solution sol is a function $(\emptyset \to \mathbb{D}) \to \mathbb{R}^+$, i.e., a single value in $\mathbb{R}^+$, given by:

$$sol = \min_{d_1, d_2, d_3 \in \mathbb{D}} (A(d_1, d_2) + B(d_2, d_3))$$

which precisely computes the minimum of $A(x, y) + B(y, z)$.

In SCSPs the solution does not depend on the identity of non-interface variables, and this will also be true in our framework. We can then abstract away from those variables and take networks up to isomorphism. We say that two concrete networks $I_1 \blacktriangleright N_1$ and $I_2 \blacktriangleright N_2$ are isomorphic, written $I_1 \blacktriangleright N_1 \cong I_2 \blacktriangleright N_2$, whenever $I_1 = I_2$ and there is an isomorphism $\varphi\colon N_1 \to N_2$ such that $\varphi(x) = x$, for all $x \in I_1$.

Definition 3 (network). *A(n abstract) network $I \triangleright C$ is an isomorphism class of concrete networks. We also write $I \triangleright N$ to mean that $I \blacktriangleright N$ is a canonical representative of its class.*

In the following, we will depict abstract networks in the same way as concrete networks (see Example 1), implicitly assuming the choice of a canonical representative.

2.2 Tree Decomposition

A decomposition of a graph can be represented as a *tree decomposition* [19,22], i.e., a tree where each vertex is a piece of the graph. We introduce a notion of *rooted tree decomposition*. Recall that a *rooted tree* $T = (V_T, E_T)$ is a set of vertices V_T and a set of edges $E_T \subseteq V_T \times V_T$, such that there is a *root*, i.e. a vertex $r \in V_T$:

- with no ingoing edges: there are no edges (v, r) in E_T;
- such that, for every $v \in V_T$, $v \neq r$, there is a unique path from r to v, i.e., a unique sequence of edges $(r, u_1), (u_1, u_2), \ldots, (u_n, v)$, $n \geq 0$.

Definition 4 (Rooted tree decomposition of a hypergraph). *A* rooted tree decomposition *of a hypergraph G is a pair $\mathcal{T} = (T, X)$, where T is a rooted tree and $X = \{X_t\}_{t \in V_T}$ is a family of subsets of V_G, one for each vertex of T, such that:*

1. *for each vertex $v \in V_G$, there exists a vertex t of T such that $v \in X_t$;*
2. *for each hyperedge $e \in E_G$, there is a vertex t of T such that $a_G(e) \subseteq X_t$;*
3. *for each vertex $v \in V_G$, let $S_v = \{t \mid v \in X_t\}$, and $E_v = \{(x, y) \in E_T \mid x, y \in S_v\}$; then (S_v, E_v) is a rooted tree.*

We gave a slightly different definition of tree decomposition: the original one refers to a non-rooted, undirected tree. All tree decompositions in this paper are rooted, so we will just call them tree decompositions, omitting "rooted".

Tree decompositions are suited to decompose networks: we require that interface variables are located at the root.

Definition 5 (Decomposition of a network). *The decomposition of a network $I \rhd N$ is a decomposition of N rooted in r, such that $I \subseteq X_r$.*

2.3 Dynamic Programming via Tree Decompositions

The general issue of assigning a tree-like structure to graphs and networks in order to efficiently solve optimization problems is an issue of paramount importance in optimization theory. It is known as the *dynamic programming secondary optimization problem* [3].

The dynamic programming strategy of reducing problems to subproblems needs to express optimal solutions in terms of parameters, which represent shared variables between subproblems. Such a decomposition can be formalized via a tree decomposition $\mathcal{T}$ of the graph, where each node t is a problem, its children are subproblems, and X_t are the problem's variables. The dynamic programming algorithm then is based on a bottom-up visit of the tree.

Usually, time and space requirements for computing parametric solutions are at least exponential in the number of variables. Thus the complexity of a problem is defined as the maximal number of parameters in its reductions, called *width*. Formally, we have $\texttt{width}(\mathcal{T}) = \max_{t \in T}\{|X_t|\}$. The *treewidth* of a graph is the minimal width among all of its tree decompositions[1]. If graphs in a certain class have bounded treewidth, then their complexity becomes linear in their size – possibly with a big coefficient which depends on the treewidth bound – usually a tremendous achievement. Finding the treewidth, which involves a minimization over all the decomposition of a graph, is NP-complete. Even if expensive, an efficient solution of the secondary optimization problem may be essential whenever the original problem must be solved many times with different data and thus several approaches have been proposed for solving the secondary problem approximately.

[1] Width is conventionally defined as $\max_{t \in T}\{|X_t|\} - 1$. We drop "$-1$" so that it gives the actual number of parameters.

3 Soft Constraint Evaluation Problems (SCEPs)

In this section we introduce *Soft Constraint Evaluation Problems* (SCEPs). They are problems involving soft constraints, generalizing SCSPs. We work in an algebraic setting: elements of the initial algebra describe the structure of SCEPs, and *evaluations* of such structure can be given in any other algebra satisfying the SCEP specification.

We write $Perm(\mathbb{V})$ for the set of permutations over $\mathbb{V}$, i.e., bijective functions $\pi\colon \mathbb{V} \to \mathbb{V}$. A *permutation algebra* is an algebra for the signature comprising all permutations and the formal equations $x\ \mathsf{id} = x$ and $(x\ \pi_1)\ \pi_2 = x\ (\pi_2 \circ \pi_1)$ (the application of a permutation is written in postfix notation). The SCEP signature equips permutation algebras with additional operators and equations.

Definition 6 (SCEP signature). *Recall that $\mathcal{E}_C$ is the ranked alphabet of constraints. The* SCEP signature *(s-signature in short) is given by the following grammar*

$$p, q := p \parallel q \ \mid\ (x)p \ \mid\ p\ \pi \ \mid\ A(\tilde{x}) \ \mid\ \mathtt{nil}$$

where $A \in \mathcal{E}_C$, $\pi \in Perm(\mathbb{V})$, $\{x\} \cup \tilde{x} \subseteq \mathbb{V}$ and $|\tilde{x}| = ar(A)$.

The *parallel composition* $p \parallel q$ represents the problem consisting of two subproblems p and q, possibly sharing some variables. The *restriction* $(x)p$ represents the fact that p has been solved w.r.t. x. The *permutation* $p\pi$ is p where variables have been renamed according to π. The *atomic SCEP* $A(\tilde{x})$ only involves an instance of the constraint A over variables $\tilde{x}$ (notice that the same variable may occur more than once in $\tilde{x}$). The constant `nil` represents the *empty problem*.

The *free* variables $fv(p)$ of p are

$$fv(p \parallel q) = fv(p) \cup fv(q) \qquad fv(x)p) = fv(p) \setminus \{x\} \qquad fv(p\pi) = \pi(fv(p))$$
$$fv(A(\tilde{x})) = \tilde{x} \qquad fv(\mathtt{nil}) = \emptyset$$

We write $v(p)$ for the set of all the variables occurring in p.

Definition 7 (Strong SCEP specification). *The* strong SCEP specification *(s-specification, in short) is formed by the signature in Definition 6 and the axioms in Fig. 1.*

The operator $\parallel$ forms a commutative monoid, meaning that problems in parallel can be solved in any order ($\mathbf{AX}_{\parallel}$). Restrictions can be α-converted ($\mathbf{AX}_{\alpha}$), i.e., the name of the variable w.r.t. which we solve the problem is irrelevant. Restrictions can also be swapped, i.e., we can solve w.r.t. variables in any order, and can be removed, whenever their scope is `nil` ($\mathbf{AX}_{(x)}$). The scope of restricted variables can be narrowed to terms where they occur free ($\mathbf{AX}_{SE}$). Notice that restriction is idempotent, namely $(x)(x)p \equiv_s (x)p$. Axioms regarding permutations say that identity and composition behave as expected ($\mathbf{AX}_{\pi}$) and that permutations distribute over syntactic operators ($\mathbf{AX}_{\pi}^{p}$). Permutations behave

$(\mathbf{AX}_{\|})$
$p \parallel q \equiv_s q \parallel p \qquad (p \parallel q) \parallel r \equiv_s p \parallel (q \parallel r) \qquad p \parallel \mathtt{nil} \equiv_s p$

$(\mathbf{AX}_{(x)})$
$(x)(y)p \equiv_s (y)(x)p \quad (x)\mathtt{nil} \equiv_s \mathtt{nil}$

$(\mathbf{AX}_{\alpha})$
$(x)p \equiv_s (y)p[x \mapsto y] \qquad (y \notin fv(p))$

$(\mathbf{AX}_{SE})$
$(x)(p \parallel q) \equiv_s (x)p \parallel q \qquad (x \notin fv(q))$

$(\mathbf{AX}_{\pi})$
$p\ \mathsf{id} \equiv_s p \qquad (p\pi')\pi \equiv_s p(\pi \circ \pi')$

$(\mathbf{AX}_{\pi}^{p})$
$A(x_1, \ldots, x_n)\pi \equiv_s A(\pi(x_1), \ldots, \pi(x_n)) \qquad \mathtt{nil}\,\pi \equiv_s \mathtt{nil} \qquad (p \parallel q)\pi \equiv_s p\pi \parallel q\pi$
$((x)p)\pi \equiv_s (\pi(x))(p\pi)$

Fig. 1. Axioms of the strong SCEP specification.

in a capture avoiding way, by replacing all names bijectively, including the bound one x. This can be understood as applying, at the same time, α-conversion and renaming of free variables on $(x)p$.

We assume a standard operation of definition $P(x_1, \ldots, x_n) \stackrel{\text{def}}{=} p$ where $x_1, \ldots, x_n$ is a sequence of distinct variables including $fv(p)$. We write $P(y_1, \ldots, y_n)$ for $p[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$, where the substitution (not just a permutation) on p acts syntactically in a capture avoiding way. In this paper, we are interested in non-recursive (but well founded) definitions only. Definitions respect permutations, namely $P(x_1, \ldots, x_n)\pi \equiv_s P(\pi(x_1), \ldots, \pi(x_n))$.

We call *s-algebras* the algebras of the s-specification. Given an operation *op* in the s-specification, $op^{\mathcal{A}}$ denotes the interpretation of *op* in the s-algebra $\mathcal{A}$. We consider terms freely generated, modulo axioms of Fig. 1, in the style of [13], and we call them s-terms. They form an initial s-algebra $\mathcal{T}_s$. By initiality, for any s-algebra $\mathcal{A}$ and $p \in \mathcal{T}_s$, there is a unique interpretation $[\![p]\!]^{\mathcal{A}}$ of p as an element of $\mathcal{A}$, inductively defined as follows:

$$[\![p \parallel q]\!]^{\mathcal{A}} = [\![p]\!]^{\mathcal{A}} \parallel^{\mathcal{A}} [\![q]\!]^{\mathcal{A}} \qquad [\![(x)p]\!]^{\mathcal{A}} = (x)^{\mathcal{A}}[\![p]\!]^{\mathcal{A}} \qquad [\![p\pi]\!]^{\mathcal{A}} = [\![p]\!]^{\mathcal{A}}\pi^{\mathcal{A}}$$
$$[\![A(\tilde{x})]\!]^{\mathcal{A}} = A(\tilde{x})^{\mathcal{A}} \qquad [\![\mathtt{nil}]\!]^{\mathcal{A}} = \mathtt{nil}^{\mathcal{A}}$$

Here we use infix, prefix or postfix notation for functions $op^{\mathcal{A}}$ to reflect the syntax of s-terms. We use the expression *concrete* terms to indicate syntactic terms that are not considered up to axioms.

Permutations in the specification allow computing the set of "free" variables, called *(minimal) support*, in any s-algebra.

Definition 8 (Support). *Let $\mathcal{A}$ be an s-algebra. We say that a finite $X \subset \mathbb{V}$* supports *$a \in \mathcal{A}$ whenever, for all permutations π acting as the identity on X, we have $a\pi^{\mathcal{A}} = a$. The minimal support* $\mathtt{supp}(a)$ *is the intersection of all sets supporting a.*

For instance, given an s-term $p \in \mathcal{T}_s$, $p\pi^{\mathcal{T}_s}$ applies π to all free names of p in a capture avoiding way. It is easy to verify that $\mathtt{supp}(p) = fv(p)$.

An important property of SCEP algebras, following from the theory of permutation algebras, is that $[\![p]\!]^{\mathcal{A}}$ depends on (at most) the free variables of p, formally:

Lemma 1. $\mathtt{supp}([\![p]\!]^{\mathcal{A}}) \subseteq \mathtt{supp}(p)$*, for all s-terms* p *and s-algebras* $\mathcal{A}$*.*

3.1 Weak Specification

Our syntax is expressive enough to describe both the problem's structure and its decomposition into subproblems. For instance, the structurally congruent concrete terms

$$(y)(x)(z)(A(x,y) \parallel B(y,z)) \qquad (y)((x)A(x,y) \parallel (z)B(y,z))$$

are equivalent s-terms, and so they describe the same problem, but the information about which subproblems to solve w.r.t. x and z, represented as the subterms in the scope of (x) and (z), is different. To distinguish different decompositions, we introduce a *weak* SCEP specification where $(\mathbf{AX}_{SE})$ is dropped to avoid the rearrangement of restrictions.

Definition 9 (Weak SCEP specification). *The* weak SCEP specification *(w-specification, in short), is the s-specification without* $(\boldsymbol{AX}_{SE})$*, and where the axiom* $(x)\mathtt{nil} \equiv_s \mathtt{nil}$ *is replaced with*

$$(\boldsymbol{AX}^{w}_{(x)}) \qquad (x)p \equiv_w p \qquad (x \notin fv(p)).$$

The axiom $(\mathbf{AX}^{w}_{(x)})$ is needed to discard "useless" variables. In the s-specification, it can be derived using other axioms, including $(\mathbf{AX}_{SE})$. This is not possible in the w-specification, so we need to state it explicitly.

Algebras of the w-specification are called w-algebras and the terms modulo its axioms are called w-terms, forming the initial w-algebra; w-terms can be understood as networks having a *hierarchical structure*, made of scopes determined by restrictions. We are interested in two forms of w-terms.

Definition 10 (Normal and canonical forms). *A w-term is said to be in* normal form *whenever it is of the form* $(\tilde{x})(A_1(\tilde{x}_1) \parallel A_2(\tilde{x}_2) \parallel \cdots \parallel A_n(\tilde{x}_n))$*, where* $\tilde{x} \subseteq \tilde{x}_1 \cup \cdots \cup \tilde{x}_n$*. It is in* canonical form *whenever it is obtained by the repeated application of the directed version of* $(\boldsymbol{AX}_{SE})$*:* $(x)(p \parallel q) \to (x)p \parallel q$ $(x \notin fv(q))$ *until termination. For both forms, we assume that subterms of the form* $(\tilde{x})\mathtt{nil}$ *(where* $\tilde{x}$ *may be empty) are removed using* $(\boldsymbol{AX}_{(x)})$ *and* $(\boldsymbol{AX}_{\parallel})$*.*

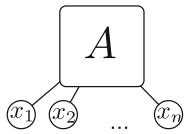
Normal and canonical forms exist in both concrete (no axioms) and abstract (up to weak axioms) versions. Normal and canonical forms are somewhat dual: normal forms have all restrictions at the top level, whereas in canonical forms every restriction (x) is as close as possible to the atomic terms where x occurs. Notice that an s-term may have more than one canonical form, whereas normal forms are unique (both up to w-specification axioms).

3.2 Soundness and Completeness of Networks

We now show that networks form an s-algebra, and that this algebra is isomorphic to $\mathcal{T}_s$. In other words, we show that the s-specification is sound and complete w.r.t. networks.

Theorem 1. *Let $\mathcal{N}$ be the smallest algebraic structure defined as follows. Constants are:*

$$A^{\mathcal{N}}(x_1, x_2, \ldots, x_n) = \quad \texttt{nil}^{\mathcal{N}} = \emptyset \rhd 1_N$$



and operations are:

$$(I \rhd N)\pi^{\mathcal{N}} = \pi(I) \rhd N_\pi \qquad (x)^{\mathcal{N}}(I \rhd N) = I \setminus \{x\} \rhd N$$
$$I_1 \rhd N_1 \parallel^{\mathcal{N}} I_2 \rhd N_2 = I_1 \cup I_2 \rhd N_1 \uplus_{I_1, I_2} N_2$$

where: N_π is N where each vertex v is replaced with $\pi(v)$; $N_1 \uplus_{I_1,I_2} N_2$ is the disjoint union of N_1 and N_2 where vertices in $I_1 \cup I_2$ with the same name are identified; and 1_N is the network with no vertices and edges. Then $\mathcal{N}$ is an s-algebra.

Even if not depicted, when the same variable x occurs twice in $A(x_1, x_2, \ldots, x_n)$, the corresponding hyperedge has two tentacles connected to the same vertex x. Theorem 1 implies that there is a unique evaluation of s-terms: given p, the corresponding network $[\![p]\!]^{\mathcal{N}}$ can be computed by structural recursion. We show that any network is the evaluation of an s-term. In order to do this, we first give translations between concrete networks and s-terms in normal forms over the same set of variables, which will also be useful later.

Definition 11 (Translation functions). *Let $I \blacktriangleright N$ be a concrete network. Let $e_1, \ldots, e_n$ be its edges, and let $A_i = lab_N(e_i)$, $\tilde{x}_i = a_N(e_i)$. Then we define*

$$\texttt{term}(I \blacktriangleright N) = (V_N \setminus I)(A_1(\tilde{x}_1) \parallel \cdots \parallel A_n(\tilde{x}_n))$$

Vice versa, given a concrete term in normal form $p = (\tilde{x})(A_1(\tilde{x}_1) \parallel \cdots \parallel A_n(\tilde{x}_n))$ we define $\texttt{net}(p) = fv(p) \blacktriangleright N_p$, where:

- *$V_{N_p} = v(p)$;*
- *$E_{N_p} = \{e^{(i)}_{A_i(\tilde{x}_i)} \mid A_i(\tilde{x}_i)$ is an atomic subterm of $p\}$;*
- *a_{N_p} and lab_{N_p} map $e^{(i)}_{A_i(\tilde{x}_i)}$ to $\tilde{x}_i$ and A_i, respectively.*

Notice that we assume an indexing on atomic subterms of p. This allows `net` to map two identical subterms to different edges.

Example 3. Consider the term in normal form $p = (x)(z)(A(x, y) \parallel B(y, z))$, then $\texttt{net}(p)$ is the concrete network depicted in Example 1.

Completeness is a consequence of the following theorem.

Theorem 2. *Given two s-terms in normal form n_1 and n_2, if* $\mathtt{net}(n_1) \cong \mathtt{net}(n_2)$ *then* $n_1 \equiv_s n_2$. *As a consequence,* $[\![p_1]\!]^{\mathcal{N}} = [\![p_2]\!]^{\mathcal{N}}$ *implies* $p_1 \equiv_s p_2$, *for any two s-terms.*

4 SCSPs as SCEPs

We now show how SCSPs are represented and solved as SCEPs. Consider the SCSPs definable over a fixed c-semiring S, a fixed domain of variable assignments $\mathbb{D}$ and a fixed family of value functions val_A, one for each atomic constraint. SCEPs for such SCSPs can be defined as follows: networks are the underlying ones of SCSPs, and the SCEP algebra for evaluations is formed by value functions. Here by value function we mean functions of the form $(\mathbb{V} \to \mathbb{D}) \to S$. This is different from Sect. 2.1, where the domain of value functions are variable assignments $I \to \mathbb{D}$, with I a finite set. We will see that the new formulation is equivalent, and allows for simpler algebraic operations, because they do not depend on the "types" of assignments.

Theorem 3. *Let $\mathcal{V}$ be the smallest algebraic structure defined as follows. For any $\rho\colon \mathbb{V} \to \mathbb{D}$, constants are:*

$$A^{\mathcal{V}}(x_1, x_2, \ldots, x_n)\rho = val_A(\rho \downarrow_{\{x_1,x_2,\ldots,x_n\}} \circ \hat{\sigma}) \qquad \mathtt{nil}^{\mathcal{V}}\rho = 1$$

and operations are:

$$((x)^{\mathcal{V}}\phi)\rho = \sum_{d\in\mathbb{D}} \phi(\rho[x \mapsto d]) \qquad (\phi\pi^{\mathcal{V}})\rho = \phi(\rho \circ \pi) \qquad (\phi_1 \parallel^{\mathcal{V}} \phi_2)\rho = \phi_1\rho \times \phi_2\rho$$

where $\hat{\sigma}$ maps $var(A)$ to $\langle x_1, x_2, \ldots, x_n\rangle$, component-wise. Then $\mathcal{V}$ is an s-algebra.

Notice that $\parallel^{\mathcal{V}}$ is the extension of the $\otimes$ operator of Sect. 2.1 to arbitrary value functions, but it is simpler: projections are not needed here, because variable assignments all have the same type, namely $\mathbb{V} \to \mathbb{D}$.

Now we show that the evaluation function $[\![-]\!]^{\mathcal{V}}$, applied to a network $I \rhd N$, gives the solution of the SCSP defined over that network. Notice that $[\![I \rhd N]\!]^{\mathcal{V}}$ has type $(\mathbb{V} \to \mathbb{D}) \to S$, but its domain should be of the form $I \to \mathbb{D}$. However, $[\![I \rhd N]\!]^{\mathcal{V}}$ has the following property.

Property 1 (Compactness). We say that $\phi\colon (\mathbb{V} \to \mathbb{D}) \to S$ is *compact* if $\rho \downarrow_{\mathtt{supp}(\phi)} = \rho' \downarrow_{\mathtt{supp}(\phi)}$ implies $\phi\rho = \phi\rho'$, for all $\rho, \rho'\colon \mathbb{V} \to \mathbb{D}$.

Now, by Lemma 1, we have $\mathtt{supp}([\![I \rhd N]\!]^{\mathcal{V}}) \subseteq \mathtt{supp}(I \rhd N) = I$. Therefore compactness means that $[\![I \rhd N]\!]^{\mathcal{V}}$ only depends on assignments to interface variables. The interpretation of constants is clearly compact and, by structural induction, we can show that compound terms are. We have our main result.

Theorem 4. *Given an SCSP with underlying network $I \rhd N$ and value functions val_A, we have that $I \rhd N$ evaluated in $\mathcal{V}$, namely $[\![I \rhd N]\!]^{\mathcal{V}}$, is its solution.*

We stress that SCEPs are more general than SCSPs: an example will be shown in Sect. 8.

5 Evaluation Complexity

Although all the s-terms corresponding to the same network have the same evaluation in any algebra $\mathcal{A}$, different ways of computing such an evaluation, represented as different w-terms, may have different computational costs. As already mentioned, finding the best one amounts to giving a solution for the secondary optimization problem.

We introduce a notion of complexity of w-terms to measure the computational costs of such evaluations.

Definition 12. *Given a w-term p, its complexity $\langle\!\langle p \rangle\!\rangle$ is defined as follows:*

$$\langle\!\langle p \parallel q \rangle\!\rangle = \max\{\langle\!\langle p \rangle\!\rangle, \langle\!\langle q \rangle\!\rangle, |fv(p \parallel q)|\} \quad \langle\!\langle (x)p \rangle\!\rangle = \langle\!\langle p \rangle\!\rangle \quad \langle\!\langle p\pi \rangle\!\rangle = \langle\!\langle p \rangle\!\rangle$$
$$\langle\!\langle A(\tilde{x}) \rangle\!\rangle = |set(\tilde{x})| \quad \langle\!\langle \mathtt{nil} \rangle\!\rangle = 0$$

The complexity of p is the maximum "size" of elements of $\mathcal{A}$ computed while inductively constructing $[\![p]\!]^{\mathcal{A}}$, the size being given by the number of variables in the support. Notice that all the concrete terms corresponding to the same abstract w-term have the same complexity.

Example 4. Consider the w-terms from Sect. 3.1

$$p = (y)(x)(z)(A(x,y) \parallel B(y,z)) \qquad q = (y)((x)A(x,y) \parallel (z)B(y,z)).$$

Even though they are s-congruent, and thus represent the same problem, we have $\langle\!\langle p \rangle\!\rangle = 3$ and $\langle\!\langle q \rangle\!\rangle = 2$. In fact, in order to evaluate p in any algebra, one has to evaluate $A(x,y) \parallel B(y,z)$, and then solve it w.r.t. all its variables. Intuitively, $A(x,y) \parallel B(y,z)$ is the most complex subproblem one considers in p, with 3 variables, hence $\langle\!\langle p \rangle\!\rangle = 3$. Instead, the evaluation of q requires solving $A(x,y)$ and $B(y,z)$ w.r.t. x and z, which are problems with 2 variables, and then putting the resulting partial solutions in parallel. The solution process for q never considers subproblems with more than 2 variables, hence $\langle\!\langle q \rangle\!\rangle = 2$.

The soundness of this definition follows from Lemma 1: if $[\![p']\!]^{\mathcal{A}}$ is computed while constructing $[\![p]\!]^{\mathcal{A}}$, we have $\mathtt{supp}([\![p']\!]^{\mathcal{A}}) \subseteq \mathtt{supp}(p')$, and this relation among supports does not depend on the choice of $\mathcal{A}$. The interesting cases are $(x)p$ and $p \parallel q$: the computation of $[\![(x)p]\!]^{\mathcal{A}}$ relies on that of $[\![p]\!]^{\mathcal{A}}$, whose support may be bigger, so we set the complexity of $(x)p$ to that of p; computing $[\![p \parallel q]\!]^{\mathcal{A}}$ requires computing $[\![p]\!]^{\mathcal{A}}$ and $[\![q]\!]^{\mathcal{A}}$, but the support of the resulting element of $\mathcal{A}$ is (at most) the union of those of p and q, so we have to find the maximum value among $\langle\!\langle p \rangle\!\rangle$, $\langle\!\langle q \rangle\!\rangle$ and the overall number of free variables.

Complexity is well-defined only for w-terms, because applying $(\mathbf{AX}_{SE})$ may change the complexity. Indeed, we have the following results for w-terms.

Lemma 2. *Given $(x)(p \parallel q)$, with $x \notin fv(q)$, we have $\langle\!\langle (x)p \parallel q \rangle\!\rangle \leq \langle\!\langle (x)(p \parallel q) \rangle\!\rangle$.*

As an immediate consequence, all the canonical forms of a term always have lower or equal complexity than the normal form.

Theorem 5. *Given a term p, let n be its normal form. Then, for all canonical forms c of p we have $\langle\langle c \rangle\rangle \leq \langle\langle n \rangle\rangle$.*

Of course, different canonical forms may have different complexities. However, due to Lemma 2, canonical forms may be considered as *local minima* of complexity w.r.t. the application of axioms of the strong specification.

6 Tree Decompositions as w-terms

In this section we provide a translation from tree decompositions to w-terms. This enables applying algebraic techniques to tree decompositions, and improving their complexity by bringing the corresponding w-terms in canonical form.

Given a network $I \rhd N$, let $\mathcal{T} = (T, X)$ be one of its tree decompositions. Its *completed* version $\mathcal{CT} = (\mathcal{T}, \{t_x\}_{x \in E_N \cup V_N})$ explicitly associates components of N to vertices of T: for each $v \in V_N$ (resp. $e \in E_N$), t_v (resp. t_e) is the vertex closest to the root of T such that $v \in X_{t_v}$ (resp. $a_N(e) \subseteq X_{t_e}$). By the definition of rooted tree decomposition (Definition 4), such vertices t_x exist (properties 1 and 2), and can be characterized as the roots of the subtrees of T induced by x (by $a_N(x)$, if x is an edge), according to property 3.

We now translate $\mathcal{CT}$ into a w-term. Given a vertex t of T, let

$$V(t) = \{v \in V_N \mid t_v = t\} \qquad E(t) = \{e \in E_N \mid t_e = t\}.$$

Suppose t has children $t_1, \ldots, t_n$ and $E(t) = \{e_1, \ldots, e_k\}$, with $n, k \geq 0$. Let $\tilde{x} = V(t) \setminus I$. The w-term $\chi(t)$ is inductively defined as follows:

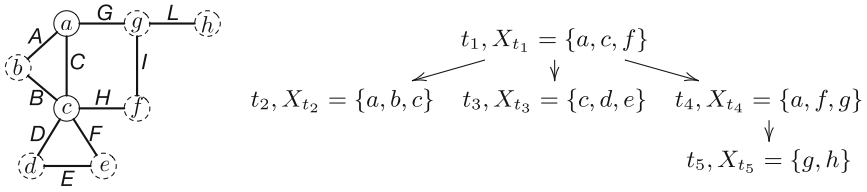
$$\chi(t) = (\tilde{x})(A_1(\tilde{x}_1) \parallel \cdots \parallel A_k(\tilde{x}_k) \parallel \chi(t_1) \parallel \cdots \parallel \chi(t_n))$$

where $A_i = lab_N(e_i)$ and $\tilde{x}_i = a_N(e_i)$. When $k = 0$ and/or $n = 0$, the corresponding part of the parallel composition degenerates to `nil`. We assume that subterms of the form $(\tilde{x})$`nil` are removed via $(\mathbf{AX}_{(x)})$ and $(\mathbf{AX}_{\parallel})$.

Example 5. Consider the network in Fig. 2a, whose underlying graph is taken from [6]. A tree decomposition for it is shown in Fig. 2b. Recall that interface variables have solid outline, namely they are a and c. Its completed version has: $t_a = t_c = t_f = t_1$, $t_b = t_2$, $t_e = t_d = t_3$, $t_h = t_5$ and $t_g = t_4$, $t_{(a,b)} = t_2$, $t_{(a,c)} = t_1$, $t_{(a,g)} = t_4$, $t_{(b,c)} = t_2$, $t_{(c,d)} = t_3$, $t_{(c,e)} = t_3$, $t_{(c,f)} = t_1$, $t_{(d,e)} = t_3$, $t_{(f,g)} = t_4$, $t_{(g,h)} = t_5$. Therefore we have

$$\begin{aligned}
\chi(t_1) &= (f)(C(a,c) \parallel H(c,f) \parallel \chi(t_2) \parallel \chi(t_3) \parallel \chi(t_4)) \\
\chi(t_2) &= (b)(A(a,b) \parallel B(b,c)) \\
\chi(t_3) &= (e)(d)(D(c,d) \parallel E(d,e) \parallel F(c,e)) \\
\chi(t_4) &= (g)(I(f,g) \parallel G(a,g) \parallel \chi(t_5)) \\
\chi(t_5) &= (h)L(g,h)
\end{aligned}$$

Again, notice that interface variables a and c are not restricted in $\chi(t_1)$.



(a) Network (b) One of its tree decompositions

Fig. 2. Example network and tree decomposition.

Definition 13 (wterm). *Given a tree decomposition $\mathcal{T}$ rooted in r, the corresponding w-term* $\texttt{wterm}(\mathcal{T})$ *is $\chi(r)$ computed on the completed version of $\mathcal{T}$.*

We have that $\texttt{wterm}(\mathcal{T})$ correctly represents the network $\mathcal{T}$ decomposes.

Proposition 1. *Let $\mathcal{T}$ be a rooted tree decomposition for $I \rhd N$. Then* $[\![\texttt{wterm}(\mathcal{T})]\!]^{\mathcal{N}} = I \rhd N$.

We now have one of our main results, relating the width of $\mathcal{T}$ and the complexity of the corresponding w-term.

Proposition 2. *Given a tree decomposition $\mathcal{T}$,* $\langle\!\langle \texttt{wterm}(\mathcal{T}) \rangle\!\rangle \leq \texttt{width}(\mathcal{T})$.

7 Computing Canonical Decompositions

We now give a simple algorithm to compute canonical term decompositions. The algorithm is shown in Fig. 3. It is based on *bucket elimination* [23, Sect. 5.2.4], also known as adaptive consistency. However, we will show that bucket elimination may also produce non-canonical decompositions, whereas our algorithm produces all and only canonical terms.

Bucket elimination works as follows. Given a CSP network of constraints, its variables are ordered, and constraints are partitioned into buckets: each constraint is placed in the bucket of its last variable in the order. At any step the bucket of the last variable, say x, is eliminated by synthesising a new constraint involving all and only the variables in the bucket different than x. This constraint is put again in the bucket of its last variable. The solution is produced when the last bucket is eliminated. Notice that one can also eliminate a subset of the variables, and obtain a solution parametric in the remaining variables.

In our algorithm, putting a constraint in the bucket of its last variable corresponds to applying the scope extension axiom. The algorithm takes an s-term in normal form as input, represented as $(R)A$, where A is a multiset of atomic terms and R is the set of variables to be eliminated. This notation amounts to taking the term up to weak axioms. A total order on R is given as input as well. The algorithm operates as follows. It picks the max variable (line 3) and partitions the input w-term into subterms according to whether the chosen

Inputs: s-term $(R)A$ in normal form; a total order O_R over R.
Output: w-term P in canonical form.

```
1  P ← (R)A
2  while O_R ≠ ∅
3        x ← extract max O_R
4        O_R ← O_R \ {x}
5        find all terms A' ⊆ A such that x ∈ fv(A')
6        if A' = {(R')P'} where P' has no top-level restriction
7              Q ← call the algorithm on (x)P' with order {(x, x)}
8              P'' ← (R')Q
9        else P'' ← (x)A'
10       P ← (R \ {x})A \ A' ∪ {P''}
11 return P
```

Fig. 3. Algorithm to compute canonical w-terms: P, P', P'' and Q denote w-terms, R and R' are sets of restricted variables, and A, A' are multisets of atomic or restriction-rooted w-terms.

variable occurs free or not (line 5). When line 5 returns a singleton $\{(R')P'\}$, the algorithm attempts at pushing the variable x further inside P', achieving the same effect as $(\mathbf{AX}_{SE})$. This is done by first calling the algorithm on $(x)P'$ and then restricting R' in the resulting term. This operation can be understood as a sequence of restriction swaps that bring x closer to P'. We have that the algorithm returns all and only the canonical forms of $(R)A$.

Theorem 6. *C is a canonical form of $(R)A$ if and only there is O_R^C such that the algorithm in Fig. 3 with inputs $(R)A$ and O_R^C outputs C.*

It is easy to see that the worst case complexity for the algorithm is given by the product of the number of variables by the number of atomic terms. In fact, this is the maximal number of times the test $x \in fv(A')$ is executed in line 5. The same worst case complexity holds for the ordinary bucket algorithm. However, for every total ordering assigned to variables, the complexity of the canonical form produced by our algorithm is lower or equal than that of the bucket elimination algorithm.

Example 6. Let us apply the algorithm to the following term in normal form:

$$P = (\{x_1, x_2, x_3, x_4\})\{A(x_1, x_2), B(x_1, x_4), C(x_1, x_3), D(x_3, x_4)\}$$

with $O_R = x_4 < x_3 < x_2 < x_1$. Line 3 picks x_1 and line 5 gives $A' = \{A(x_1, x_2), B(x_1, x_4), C(x_1, x_3)\}$. As A' is not a singleton, P becomes

$$(\{x_2, x_3, x_4\})\{D(x_3, x_4), (x_1)\{A(x_1, x_2), B(x_1, x_4), C(x_1, x_3)\}\}.$$

In the next iteration x_2 is picked from O_R, and we have $A' = (x_1)\{A(x_1, x_2), B(x_1, x_4), C(x_1, x_3)\}$. Now A' is a singleton, so the algorithm is called on

$$(x_2)\{A(x_1, x_2), B(x_1, x_4), C(x_1, x_3)\}$$

with $\{(x_2, x_2)\}$ order. The restriction (x_2) is pushed further inside, and the term

$$\{(x_2)A(x_1, x_2), \{B(x_1, x_4), C(x_1, x_3)\}\}$$

is returned. Line 8 will prepend (x_1) to the term above, and line 9 will construct the following term

$$(\{x_3, x_4\})\{D(x_3, x_4), (x_1)\{(x_2)A(x_1, x_2), \{B(x_1, x_4), C(x_1, x_3)\}\}\}.$$

which is then returned. The next two iterations will pick x_3 and x_4, and the **then** and **else** cases of line 6 are executed respectively. In the end we get the term (in usual notation):

$$C = (x_4)(x_3)(D(x_3, x_4) \parallel (x_1)((x_2)A(x_1, x_2) \parallel B(x_1, x_4) \parallel C(x_1, x_3)))$$

Bucket elimination corresponds to always executing line 9, even when A' is a singleton. In this case the result would be:

$$P' = (x_4)(x_3)(D(x_3, x_4) \parallel (x_2)(x_1)(A(x_1, x_2) \parallel B(x_1, x_4) \parallel C(x_1, x_3)))$$

which is not in canonical form and has worse complexity. In fact, we have $\langle\langle C \rangle\rangle = 3 < \langle\langle P' \rangle\rangle = 4$.

8 Example

In this section we present an example of an optimization problem which is an SCEP and cannot be represented as an SCSP.

Consider a social network, based on an overlay network, where certain meeting activities for a group of sites require the existence of routing paths between every pair of collaborating sites. Under the assumption that the network is composed of end-to-end two-way connections with independent probabilities of failure, we want to find the probability of a given group of sites staying connected.

We formalize the problem as an SCEP as follows. We consider networks that are undirected, binary graphs with no loops (but possibly with circuits), modelling the overlay network. Each edge has an associated probability of failure. The solution of the problem is the probability of some interface vertices staying connected. To achieve this, the idea is evaluating networks $I \rhd N$ into an algebra of probability distributions P on the partitions $Part(I)$ of I. Thus every partition of I, characterizing a certain level of connectivity, is assigned a probability. Consequently, if J is the group of sites we are interested in and N is the hypergraph for the whole network, then the solution is obtained by computing the probability distribution P for $J \rhd N$ and by selecting $P(\{J\})$. Notice that the size of the values of our algebra grows very rapidly with the cardinality n of I. In fact, the number of possible partitions for a set of n elements is the *Bell number*, inductively given by $B_0 = 1$, $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$. Thus if a vector

representation is chosen, the amount of memory needed to represent a value of the algebra grows very rapidly with the number of interface vertices.

We now define the evaluation from networks and we show that it induces an s-algebra. For the case of constants, we assume for simplicity that we have two kinds of edges: A-labelled ones (more reliable) and B-labelled ones (less reliable), both with two vertices x, y. Given $\Pi_1 = \{\{x\}, \{y\}\}$ and $\Pi_2 = \{\{x, y\}\}$, we have

$$\begin{array}{ll} [\![I \rhd N_A]\!]^{\mathcal{D}} \Pi_1 = q_A & [\![I \rhd N_A]\!]^{\mathcal{D}} \Pi_2 = 1 - q_A \\ [\![I \rhd N_B]\!]^{\mathcal{D}} \Pi_1 = q_B & [\![I \rhd N_B]\!]^{\mathcal{D}} \Pi_2 = 1 - q_B \end{array}$$

where N_A (resp. N_B) is a network with a single A-labelled (resp. B-labelled) hyperedge, and q_A (resp. q_B) is the probability of the former (resp. latter) hyperedge failing, i.e., of x and y being in different sets of the partition. We have $\mathtt{nil}^{\mathcal{D}} \emptyset = 1$. Permutations are defined straightforwardly:

$$[\![I \rhd N\pi]\!]^{\mathcal{D}} \Pi = [\![I \rhd N]\!]^{\mathcal{D}} \Pi \pi^{-1},$$

where $\Pi \in Part(I\pi)$. Permutations are applied to sets and partitions in the obvious way. Parallel composition is more complicated:

$$[\![I_1 \rhd N_1 \parallel I_2 \rhd N_2]\!]^{\mathcal{D}} \Pi = \sum_{\{(\Pi_1, \Pi_2) | \Pi_1 \cup \Pi_2 = \Pi\}} [\![I_1 \rhd N_1]\!]^{\mathcal{D}} \Pi_1 \times [\![I_2 \rhd N_2]\!]^{\mathcal{D}} \Pi_2.$$

where $\Pi \in Part(I_1 \cup I_2)$, and each Π_1, Π_2 must belong to $Part(I_1)$ and $Part(I_2)$, respectively. Here the union operation $\cup$ produces the finest partition coarser than the two components and $\times$ is the multiplication on reals. The last operation is restriction:

$$[\![(x)I \rhd N]\!]^{\mathcal{D}} \Pi = \sum_{\{\Pi' \in Part(I \cup \{x\}) | \Pi' - x = \Pi\}} [\![I \rhd N]\!]^{\mathcal{D}} \Pi'$$

where $\Pi' - x$ removes x from its set in Π'. Here probability values are accumulated for all the cases where a certain partition of interface vertices is guaranteed, independently of the set where variable x is located.

Theorem 7. *The image of $[\![-]\!]^{\mathcal{D}}$ is an s-algebra.*

As a family of overlay networks we choose *wheels* of N vertices where each vertex is also connected to a central *control* vertex. Accordingly, connections in the ring have low failure probability (label A), while the connections to the center have high failure probability (label B). We want to find out how much the connection probability between two adjacent vertices in the ring deteriorates when the direct link between them breaks down.

The formal definition of our networks is given in Fig. 4. They consist of radius elements R_i, recursively composed in parallel; rings are closed ($W_k(v, x)$) by connecting the last (v) and the first (x) radius; the failed network is $FW_k(v, x)$ where the ring is interrupted because $A(v, x)$ is missing. Figure 4 shows $W_2(v, x)$.

$$R_0(x,y,z) = A(x,y) \parallel B(x,z)$$
$$R_{i+1}(x,y,z) = (v)(R_i(x,v,z) \parallel R_i(v,y,z))$$
$$W_k(v,x) = (z)(R_k(x,v,z) \parallel A(v,x) \parallel B(v,z))$$
$$FW_k(v,x) = (z)(R_k(x,v,z) \parallel B(v,z))$$

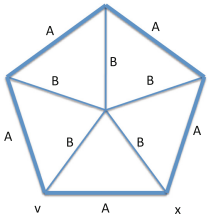


Fig. 4. Formal specification of a *wheel* network and depiction of W_2.

It is easy to see that $W_k(v,x)$ is a wheel with $N = 2^k + 1$ radii, which is specified by a number of simple well-founded non-recursive defining equations linear in k. The top-down recursive evaluation of $W_k(v,x)$ is clearly exponential in k. The complexity of bucket elimination is the same. However, the bottom-up dynamic programming evaluation is much more efficient: its complexity is linear in k and logarithmic in the size N of the problem, thanks to the presence of repetitive subterms.

8.1 Non-existence of a SCSP Formulation

As mentioned, the problem does not fit the SCSP format. To show why, given the SCEP defined above, let us try to construct an equivalent SCSP.

We can safely assume that the network $I \rhd N$ is the same in both cases. The carrier of our algebra consists of the probability distributions on the partitions $Part(I)$ of the interface variables I of the network. To fit the SCSP definition, a partition in $Part(I)$ can be represented as (the kernel of) an assignment of variables I. Thus the solution function $sol : D(Part(I))$ computes the probability $sol(\Pi)$ associated to a given partition Π of interface variables. Without discussing how to impose a semiring structure on probabilities, notice that the solution in the SCSP case, for any two networks N_1 and N_2 whose union is N, is given by $val_{N_1}(\Pi_1) \otimes val_{N_2}(\Pi_2)$, where Π_1 and Π_2 are the restrictions (projections) of Π to the vertices of N_1 and N_2, respectively. The solution only examines the probabilities caused by *the same* Π on the two sub-networks. This limitation is incompatible with the definition of parallel composition in our example, where to compute the outcome of a resulting partition in the composed network, the probabilities must be considered computed by all pairs of partitions in the component networks whose union (as described earlier when defining $\parallel^{\mathcal{D}}$) is the given partition.

8.2 Implementation

The main issue in the implementation is how to represent the values of the domain and how to implement the operations. Probability distributions can be represented as vectors indexed by the partitions of the set of the interface vertices, which grow very rapidly with the number of vertices. To allow for fast

insertion and retrieval, it is convenient to represent partitions as strings and to order them. A simple representation starts ordering the vertices within the sets of vertices, and eventually the sets of vertices in the partitions according to their first element. It is interesting to observe that it is convenient not to represent sets of vertices which are singletons. Omitting them makes partitions untyped, and thus simplifies the computation of parallel composition.

A natural way to compute parallel composition takes all pairs (Π_1, Π_2) of partitions, determines $\Pi_1 \cup \Pi_2 = \Pi$ and increments by $p_1\Pi_1 \times p_2\Pi_2$ the entry of Π in the result. The union can be computed efficiently with merge-find-like algorithms, thus the cost of multiplication is essentially quadratic with the number of partitions. Similarly, the cost of $(x)p$ is essentially linear with the number of partitions of p: every value $p\Pi$ increments the entry $\Pi - x$ of the result.

Table 1. Example values.

k	N	A	B	F	msec	W	msec	A	B	F	msec	W	msec
1	3	0.01	0.1	0.00217	17	0.00002	22	0.1	0.3	0.07043	17	0.00704	19
2	5	0.01	0.1	0.03154	78	0.00031	105	0.1	0.3	0.30817	74	0.03081	80
3	9	0.01	0.1	0.0697	183	0.00069	190	0.1	0.3	0.54609	172	0.00546	184
4	17	0.01	0.1	0.14157	409	0.00141	426	0.1	0.3	0.8046	435	0.00804	452
5	33	0.01	0.1	0.26908	620	0.00269	623	0.1	0.3	0.96379	625	0.09637	661

We ran experiments on a 2.2 GHz Intel Core i7 with 4 GB RAM. In Table 1 we see the connection probability between v and x with and without failure (i.e. for F_k and W_k) for various values of k, together with the corresponding computing time. Each case is computed for failure probabilities $q_A = 0.01, q_B = 0.1$, and $q_A = 0.1, q_B = 0.3$. Notice that it is always the case that failure probability for W_k equals the product of the failure probability for F_k and of q_A. This is obvious, since the edge A and the network F_k are composed in parallel to obtain W_k, and thus their failure probabilities should be multiplied.

9 Conclusion

We have presented a class of constraint algebras, which generalize SCSPs. Vertices of constraint networks are implicitly represented as support elements of a permutation algebra. This allows for the evaluation of terms of the algebras in rather abstract domains. Applying directionally the scope extension axiom until termination yields terms for efficient dynamic programming strategies. An example has also been shown about computing the connection probability of communication networks. This problem can be represented using our algebras, but not as an SCSP.

Our framework is a significant step towards the use of existing techniques and tools for algebraic specifications in the context of constraint-based satisfaction

and optimization. While some evidence of the approach we foresee are given in the paper (improved bucket elimination and doubly exponential speed up in a recursive, well-founded definition), further results are left for future work. A direction to explore is using more sophisticated term substitutions (e.g., second order substitutions, in the line of [14]) for defining complex networks inductively. In this paper definitions are restricted to deterministic, non-recursive instances: dropping these restrictions would lead us to the realm of DATALOG constraint programming, with tabling, possibly suggestive in the presence of programmable evaluation strategies.

Related Work. Other compositional constraint definitions have been proposed in the literature: in [7] constraints are modeled in a *named* semiring, and in [4] the semiring operations are extended point-wise to functions mapping variable assignments to semiring values. However, in the former case no explicit evaluation is performed, while in the latter no restriction operation is considered. Other approaches are: [5], where compositionality is achieved via complex categorical structures, and [25], where compositionality is not tackled. In a previous workshop paper [18], some early results were given by two of the authors. However, while the algebraic specification is essentially the same, the interpretation domain was restricted to SCSPs for optimization, without reference to SCEPs. Moreover, no proof was given that SCSPs actually satisfy the specification. Furthermore, the connection with the classical tree decomposition was just hinted.

The problem of how to represent parsing trees for (hyper)graphs has been studied in depth in the literature. In particular, we mention the notion of Courcelle graph algebras [9] and of graph grammars for hyperedge replacement [8], which assign a complexity value to the parsing steps. Typical results are about classes of graphs with parsings of bound complexity, having properties that can be proved or computed in linear time. While these results are analogous to ours for some aspects, they do not apply specifically to SCSPs or SCEPs. Instead, tree decomposition and secondary optimization problems have been studied for CSP in [16]. However our approach has a simpler and more effective compositional structure and an up-to-date foundation for name handling.

The role of bounded treewidth CSP has been studied also in connection with the general area of computing homomorphisms between relational structures [10, 11, 17] and k-consistency [1].

Acknowledgements. We thank Nicklas Hoch and Giacoma Valentina Monreale for their collaboration in an earlier version of this work. We also thank an anonymous reviewer for suggesting the example where bucket elimination does not produce a canonical term.

References

1. Atserias, A., Bulatov, A., Dalmau, V.: On the power of k-consistency. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 279–290. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_26
2. Bellman, R.: The theory of dynamic programming. Bull. Am. Math. Soc. **60**(6), 503–516 (1954)
3. Bertelè, U., Brioschi, F.: On non-serial dynamic programming. J. Comb. Theory, Ser. A **14**(2), 137–148 (1973)
4. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM **44**(2), 201–236 (1997)
5. Blume, C., Sander Bruggink, H.J., Friedrich, M., König, B.: Treewidth, pathwidth and cospan decompositions with applications to graph-accepting tree automata. J. Vis. Lang. Comput. **24**(3), 192–206 (2013)
6. Bodlaender, H.L., Koster, A.M.C.A.: Combinatorial optimization on graphs of bounded treewidth. Comput. J. **51**(3), 255–269 (2008)
7. Buscemi, M.G., Montanari, U.: CC-Pi: a constraint-based language for specifying service level agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 18–32. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_3
8. Chiang, D., Andreas, J., Bauer, D., Hermann, K.M., Jones, B., Knight, K.: Parsing graphs with hyperedge replacement grammars. In: ACL, pp. 924–932 (2013)
9. Courcelle, B., Mosbah, M.: Monadic second-order evaluations on tree-decomposable graphs. Theor. Comput. Sci. **109**(1–2), 49–82 (1993)
10. Dalmau, V., Jonsson, P.: The complexity of counting homomorphisms seen from the other side. Theor. Comput. Sci. **329**(1–3), 315–323 (2004)
11. Dalmau, V., Kolaitis, P.G., Vardi, M.Y.: Constraint satisfaction, bounded treewidth, and finite-variable logics. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 310–326. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_21
12. Dechter, R.: Constraint Processing. Morgan Kaufmann Series. Elsevier, New York (2003)
13. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations und Initial Semantics. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69962-7
14. Fiore, M., Mahmoud, O.: Second-order algebraic theories. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 368–380. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15155-2_33
15. Gadducci, F., Miculan, M., Montanari, U.: About permutation algebras, (pre)sheaves and named sets. High. Order Symbol. Comput. **19**(2–3), 283–304 (2006)
16. Gogate, V., Dechter, R.: A complete anytime algorithm for treewidth. In: UAI, pp. 201–208 (2004)
17. Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. J. ACM **54**(1), 1:1–1:24 (2007)
18. Hoch, N., Montanari, U., Sammartino, M.: Dynamic programming on nominal graphs. In: GaM 2015, pp. 80–96 (2015)
19. Kloks, T. (ed.): Treewidth: Computations and Approximations. LNCS, vol. 842. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0045375

20. Montanari, U.: Networks of constraints: fundamental properties and applications to picture processing. Inf. Sci. **7**, 95–132 (1974)
21. Pitts, A.M.: Nominal Sets: Names and Symmetry in Computer Science. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (2013)
22. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. J. Comb. Theory, Ser. B **36**(1), 49–64 (1984)
23. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Foundations of Artificial Intelligence, vol. 2. Elsevier, New York (2006)
24. Rossi, F., van Beek, P., Walsh, T.: Constraint programming. In: Handbook of Knowledge Representation, pp. 181–211 (2008)
25. Schiendorfer, A., Knapp, A., Steghöfer, J.-P., Anders, G., Siefert, F., Reif, W.: Partial valuation structures for qualitative soft constraints. In: De Nicola, R., Hennicker, R. (eds.) Software, Services, and Systems. LNCS, vol. 8950, pp. 115–133. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15545-6_10

Overview of Reconfigurable Petri Nets

Julia Padberg[1(✉)] and Laid Kahloul[2]

[1] Hamburg University of Applied Sciences, Hamburg, Germany
julia.padberg@haw-hamburg.de
[2] LINFI Laboratory, Computer Science Department,
Biskra University, Biskra, Algeria
l.kahloul@univ-biskra.dz

Abstract. The evolution in software and hardware systems from classical systems with rigid structures to open, dynamic, and flexible structures has inspired the extension of Petri nets to reconfiguration. The idea of reconfiguring Petri nets was launched in the early nineties and since then has been developed by several researchers at different levels of formalization. Researchers in this field have achieved a large amount of theoretical results and of practical applications. The aim of this paper is to present an overview of reconfigurable Petri nets dealing with several aspects including: the fundamental, theoretical basis, application domains, results at the verification/analysis level as well as practical tools. The paper finally discusses some future research directions.

Keywords: Reconfigurable Petri nets · Petri net transformations
Dynamic infrastructures

1 Introduction

The characteristic feature of reconfigurable Petri nets, consisting of a Petri net and a set of rules that can modify it, is the possibility to discriminate between different levels of change. They provide powerful and intuitive formalisms to model dynamic software or hardware systems that are executed in dynamic infrastructures. These infrastructures are dynamic since they are subject to change as well and since they support various applications that may share some resources. Such dynamic software or hardware systems have become increasingly more common but are difficult to handle. Modelling and simulating dynamic systems require both the representation of their processes and of the system changes within one model. As the underlying type of Petri net can vary (for example being place/transition nets, object nets, timed and/or stochastic nets, or high-level nets) this approach can be considered a family of formal modelling techniques. Reconfigurable Petri nets are an instantiation of abstract transformation systems that are formulated in category theory. The fundamental idea is to characterize those categories that allow double-pushout transformations: therefore only the diagrammatic descriptions are needed. This has the advantage of a thorough theory that yields a vast amount of results concerning the transformation part.

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 201–222, 2018.
https://doi.org/10.1007/978-3-319-75396-6_11

Reconfigurable Petri nets have been applied in various application areas where complex coordination and structural adaptation at run-time are required (e.g. mobile ad-hoc networks [62], communication spaces [21,51], ubiquitous computing [10,24], concurrent systems [45], workflows in dynamic infrastructures [29], flexible manufacturing systems [71], reconfigurable manufacturing systems [31]). They improve the expressiveness of Petri nets as they increase flexibility and change while allowing the transitions to fire. This greater expressiveness yields rich models with very large state spaces. The state space of the model is even more complex because states are not capturing only the change of the markings but also structure and connectivity changes [2,57]. The reachability graph is composed of several subgraphs [57], each representing the state space of each accessible configuration in the modelled reconfigurable system. To deal with such growing complexity there are two main ideas: relying on invariant properties or to check the state space for such properties. The first idea is followed mainly in a more informal approach, the second leads to explicit model checking of reconfigurable Petri nets, both are discussed in Sect. 5.

The paper is organized as follows: The next section sketches related work, followed by the section introducing reconfigurable Petri nets. Section 3 gives their formal definition as well as an ongoing example and in Sect. 4 this notion is extended to several different types of Petri nets. Subsequently, we illustrate various application areas and concentrate on modelling reconfigurable manufacturing systems in Sect. 6. Then we discuss two tools that have been developed explicitly as tools for reconfigurable Petri nets (see Sect. 7). Finally we discuss some ideas concerning future work in the conclusion.

This paper overlaps with previously published papers, it presents and structures their results.

2 Related Work

The work in this area started in the beginning of the nineties [20] with transformation of various Petri net types as a refinement concept. In several papers [20,54,60] the use of net transformations in algebraic high-level nets had been investigated and a rule-based refinement concept had been developed that ensured safety and liveness properties under specific conditions (e.g. in [55,61,63]). Moreover, based on a categorical framework, namely abstract Petri nets [56] that comprises various low- and high-level types of Petri nets, the results on horizontal and vertical structuring had been made available for these net types. At the turn of the century the idea of adaptation of dynamic systems became an important research topic. The notion "reconfigurable Petri nets" had been coined at INRIA [3] where the reconfiguration had first been the replacement of places.

Zero-safe nets are place-transition nets [12] that allow the distinction between observable and hidden states. In [11] reconfigurable nets are defined as a special case of zero-safe nets. In this approach the post-domain of a transition is not static, but depends on the colours of the consumed tokens.

In [2] net rewriting systems have introduced rules based on a partial morphism between left-hand and right-hand side of the rule. In [47] marked-controlled net rewriting system have been based on a place/transition nets and a graph rewriting for changing configurations. In [4] open nets (a generalisation of place/transition nets suited to model open systems interacting with the surrounding environment) are equipped with suitable classes of reconfiguration rules whose application preserves the observational semantics of the net. The "nets-within-nets" formalism was introduced by Valk in [74] and was combined with workflow Petri nets [67] to develop an approach for the specification and code generation of dynamically reconfigurable embedded systems in [75].

Another approach, called "improved net rewriting systems" (INRS) [41–43] concentrates on preserving important Petri net properties, namely liveness, boundedness and reversibility. It is based on a set of fixed building blocks and rewrite rules with fixed interfaces for the left-hand and the right-hand side. INRS is the basis for the application we investigate in Sect. 6.1, namely Reconfigurable Manufacturing Systems (RMSs) [39]. RMSs allow changeable structures at runtime for various kinds of industrial production systems. To model explicitly the reconfiguration of RMSs several variants of reconfigurable Petri have been proposed and applied. The different proposals can be classified into three principal classes: graph transformation based approaches, approaches based on rewriting net systems, and finally hybrid approaches. Based on [19] formalisms and methods have been developed to design, simulate and verify RMSs [75]. Reconfigurable Object Nets (RON) [8,68] have been used to propose an approach for the design, simulation and verification of RMSs [31–34].

Besides graph transformation based approaches and the INRSs based approach there are proposals that combine Petri nets with rewriting logics, π-calculus, or algebraic specifications to define reconfigurable models. In [35,36] rewriting logic is combined with a variant of the recursive Petri nets [26] to describe the reconfigurability through the ability to model dynamic creation of threads. In [77] object Petri nets are combined with π-calculus, where object-oriented Petri net are employed to depict the static structure and behaviours of the RMS while the π-calculus is used to describe the dynamic structure of the system. In [15] adaptive Petri nets are proposed to specify self-adaptive systems. The adaptation is achieved by the learning ability of neural networks. In [13] the authors deal with transformations over Petri nets as algebraic specifications, thus they developed a tool to set up a basic set of transformation primitives, including adding/removal of nodes, changing the marking and setting connections.

3 Reconfigurable Petri Nets

We now define place/transition nets formally, to have a basis for the definition of rules and transformations later on. Subsequently, we present an example from dynamic hardware reconfiguration.

3.1 Basic Concepts

We use the algebraic approach to Petri nets, where the pre- and post-domain functions $pre, post : T \to P^{\oplus}$ map the transitions T to a multiset of places $P^{\oplus}$ given by the set of all linear sums over the set P. A marking is given by $m \in P^{\oplus}$ with $m = \sum_{p \in P} k_p \cdot p$. The multiplicity of a single place p is given by $(\sum_{p \in P} k_p \cdot p)_{|p} = k_p$. The $\leq$ operator can be extended to linear sums: For $m_1, m_2 \in P^{\oplus}$ with $m_1 = \sum_{p \in P} k_p \cdot p$ and $m_2 = \sum_{p \in P} l_p \cdot p$ we have $m_1 \leq m_2$ if and only if $k_p \leq l_p$ for all $p \in P$. The operations "+" and "–" can be extended accordingly.

Definition 1 (Place/transition nets). *A (marked place/transition) net is given by $N = (P, T, pre, post, cap, lab_P, lab_T, m)$ where P is a set of places, T is a set of transitions. $pre : T \to P^{\oplus}$ maps a transition to its pre-domain and $post : T \to P^{\oplus}$ maps it to its post-domain. Moreover $cap : P \to \mathbb{N}_+^{\omega}$ assigns to each place a capacity (either a natural number or infinity ω), $lab_P : P \to A_P$ is a label function mapping places to a name space, $lab_T : T \to A_T$ is a label function mapping transitions to a name space and $m \in P^{\oplus}$ is the marking denoted by a multiset of places.*
A transition $t \in T$ is m-enabled for a marking $m \in P^{\oplus}$ if we have $pre(t) \leq m$ and $\forall p \in P : (m + post(t))_{|p} \leq cap(p)$. The follower marking m' is computed by $m' = m - pre(t) + post(t)$ and represents the result of a firing step $m[t > m'$.

Net morphisms are given as a pair of mappings for the places and the transitions preserving the structure, the decoration and the marking. Given two nets N_1 and N_2 as in Definition 1 a net morphism $f : N_1 \to N_2$ is given by $f = (f_P : P_1 \to P_2, f_T : T_1 \to T_2)$, so that $pre_2 \circ f_T = f_P^{\oplus} \circ pre_1$ and $post_2 \circ f_T = f_P^{\oplus} \circ post_1$ and $m_1(p) \leq m_2(f_P(p))$ for all $p \in P_1$. The labels and the capacity need to remain the same when mapping one net to another. Moreover, the morphism f is called strict if both f_P and f_T are injective and $m_1(p) = m_2(f_P(p))$ holds for all $p \in P_1$. A rule in the algebraic transformation approach is given by three nets called left-hand side L, interface K and right-hand side R, respectively, and a span of two strict net morphisms $K \to L$ and $K \to R$. Then an occurrence morphism $o : L \to N$ is required that identifies the relevant parts of the left hand side in the given net N.

A transformation step $N \overset{(r,o)}{\Longrightarrow} M$ via rule r can be constructed in two steps by the commutative squares (1) and (2) in Fig. 1. Given a rule with an occurrence $o : L \to N$ the *gluing condition* has to be satisfied in order to apply a rule at a given occurrence. Its satisfaction requires that the deletion of a place implies the deletion of the adjacent transitions, and that the deleted place's marking does not contain more tokens than the corresponding place in L. In this collection [37] such double-pushout transformations are explained in more detail for attributed graphs.

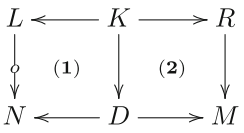


Fig. 1. Net transformation

Reconfigurable place/transition nets exhibit dynamic behaviour using the token game of place/transition nets and using net transformations by applying

rules. So, a reconfigurable net as in Definition 2 combines a net with a set of rules that modify the net [18,19].

Definition 2 (Reconfigurable place/transition nets). *A reconfigurable place/transition net $RN = (N, \mathcal{R})$ is given by a net N and a set of rules $\mathcal{R}$.*

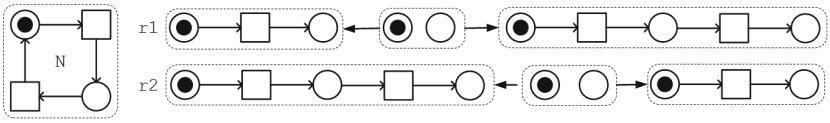


Fig. 2. Cyclic net with rules

Example 1 (Modifying a cyclic process). As an abstract example of a dynamic system we model a cyclic process that can either be executed or modified using the reconfigurable Petri net (N, {r1, r2}). Fig. 2 depicts a simple place/transition net N and the rules r1 and r2. The net describes a cyclic process that executes one step and then returns to the start. The modifications in rule r1 change the process by inserting additional sequential steps. Rule r2 deletes an intermediate step. In Fig. 3 the application of rule r1 to N is given. First a match of the left hand side of the rule is given by the occurrence morphism indicated by the light grey colour of the places and transitions in L and N. The gluing condition holds since the occurrence morphism preserves the token.

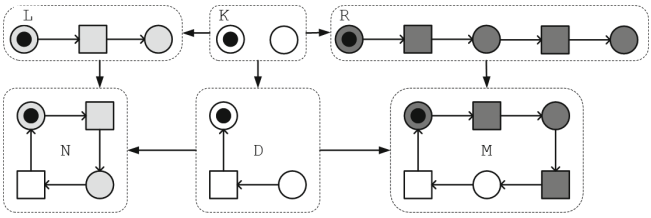


Fig. 3. Application of rule r1 to N

In the first step the transition, which is coloured light grey, is deleted by the construction of the net D and in the second step the intermediate place and its adjacent transitions (coloured dark grey) are added.

3.2 Reconfigurable Computing

Here we give a realistic example that illustrates the use of reconfigurable place/transition nets in dynamic hardware reconfiguration. Reconfigurable computing allows performing several functions on the same hardware with only few modifications. This economic solution avoids the re-fabrication of new hardware when new functions are required. Reconfigurable computing replaces classical fixed digital circuits by FPGA (Field Programmable Gate Array) technologies. An FPGA is a matrix of interconnected logic blocs and it is characterized by its flexibility on both interconnections and logic blocs levels. This flexibility is ensured by programming bits which can be updated rapidly, thus

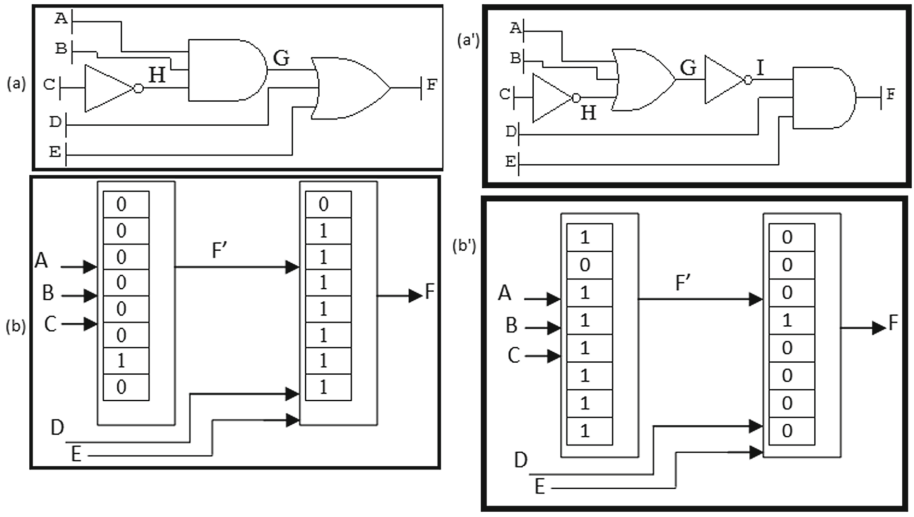


Fig. 4. Reconfigurable computing using LUT

enabling the implementation of several functions in a short time. Usually, the routing mechanism is implemented through programmable gates and the computation exploits lookup-tables (LUTs). A programmable gate is closed or opened due to the value of the programming bit P. A lookup-table is a small circuit which can compute any basic logic function of n inputs by programming its 2^n programming bits. The values of the set of programming bits are saved in an SRAM (Static Random Access Memory). Modifying the values of these programming bits in the SRAM reconfigures the FPGA at two levels: routing connections and computational structure, thus the behaviour of the FPGA is reconfigured. We present in Fig. 4(a) a first configuration which implements the logic function: $F = (A \times B \times \neg C) + D + E$ (known as seat-belt warning light system) as an example. The realisation of this circuit using a LUT table is made using two 3-LUTs (with 8 programming bits) connected as shown in Fig. 4(b). The first LUT uses A, B and C as inputs and generates F'. F', D and E will be the inputs for the second 3-LUT. Figure 4(a′) shows the circuit after a reconfiguration and how this reconfiguration is made through LUT is Fig. 4(b′). In the approach proposed by [38,76] modelling digital circuit (as digital gates) using Petri nets is based on the modelling of signals. The behaviour of a signal x is modelled using two places denoted $x0$ and $x1$ (representing respectively the two signal states 0 and 1) and two transitions $+x$

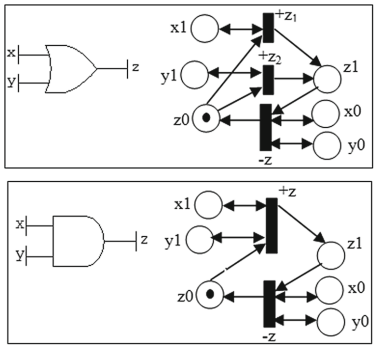


Fig. 5. Petri net models for the "Or gate" and the "And Gate"

and $-x$ (which model respectively the rising of the signal x from 0 to 1 and the falling of the signal x from 1 to 0).

If there are many transitions which rise (resp. fall) the signal x then we denote these transitions as $+x_i$ (resp. $-x_i$). The signal y rises through the transition $+y$ when the signal x is in a 0 level (place $x0$). Then, the signal y will fall by firing the transition $-y$ when the signal x is in the level 1 (place $x1$). The signal x is the input of the gate which will be controlled, eventually, by another circuit.

Using the previous method, the Fig. 5 presents the Petri net models for the AND and OR gates. Hence, we depict in Fig. 6 the models of the two configurations presented in Fig. 4. The formalisation of reconfiguration using the double-pushout (see Fig. 7) requires the definition of a transformation rule from the net TN_1 representing the model of Fig. 6(a) toward the net TN_2 representing the net of Fig. 6(b).

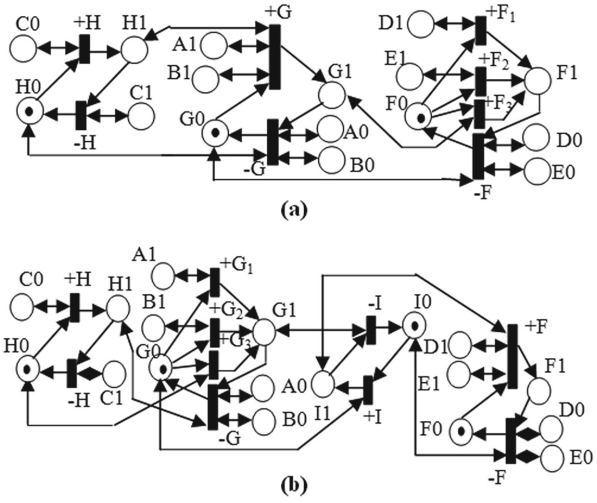


Fig. 6. Net models for the configurations in Fig. 4

4 Types of Reconfigurable Petri Nets

4.1 Reconfigurable Low-level Nets

Place/transition nets as given in Sect. 3 are well-known and widely used. Various types of reconfigurable place/transition nets have been proposed, mostly differing in the additional control structures. In [57] new features have been added to gain an adequate modelling technique where transition labels have been introduced that may change, when the transition is fired. This allows a better coordination of transition firing and rule application, for example one can ensure that a transition has fired (repeatedly) before a transformation may take place. This last extension is conservative with respect to Petri nets as it does not change the net behaviour, but it is crucial for the coordination of rule application and transition firing.

Reconfigurable place/transition nets with individual tokens [51] have tokens that can be identified as individual objects. Hence, markings are multi-sets of distinguished elements rather than amounts of indistinguishable black tokens. This notion of token facilitates the definition of net processes and hence yields a process semantics [21]. Petri nets do not inherently provide a way to model time but various approaches have been suggested to extend Petri nets by notions of time, as for example [6] or [30]. In [22] timed Petri nets extend place/transition nets attaching time durations to transitions and timestamps to tokens and are

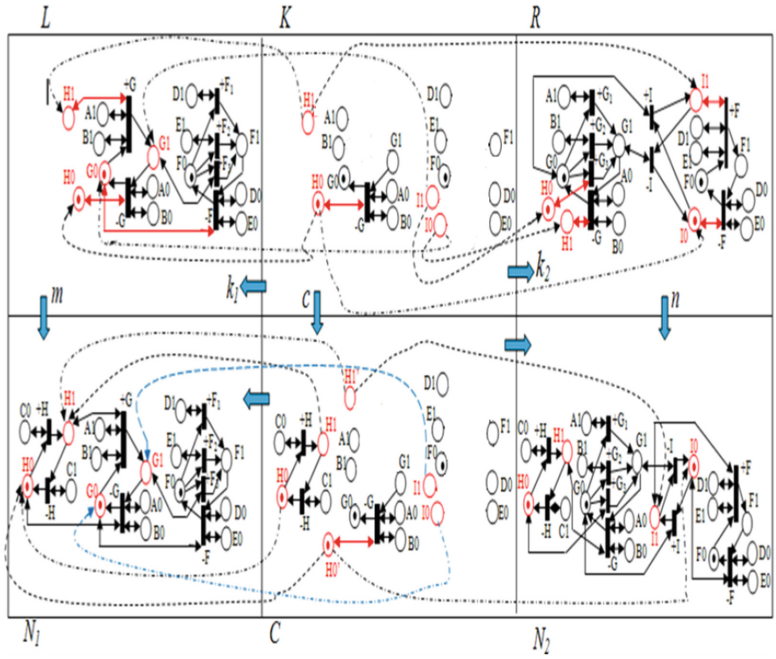


Fig. 7. Double pushout for the example

equipped with rules and transformations. Elementary nets have been shown to be a special case of abstract nets [56]. Hence reconfigurable elementary nets can be considered as well, but they have not be used explicitly.

The above mentioned Petri net types have been proven to yield $\mathcal{M}$-adhesive categories, this means that the results given in Sect. 5.1 are valid for each of these net types.

4.2 Reconfigurable Stochastic Nets

Stochastic Petri nets (SPN) are high level Petri nets that have been proposed to model stochastic and random systems. The most used variant are SPNs in the sense of [48]. They are an extension of timed transition Petri nets where the duration is no longer deterministic but stochastic with a predefined probabilistic law. The Generalised Stochastic Petri Nets (GSPN) [5] are an extension of SPNs where transitions can be of two kinds: stochastic or immediate. The use of GSPNs allows designers to evaluate performance for the modelled systems and to study several quantitative parameters. Introducing reconfiguration into GSPNs is an ambitious issue which will yield a new reconfigurable stochastic Petri net formalism. A first approach has been developed in some recent work [69,70]. There GSPNs have been extended to reconfigurable GSPN using the Improved Net Rewriting Systems [43] to provide a new formalism called INRS-GSPN. This approach has been used to design Stochastic RMSs and to allow performance

evaluation of configurations. Based on the INRS approach the reconfiguration of GSPN is expected to preserve specific required qualities as: liveness, boundedness and reversibility.

4.3 Reconfigurable High-level Nets

Algebraic high-level (AHL) nets are Petri nets combined with algebraic specifications [60]. In contrast to low-level nets AHL nets comprise a data type part, so that the tokens are values in an underlying algebra of the given signature rather than indistinguishable black tokens. The arcs are inscribed by terms over the signature and firing of a transition requires the assignment of these variables to the values of the available tokens. Moreover, transitions are provided with guards, called firing conditions and given by equations. Guards allow the firing of a transition only if the tokens that are to be consumed satisfy the firing conditions of that transition. The operational behaviour of AHL nets is given analogously to the operational behaviour of low-level nets. The activation of a transitions requires an assignment of the variables in the environment of the transition, such that the assigned pre-domain is included in the marking and the firing conditions of the transition hold. This assignment is then used to compute the follower marking, obtained by decreasing the marking by the assigned pre-domain and increasing the result by the assigned post-domain. Algebraic higher-order (AHO) net are high-level nets where tokens can be place/transition nets and net transformation rules. Thus AHO nets follow the paradigm "nets as tokens", introduced by Valk in [73] but extend this paradigm to "nets and rules as tokens". AHO nets are used for controlling firing steps and transformations of low-level nets that has been introduced in [28] with AHL nets that contain place/transition nets and transformation rules as tokens. Reconfigurable Object Nets (RONs) as given in [28] are a restriction of AHO nets, so that firing of RON-transitions may only involve firing of object net transitions, transporting object net tokens through the high-level net, or applying net transformation rules to object nets. Net transformation rules model net modifications such as merging or splitting of object nets, and net refinements. Both AHL nets as well as AHO nets are available with individual tokens [21,51].

The above mentioned high-level net types have also been proven to yield $\mathcal{M}$-adhesive categories, so the results given in Sect. 5.1 hold for each of them.

5 Results

First, we sketch the basics for abstract transformation systems and the results obtained in that way. We now discuss some of the results concerning control structures and verification of reconfigurable nets.

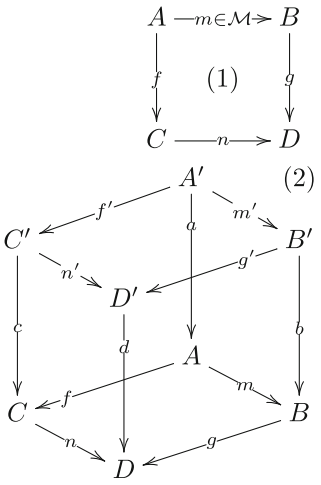
5.1 Results for Abstract Transformation Systems

The basic idea of transforming Petri nets is the same as transforming graphs in the algebraic approach, e.g. in [16]. The theoretical backbone of these

transformations are $\mathcal{M}$-adhesive transformation systems that yield the abstract transformation system. They are formulated in terms of category theory and can be considered as a unifying framework for graph and Petri net transformations providing enough structure that most notions and results from algebraic graph transformation systems hold. Such a categorical approach has the advantage that the results in this framework hold for any category which satisfies the set of assumptions for specific classes of morphisms. $\mathcal{M}$-adhesive transformation systems have been instantiated with various types of graphs, as hypergraphs, attributed and typed graphs, structures, algebraic specifications, various Petri net classes, elementary nets, place/transition nets, Colored Petri nets, or algebraic high-level nets, and more (see [16]). $\mathcal{M}$-adhesive transformation systems allow a uniform description of the different notions and results based on a class $\mathcal{M}$ of specific monomorphisms that have to be PO-PB-compatible, that is:

- Pushouts along $\mathcal{M}$-morphisms exist and $\mathcal{M}$ is stable under pushouts.
- Pullbacks along $\mathcal{M}$-morphisms exist and $\mathcal{M}$ is stable under pullbacks.
- $\mathcal{M}$ contains all identities and is closed under composition.

The fundamental construct for $\mathcal{M}$-adhesive categories and systems are van Kampen squares of $\mathcal{M}$-morphisms.



Definition 3 ($\mathcal{M}$-Adhesive Category). *Given a class M of PO-PB compatible monomorphisms in a category $\mathbf{C}$, then $(\mathbf{C}, \mathcal{M})$ is called $\mathcal{M}$-adhesive category, if pushouts along $\mathcal{M}$-morphisms are $\mathcal{M}$-van Kampen squares, that is for any commutative cube (2) with (1) in the bottom and back faces being pullbacks, the following holds: the top is pushout $\Leftrightarrow$ the front faces are pullbacks.*

An $\mathcal{M}$-adhesive transformation system $AHS = (\mathbf{C}, \mathcal{M}, \mathcal{R})$ consists of an adhesive $\mathcal{M}$-category $(\mathbf{C}, \mathcal{M})$ and a set of rules $\mathcal{R}$.

Based on this categorical framework we have the following results for those Petri net types that have been shown to be adhesive categories:

- Negative application conditions [25] allow specifying undesired context. The rules are equipped with additional nets that show the context in which the rule should not be applied, see Fig. 12 in Sect. 7.2.
- Confluence and independency results as parallel and sequential independence, local Church-Rosser, conflict and causal dependency describe how rules behave in specific contexts. Indepenency conditions are given for two direct transformations being applied to the same net, so that they can be applied in arbitrary order leading to the same result. Properties of dependent transformations have been investigated as well (see e.g. [16]).

- Critical pair analysis as known from term rewriting are used to check for confluence. Critical pairs specify the minimal instance of a conflicting situation. From the set of all critical pairs the items causing conflicts or dependencies are extracted. Local confluence can be shown for abstract transformation systems using the concept of critical pairs (see e.g. [17]).
- Net transformation units have been instantiated from HLR units [9] that are a generalisation of graph transformation units (e.g. [40]). Net transformation units [27] provides syntactic and semantic means for structuring net transformations. They regulate the application of rules to nets encapsulating the rules and control expressions, see also Fig. 12 in Sect. 7.2.

5.2 Control Structures

Control structures to reconfigurable Petri nets are required due to the expressive power of the interplay between rule application and firing behaviour. The available control structures can be differentiated into those that arise from Petri nets, as transition priorities, inhibitor arcs or capacities and those that arise from graph transformation systems as negative application conditions or transformation units.

In [58] priorities for transitions and inhibitor arcs – both well-known concept in Petri nets – have been introduced to reconfigurable Petri nets. The results of $\mathcal{M}$-adhesive transformation system for reconfigurable Petri nets with priorities are ensured by proving the corresponding category to be $\mathcal{M}$-adhesive. Moreover, it was shown that Petri nets with inhibitor arcs yield an $\mathcal{M}$-adhesive category as well.

Other control structures determine the application of rules. They concern the situation that may or may not be given or they concern the order of the rules to be applied. Net transformation units are the transfer of graph transformation units (see [40]) to reconfigurable Petri nets and have been achieved using the abstract formulation of HLR units [9]. Control conditions can be given by regular expressions, describing in which order and how often the rules and imported units are to be applied. For an example see Sect. 7.2.

Negative application conditions for reconfigurable Petri nets have been introduced in [66] and provide the possibility to forbid certain rule applications. These conditions restrict the application of a rule forbidding a certain structure to be present before or after applying a rule in a certain context. Such a constraint influences thus each rule application or transformation and therefore changes significantly the properties of the net transformation system, again see Sect. 7.2.

5.3 Verification

Ensuring that relevant properties as liveness, reachability of specific states or boundedness hold, is central for the adequate use of a modelling technique. Since the classical analysis techniques fail there are two main possibilities for reconfigurable Petri nets that we discuss subsequently: either these properties are preserved during the transformation or they are proven explicitly.

Invariants. Invariant properties can be achieved if the rules preserve the corresponding properties. Conditions for rules have been formulated so that the rule application ensures safety properties [61] and liveness [72] in the resulting net provided that the original net satisfies these properties.

Although Net Rewriting Systems (NRSs) allow the formalisation of dynamics in Petri nets, they do not preserve properties such as liveness, boundedness (or safeness), and reversibility. Based on the approach of NRSs developed in [2,45], the authors of [42–44] propose Improved Net Rewriting Systems (INRS). The INRS can not only change dynamically the structure of a Petri net but also preserve important behavioural properties. In fact, preserving liveness, boundedness (or safeness), and reversibility is important in several systems such as Reconfigurable Manufacturing Systems (RMSs), where the previous properties are vital to guarantee that the RMS is free from deadlocks, has finite states, and behaves cyclically, respectively. Reconfiguration in INRS replaces some subnet from the source net by another subnet yielding a new net. The approach defines net block class libraries (well-formed net blocks) and the reconfiguration process substitutes a well-formed subnet of any live bounded reversible net with another well-formed net block of the same interface type. The INRS approach was applied in [43] to design reconfigurable Petri net controllers for the supervision of RMSs.

Model Checking. The non-deterministic and concurrent behaviour of reconfigurable Petri nets inhibit the determination of emerging properties. In [64] model checking of reconfigurable place/transition nets has been developed, implemented and proven to be correct. Maude is a mature theory of rewriting logic and is feasible for modelling reconfigurable Petri nets, e.g. [14]. Model-checking of reconfigurable Petri nets [64] is achieved by a conversion of a net and a set of rules into a Maude specification. This specification can be model-checked for properties expressed in linear temporal logic (LTL) using the Maude module LTLR with extensions for rewrite rules and properties such as fairness. The model-checking of reconfigurable nets allows the verification of reachability of states or the absence of deadlocks.

6 Applications

In this section we introduce some of the application areas for reconfigurable Petri nets. First we investigate the use reconfigurable nets for manufacturing systems in some detail. Subsequently, other application areas are merely sketched.

6.1 Reconfigurable Manufacturing Systems

Reconfigurable Manufacturing Systems (RMSs) [39] represent a new innovative approach providing “production systems” with a changeable structure at runtime. Changing the structure can be a solution to satisfy dynamic customers requirement as well as to resolve unpredictable system failures. The use of reconfigurable Petri nets in the design of RMSs offers high level specification, simulation, verification, performance analysis, and code generation at the software

level. Using Reconfigurable Object Nets (RONs) [8] a formal approach [31] for the design, simulation, and verification of RMSs is proposed.

It starts with an informal or semi-formal description of the RMS, builds a RON model for simulation or formal verification. The semi-formal description of RMSs is often given as bloc-diagrams describing tasks and the flow control in the RMS. Figure 8 depicts how the approach is applied by the designer. As a demonstration of the depicted approach in Fig. 8, let's consider the following RMS inspired from [50]. This RMS contains two manufacturing cells (MC_1, MC_2), a storage AS/AR (automated storage and retrieval system), and an AGV (automated guided vehicle). The system requires two raw materials R_1 and R_2 and it produces one final product A (see Fig. 9(a)). The production process passes respectively by MC_1 (Fig. 9(b)) then MC_2 (Fig. 9(c)). The MC_1 is composed of a CNC lathe machine, a CNC milling machine, a robot and a buffer. In MC_1, R_1 and R_2 are processed firstly by the lathe machine before the milling machine. The MC_2 is composed by an assembly machine (which assembles the two products into one product A), a robot and a buffer. During the life cycle of the system, the production process meets

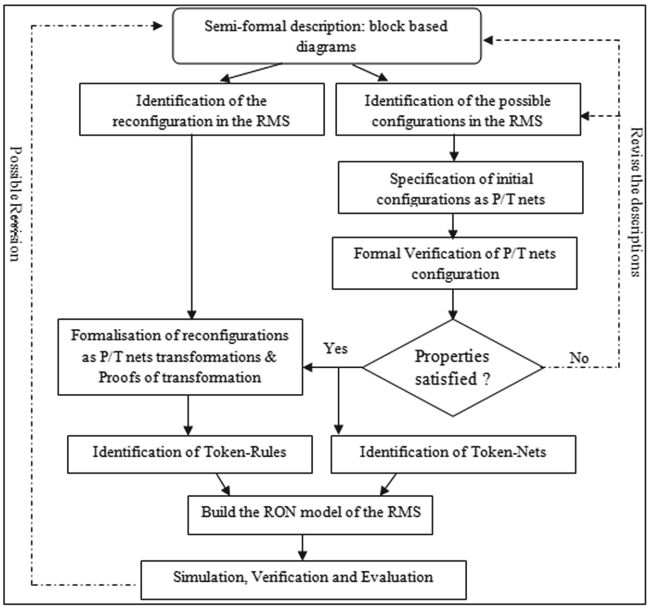


Fig. 8. Designing RMSs using RONs based-approach

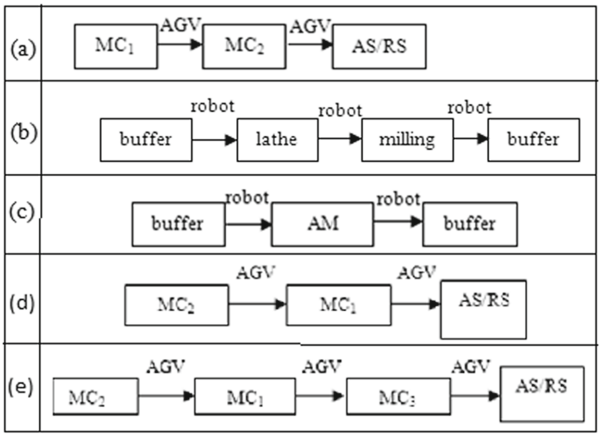


Fig. 9. (a) Manufacturing of product A, (b) Flow in MC_1, (c) Flow in MC_2, (d) Reconfiguration 1: a new product B, (e) Reconfiguration 2: introducing MC_3.

two reconfigurations. The first reconfiguration is triggered by a new customers' requirement for a product B (see Fig. 9(d)). To produce B the flow must be converted (i.e. the assembly is done before the lathe and the milling). The second reconfiguration (see Fig. 9(e)) is triggered by the inspection team of the production process requiring the introduction of a new manufacturing cell MC_3 (inspection cell). The inspection cell MC_3 is composed of a coordinate measuring machine (CMM) and a set of buffers. Using the RON formalism the three configurations are considered as token nets and the reconfigurations are considered as token rules. Figure 10(a) and (b) represent the two first configurations. The reconfiguration is described as a double-pushout (See Fig. 10(c)). As an example, the Fig. 10(d) shows the production (L, I, R) used in the double-pushout.

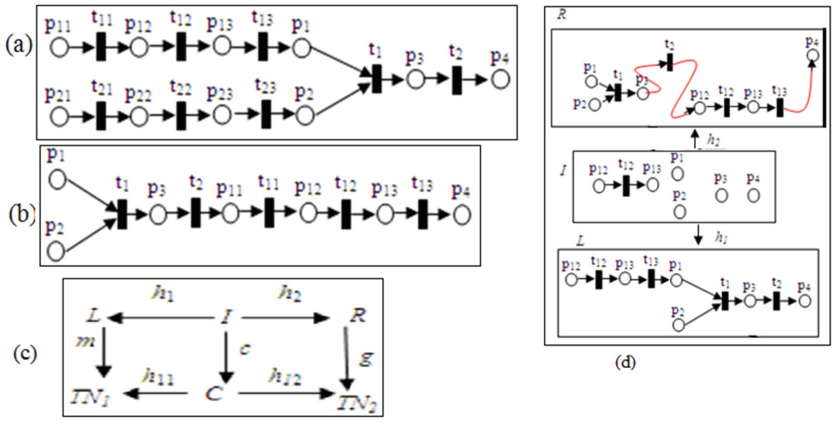


Fig. 10. (a) Initial configuration, (b) Second configuration, (c) Double Pushout, (d) The production rule (L, I, R).

By determining the set of token nets and the set of rules the whole RMS behaviour can be modelled by a RON model as shown in Fig. 11 in Sect. 7.1. This RON model can be used to visualise, simulate, and verify the RMS behaviour.

6.2 Other Applications

Communication Platforms. A general modelling framework for communication platforms and scenarios has been presented in [21] using reconfigurable algebraic high-level nets. This framework employs an integration of Petri nets, algebraic data types and net transformation techniques. It allows the analysis of the evolution of communication platforms, the analysis of scenario evolutions and the investigation of user interactions on communication platforms. Reconfigurable AHL nets have also been used in [51,52] for a case study on modelling a concrete communication platform – namely Skype. The behaviour of the Skype clients has been modelled in detail and the whole system specification has been demonstrated for concrete use case scenarios. For these scenarios model properties have been formulated and validated.

Ubiquitous Computing Systems (UCSs). Lets computing appear to occur using any device, in any location, and in any format. Underlying technologies comprise the internet, advanced middleware, mobile devices, constantly available networked sensors and microprocessors, and so on. USCs penetrates almost imperceptibly in everyday life. To ensure a solid operation, a UCS needs reliable and efficient communication between its distributed computing components. [24] presents a formal approach based on reconfigurable algebraic higher order nets with individual tokens (AHOI) nets [51]. This approach allows modelling the synchronous and asynchronous communication in UCSs and is used for modelling a smart home. Emergency scenarios using mobile ad-hoc networks have been investigated extensively [10,23,62]. In emergency scenarios, we can obtain an effective coordination among team members constituting a mobile ad-hoc network through the use of net system and rule tokens. From an abstract point of view, mobile ad-hoc networks consist of mobile nodes which communicate with each other independent of a stable infrastructure, while the topology of the network constantly changes depending on the current position of the nodes and their availability. The net structure can be adapted to new requirements of the environment during run time by a set of rules, i.e. token firing and net transformation can be interleaved with each other.

7 Tools

Since the rewriting in reconfigurable Petri nets is in most cases given as a kind of graph transformation, general purpose graph transformation tools as AGG [1] that supports the modelling, the simulation and the analysis of typed attributed graph transformation systems, are likely candidates. But specific tools have been developed as well. In Sects. 7.1 and 7.2 we introduce tools that are in use at this point. The MCReNet-tool [49] is a tool for the modelling and verification of marked-controlled reconfigurable Petri nets [46] and was the first implementation that has explicitly dealt with reconfigurable Petri nets.

7.1 RON-Editor

One of the tools concerned with reconfigurable Petri nets is RON-Editor [7]. The RON-editor is based on reconfigurable object nets [28]. It is an open source and free tool [68]. The RON-editor supports users to create, delete and edit parts of the model like object nets, net transformations rules and a top-level RON. The RON-editor makes several checks (e.g. for correct typing of tokens on RON places, to guarantee that mappings in rules satisfy net morphism properties) that help the user to obtain consistent RONs. Additionally, the editor comprises a simulator using the AGG engine to simulate the application of rules and thus firing of high-level transitions in the RONs created with the editors. The set of visual editors have been realized as Eclipse plug-ins using the Eclipse Modelling Framework (EMF) and Graphical Editor Framework (GEF) plug-ins. As an example, the simulation of the RON model presented in Sect. 6.1 is depicted

in Fig. 11. There the top-right window depicts the system level net (the RON model), the top-left window depicts the object net TN_1 in the place np_1 and the lower window shows the transformation rule (token-rule r_1) which is applied to the object net TN_1. The RON-editor can simulate the behaviour of the system level net as well as the behaviour of the object nets. The transition "transform-transition 1" is green which means that it is enabled.

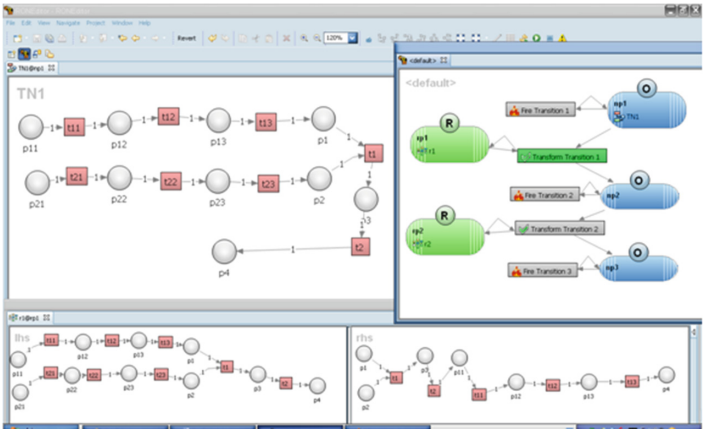


Fig. 11. An example of reconfigurable object nets with RON-Editor. (Color figure online)

7.2 ReConNet

ReConNet [59,65] is an open source project that has been initiated at the HAW Hamburg developing a tool for editing and simulating reconfigurable decorated nets. It provides an intuitive graphic-based user interface that allows the user to create, modify and simulate reconfigurable nets. It facilitates non-deterministic application of rules and firing steps. There are different simulation options executing a definable amount of steps: only transitions are fired, only rules are applied, or both may happen. Control structures that are available comprise negative application conditions, transformation units and dynamic transition labels (see Sect. 5). In Fig. 12 the net `N` from Example 1 in Sect. 3 is depicted together with a third rule `r3` that reverts (various instances and derivations) the arrows of the net. The requirement "the arrows only may be turned if there is no token on the second place" ensures that token do not directly go back. This rule is shown in four windows: the first presents the negative application condition that ensures the requirement. The next three windows present a rule where the intermediate transition is deleted and then a transition with a new label is inserted so that the arrow point in the other direction. Note, that in this illustration the interface's window does not show a net explicitly. The transformation unit's control structure `tu1: (r1 | r2)*;(r3!)` guarantees that `r1` and `r2` (as given in Example 1) are executed arbitrarily often, subsequently they

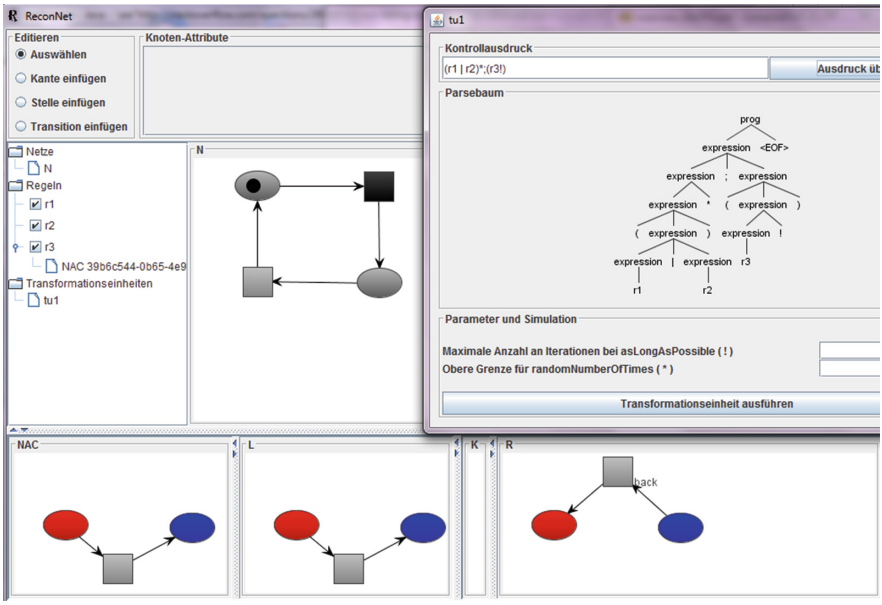


Fig. 12. GUI of ReConNet

are followed by rule `r3`. The exclamation mark `!` denotes that this expression is repeated *as long as possible*. Hence, once the turning has started, it goes on until all arrows have been turned around.

8 Conclusion

In this contribution we have given a comprehensive compilation of the results that have been achieved for various types of reconfigurable Petri nets in the last two decades. Reconfiguration is a topic that is quite virulent in very different areas. For some of these modelling the dynamic change with reconfigurable Petri nets seems to be very promising. We have sketched some application areas and have given examples how to tackle the issue with reconfigurable nets. The theoretical research in reconfigurable Petri nets has provided important results on several types of nets. However, the proposed case studies are often academic ones illustrated only to explain the feasibility of the proposed formal approaches. In order to tackle with these limits, future work has to focus on the following aspects:

– Enlarge the application domain of reconfigurable Petri nets to handle new technologies, for example cloud computing systems and the internet of things. These later are the most suitable systems where mobility, flexibility, and dynamics are inherent characteristics. These systems require new reliable hardware devices and new reliable software protocols and drivers, thus reconfigurable Petri nets should be suggested as a validation and verification technique.

– Invest in the automatic tools for modelling, simulation, and verification of reconfigurable Petri nets. The current tools remain at the prototypic and academic level.
– Extend reconfigurable Petri nets to performance analysis. Most verification results for reconfigurable Petri nets concern qualitative verification. However, in real systems the designer expects often performance evaluation and quantitative measurements. Such analysis is well developed in stochastic Petri nets and stochastic automata with some improved tools like GreatSPIN or UPPAAL. Stochastic graph transformations [4] provide attributed typed graph transformations systems for analysing transformation systems with stochastic methods and is a good basis for Future work to integrate stochastic features to reconfigurable Petri nets more formally.
  This comprises work on the integration of the INRS approach and the abstract transformation systems. We want to achieve the strong theoretical basis of the abstract transformation systems also for the INRS approach. The formulation of building blocks used in INRS independently of the underlying net types (similar to [53]) is the basis for a formal correctness proof.
– Integrate optimisation problems. Another, new application of reconfigurable Petri nets is the optimal configuration in reconfigurable systems. These use often evolutionary algorithms that find the optimal configuration after several reconfigurations of a random initial configuration. Combining reconfigurable Petri nets with evolutionary processing can yield new hybrid methods where both objectives are captured: optimisation and formal verification.

References

1. AGG. http://www.user.tu-berlin.de/o.runge/agg/index.html. Accessed 02 June 2017
2. Badouel, E., Llorens, M., Oliver, J.: Modeling concurrent systems: reconfigurable nets. In: Arabnia, H.R., Mun, Y. (eds.) International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 1568–1574 (2003)
3. Badouel, E., Oliver, J.: Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes within Workflow Systems. Research Report RR-3339. INRIA (1998)
4. Baldan, P., Corradini, A., Ehrig, H., Heckel, R., König, B.: Bisimilarity and behaviour-preserving reconfigurations of open Petri nets. Log. Methods comput. Sci. **4**, 126–142 (2008)
5. Bause, F., Kritzinger, P.S.: Stochastic Petri Nets: An Introduction to the Theory. Vieweg+Teubner Verlag, Cape Town (2002)
6. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. Softw. Eng. **17**(3), 259–273 (1991)
7. Biermann, E., Ermel, C., Hermann, F., Modica, T.: A visual editor for reconfigurable object nets based on the ECLIPSE graphical editor framework. In: 14th Workshop on Algorithms and Tools for Petri Nets (2007)
8. Biermann, E., Modica, T.: Independence analysis of firing and rule-based net transformations in reconfigurable object nets. Electron. Commun. EASST **10**, 1–13 (2008)

9. Bottoni, P., Hoffmann, K., Parisi-Presicce, F., Taentzer, G.: High-level replacement units and their termination properties. J. Vis. Lang. Comput. **16**(6), 485–507 (2005)
10. Bottoni, P., Rosa, F.D., Hoffmann, K., Mecella, M.: Applying algebraic approaches for modeling workflows and their transformations in mobile networks. Mob. Inf. Syst. **2**(1), 51–76 (2006)
11. Bruni, R., Melgratti, H., Montanari, U.: Extending the zero-safe approach to coloured, reconfigurable and dynamic nets. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 291–327. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_7
12. Bruni, R., Montanari, U.: Transactions and zero-safe nets. In: Ehrig, H., Padberg, J., Juhás, G., Rozenberg, G. (eds.) Unifying Petri Nets. LNCS, vol. 2128, pp. 380–426. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45541-8_12
13. Capra, L.: A pure SPEC-inscribed PN model for reconfigurable systems. In: 2016 13th International Workshop on Discrete Event Systems (WODES), pp. 459–465, May 2016
14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285**(2), 187–243 (2002)
15. Ding, Z., Zhou, Y., Zhou, M.: Modeling self-adaptive software systems with learning Petri nets. IEEE Trans. Syst. Man Cybern. Syst. **46**(4), 483–498 (2016)
16. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in TCS. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
17. Ehrig, H., Golas, U., Habel, A., Lambers, L., Orejas, F.: $\mathcal{M}$-adhesive transformation systems with nested application conditions. part 2: embedding, critical pairs and local confluence. Fundam. Inform. **118**(1–2), 35–63 (2012)
18. Ehrig, H., Hoffmann, K., Padberg, J., Prange, U., Ermel, C.: Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 104–123. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73094-1_9
19. Ehrig, H., Padberg, J.: Graph grammars and Petri net transformations. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 496–536. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_14
20. Ehrig, H., Padberg, J., Ribeiro, L.: Algebraic high-level nets: Petri nets revisited. In: Ehrig, H., Orejas, F. (eds.) ADT/COMPASS -1992. LNCS, vol. 785. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57867-6
21. Gabriel, K., Ehrig, H.: Modelling of communication platforms using algebraic high-level nets and their processes. In: Heisel, M. (ed.) Software Service and Application Engineering. LNCS, vol. 7365, pp. 10–25. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30835-2_2
22. Gabriel, K., Lingnau, P., Ermel, C.: Algebraic approach to timed Petri nets. Electron. Commun. EASST **47**, 1–14 (2012)
23. Golas, U., Hoffmann, K., Ehrig, H., Rein, A., Padberg, J.: Functorial analysis of algebraic higher-order net systems with applications to mobile ad-hoc networks. ECEASST **40**, 1–24 (2010)
24. Gottmann, S., Nachtigall, N., Hoffmann, K.: On modelling communication in ubiquitous computing systems using algebraic higher order nets. ECEASST **51**, 1–12 (2012)
25. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. **26**(3/4), 287–313 (1996)

26. Haddad, S., Poitrenaud, D.: Recursive Petri nets. Acta Informatica **44**(7), 463–508 (2007)
27. Hoff, C.: Transformationseinheiten als Kontrollstruktur für rekonfigurierbare Petrinetze in ReConNet. Master's thesis, University of Applied Sciences Hamburg (2016)
28. Hoffmann, K., Ehrig, H., Mossakowski, T.: High-level nets with nets and rules as tokens. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 268–288. Springer, Heidelberg (2005). https://doi.org/10.1007/11494744_16
29. Hoffmann, K., Ehrig, H., Padberg, J.: Flexible modeling of emergency scenarios using reconfigurable systems. ECEASST **12**, 1–20 (2008)
30. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009). https://doi.org/10.1007/b95112
31. Kahloul, L., Bourekkache, S.: Djouani, K: Designing reconfigurable manufacturing systems using reconfigurable object Petri nets. Int. J. Comput. Integr. Manuf. **29**, 1–18 (2016)
32. Kahloul, L., Bourekkache, S., Djouani, K., Chaoui, A., Kazar, O.: Using high level Petri nets in the modelling, simulation and verification of reconfigurable manufacturing systems. Int. J. Softw. Eng. Knowl. Eng. **24**(03), 419–443 (2014)
33. Kahloul, L., Chaoui, A., Djouani, K., Bourekkache, S., Kazar, O.: Using high level nets for the design of reconfigurable manufacturing systems. In: 1st International Workshop on Petri Nets for Adaptive Discrete-Event Control Systems, pp. 1–19 (2014)
34. Kahloul, L., Djouani, K., Chaoui, A.: Formal study of reconfigurable manufacturing systems: a high level Petri nets based approach. In: Mařík, V., Lastra, J.L.M., Skobelev, P. (eds.) HoloMAS 2013. LNCS (LNAI), vol. 8062, pp. 106–117. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40090-2_10
35. Kheldoun, A., Barkaoui, K., Zhang, J.F., Ioualalen, M.: A high level net for modeling and analysis reconfigurable discrete event control systems. In: Amine, A., Bellatreche, L., Elberrichi, Z., Neuhold, E.J., Wrembel, R. (eds.) CIIA 2015. IAICT, vol. 456, pp. 551–562. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19578-0_45
36. Kheldoun, A., Zhang, J., Barkaoui, K., Ioualalen, M.: A high-level nets based approach for reconfigurations of distributed control systems. In: ADECS Petri Nets, pp. 36–51 (2014)
37. König, B., Nolte, D., Padberg, J., Rensink, A.: A tutorial on graph transformation. In: Festschrift in Memory of Hartmut Ehrig. Springer (2018, accepted)
38. Kondratyev, A., Cortadella, J., Kishinevsky, M., Lavagno, L., Taubin, A.: The use of Petri nets for the design and verification of asynchronous circuits and systems. J. Circuits Syst. Comput. **8**(1), 67–118 (1998)
39. Koren, Y., Shpitalni, M.: Design of reconfigurable manufacturing systems. J. Manuf. Syst. **29**(4), 130–141 (2010)
40. 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
41. Li, J., Dai, X., Meng, Z.: Improved net rewriting systems-based rapid reconfiguration of Petri net logic controllers. In: 31st 2005 Annual Conference of IEEE Industrial Electronics Society, 6 pp. IEEE (2005)
42. Li, J., Dai, X., Meng, Z.: Improved net rewriting system-based approach to model reconfiguration of reconfigurable manufacturing systems. Int. J. Adv. Manuf. Technol. **37**(11–12), 1168–1189 (2008)

43. Li, J., Dai, X., Meng, Z.: Automatic reconfiguration of Petri net controllers for reconfigurable manufacturing systems with an improved net rewriting system-based approach. IEEE Trans. Autom. Sci. Eng. **6**(1), 156–167 (2009)
44. Li, J., Dai, X., Meng, Z., Xu, L.: Improved net rewriting system-extended Petri net supporting dynamic changes. J. Circuits Syst. Comput. **17**(06), 1027–1052 (2008)
45. Llorens, M., Oliver, J.: Structural and dynamic changes in concurrent systems: reconfigurable Petri nets. IEEE Trans. Comput. **53**(9), 1147–1158 (2004)
46. Llorens, M., Oliver, J.: MCReNet: a tool for marked-controlled reconfigurable nets. In: International Conference on Quantitative Evaluation of Systems, pp. 255–256 (2005)
47. Llorens, M., Oliver, J.: A basic tool for the modeling of marked-controlled reconfigurable Petri nets. ECEASST **2**, 1–13 (2006)
48. Marsan, M.A.: Stochastic Petri nets: an elementary introduction. In: Rozenberg, G. (ed.) APN 1988. LNCS, vol. 424, pp. 1–29. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52494-0_23
49. MCReNet. http://users.dsic.upv.es/~mllorens/MCReNet.htm. Accessed 14 May 2017
50. Meng, X.: Modeling of reconfigurable manufacturing systems based on colored timed object-oriented Petri nets. J. Manuf. Syst. **29**(2–3), 81–90 (2010)
51. Modica, T., Gabriel, K., Hoffmann, K.: Formalization of Petri nets with individual tokens as basis for DPO net transformations. ECEASST **40**, 1–21 (2010)
52. Modica, T., Homann, K.: Formal modeling of communication platforms using reconfigurable algebraic high-level nets. ECEASST **30**, 1–24 (2010)
53. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
54. Padberg, J.: Algebreic high-level net transformation systems: a survey over theory and applications. Bull. EATCS **51**, 102–110 (1993)
55. Padberg, J.: Categorical approach to horizontal structuring and refinement of high-level replacement systems. Appl. Categ. Struct. **7**(4), 371–403 (1999)
56. Padberg, J.: Classification of Petri nets using adjoint functors. In: Salomaa, A., Gheorghe, P., Rozenberg, G. (eds.) Current Trends in Theoretical Computer Science, pp. 171–179. World Scientific, Singapore (2001)
57. Padberg, J.: Abstract interleaving semantics for reconfigurable Petri nets. ECEASST **51**, 1–14 (2012)
58. Padberg, J.: Reconfigurable Petri nets with transition priorities and inhibitor arcs. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 104–120. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_7
59. Padberg, J., Ede, M., Oelker, G., Hoffmann, K.: Reconnet: a tool for modeling and simulating with reconfigurable place/transition nets. ECEASST **54**, 1–11 (2012)
60. Padberg, J., Ehrig, H., Ribeiro, L.: Algebraic high-level net transformation systems. Math. Struct. Comput. Sci. **5**(2), 217–256 (1995)
61. Padberg, J., Gajewsky, M., Ermel, C.: Rule-based refinement of high-level nets preserving safety properties. Sci. Comput. Program. **40**(1), 97–118 (2001)
62. Padberg, J., Hoffmann, K., Ehrig, H., Modica, T., Biermann, E., Ermel, C.: Maintaining consistency in layered architectures of mobile ad-hoc networks. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 383–397. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71289-3_29
63. Padberg, J., Hoffmann, K., Gajewsky, M.: Stepwise introduction and preservation of safety properties in algebraic high-level net systems. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 249–265. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46428-X_18

64. Padberg, J., Schulz, A.: Model checking reconfigurable Petri nets with maude. In: Echahed, R., Minas, M. (eds.) ICGT 2016. LNCS, vol. 9761, pp. 54–70. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40530-8_4
65. ReConNet. https://reconnetblog.wordpress.com/. Accessed 16 May 2017
66. Rein, A., Prange, U., Lambers, L., Hoffmann, K., Padberg, J.: Negative application conditions for reconfigurable place/transition systems. ECEASST **10**, 1–14 (2008)
67. Richta, T., Janousek, V., Kocí, R.: Petri nets-based development of dynamically reconfigurable embedded systems. PNSE+ ModPE **989**, 203–217 (2013)
68. RON-Editor. http://www.user.tu-berlin.de/o.runge/tfs/projekte/roneditor/. Accessed 24 May 2017
69. Tigane, S., Kahloul, L., Bourekkache, L.: Net rewriting system for GSPN: A RMS case study. In: 2016 International Conference on Advanced Aspects of Software Engineering (ICAASE), pp. 38–45. IEEE (2016)
70. Tigane, S., Kahloul, L., Bourekkache, S.: Reconfigurable stochastic Petri nets for reconfigurable manufacturing systems. In: Borangiu, T., Trentesaux, D., Thomas, A., Leitão, P., Barata Oliveira, J. (eds.) Service Orientation in Holonic and Multi-Agent Manufacturing. SCI, vol. 694, pp. 383–391. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51100-9_34
71. Târnauc, B., Puiu, D., Comnac, V., Suciu, C.: Modelling a flexible manufacturing system using reconfigurable finite capacity Petri nets. In: 13th International Conference on Optimization of Electrical and Electronic Equipment, pp. 1079–1084, May 2012
72. Urbášek, M.: Preserving properties in system redesign: rule-based approach. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 442–456. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_26
73. Valk, R.: Petri nets as token objects. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–24. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-69108-1_1
74. Valk, R.: Object Petri nets. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 819–848. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_23
75. Van Der Aalst, W., Van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT press, Cambridge (2004)
76. Yakovlev, A., Koelmans, A., Semenov, A., Kinniment, D.: Modelling, analysis and synthesis of asynchronous control circuits using Petri nets. Integr. VLSI J. **21**(3), 143–170 (1996)
77. Yu, Z., Guo, F., Ouyang, J., Zhou, L.: Object-oriented Petri nets and π-calculus-based modeling and analysis of reconfigurable manufacturing systems. Adv. Mech. Eng. **8**(11), 1–11 (2016). https://doi.org/10.1177/1687814016677698

A Category of "Undirected Graphs"
A Tribute to Hartmut Ehrig

John L. Pfaltz(✉)

Department of Computer Science, University of Virginia, Charlottesville, USA
jlp@virginia.edu

Abstract. In this paper, a category of undirected graphs is introduced where the morphisms are chosen in the style of mathematical graph theory rather than as algebraic structures as is more usual in the area of graph transformation.

A representative function, ω, within this category is presented. Its inverse, ω^{-1}, is defined in terms of a graph grammar, ε.

1 The Abstract Category

Hartmut Ehrig was one who helped introduce the graph grammar community (GraGra) to the concept of categories [6]. In this short paper we apply some of his vision to develop a category of *undirected graphs*. A graph (V, E) is undirected if its edge set E consists of sets $\{x, y\}$, not ordered pairs. It is not hard to characterize one version of this category. It consists of $obj = UG$, the collection of all finite undirected graphs, together with hom = *all functions,* $f : G \to G'$, where $G, G' \in UG$, with composition, that is $f : G \to G', g : G' \to G''$ implies $f{\cdot}g : G \to G'' \in hom$. Let $G = (V, E)$ and $G' = (V', E')$. By $f : G \to G'$ we actually mean $f : 2^V \to 2^{V'}$ subject to appropriate constraints with respect to the edge sets E and E'.[1] But, without specifying these constraints this kind of category conveys little information.

More interesting is the subcategory whose functions f, g are continuous (see below). Continuity in the familiar continuous manifolds, such as $\mathbf{R}$ or $\mathbf{C}$, is defined in terms of open sets. With discrete, or finite, graphs it can be better defined in terms of closed sets.

Let φ denote an arbitrary closure operator on an arbitrary collection, 2^V, of sets, that is for all subsets $X, Y, \in 2^V$, φ is expansive ($Y \subseteq Y.\varphi$), monotone ($X \subseteq Y$ implies $X.\varphi \subseteq Y.\varphi$) and idempotent ($Y.\varphi.\varphi = Y.\varphi$).[2] Such closure systems $(2^V, \varphi)$ are rather well studied, since they include matroids and antimatroids [2–5,8]. More importantly, we can now define what we mean by a continuous,

[1] The codomain $2^{V'}$ of f need not be 2^V, and its edge set E' need not have the same structure as E. Therefore, elements of the codomain are denoted with a prime.

[2] We use suffix notation to denote the application of set-valued operators and functions.

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 223–230, 2018.
https://doi.org/10.1007/978-3-319-75396-6_12

discrete, set-valued function f. A function $f : (2^V, \varphi) \rightarrow (2^{V'}, \varphi')$ is said to be continuous [11,12] if for all $Y \subseteq V$,

$$Y.\varphi.f \subseteq Y.f.\varphi'$$

We observe that the closure operator, φ' on V' need not be the same as φ on V. To obtain a category, we must now show that the composition of continuous functions $f \cdot g$ is continuous. But, they need not be. The composition $f \cdot g$ of continuous, set-valued functions will be continuous provided f and g are also monotone [12]. To create a subcategory, we need both properties.

Suppose the functions f and g are also "closure preserving", that is the image of any set Y, closed with respect to φ will be closed with respect to φ'. In this case,

$$Y.f.\varphi' \subseteq Y.\varphi.f$$

so $Y.\varphi.f = Y.f.\varphi'$, yielding the categorical diagram.

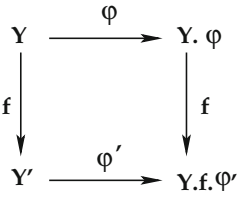


Fig. 1. A typical categorical diagram

The preceding discussion creates a subcategory of continuous set-valued functions. But as yet, it has nothing to do with undirected graphs!

As before, let *obj* be the set of all undirected graphs, $G = (V, E)$ where V is a set of vertices, points, or nodes and E is a symmetric binary relation on V, commonly called the edge set. Now, we consider *hom* to be the collection of all continuous, monotone, set-valued functions mapping subsets of the vertex (point, node) set, V of G into subsets of the vertex set V' of G'. We expect, somehow, that the closure operator on these graphs should reflect their edge structure. Let η be an operator on 2^V such that $y \in \{x\}.\eta$ and $x \in \{y\}.\eta$ if and only if $\{x, y\}$ is an edge in G. It is convenient if η, a *neighbor* operator is reflexive, that is $x \in \{x\}.\eta$. We, now, extend η to subsets $Y \subseteq V$ by $Y.\eta = \cup_{y \in Y}\{y\}.\eta$. Some texts call these "closed neighborhoods".[3] In the case of undirected graphs we prefer to use *neighborhood closure* φ_η, defined below

$$Y.\varphi_\eta = \{z | \{z\}.\eta \subseteq Y.\eta\} \tag{1}$$

Because η is reflexive, φ_η is expansive; it is monotone by construction; and idempotency is not hard to prove [14,15].

[3] This is a common terminology, but unfortunately such "closed neighborhoods" are not "closed". The intersection of closed sets must be closed, but it easy to show that this is seldom true with "closed neighborhoods".

Now we have the makings of a category, UG, of undirected graphs consisting of obj = the collection of all undirected graphs, and hom = all monotone, set-valued functions $f : 2^V \to 2^{V'}$ that are continuous with respect to φ_η. It is worth observing that this development allows us to continuously enlarge graphs by a function $f : 2^V \to 2^{V'}$ in which $\emptyset.f = X' \subseteq V'$[4] and to contract graphs with $g : 2^{V'} \to 2^{V''}$ where $Y.g = \emptyset \subseteq V''$. It is convenient to employ the notation $f : G \to G'$ with the understanding that f is really defined on the power sets of V and V' and that f is continuous with respect to a closure operator φ on the edge set/relation E.

Is UG anything more than an abstract category? Are there really functions in hom?

In the next section we present two graph transformations which define $\omega \in hom$ and $\varepsilon \in hom$. Both have been implemented as algorithmic computer programs.

2 Two Functions in $hom(UG)$

Let G be a graph (V, E), with a neighborhood operator η. Suppose $z \in \{y\}.\varphi_\eta$, implying by (1) that $\{z\}.\eta \subseteq \{y\}.\eta$. Since $\{z\}.\varphi_\eta = \{y\}.\varphi_\eta$, the set $\{z\}$ contributes nothing to the closure structure of G; it can be removed from G with little loss of information. We define the transformation $\omega_z : G \to G'$ by $\{z\}.\omega_z = \emptyset$ where ω_z is the identity map on $V - \{z\}$, $Y \subseteq V$, and $\{u', v'\} \in E'$ if and only if $\{u, v\} \in E$, $u, v \neq z$. We say z has been *subsumed* by y. It is not hard to show that ω_z is both monotone and continuous since $z \in \{y\}.\varphi_\eta$.

2.1 Reduction, ω

A computer procedure, *reduce* implements ω. It repeatedly sweeps through all vertices $y \in V$, deleting any vertices $z_i \in \{y\}.\varphi_\eta$, together with all edges incident to z_i, until no such z remain in V.[5] That is, $\omega = \omega_{z_1} \cdot \omega_{z_2} \cdot \ldots \cdot \omega_{z_n}$. Since each ω_{z_i} is monotone and continuous, ω is as well, that is $Y.\varphi_\eta.\omega \subseteq Y.\omega.\varphi_\eta{}'$. The process terminates when every singleton subset $\{y\} \subseteq V$ is closed. Such a graph is said to be *irreducible*.

It can be shown that $G' = G.\omega$ is unique (up to isomorphism) regardless of the order in which the vertices $y \in V$ are visited by ω or the order in which vertices $z \in \{y\}.\varphi_\eta$ are deleted [14–16]. So ω is a well defined function in $hom(UG)$. Because every singleton set (vertex) in G' is closed, ω must also be closure preserving, with $Y.\omega.\varphi_\eta{}' \subseteq Y.\varphi_\eta.\omega$, so the diagram of Fig. 1 is applicable when $f = \omega$.

In Fig. 2, the graph G of 18 vertices is reduced to $G' = G.\omega$ with 10 remaining vertices. In G, the dashed lines encircle the vertices that were subsumed by $2', 3', 15'$ and $17'$.

[4] We modify the usual definition of monotonicity to read: $X \subseteq Y$ implies $X.f \subseteq Y.f$, provided $X \neq \emptyset$.

[5] This procedure has been quite effective reducing large graphs $|V| \geq 1,000$, with at worst 6 iterative sweeps of V.

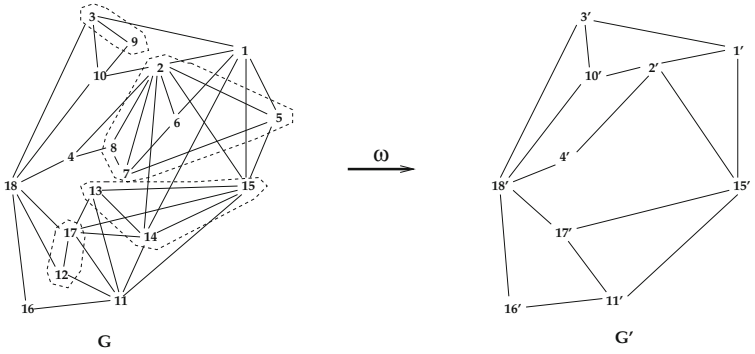


Fig. 2. Reduction, ω, of a graph G

Irreducible graphs, such as G', have a number of interesting properties. It is not hard to show that G' consists of a collection of chordless cycles of length ≥ 4. By a "chordless cycle" we mean a sequence of vertices $< y_1, y_2, \ldots y_n, y_1 >$, where $\{y_i, y_{i+1}\} \in E$, $1 \leq i \leq n-1$, and where $\{y_i, y_{i+k}\} \notin E$ for $k \geq 2$. Of course, we also require $\{y_n, y_1\} \in E$. It's a "pearl necklace" without cross connections. Because there can be no cross connecting edges of the form $\{y_i, y_{i+k}\}, k \geq 2$, each cycle C_α, when considered strictly as a "set" of vertices, is a member of a Sperner set [7]. That is, given a ground set V, for all cycles $C_\alpha, C_\beta \subset V$, $C_\alpha \not\subseteq C_\beta$. Besides the interesting combinatorics associated with Sperner sets, this permits various computer algorithms to process irreducible graphs solely as set systems without regard to individual edges. This reduction, $G.\omega$, of G to an irreducible graph G' has a number of other intriguing properties [16], such as the preservation of paths, of the graph "centers", but this is not relevant to this paper.

2.2 Expansion, ε

It is fairly easy to define the treatment of edges in a function, such as ω, that contracts a graph. If $Y \xrightarrow{f} \emptyset$, then all edges $\{y, z\}$ such that $y \in Y, z \in Y.\eta$ can be deleted. Expanding a graph, $\emptyset \xrightarrow{g} Y'$, presents more problems. How is Y' to be embedded in G'? One option is to employ an *expansion grammar* ε, such as explored in [13]. Expansion grammars are quite different from phrase-structured grammars in which a non-terminal symbol A is expanded with a rewrite rule of the form $A \rightarrow \sigma$ [19]. The problematic aspect of a phase-structured grammar, explored by Ehrig in [6], is how will the right side σ of the rewrite rule be embedded in the growing, non-linear structure.

In an expansion grammar, a subset Y of a growing structure is first identified to be the neighborhood of a new element p'. That is $\{p'\}.\eta' = Y \subseteq V$ in the rewritten structure. More precisely, $\varepsilon_i : (V_i, E_i) \rightarrow (V_{i+1}, E_{i+1})$ where $V_{i+1} = V_i \cup \{p'_i\}$, $E_{i+1} = E_i \cup \{\{y_k, p'_i\}, y_k \in Y \subset V_i\}$ and $\varepsilon_i : \emptyset = \{p'_i\}$.

The set-valued procedure, ε can then be defined as a graph grammar with any set of specified rewrite rules, or productions. The following example of an expansion grammar is also given in [13]. Consider the rewrite rule $r1$ below,

$$r1 : K_n \xrightarrow{\varepsilon} : p' \quad n \geq 1$$

which specifies that any complete subgraph, K_n, (or clique) of order n in V can serve as the neighborhood of a new point p' provided $n \geq 1$.[6] Every point in K'_n will be adjacent to p' in G'. Call the application of a rewrite rule a step, ε_i, in the process ε. It is a well defined operation in which $\emptyset.\varepsilon_i = \{p'\}$. The left side of the rewrite rule defines its embedding neighborhood. The right-most part defines any conditions on this neighborhood.

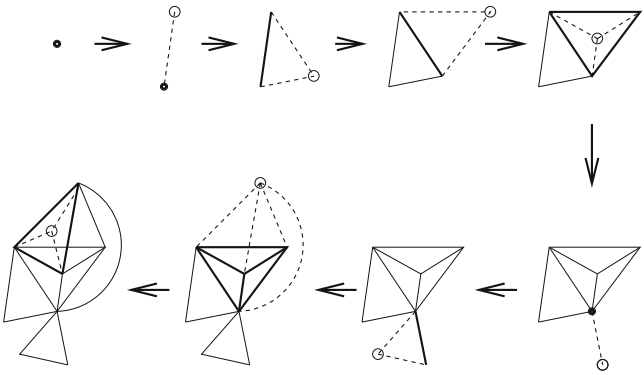


Fig. 3. A sequence of neighborhood expansions generating chordal graphs

Application of $r1$ is illustrated in Fig. 3. Each expanded neighborhood (in this case clique) has been made bold; and the expansion point, p', circled. The dashed edges indicate those links which define the clique as the neighborhood of the expansion point p'. It is not hard to see that any graph generated in this fashion must be chordal.[7]

A more relaxed version of the rewrite rule $r1$ above, will allow Y, the new neighborhood of p', to be any subset of the neighborhood of an existing vertex $y \in V_i$. Specified as a rewrite rule $r2$ it is,

$$r2 : Y \xrightarrow{\varepsilon} : p' \quad \exists y \in V_i, Y \subseteq \{y\}.\eta$$

[6] A graph, K_n is **complete** if all n nodes are mutually connected by an edge.

[7] Because extreme points are simplicial (neighborhood is a clique), and because every chordal graph must have at least two extreme points [8,9], every chordal graph can be so generated.

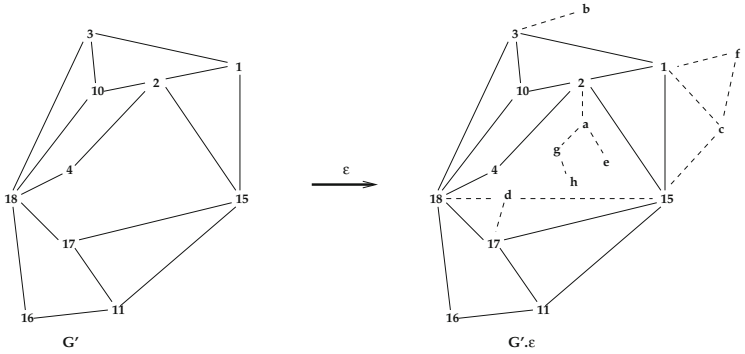


Fig. 4. A member of $G'.\varepsilon$ where $G' = G.\omega$ in Fig. 2.

Fig. 4 shows one possible application of this expansion grammar ε to the graph G' of Fig. 2. Here, the rewrite rule $r2$ has been used 8 times, to create $a, b, \ldots h$. The vertex d is generated by $r2$ using the neighborhood $\{17\}.\eta = \{15, 17, 18\} = \{d\}.\eta'$. The new vertex c was attached to $\{1, 15\} \subset \{1, 2, 3, 15\} = \{1\}.\eta$; and f was later attached to $\{1, c\} \subset \{1\}.\eta$.

2.3 The Inverse Set, ω^{-1}

The two procedures ω and ε are intertwined. The requirement in the second rewrite rule $r2$ that $\{p'\}.\eta = Y \subseteq \{y\}.\eta$ ensures that if ω is applied to $G'.\varepsilon$, p' will at some iteration be subsumed by y. Thus, if G' is irreducible, $G'.\varepsilon.\omega = G'$ This characteristic is evident in Fig. 4 where b will be subsumed by 3, *etc.* It is also true for the graph $G'.\varepsilon$ of Fig. 5 as well. Consequently, ω is a right-inverse of ε over the subspace of irreducible undirected graphs. The inverse of ω, that is $G.\omega.\omega^{-1}$ is the collection of all undirected graphs $\{G_k\}$ such that $G_k.\omega = G' = G.\omega$. Each invocation of the non-deterministic procedure ε is single-valued; but ε is not a function. The execution of ε will yield a graph, $G_k \in G.\omega.\omega^{-1}$.

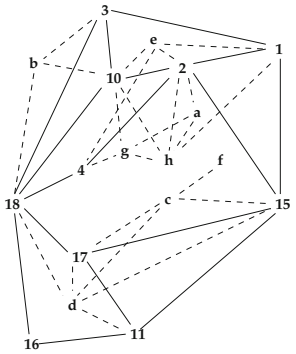


Fig. 5. Another graph $G'.\varepsilon$ in $G'.\omega^{-1}$.

In the rewrite rule $r2$ the choice of $y \in V_i$ and the choice of $Y \subseteq \{y_i\}.\eta$ are completely arbitrary. Given different choices for y and Y yields Fig. 5 which seems to be a far more interesting graph. Both Figs. 4 and 5 were generated by a computer version of ε using a random number generator.

This is not the only category of undirected graphs, but it is a promising one [17]. Unfortunately, undirected graphs, and mappings between such graphs, have little of the regular structure seen in the different abstract algebras that gave rise to the categorical approach of [1,10,18], or that of [2] which was applied to general closure operators. Yet, the rudiments are there, as this short treatise shows. In the early 70's, Hartmut Ehrig urged us to view graph grammars and graph manipulation through a categorical lens. He was ahead of his time.

References

1. Arbib, M., Manes, E.: Arrows, Structures, and Functors: The Categorical Imperative. Academic Press, New York (1975)
2. Castellini, G.: Categorical Closure Operators. Birkhauser, Boston (2003)
3. Chvátal, V.: Antimatroids, betweenness, convexity. In: László, W.C., Vygen, J. (eds.) Research Trends in Combinatorial Optimization, pp. 57–64. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-76796-1_3
4. Edelman, P.H.: Abstract convexity and meet-distributive lattices. In: Combinatorics and Ordered Sets, Arcata, CA, pp. 127–150 (1986)
5. Edelman, P.H., Jamison, R.E.: The theory of convex geometries. Geom. Dedicata **19**(3), 247–270 (1985)
6. Ehrig, H., Pfender, M., Schneider, H.J.: Graph grammars: an algebraic approach. In: IEEE Conference on SWAT (1973)
7. Engle, K.: Sperner theory. In: Hazewinkle, M. (ed.) Encyclopedia of Mathematics. Springer, Heidelberg (2001)
8. Farber, M., Jamison, R.E.: Convexity in graphs and hypergraphs. SIAM J. Algebra Discrete Methods **7**(3), 433–444 (1986)
9. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York (1980)
10. MacLane, S.: Categories for the Working Mathematician, 2nd edn. Springer, New York (1998). https://doi.org/10.1007/978-1-4612-9839-7
11. Ore, O.: Mappings of closure relations. Ann. Math. **47**(1), 56–72 (1946)
12. Pfaltz, J., Šlapal, J.: Transformations of discrete closure systems. Acta Math. Hung. **138**(4), 386–405 (2013)
13. Pfaltz, J.L.: Neighborhood expansion grammars. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 30–44. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46464-8_3
14. Pfaltz, J.L.: Finding the mule in the network. In: Alhajj, R., Werner, B. (eds.) International Conference on Advances in Social Network Analysis and Mining, ASONAM 2012, Istanbul, Turkey, pp. 667–672, August 2012
15. Pfaltz, J.L.: Mathematical continuity in dynamic social networks. Soc. Netw. Anal. Min. (SNAM) **3**(4), 863–872 (2013)
16. Pfaltz, J.L.: The irreducible spine(s) of undirected networks. In: Lin, X., Manolopoulos, Y., Srivastava, D., Huang, G. (eds.) WISE 2013. LNCS, vol. 8181, pp. 104–117. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41154-0_8

17. Pfaltz, J.L.: Computational processes that appear to model human memory. In: Figueiredo, D., Martín-Vide, C., Pratas, D., Vega-Rodríguez, M.A. (eds.) AlCoB 2017. LNCS, vol. 10252, pp. 85–99. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58163-7_6
18. Pierce, B.C.: Basic Category Theory for Computer Scientists. MIT Press, Cambridge (1991)
19. Rozenberg, G. (ed.): The Handbook of Graph Grammars. World Scientific, Singapore (1997)

Modular Termination of Graph Transformation

Detlef Plump(✉)

University of York, York, UK
detlef.plump@york.ac.uk

Abstract. We establish a machine-checkable condition which ensures that the union of two terminating hypergraph transformation systems is terminating. The condition is based on so-called sequential critical pairs which represent consecutive rule applications that are not independent. In contrast to a corresponding modularity result for term rewriting, no restrictions on the form of rules are required. Our result applies to both systems with injective rules and systems with rules that merge nodes or edges.

1 Introduction

In the area of graph transformation, the problem of proving that a system will terminate (admit only a finite number of rule applications) on arbitrary host graphs has received surprisingly little attention. This is in contrast to the central role of termination research in the area of term rewriting [1,2].

Work on proving termination of general graph transformation systems by various methods includes [3–5,16]. There are also some papers on termination in the specialised settings of term graph rewriting [14,15,17] and cycle rewriting [21,23].

In this paper, we are interested in the problem of finding conditions that guarantee that the combination of terminating (hyper-)graph transformation systems yields again a terminating system. The corresponding problem for term rewriting systems received considerable attention after Toyama showed that even the combination of systems with disjoint function symbols need not preserve termination [22]. Interestingly, the latter phenomenon vanishes into thin air for acyclic term graph rewriting [14,15] because rewrite steps create shared subgraphs instead of copying subterms.

We prove in this paper that the union of two general hypergraph transformation systems preserves termination if there are no critical overlaps between the right-hand sides of one system and the left-hand sides of the other system. This idea is inspired by a result of Dershowitz [7] which shows that the corresponding property for term rewriting systems holds if one of the systems is left-linear and the other system is right-linear. In the case of graph transformation, it turns out that no restrictions on the form of rules are needed.

The rest of this paper is organized as follows. In Sect. 2, we review the basics of hypergraph transformation. In particular, we recall the concept of sequential independence which is crucial for our main result. Sequential critical pairs are

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 231–244, 2018.
https://doi.org/10.1007/978-3-319-75396-6_13

introduced in Sect. 3, in analogy to standard critical pairs for graph transformation. Our main result is proved in Sect. 4 where we also give examples to demonstrate the application of the modularity criterion. We conclude in Sect. 5.

2 Hypergraph Transformation

In this section, we recall some definitions and results of the double-pushout approach to graph transformation which can be found, for example, in [11]. For generality, we use here the setting of hypergraphs.

2.1 Hypergraphs

Our hypergraphs are directed and labelled. We use a type system where the label of a hyperedge can restrict the number of incident nodes and their labels. A *signature* $\Sigma = \langle \Sigma_{\mathrm{V}}, \Sigma_{\mathrm{E}}, \mathrm{Type} \rangle$ consists of a set Σ_{V} of *node labels*, a set Σ_{E} of *hyperedge labels* and a function Type assigning to each $l \in \Sigma_{\mathrm{E}}$ a set of strings $\mathrm{Type}(l) \subseteq \Sigma_{\mathrm{V}}^*$. This kind of typing allows to "overload" hyperedge labels by specifying different admissible attachment sequences. Typing regimes covered by this approach include the case of a singleton set $\mathrm{Type}(l)$ for each label l (as in [6]) and the case of "untyped" hypergraphs given by $\mathrm{Type}(l) = \Sigma_{\mathrm{V}}^*$ for each l (as in [10]). Unless stated otherwise, we denote by Σ an arbitrary but fixed signature over which all hypergraphs are labelled.

A *hypergraph* over Σ is a system $G = \langle \mathrm{V}_G, \mathrm{E}_G, \mathrm{mark}_G, \mathrm{lab}_G, \mathrm{att}_G \rangle$ consisting of two finite sets V_G and E_G of *nodes* (or *vertices*) and *hyperedges*, two labelling functions $\mathrm{mark}_G \colon \mathrm{V}_G \to \Sigma_{\mathrm{V}}$ and $\mathrm{lab}_G \colon \mathrm{E}_G \to \Sigma_{\mathrm{E}}$, and an attachment function $\mathrm{att}_G \colon \mathrm{E}_G \to \mathrm{V}_G^*$ such that $\mathrm{mark}_G^*(\mathrm{att}_G(e)) \in \mathrm{Type}(\mathrm{lab}_G(e))$ for each hyperedge e. (The extension $f^* \colon A^* \to B^*$ of a function $f \colon A \to B$ maps the empty string to itself and $a_1 \dots a_n$ to $f(a_1) \dots f(a_n)$.)

In pictures, nodes and hyperedges are drawn as circles and boxes, respectively, with labels inside. Lines represent the attachment of hyperedges to nodes, where numbers specify the left-to-right order in the attachment string. For example, Fig. 1 shows a hypergraph with four nodes (all labelled with •) and three hyperedges (labelled with B and S).

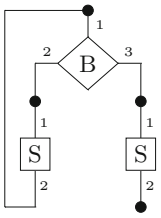


Fig. 1. A hypergraph

A hypergraph G is a *graph* if each hyperedge e is an ordinary edge, that is, if $\mathrm{att}_G(e)$ has length two. Ordinary edges may be drawn as arrows with labels written next to them.

Given hypergraphs G and H, a *hypergraph morphism* (or *morphism* for short) $g\colon G \to H$ consists of two functions $g_{\mathrm{V}}\colon \mathrm{V}_G \to \mathrm{V}_H$ and $g_{\mathrm{E}}\colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve labels and attachment to nodes, that is, $\mathrm{mark}_H \circ g_{\mathrm{V}} = \mathrm{mark}_G$, $\mathrm{lab}_H \circ g_{\mathrm{E}} = \mathrm{lab}_G$ and $\mathrm{att}_H \circ g_{\mathrm{E}} = g_{\mathrm{V}}^* \circ \mathrm{att}_G$. A morphism $incl\colon G \to H$ is an *inclusion* if $incl_{\mathrm{V}}(v) = v$ and $incl_{\mathrm{E}}(e) = e$ for all $v \in \mathrm{V}_G$ and $e \in \mathrm{E}_G$. In this case G is a *subhypergraph* of H which is denoted by $G \subseteq H$. Morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective). If g is both injective and surjective, then it is an *isomorphism*. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

The *composition* of two morphisms $g\colon G \to H$ and $h\colon H \to M$ is the morphism $h \circ g\colon G \to M$ consisting of the composed functions $h_{\mathrm{V}} \circ g_{\mathrm{V}}$ and $h_{\mathrm{E}} \circ g_{\mathrm{E}}$. The composition is also written as $G \to H \to M$ if g and h are clear from the context.

2.2 Rules and Derivations

A *rule* $r\colon \langle L \leftarrow K \to R \rangle$ consists of two hypergraph morphisms with a common domain, where $K \to L$ is an inclusion. The hypergraphs L and R are the *left-* and *right-hand side* of r, and K is the *interface*. The rule is *injective* if the morphism $K \to R$ is injective.

Let G and H be hypergraphs, $r\colon \langle L \leftarrow K \to R \rangle$ a rule and $g\colon L \to G$ an injective morphism. Then G *directly derives* H by r and g, denoted by $G \Rightarrow_{r,g} H$, if there exist two pushouts as in Fig. 2. Given a set of rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ to express that there exist $r \in \mathcal{R}$ and a morphism g such that $G \Rightarrow_{r,g} H$.

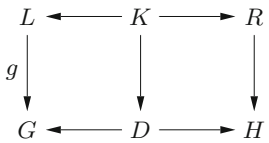


Fig. 2. A double-pushout

We refer to [20] for the definition and construction of hypergraph pushouts. Intuitively, the left pushout corresponds to the construction of D from G by removing the items in $L - K$, and the right pushout to the construction of H from D by merging items according to $K \to R$ and adding the items in R that are not in the image of K.

A double-pushout as in Fig. 2 is called a *direct derivation* from G to H and may be denoted by $G \Rightarrow_{r,g} H$ or just by $G \Rightarrow_r H$ or $G \Rightarrow H$. A *derivation* from G to H is a sequence of direct derivations $G_0 \Rightarrow \ldots \Rightarrow G_n$, $n \geq 0$, such that

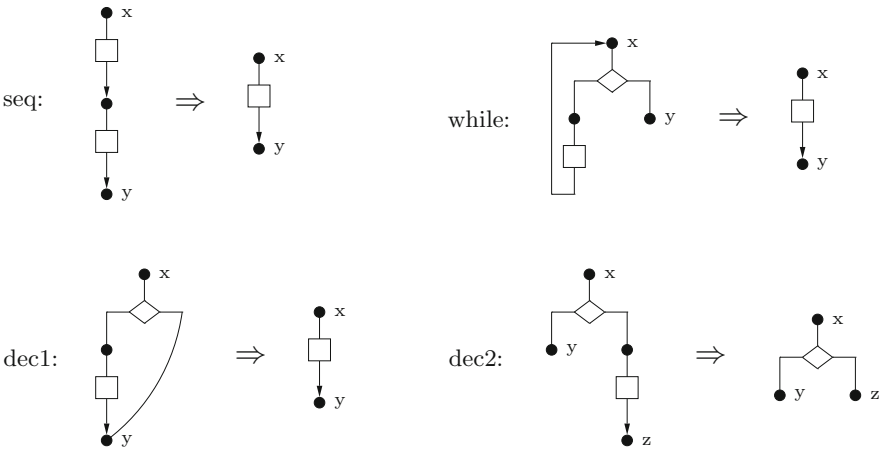


Fig. 3. Hypergraph transformation system for flow-graph reduction

$G \cong G_0$ and $G_n \cong H$. We write $G \Rightarrow^* H$ for such a derivation or $G \Rightarrow^*_{\mathcal{R}} H$ if all rules used in the derivation are from $\mathcal{R}$.

Given a rule $r\colon \langle L \leftarrow K \rightarrow R \rangle$, an injective morphism $g\colon L \rightarrow G$ satisfies the *dangling condition* if no hyperedge in $\mathrm{E}_G - g_{\mathrm{E}}(\mathrm{E}_L)$ is incident to a node in $g_{\mathrm{V}}(\mathrm{V}_L - \mathrm{V}_K)$. It can be shown that, given r and f, a direct derivation as in Fig. 2 exists if and only if g satisfies the dangling condition [11].

Definition 1 (Hypergraph transformation system). A *hypergraph transformation system* $\langle \Sigma, \mathcal{R} \rangle$ consists of a signature Σ and a finite set $\mathcal{R}$ of rules over Σ. The system is *injective* if all rules in $\mathcal{R}$ are injective. It is a *graph transformation system* if for each label l in Σ_{E}, all strings in $\mathrm{Type}(l)$ are of length two.

Note that since graph transformation systems are special hypergraph transformation systems, results for the latter usually apply to the former, too.

Example 1. Figure 3 shows hypergraph transformation rules for reducing control-flow graphs [20]. The associated signature contains a single node label $\bullet$ and two hyperedge labels which are graphically represented by hyperedges formed as squares and rhombs. Instead of using numbers to represent the attachment function, we use an arrow to point to the second attachment node of a square and define the order among the links of a rhomb to be "top-left-right". The rules are shown in a shorthand notation where only the left- and right-hand sides are depicted, the interface and the morphisms are implicitly given by the node names x, y, z. □

2.3 Sequential Independence

Given injective rules r_1 and r_2, consecutive direct derivations $G \Rightarrow_{r_1} H \Rightarrow_{r_2} M$ do not interfere with each other if the intersection of the right-hand side of r_1

with the left-hand side of r_2 in H consists of common interface items. In the presence of non-injective rules, however, independence needs to be defined in terms of the existence of certain morphisms. We give the general definition first and then consider a simpler condition for the case of injective rules. For $i = 1, 2$, let r_i denote a rule $\langle L_i \leftarrow K_i \rightarrow R_i \rangle$.

Definition 2 (Sequential independence). Direct derivations $G \Rightarrow_{r_1} H \Rightarrow_{r_2} M$ as in Fig. 4 are *sequentially independent* if there are morphisms $R_1 \rightarrow D_2$ and $L_2 \rightarrow D_1$ such that the following holds.

Commutativity: $R_1 \rightarrow D_2 \rightarrow H = R_1 \rightarrow H$ and $L_2 \rightarrow D_1 \rightarrow H = L_2 \rightarrow H$.
Injectivity: $R_1 \rightarrow D_2 \rightarrow M$ is injective.

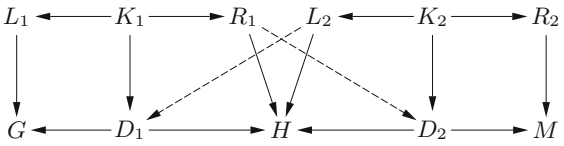


Fig. 4. Sequentially independent direct derivations

The injectivity requirement for $R_1 \rightarrow D_2 \rightarrow M$ is needed in case r_2 is non-injective. See [11] for an example that the steps $G \Rightarrow_{r_1} H \Rightarrow_{r_2} M$ may not be interchangeable without this condition.

If r_1 and r_2 are injective, the direct derivations of Fig. 4 are independent if and only if the intersection of R_1 and L_2 in H coincides with the intersection of K_1 and K_2. Moreover, the injectivity condition of Definition 2 is automatically satisfied.

Lemma 1 (Independence for injective rules). *Let rules r_1 and r_2 in Fig. 4 be injective. Then the following are equivalent:*

(1) *$G \Rightarrow_{r_1} H \Rightarrow_{r_2} M$ are sequentially independent.*
(2) *There are morphisms $R_1 \rightarrow D_2$ and $L_2 \rightarrow D_1$ such that $R_1 \rightarrow D_2 \rightarrow H = R_1 \rightarrow H$ and $L_2 \rightarrow D_1 \rightarrow H = L_2 \rightarrow H$.*
(3) *$h(R_1) \cap g(L_2) = h(b(K_1)) \cap g(K_2)$ where $h\colon R_1 \rightarrow H$, $g\colon L_2 \rightarrow H$ and $b\colon K_1 \rightarrow R_1$. (Note that $K_2 \rightarrow L_2$ is an inclusion.)*

Proof. The equivalence of (1) and (2) is an easy consequence of the fact that with injective rules, all morphisms in Fig. 4 are injective. The equivalence of (2) and (3) is shown in [9,19]. □

The following theorem was originally proved in [8], in the setting of graph transformation with arbitrary, possibly non-injective matching morphisms. This proof was adapted in [19] to the setting of hypergraph transformation with injective matching.

Theorem 1 ([11,19]). *Given sequentially independent direct derivations* $G \Rightarrow_{r_1} H \Rightarrow_{r_2} M$*, there exist direct derivations of the form* $G \Rightarrow_{r_2} H' \Rightarrow_{r_1} M$*.*

For instance, Fig. 5 shows applications of the rules seq and dec2 from Example 1. The steps are independent because the occurrences of the right-hand side of seq and the left-hand side dec2 share only nodes y and z, which are interface nodes in both rules. Hence the steps can be interchanged, leading to the same result.

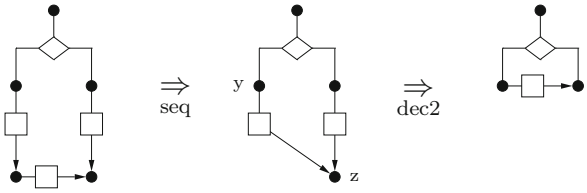


Fig. 5. Sequentially independent rule applications

3 Sequential Critical Pairs

Standard critical pairs represent conflicts between rule applications to the same graph and are used to analyse graph transformation systems for local confluence [20]. In the context of verifying termination, we adapt this concept to represent conflicts between consecutive direct derivations.

Sequential critical pairs are minimal derivations of length two whose steps are not independent. Due to the minimality, the set of critical pairs induced by two hypergraph transformation rules is finite and can be constructed. We will see that if this set is empty, then any pair of direct derivations with these rules is sequentially independent. In the proof of the main result, this will enable us to re-order the rule applications of infinite derivations.

Definition 3 (Sequential critical pair). Let $r_i \colon \langle L_i \leftarrow K_i \rightarrow R_i \rangle$ be rules, for $i = 1, 2$. A pair of direct derivations $S \Rightarrow_{r_1} T \Rightarrow_{r_2} U$ as in Fig. 6 is a *sequential critical pair* if the following holds.

Minimality: $T = h_1(R_1) \cup g_2(L_2)$.
Conflict: The steps are not sequentially independent.

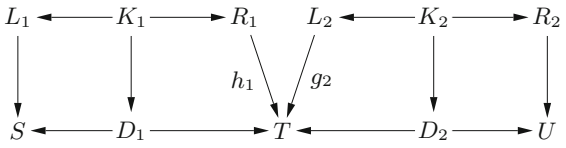


Fig. 6. A sequential critical pair

Sequential critical pairs $S \Rightarrow_{r_1,g_1} T \Rightarrow_{r_2,g_2} U$ and $S' \Rightarrow_{r_1,g_1'} T' \Rightarrow_{r_2,g_2'} U'$ are *isomorphic* if there are isomorphisms $i_1 \colon S \to S'$ and $i_2 \colon T \to T'$ such that $g_1' = i_1 \circ g_1$ and $g_2' = i_2 \circ g_2$. (This implies that U and U' are isomorphic, too.) From now on we equate isomorphic critical pairs. With the minimality condition of Definition 3 it follows that every hypergraph transformation system has only finitely many critical pairs.

Example 2. Figure 7 shows two sequential critical pairs of the rules of Fig. 3.

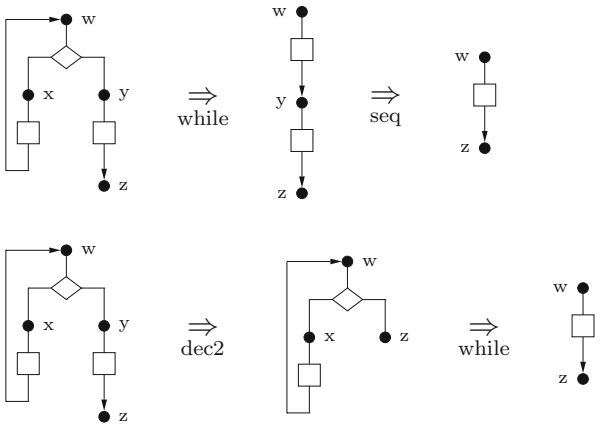


Fig. 7. Two sequential critical pairs of the system of Fig. 3

Lemma 2. *For every pair of direct derivations $G \Rightarrow_{r_1} H \Rightarrow_{r_2} M$ that are not sequentially independent, there exists a critical pair of the form $S \Rightarrow_{r_1} T \Rightarrow_{r_2} U$.*

Proof Sketch. The steps $G \Rightarrow_{r_1} H \Rightarrow_{r_2} M$ can be restricted to a critical pair by removing all nodes and edges that are not used by r_1 or r_2. That is, S is the subhypergraph of G consisting of all items in the occurrence of the left-hand side of r_1 and all items that are preserved by $G \Rightarrow_{r_1} H$ and are in the occurrence of the left-hand side of r_2. See [19] for the precise construction. It is not difficult to check that the restricted steps are minimal and in conflict. □

4 Modular Termination

A hypergraph transformation system $\langle \Sigma, \mathcal{R} \rangle$ is *terminating* if there exists no infinite derivation $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} G_2 \Rightarrow_{\mathcal{R}} \ldots$ The problem to decide whether a system $\langle \Sigma, \mathcal{R} \rangle$ is terminating is undecidable in general, even for graph transformation systems with injective rules [18].

Theorem 2 (Modular termination). *Let $\langle \Sigma, \mathcal{R} \rangle$ and $\langle \Sigma, \mathcal{S} \rangle$ be terminating hypergraph transformation systems. If there are no sequential critical pairs of shape $S \Rightarrow_{\mathcal{R}} T \Rightarrow_{\mathcal{S}} U$, then the combined system $\langle \Sigma, \mathcal{R} \cup \mathcal{S} \rangle$ is terminating.*

Note the symmetry in the statement of Theorem 2: it is sufficient that there are no critical pairs of shape $S \Rightarrow_{\mathcal{R}} T \Rightarrow_{\mathcal{S}} U$ *or* no critical pairs of shape $S \Rightarrow_{\mathcal{S}} T \Rightarrow_{\mathcal{R}} U$.

Proof of Theorem 2. Let $\langle \Sigma, \mathcal{R} \rangle$ and $\langle \Sigma, \mathcal{S} \rangle$ be terminating hypergraph transformation systems such that there are no critical pairs of shape $S \Rightarrow_{\mathcal{R}} T \Rightarrow_{\mathcal{S}} U$. We proceed by contradiction. Suppose there exists an infinite derivation

$$G_1 \underset{\mathcal{R}\cup\mathcal{S}}{\Rightarrow} G_2 \underset{\mathcal{R}\cup\mathcal{S}}{\Rightarrow} G_3 \underset{\mathcal{R}\cup\mathcal{S}}{\Rightarrow} \dots$$

Because $\Rightarrow_{\mathcal{R}}$ and $\Rightarrow_{\mathcal{S}}$ are terminating, the derivation must contain infinitely many steps of both kinds. By Lemma 2, any two steps $G_k \Rightarrow_{\mathcal{R}} G_{k+1} \Rightarrow_{\mathcal{S}} G_{k+2}$ in the sequence are sequentially independent because there are no critical pairs of shape $S \Rightarrow_{\mathcal{R}} T \Rightarrow_{\mathcal{S}} U$. By Theorem 1, the steps can be swapped such that $G_k \Rightarrow_{\mathcal{S}} G'_{k+1} \Rightarrow_{\mathcal{R}} G_{k+2}$. Thus all $\Rightarrow_{\mathcal{S}}$-steps can be pushed back to the beginning of the derivation, resulting in an infinite sequence of $\Rightarrow_{\mathcal{S}}$-steps. This contradicts the fact that $\langle \Sigma, \mathcal{S} \rangle$ is terminating. □

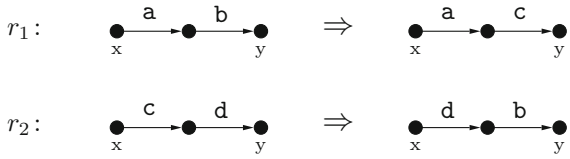
Figure 8 illustrates the infinite swapping process used to prove Theorem 2. Any infinite derivation with $\mathcal{R}\cup\mathcal{S}$ must contain infinitely many $\mathcal{S}$-steps and hence pushing them to the beginning *ad infinitum* will build up an infinite derivation of $\mathcal{S}$-steps.

$$\begin{array}{l}
G_0 \Rightarrow_{\mathcal{R}} \boxed{G_1 \Rightarrow_{\mathcal{R}} G_2 \Rightarrow_{\mathcal{S}} G_3} \Rightarrow_{\mathcal{R}} G_4 \Rightarrow_{\mathcal{S}} G_5 \Rightarrow \dots \\
\qquad\qquad \downarrow \\
\boxed{G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{S}} G'_2} \Rightarrow_{\mathcal{R}} G_3 \Rightarrow_{\mathcal{R}} G_4 \Rightarrow_{\mathcal{S}} G_5 \Rightarrow \dots \\
\qquad\qquad \downarrow \\
G_0 \Rightarrow_{\mathcal{S}} G'_1 \Rightarrow_{\mathcal{R}} G'_2 \Rightarrow_{\mathcal{R}} \boxed{G_3 \Rightarrow_{\mathcal{R}} G_4 \Rightarrow_{\mathcal{S}} G_5} \Rightarrow \dots \\
\qquad\qquad \downarrow \\
G_0 \Rightarrow_{\mathcal{S}} G'_1 \Rightarrow_{\mathcal{R}} \boxed{G'_2 \Rightarrow_{\mathcal{R}} G_3 \Rightarrow_{\mathcal{S}} G'_4} \Rightarrow_{\mathcal{R}} G_5 \Rightarrow \dots \\
\qquad\qquad \downarrow \\
G_0 \Rightarrow_{\mathcal{S}} \boxed{G'_1 \Rightarrow_{\mathcal{R}} G'_2 \Rightarrow_{\mathcal{S}} G'_3} \Rightarrow_{\mathcal{R}} G'_4 \Rightarrow_{\mathcal{R}} G_5 \Rightarrow \dots \\
\qquad\qquad \downarrow \\
G_0 \Rightarrow_{\mathcal{S}} G'_1 \Rightarrow_{\mathcal{S}} G''_2 \Rightarrow_{\mathcal{R}} G'_3 \Rightarrow_{\mathcal{R}} G'_4 \Rightarrow_{\mathcal{R}} G_5 \Rightarrow \dots \\
\qquad\qquad \downarrow \\
\qquad\qquad \vdots
\end{array}$$

Fig. 8. Infinite swapping process to create an infinite $\mathcal{S}$-derivation

We now present a sequence of examples which demonstrate the application of Theorem 2.

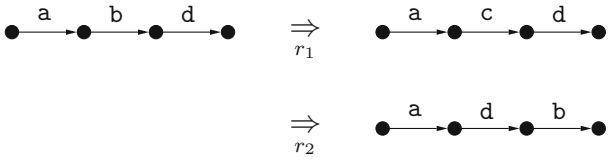
Example 3. Consider the following graph transformation system from [16]:



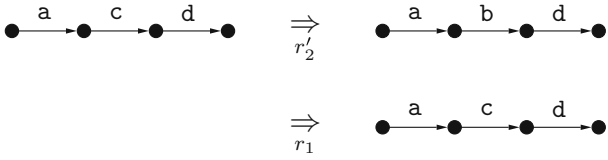
It is not obvious that this system is terminating because r_1 reduces the number of `b`-labels in a graph while r_2 increases this number. Similarly, r_2 reduces the number of `c`-labels while r_1 increases this number.

We can prove that this system is terminating by showing termination of r_1 and r_2 separately and then applying Theorem 2. Each rule individually is terminating because r_1 reduces the number of `b`'s and r_2 reduces the number of `c`'s. Moreover, there are no critical pairs of shape $S \Rightarrow_{r_2} T \Rightarrow_{r_1} U$. This is because the middle node in the right-hand side of r_2 is created by the rule and hence cannot have an incoming `a`-edge. Thus, by Theorem 2, the combined system is terminating.

Note that the system does have a critical pair of shape $S \Rightarrow_{r_1} T \Rightarrow_{r_2} U$:



Moreover, if we replace rule r_2 with r_2' by swapping labels `d` and `b` in the right-hand side, then there is also a critical pair of shape $S \Rightarrow_{r_2'} T \Rightarrow_{r_1} U$. Indeed, this change makes the system non-terminating as the new critical pair is cyclic:



□

It is worth pointing out the mechanical nature of the termination proof in Example 3. Combining a simple label counting algorithm with a generator for sequential critical pairs would automatically determine that both rules are terminating and that there are no critical pairs of shape $S \Rightarrow_{r_2} T \Rightarrow_{r_1} U$. Based on Theorem 2, this method would then report that the combined system is terminating.

Example 4. The graph transformation system in Fig. 9 is shown to be terminating in [5]. The technique used is to first simplify the system by removing rules r_3 and r_4, obtaining a system that is terminating if and only if the original system is terminating. The argument is as follows: rule r_3 decreases the number of `B`-labels while no other rule increases this number. Hence any infinite derivation contains only finitely many applications of r_3 and thus $\langle \Sigma, \{r_1, \ldots, r_4\} \rangle$ is terminating if

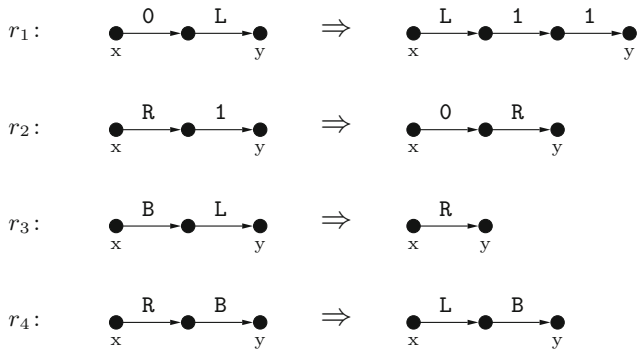


Fig. 9. A terminating graph transformation system [5]

and only if $\langle \Sigma, \{r_1, r_2, r_4\} \rangle$ is terminating. After removing r_3, one can observe that rule r_4 reduces the number of R-labels while neither r_1 nor r_2 increase this number. Hence r_4 can be removed, too.

For the simplified system $\langle \Sigma, \{r_1, r_2\} \rangle$, a so-called weighted type graph over the tropical semiring is constructed [5] which provides a decreasing measure for both rules. We can give a simpler termination argument for this system by using the approach of Example 3: r_1 is terminating as it reduces the number of 0's and r_2 is terminating as it reduces the number of 1's. Also, it is easy to check that there are no critical pairs of shape $S \Rightarrow_{r_1} T \Rightarrow_{r_2} U$ or $S \Rightarrow_{r_2} T \Rightarrow_{r_1} U$. Hence Theorem 2 guarantees that $\langle \Sigma, \{r_1, r_2\} \rangle$ is terminating. □

Similar to Example 3, our termination proof could be found automatically by a tool that counts labels, eliminates rules that decrease label counts not increased by the other rules, and finally generates sequential critical pairs.

Example 5. The hypergraph transformation system in Fig. 10 checks, when applied as long as possible, whether a host graph is 2-colourable (bipartite). If this is the case, the rules colour the graph accordingly. We assume that the initial hypergraph is a loop-free connected graph in which each node has exactly one "colour flag" attached to it (a hyperedge e with $|\mathrm{att}_G(e)| = 1$). Moreover, exactly one flag should be labelled r (or b) and all other flags should be blank. To save space, we deviate from our usual drawing convention in that all ordinary edges shown in the rules are interface edges.

If the initial graph is 2-colourable, the system of Fig. 10 will terminate with a coloured version of the graph in which each node is coloured r or b. (A graph is 2-colourable if its underlying undirected graph has no cycles of odd length.) If the graph is not 2-colourable, this will eventually be detected by the Invalid rules. The Propagate rules then make sure that all colour flags are black in the final graph.

As in the previous examples, we can prove termination of the system by label counting and Theorem 2. Subsystem Colour is terminating because all

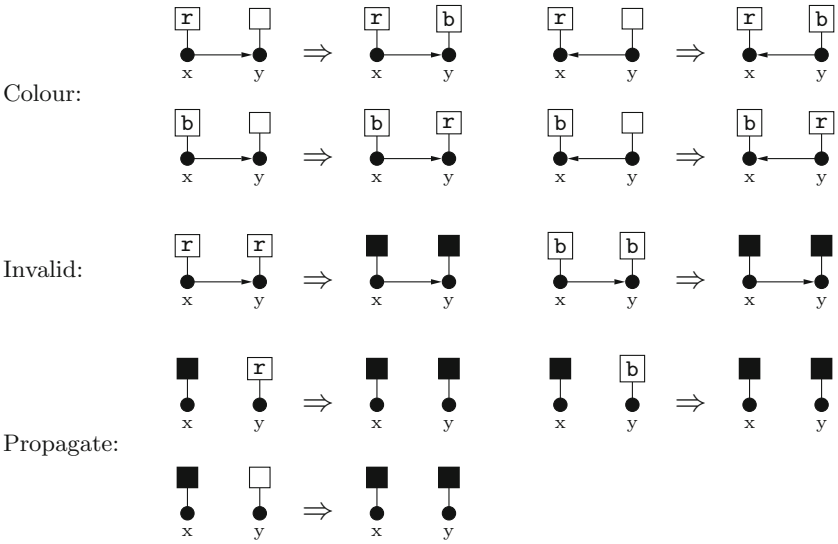


Fig. 10. Hypergraph transformation system for generating a 2-colouring

rules reduce the number of blank flags. On the other hand, Invalid ∪ Propagate is terminating as all rules decrease the number of non-black flags. It is easy to see that there are no critical pairs of shape $S \Rightarrow_{\text{Invalid}\cup\text{Propagate}} T \Rightarrow_{\text{Colour}} U$ (since all ordinary edges are interface edges) and thus the combined system is terminating. □

We remark that termination of the system of Fig. 10 can alternatively be proved by removing the Colour rules and observing that the Invalid and Propagate rules reduce the number of non-black flags. This works because the Colour rules reduce the number of blank flags while the Invalid and Propagate rules do not increase this number. However, the point of Example 5 is to demonstrate that a non-trivial system can be decomposed into subsystems such that Theorem 2 is applicable, and where the components are proved terminating with different measures.

Example 6. Our final example is about *jungle evaluation*, a framework in which hypergraphs representing functional expressions are evaluated by transformation rules [12]. Figure 11 shows the non-injective evaluation rule corresponding to the term rewriting rule $\mathtt{y} + \mathtt{0} \to \mathtt{y}$, where the notation x=y means that interface nodes x and y are merged by the right-hand morphism. This rule is clearly terminating as it reduces the size of any hypergraph it is applied to. Rule copy in Fig. 12, on the other hand, enlarges any hypergraph it is applied to. The rule copies an occurrence of the constant 0 that is shared by two s-functions. The rule

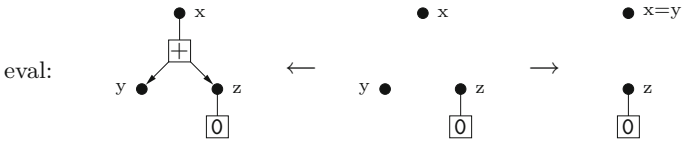


Fig. 11. Jungle evaluation rule for $\mathtt{y} + \mathtt{0} \to \mathtt{y}$

is terminating because each application reduces the measure

$$\#G = \sum_{v \in V_G} \text{indegree}(v)^2.$$

For, consider a step $G \Rightarrow_{\text{copy}} H$ and let n be the indegree of node z in G. Then $\#H = (\#G - n^2) + (n-1)^2 + 1 < \#G$ where the inequality holds because $(n-1)^2 + 1 < n^2$ for $n \geq 2$, and $n = \text{indegree}(z) \geq 2$.

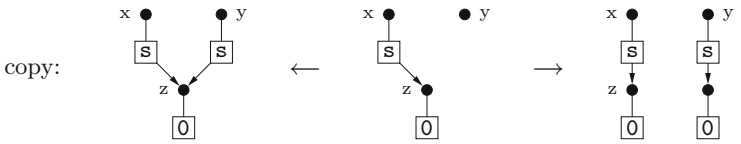


Fig. 12. Rule for copying a shared constant 0

It is not difficult to check that there are no sequential critical pairs of shape $S \Rightarrow_{\text{eval}} T \Rightarrow_{\text{copy}} U$, and thus the combined system {eval, copy} is terminating by Theorem 2. Note that it is not obvious how to combine graph size and the # value into a measure that decreases under rule applications of the combined rule set. This is because copy always increases graph size and eval increases the # value when applied to certain hypergraphs. To see the latter, consider a step $G \Rightarrow_{\text{eval}} H$ where $\text{indegree}(x) = 2$, $\text{indegree}(y) = 3$ and $\text{indegree}(z) = 1$. Then $\#H = \#G - (4 + 9 + 1) + (4^2 - 1) = \#G + 1$. □

5 Conclusion and Future Work

Termination is an undecidable property of graph transformation systems. We have established a criterion based on the absence of certain sequential critical pairs which guarantees that the union of two terminating systems is terminating. This allows to split systems into component systems whose termination is then verified separately, possibly using different techniques, and to conclude that the combination of the components is terminating if the critical pair-criterion is satisfied. The criterion is syntactic and can be machine-checked by generating all critical pairs between rules from different component systems. Moreover, the

method is a black-box approach in that the termination proofs of the component systems need not be inspected.

An obvious topic for future work is to implement a tool that given a hypergraph transformation system, generates all sequential critical pairs and calculates all possible partitions of the system into smaller components such that the condition of Theorem 2 is satisfied. For each partition, the components can then be proved to be terminating with whatever method seems suitable resp. has been implemented. The tool could be used together with a termination prover such as *Grez*, described in [5], which allows to choose different proof methods, including weighted type graphs, label counting and node counting.

Future research may also attempt to generalize Theorem 2 in various ways. It may be possible to allow critical pairs $S \Rightarrow_{\mathcal{R}} T \Rightarrow_{\mathcal{S}} U$ and formulate conditions under which arbitrary steps $G \Rightarrow_{\mathcal{R}} H \Rightarrow_{\mathcal{S}} M$ can still be swapped. A naive try is to require that there exits a graph T' such that $S \Rightarrow_{\mathcal{S}} T' \Rightarrow_{\mathcal{R}} U$, which however is insufficient as the dangling condition may prevent embedding the steps into context. The situation has some similarity with the analysis of conventional critical pairs to verify confluence: the mere joinability of all critical pairs does not guarantee local confluence of a set of rules [20].

Finally, it would be desirable to extend the approach of this paper such that rules with application conditions (of some form) can be handled. Even more challenging is an extension to attributed graph transformation on which graph programming languages such as GP 2 are based. This is because finite sets of attributed rules typically induce infinite sets of sequential critical pairs in the sense of this paper (see [13] for the corresponding problem with conventional critical pairs).

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Bezem, M., Klop, J.W., de Vrijer, R. (eds.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
3. Bruggink, H.J.S.: Towards a systematic method for proving termination of graph transformation systems. Electron. Notes Theor. Comput. Sci. **213**(1), 23–38 (2008). https://doi.org/10.1016/j.entcs.2008.04.072
4. Bruggink, H.J.S., König, B., Nolte, D., Zantema, H.: Proving termination of graph transformation systems using weighted type graphs over semirings. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 52–68. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_4
5. Bruggink, H.J.S., König, B., Zantema, H.: Termination analysis for graph transformation systems. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) TCS 2014. LNCS, vol. 8705, pp. 179–194. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44602-7_15
6. Courcelle, B.: Graph rewriting: an algebraic and logic approach. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, Chap. 5. Elsevier (1990)
7. Dershowitz, N.: Termination of linear rewriting systems (preliminary version). In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 448–458. Springer, Heidelberg (1981). https://doi.org/10.1007/3-540-10843-2_36

8. Ehrig, H., Kreowski, H.-J.: Parallelism of manipulations in multidimensional information structures. In: Mazurkiewicz, A. (ed.) MFCS 1976. LNCS, vol. 45, pp. 284–293. Springer, Heidelberg (1976). https://doi.org/10.1007/3-540-07854-1_188
9. Ehrig, H., Rosen, B.K.: Commutativity of independent transformations on complex objects. Research Report RC 6251. IBM Thomas J. Watson Research Center, Yorktown Heights (1976)
10. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0013875
11. Habel, A., Müller, J., Plump, D.: Double-pushout graph transformation revisited. Math. Struct. Comput. Sci. **11**(5), 637–688 (2001). https://doi.org/10.1017/S0960129501003425
12. Hoffmann, B., Plump, D.: Implementing term rewriting by jungle evaluation. RAIRO Theor. Inform. Appl. **25**(5), 445–472 (1991). https://doi.org/10.1051/ita/1991250504451
13. Hristakiev, I., Plump, D.: Towards critical pair analysis for the graph programming language GP 2. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 153–169. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72044-9_11
14. Krishna Rao, M.R.K.: Modular aspects of term graph rewriting. Theor. Comput. Sci. **208**(1–2), 59–86 (1998). https://doi.org/10.1016/S0304-3975(98)00079-6
15. Plump, D.: Implementing term rewriting by graph reduction: termination of combined systems. In: Kaplan, S., Okada, M. (eds.) CTRS 1990. LNCS, vol. 516, pp. 307–317. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54317-1_100
16. Plump, D.: On termination of graph rewriting. In: Nagl, M. (ed.) WG 1995. LNCS, vol. 1017, pp. 88–100. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60618-1_68
17. Plump, D.: Simplification orders for term graph rewriting. In: Prívara, I., Ružička, P. (eds.) MFCS 1997. LNCS, vol. 1295, pp. 458–467. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0029989
18. Plump, D.: Termination of graph rewriting is undecidable. Fundam. Inform. **33**(2), 201–209 (1998). https://doi.org/10.3233/FI-1998-33204
19. Plump, D.: Computing by Graph Rewriting. Habilitation thesis, Universität Bremen, Fachbereich Mathematik und Informatik (1999)
20. Plump, D.: Confluence of graph transformation revisited. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) Processes, Terms and Cycles: Steps on the Road to Infinity. LNCS, vol. 3838, pp. 280–308. Springer, Heidelberg (2005). https://doi.org/10.1007/11601548_16
21. Sabel, D., Zantema, H.: Termination of cycle rewriting by transformation and matrix interpretation. Log. Methods Comput. Sci. **13**(1), 38 (2017). https://doi.org/10.23638/LMCS-13(1:11)2017
22. Toyama, Y.: Counterexamples to termination for the direct sum of term rewriting systems. Inf. Process. Lett. **25**, 141–143 (1987). https://doi.org/10.1016/0020-0190(87)90122-0
23. Zantema, H., König, B., Bruggink, H.J.S.: Termination of cycle rewriting. In: Dowek, G. (ed.) RTA-TLCA 2014. LNCS, vol. 8560, pp. 476–490. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_33

Graph Attribution Through Sub-Graphs

Harmen Kastenberg and Arend Rensink(✉)

Department of Computer Science, University of Twente, Enschede, The Netherlands
arend.rensink@utwente.nl

Abstract. We offer an alternative to the standard way of formalising attributed graphs. We propose to represent them as graphs with a marked sub-graph that represents the data domain, rather than as tuples of graph and algebra. This is a general construction which can be shown to preserve adhesiveness of categories; it has the advantage of uniformity and gives more flexibility in defining data abstractions. We show equivalence of our formalisation with the standard one, under a suitable encoding of algebras as graphs.

1 Introduction

Graph transformation has many strengths and pleasant characteristics, but the treatment of data values, such as integers, booleans and strings, is not among them. In fact, the core idea of graph-based modelling is that concrete node and edge identities are irrelevant, and so graphs can be regarded up to isomorphism; this, however, is simply no longer true if the nodes stand for data values.

Nevertheless, the large majority of systems for which graph-based modelling is appropriate do include primitive data, in the form of attributes. There is therefore no question but that graph transformation has to cope with data in order to be practically useful in modelling real-world applications. And so, a model for attributed graphs has been worked out by Ehrig et al. [3], which we will refer to as the *standard model*.

The standard model explicitly combines the world of graphs and that of algebras; the manipulation of the data is deferred to the second, whereas the data values appear as nodes in the graphs, to which it is possible to define edges from ordinary nodes. Such edges then stand for attributes. Although this is theoretically satisfactory, in that the model allows us to use attributes, and is, moreover, a "nice" category for graph transformation (meaning that it is adhesive HLR — more about this later), we feel that the standard model leaves some things to be desired.

– Due to the presence of both graphs and algebras in the standard model, some things are solved twice. In particular, in transformation rules, the algebra component uses variables, terms and (in)equations, whereas the graph component uses nodes, edges and (non)injectivity constraints, for essentially the same functionality. This means that users have two different formalisms to

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 245–265, 2018.
https://doi.org/10.1007/978-3-319-75396-6_14

cope with, and the visual presentation of rules needs to combine graphical and textual parts. Moreover, an implementation also needs to contain distinct algorithms for matching the graph and algebra parts.
– We are studying abstraction in graph transformation, in particular also data abstraction. A very limited form of abstraction is possible using algebras, by moving from the standard algebra of a given signature (for instance, the integers with successor, addition and multiplication) by a surjective homomorphism to another algebra (for instance, the integers modulo an upper bound). However, many interesting abstractions cannot be formulated as algebra homomorphisms. For instance, the classical abstraction of the integers into the three-valued set of "strict negative", "zero" and "strict positive" either does not give rise to an algebra (the operations are not deterministic); or, if we add the joined elements "negative", "positive" and "all", then there is no homomorphism from the standard algebra to this one.

The first of these issues has prompted us to consider a version of attributed graphs in which the algebras are entirely encoded as sub-graphs. In particular, the operations are also coded up, by adding corresponding nodes and edges. A preliminary version of this idea was presented in [7]. Since (at need) these sub-graphs are easily distinguishable from the surrounding "real" graphs by typing, in most circumstances we can proceed as if we were dealing with standard graphs.

A side benefit is that this sub-graph arrangement can be understood as a general categorical construction: namely, it gives rise to a category of *reflected monos*, in which the objects are monos (corresponding to embedded graphs) and the arrows are pullbacks. The proof of adhesiveness of the resulting category can therefore be established on a more general level than for the standard model.

It turns out that this also provides a solution to the second issue. By extending the set of "algebra graphs" allowed as sub-graphs with graphs in which the algebraic operations are not deterministic (and so are no longer truly operations), we can easily cope with data abstractions such as the one mentioned above. Our proof of adhesiveness carries over to the extended category without any changes. Now the embedding theorem implies that the abstract graphs over-approximate the behaviour of the concrete graphs. We also extend the embedding theorem to rules with negative application conditions, provided that these do not test (negatively) for the data part.

The paper is structured as follows: in Sect. 2 we define our attributed graph category and establish equivalence with the standard model. In Sect. 3 we give an independent proof that the construction gives rise to an adhesive category. In Sect. 4 we discuss data abstraction, and we show that the embedding theorem still holds in the presence of NACs which do not test for data. In Sect. 5 we briefly discuss the implementation of these concepts in the graph transformation tool GROOVE. Section 6 concludes the paper.

Almost all of the proofs are silently omitted from this version of the paper. For the full technical report, including all proofs, see [8].

2 The Model

In this section, we show how the structure of any algebra can be encoded as a graph. We then combine these *algebra graphs* with the graphs that need attribution, giving rise to larger graphs of which the algebra graphs are sub-graphs; attributes then take the form of edges from the surrounding graph into the algebra sub-graph.

Some general notational conventions: if $s \in A^*$ is a sequence, say $s = s_1 \cdots s_n$, then $|s|$ denotes the length (n), $[s]$ denotes the set of elements in s ($\{s_1, \ldots, s_n\}$), and for all $1 \leq i \leq n$, $s|_i$ denotes the ith element (s_i). The empty sequence is denoted ε.

2.1 Algebra Graphs

Let us first recall the standard definitions of signatures and algebras. We assume a global set **Name** of names, which are symbols that are of themselves uninterpreted; the interpretation is given by their use.

Definition 1 (signature). *A* signature *is a tuple* $\Sigma = \langle S, O, \sigma, \tau \rangle$ *where* $S \subseteq$ **Name** *is a set of* sorts*,* $O \subseteq$ **Name** *is a set of* operators*, disjoint from* S*,* $\sigma \colon O \to S^*$ *is the* source typing *of the operators, and* $\tau \colon O \to S$ *is the* target typing *of the operators.*

We call a sort s of a given signature *spurious* if there is no operator that uses it, i.e., $s \notin [\sigma(o)]$ for all $o \in O$. In this paper we assume that signatures have no spurious sorts.

Given a signature, the *arity* of an operator $o \in O$ is given by $\alpha(o) = |\sigma(o)|$. We call a signature *unary* if $\alpha(o) = 1$ for all $o \in O$.

Example 2. As a running example we use the algebra of booleans and integers with a few operations. This is given by the signature Prim with $S = \{\mathsf{Int}, \mathsf{Bool}\}$ and O, σ and τ given by the following table. (lt stands for *lesser than.*)

O	zero	succ	pred	add	lt	pos	true	false	not
σ	ε	Int	Int	Int Int	Int Int	Int	ε	ε	Bool
τ	Int	Int	Int	Int	Bool	Bool	Bool	Bool	Bool

Definition 3 (algebra). *An* algebra *over a signature* Σ *is a tuple* $A = \langle D, F \rangle$ *where*

- $D = (D^s)_{s \in S}$ *is an* S*-indexed family of disjoint* data sets*;*
- $F = (f^o)_{o \in O}$ *is an* O*-indexed family of* functions *typed by the signature; i.e., for all* $o \in O$*, if* $\sigma(o) = s_1 \cdots s_n$ *then* $f^o \colon D^{s_1} \times \cdots \times D^{s_n} \to D^{\tau(o)}$*.*

Given two algebras $A_i = \langle D_i, F_i \rangle$ *over* Σ *(*$i = 1, 2$*), an* algebra morphism *is a family of functions* $h = (h^s \colon D_1^s \to D_2^s)_{s \in S}$ *such that for all* $o \in O$ *with* $\sigma(o) = s_1 \cdots s_n$ *and for all* $d_j \in D_1^{s_j}$ *(*$j = 1, \ldots, n$*):*

$$h^s(f_1^o(d_1, \ldots, d_n)) = f_2^o(h^{s_1}(d_1), \ldots, h^{s_n}(d_n)) \ .$$

We commonly use D_A and F_A to denote the data sets and functions of an algebra A; we omit the subscript A if it is clear from the context. The algebras over a signature Σ together with the algebra morphisms form a category, which we call $\mathbf{Alg}(\Sigma)$.

Example 4. For the signature of Example 2, one may consider the following algebras:

- The *initial* or *term* algebra A_{Term}, where all terms built over Prim denote distinct elements. The data sets of this algebra consist of the (syntax trees of the) terms themselves.
- The *natural* or *standard* algebra A_{Std}, consisting of the "real" integers and booleans.
- The *final* or *point* algebra A_{Point}, where the data sets are all singletons, i.e., all values are collapsed to a single one.

There are unique algebra morphisms from A_{Term} to A_{Std} and from A_{Std} to A_{Point}; for instance, if $h\colon A_{\mathsf{Term}} \to A_{\mathsf{Std}}$ then $h^{\mathsf{Int}}(\mathsf{succ}(\mathsf{zero}())) = 1$ and $h^{\mathsf{Bool}}(\mathsf{true}()) = h^{\mathsf{Bool}}(\mathsf{not}(\mathsf{false}())) = \mathit{true}$.

The encoding of algebras as graphs is essentially straightforward:

- The data values (i.e., the elements of the carrier sets) are represented by nodes
- The functions are interpreted as sets of pairs of elements from the function domain, respectively codomain; these pairs are then represented by edges.

The only complication is that, for operators with arity > 1, the domain of the corresponding function is a cartesian product; in order to interpret such a function as a set of edges we need to introduce nodes for the elements of the domain, i.e., nodes that stand for *tuples* of data values. For unary signatures, this complication does not arise, hence we concentrate on these first; we then show a way to transform algebras over arbitrary signatures into equivalent algebras over unary signatures.

Definition 5 (graph). *A* graph *is a tuple* $G = \langle N, E, src, tgt, lab\rangle$ *where* N *is a set of* nodes*,* E *is a set of* edges*,* $src\colon E \to N$ *is a* source function*,* $tgt\colon E \to N$ *is a* target function*, and* $lab\colon E \to \mathbf{Name}$ *is a* labelling*.*

Given two graphs $G_i = \langle N_i, E_i, src_i, tgt_i, lab_i\rangle$ *for* $i = 1, 2$*, a* graph morphism *from* G_1 *to* G_2 *is a pair* $h = (h^N\colon N_1 \to N_2, h^E\colon E_1 \to E_2)$ *such that, for all* $e \in E_1$*,*

$$\begin{aligned} src_2(h^E(e)) &= h^N(src_1(e)) \\ tgt_2(h^E(e)) &= h^N(tgt_1(e)) \\ lab_2(h^E(e)) &= lab_1(e). \end{aligned}$$

We commonly use N_G, E_G etc. to denote the components of a graph G; we omit the subscript G if it is clear from the context. Graphs and graph morphisms form a category, which we call **Graph** (identity arrows are pairs of identity functions

over the node and edge sets, and arrow composition is component-wise composition of the node and edge functions). We call a graph G *discrete* if $E_G = \emptyset$, i.e., the graph consists of nodes only. The full sub-category of **Graph** consisting of discrete graphs will be denoted **dGraph**. Note that a unary signature Σ can be seen as a graph where the nodes are sorts and the edges are operators. For edge labels we can use the operators themselves. This gives rise to the *signature graph* $G_\Sigma = \langle S, O, \sigma, \tau, id_O \rangle$.

Definition 6 (algebra graph). *Let Σ be a unary signature. An* algebra graph *over Σ is a graph G with a morphism t to G_Σ such that for all $n \in N_G$ and $o \in O$, if $t^N(n) = \sigma(o)$ then there is an edge $e \in E_G$ such that $src(e) = n$ and $t^E(e) = o$. G is called* deterministic *if this edge e is always unique.*

For a given unary signature Σ, we use $\mathbf{AlgGraph}^+(\Sigma)$ to denote the full sub-category of **Graph** consisting of all algebra graphs over Σ, and $\mathbf{AlgGraph}(\Sigma)$ for the full (further) sub-category of deterministic algebra graphs.

(The upshot of the above definition is that t acts as a typing morphism from G to G_Σ; the additional conditions on the existence and, in the case of determinism, uniqueness of edges can be understood as multiplicity constraints in the type graph G_Σ: all edges have outgoing multiplicity $1..*$ or, in the case of determinism, 1.)

Example 7. Figure 1 shows an algebra graph for a variation on Prim, viz., the unary signature Σ with $S_\Sigma = S_{\mathsf{Prim}}$ and $O_\Sigma = \{\mathsf{succ}, \mathsf{odd}, \mathsf{not}\}$. Here, odd tests if a number is odd; it has $\sigma(\mathsf{odd}) = \mathsf{Int}$ and $\tau(\mathsf{odd}) = \mathsf{Bool}$.

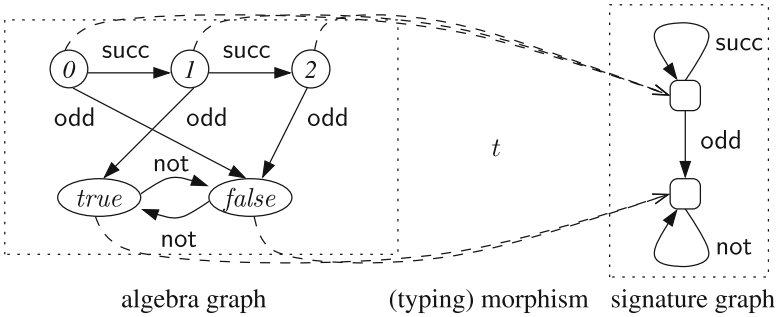


Fig. 1. Algebra graph with typing into the signature graph. Italic node labels stand for algebra values.

The following proposition is important in that it implies that it is enough to know that a graph is in $\mathbf{AlgGraph}^+(\Sigma)$ (for a given unary signature Σ) in order to reconstruct the actual typing morphism. This relies on our assumption that Σ has no spurious sorts.

Proposition 8. *For any Σ and $G \in \mathbf{AlgGraph}^+(\Sigma)$, there exists exactly one (typing) morphism $t\colon G \to G_\Sigma$.*

The following theorem essentially states that our encoding of algebras as graphs works.

Theorem 9. *For any unary Σ,* $\mathbf{Alg}(\Sigma)$ *and* $\mathbf{AlgGraph}(\Sigma)$ *are equivalent.*

This is proved by two functors, one of which turns data values into nodes and codes up the operations as edges, and the other of which undoes this by reconstructing the operations from the edges. The full proof can be found in [8].

For non-unary signatures, the situation is more complicated: first we have to *flatten* the signatures and algebras, but we also have to impose some additional constraints on the flattened algebras in order to get an equivalent category.

Definition 10 (product sorts).

1. *A* signature with products *is a pair $\Sigma|\pi$ where $\Sigma = \langle S, O\rangle$ is a unary signature and $\pi\colon S \rightharpoonup O^*$ is a partial function that assigns to some of the sorts (called the product sorts) a sequence of distinct* projection operators, *such that $src(o) = s$ for all $o \in [\pi(s)]$. For product sorts $p \in dom(\pi)$ we use $w(p) = |\pi(p)|$ to denote the* width *of p, and $\pi_{p,i}$ $(1 \leq i \leq w(p))$ to denote the individual elements of $\pi(p)$ (hence $\pi(p) = \pi_{p,1} \cdots \pi_{p,w(p)}$).*
2. *An* algebra over $\Sigma|\pi$ *is an algebra over Σ such that, in addition, for all sorts $p \in dom(\pi)$ and all combinations of data values $(d_i \in D^{tgt(\pi_{p,i})})_{1\leq i\leq w(p)}$ from the target sorts of the projection operators, there is a unique $d \in D^p$ with $f^{\pi_{p,i}}(d) = d_i$ for all $1 \leq i \leq w(p)$.*
3. *An* algebra graph *G over $\Sigma|\pi$ is an algebra graph over Σ, with typing t, such that, in addition, for all product sorts $p \in dom(\pi)$ and all combinations of nodes $(n_i \in t^{N,-1}(tgt(\pi_{p,i})))_{1\leq i\leq w(p)}$ typed by the target sorts of the projection operators, there is an $n \in N$ and a family of edges $(e_i \in E)_{1\leq i\leq w(p)}$ such that for all $1 \leq i \leq w(p)$, $t^E(e) = \pi_{p,i}$, $src(e) = n$ and $tgt(e) = n_i$. G is called deterministic if, in addition to the conditions of Definition 6, this n is unique.*

The underlying intuition is as follows: if p is a product sort with projection operators $\pi(p) = o_1 \cdots o_n$, and respective target sorts $s_1 \cdots s_n$, then Clause 10.2 above guarantees that D^p is essentially the cartesian product $D^{s_1} \times \cdots D^{s_n}$ and the o_i project the values of D^p to their ith components; and analogously for algebra graphs.

If $\Sigma|\pi$ is a signature with products, we use $\mathbf{Alg}(\Sigma|\pi)$ to denote the category of algebras over $\Sigma|\pi$ and $\mathbf{AlgGraph}^+(\Sigma|\pi)$ [resp. $\mathbf{AlgGraph}(\Sigma|\pi)$] to denote the category of [deterministic] algebra graphs over $\Sigma|\pi$. The following extends Theorem 9 to signatures with products.

Theorem 11. *For any $\Sigma|\pi$,* $\mathbf{Alg}(\Sigma|\pi)$ *and* $\mathbf{AlgGraph}(\Sigma|\pi)$ *are equivalent.*

The following result states that we can indeed flatten arbitrary signatures into signatures with products, and obtain equivalent categories of algebras.

Theorem 12. *For any Σ, there is a signature with products* $\mathsf{flat}(\Sigma)$ *such that* $\mathbf{Alg}(\Sigma)$ *and* $\mathbf{Alg}(\mathsf{flat}(\Sigma))$ *are equivalent.*

To construct $\mathsf{flat}(\Sigma)$, we need to add product sorts and projection operators. For this purpose, assume disjoint subsets of product sort names and projection operator names, which are also disjoint from S and O. For all $z \in S^*$, let s_z denote a distinct fresh product sort name corresponding to z, and for all $1 \leq i \leq |z|$, let $p_{z,i}$ denote a distinct fresh projection operator name from s_z-values to their ith components. Now $\mathsf{flat}(\Sigma)$ is defined as $\Sigma_1|\pi$, where Σ_1 consists of[1]

$$
\begin{aligned}
S_1 &= S \cup \{s_{\sigma(o)} \mid o \in O\} \\
O_1 &= O \cup \{p_{\sigma(o),i} \mid o \in O, 1 \leq i \leq \alpha(o)\} \\
\sigma_1 &= \{(o, s_{\sigma(o)}) \mid o \in O\} \cup \{(p_{\sigma(o),i}, s_{\sigma(o)}) \mid o \in O, 1 \leq i \leq \alpha(o)\} \\
\tau_1 &= \tau \cup \{(p_{\sigma(o),i}, \sigma(o)|_i) \mid o \in O, 1 \leq i \leq \alpha(o)\} \\
\pi &= \{(s_{\sigma(o)}, p_{\sigma(o),1} \cdots p_{\sigma(o),\alpha(o)}) \mid o \in O\}.
\end{aligned}
$$

By combining the above results, we get

Corollary 13. *For any Σ,* $\mathbf{Alg}(\Sigma)$ *and* $\mathbf{AlgGraph}(\mathsf{flat}(\Sigma))$ *are equivalent.*

2.2 Reflected Graph Embeddings

To achieve graph attribution, we embed algebra graphs into larger graphs. To define the necessary constructs, let $\subseteq$ define the component-wise subset relation over graphs.

Definition 14 (graph embedding). *Let* $\mathbf{G}$ *be a sub-category of* **Graph**. *A* graph embedding *over* $\mathbf{G}$ *is a pair* (G^-, G) *such that* $G^- \in \mathbf{G}$ *and* $G^- \subseteq G \in$ **Graph**. *If* $(G^-, G), (H^-, H)$ *are graph embeddings, then a* reflection *from* (G^-, G) *to* (H^-, H) *is a graph morphism* $h\colon G \to H$ *such that for all* $n \in N_G$, $h^N(n) \in N_{H^-}$ *implies* $n \in N_{G^-}$, *and for all* $e \in E_G$, $h^E(e) \in E_{H^-}$ *implies* $e \in E_{G^-}$. $\mathbf{REmb}(\mathbf{G})$ *denotes the category of graph embeddings over* $\mathbf{G}$ *with reflections as arrows.*

A graph embedding (G^-, G) *is said to be* glued over *a discrete graph* $G^{--} \subseteq G^-$, *if for all* $e \in E_G \setminus E_{G^-}$ *and incident nodes* $n \in \{src(e), tgt(e)\}$, $n \in N_{G^-}$ *implies* $n \in N_{G^{--}}$. *An* embedding functor *is a functor* $\mathcal{E}\colon \mathbf{G} \to \mathbf{dGraph}$ *such that* $\mathcal{E}(G) \subseteq G$ *for all* $\mathbf{G}$*-graphs* G *and* $\mathcal{E}(f) = f \restriction \mathcal{E}(G)$ *for all* $\mathbf{G}$*-morphisms* $f\colon G \to H$. $\mathbf{REmb}(\mathcal{E})$ *denotes the full sub-category of* $\mathbf{REmb}(\mathbf{G})$ *consisting of embeddings* (G^-, G) *glued over* $\mathcal{E}(G^-)$.

The term *reflection* is chosen to stress that the structure of the subgraph H^- is reflected (as the dual of preserved) in G^-.

[1] It should be noted that Σ_1 has a bipartite signature graph (and hence bipartite algebra graphs) as *every* operation is redefined to have a product sort as its source; even the operations that were already unary to start with. This is not at all necessary for the results in this paper: other constructions for $\mathsf{flat}(\Sigma)$ may be more intuitive in practice.

Thus, if an embedding (G^-, G) is glued over a graph G^{--}, this means that only nodes in G^{--} may be connected (by G-edges) to nodes outside G^-. For instance, in this paper we do not want to allow attribute edges to point to product nodes as these are meant only as auxiliaries,[2] so our embeddings will be glued over the sub-graph of the algebra graph with only non-product nodes. Very often we just use G to denote graph embeddings (G^-, G).

Based on this, we can define our category of attributed graphs. In this definition, $\mathcal{E}_{\Sigma|\pi}$: $\mathbf{AlgGraph}(\Sigma|\pi) \to \mathbf{dGraph}$ (for an arbitrary signature with products $\Sigma|\pi$) is the embedding functor mapping every $\Sigma|\pi$-algebra graph G to the discrete sub-graph with nodes $\{n \in N_G \mid t(n) \in S_\Sigma \setminus dom(\pi)\}$, where t is the typing of G into G_Σ.

$$\mathbf{AttGraph}(\Sigma) = \mathbf{REmb}(\mathcal{E}_{\mathsf{flat}(\Sigma)}). \tag{1}$$

Although the formal definition may appear complicated (partially because we have set it up so that it is a special case of the general framework introduced in the next section), the basic idea is still conceptually simple: an attributed graph is a graph with an embedded deterministic algebra graph. This means that there are three types of edges in the overall graph:

- Edges within the algebra graph. These encode the algebra, as discussed above.
- Edges entirely outside the algebra graph, i.e., with end nodes also outside the algebra graph. These represent the "ordinary" graph structure.
- Edges not in the algebra graph, but with one or more end nodes in the algebra graph. These are *attribute edges*, i.e., they provide the kind of information that we introduced attributed graphs for in the first place.

Example 15. Figure 2 shows an example attributed graph for the signature Prim of Example 2, using the standard algebra, encoded into the graph structure. (Obviously the algebra graph is only partially shown.) Examples of algebra-only edges are the succ- and π-labelled edges; A, B and next are ordinary graph edges; and x and y are attribute edges. The italic inscriptions *0*, *1* and *true* represent the algebra values and are formally not part of the actual graph. Note that only non-product nodes are used as glue between the algebra graph to the "real" graph.

For arbitrary signatures, we first have to construct the algebra graph with product sorts; an attributed graph is then a graph with this algebra graph embedded, such that, moreover, only the non-product sorts are eligible as end nodes of the attribute edges.

With a fairly light discipline on the choice of labels, we can in fact make the definitions even easier. Namely, if we assume that operators of the signature Σ are never used to label edges in $E_G \setminus E_{G^-}$, then G^- can be constructed from G by restricting to the O-labelled edges.

[2] This is a choice, not a necessity: one might actually want to have sorts that stand for tuples in the original, unflattened signature.

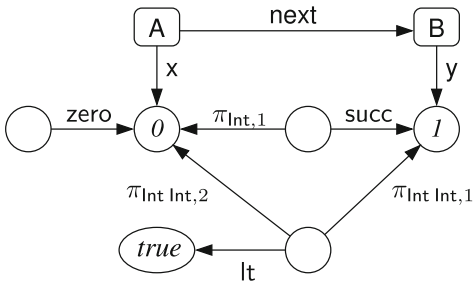


Fig. 2. Example attributed graph; rectangular nodes are ordinary graph nodes, ellipsoid ones represent algebra values.

We now show that this category is essentially equivalent to the standard model of [3]. We reformulate their definition so as to make the equivalence proof easier.

Definition 16. *Let* $\mathcal{D}\colon \mathbf{C} \to \mathbf{dGraph}$ *be a functor to discrete graphs. The category of* $\mathcal{D}$*-attributed graphs* $\mathbf{SAttGraph}(\mathcal{D})$ *is defined by*

- *Objects* $\langle G, C\rangle$ *where* G *is a graph and* C *an object of* $\mathbf{C}$*, such that* $\mathcal{D}(C) \subseteq G$*.*
- *Arrows* $(f\colon G \to H, g\colon B \to C)$*, where* f *is a graph morphism and* g *an arrow from* $\mathbf{C}$*, such that* $\mathcal{D}(g) = f \restriction \mathcal{D}(dom(g))$ *— in other words,* f *and* g *agree upon the discrete graph.*

Examples of functors $\mathcal{D}$ that can be "plugged in" here are:

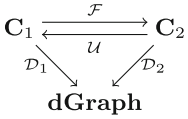
- $\mathcal{A}_\Sigma\colon \mathbf{Alg}(\Sigma) \to \mathbf{dGraph}$ for an arbitrary signature Σ, mapping every Σ-algebra A to the discrete graph with $N = \bigcup_{s\in S} D^s$;
- $\mathcal{A}_{\Sigma|\pi}\colon \mathbf{Alg}(\Sigma|\pi) \to \mathbf{dGraph}$ for a signature with product sorts $\Sigma|\pi$, mapping every $\Sigma|\pi$-algebra A to the discrete graph with $N = \bigcup_{s\in S\setminus dom(\pi)} D^s$;
- The functor $\mathcal{E}_{\Sigma|\pi}\colon \mathbf{AlgGraph}(\Sigma|\pi) \to \mathbf{dGraph}$ defined above.

The standard category of node-attributed graphs, as defined in [3], is essentially given by $\mathbf{SAttGraph}(\mathcal{A}_\Sigma)$ — where "essentially" means that we ignore some differences:

- In the standard model, attributed graphs are *typed*. We leave out typing because we find it complicates the presentation; moreover, enriching graphs with typing is a standard construction — see, e.g., [10].
- In the standard model, the only connections allowed between the non-attribute part of the graph and attribute (i.e., algebra) values are edges with non-data nodes as sources. We find that this constraint unnecessarily complicates the presentation and does not affect the formalism in any way; moreover, we believe that attribute edges starting in data nodes may be useful as well. Furthermore, this constraint can always be imposed on top of our definition, if so desired.

– The standard model includes *edge attributes*, which are essentially edges whose sources are edges. These present a technical complication which we have omitted, but which could be catered for by extending the category **Graph** with such edges in general.[3]

Definition 17. *Two functors* $\mathcal{D}_i \colon \mathbf{C}_i \to \mathbf{dGraph}$ *(*$i = 1, 2$*) are* source equivalent *if there are functors* $\mathcal{F} \colon \mathbf{C}_1 \to \mathbf{C}_2$ *and* $\mathcal{U} \colon \mathbf{C}_2 \to \mathbf{C}_1$ *which establish an equivalence between* $\mathbf{C}_1$ *and* $\mathbf{C}_2$*, and such that, moreover, the following diagram of functors commutes:*



For instance, the functors $\mathcal{A}_\Sigma$, $\mathcal{A}_{\mathsf{flat}(\Sigma)}$ and $\mathcal{E}_{\mathsf{flat}(\Sigma)}$ introduced above are pairwise source equivalent for arbitrary Σ, due to (respectively) Theorems 11 and 12.

The reason for introducing source equivalence is the following theorem, which states that replacing the "data component" in the standard model by a source equivalent one does not change the category.

Theorem 18. *If* $\mathcal{D}_i \colon \mathbf{C}_i \to \mathbf{dGraph}$ *for* $i = 1, 2$ *are two source equivalent functors, then* $\mathbf{SAttGraph}(\mathcal{D}_1)$ *and* $\mathbf{SAttGraph}(\mathcal{D}_2)$ *are equivalent categories.*

This is shown by functors between $\mathbf{SAttGraph}(\mathcal{D}_1)$ and $\mathbf{SAttGraph}(\mathcal{D}_2)$ that coincide with $\mathcal{D}_1$ and $\mathcal{D}_2$ on the algebra component and with the identity functor on the graph component. Note that the source equivalence precisely guarantees that the part of the algebra used in the graph remains untouched when replacing $\mathcal{D}_1$ by $\mathcal{D}_2$, and hence the identity functor can be used.

The final auxiliary result on the road to proving equivalence between the standard model and our formalisation is the following.

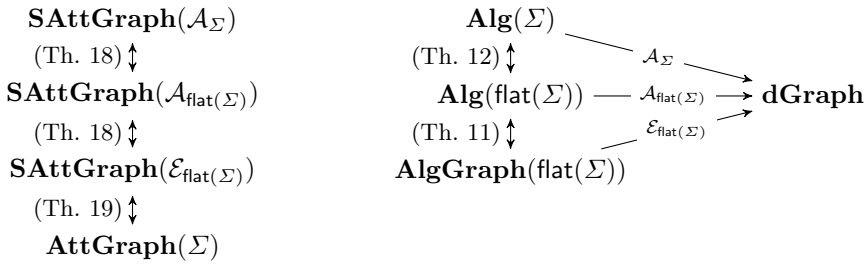
Theorem 19. *For any* $\Sigma|\pi$, $\mathbf{SAttGraph}(\mathcal{E}_{\Sigma|\pi})$ *and* $\mathbf{REmb}(\mathcal{E}_{\Sigma|\pi})$ *are equivalent.*

This results in the following corollary, which is the first main result of this paper:

Corollary 20. *For any* Σ, $\mathbf{SAttGraph}(\mathcal{A}_\Sigma)$ *and* $\mathbf{AttGraph}(\Sigma)$ *are equivalent.*

Proof. This follows from a chain of equivalences sketched in the following diagram.

[3] Methodologically, we believe that edge attributes are not a useful concept, since they can always be encoded by using attributed nodes instead. In a context where the increase in expressiveness is felt to be worth the price of a more complicated formalism, we believe that an extension to *hyper-edges* is typically more appropriate than edges over edges.

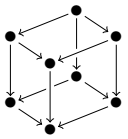


Here, $\leftrightarrow$ denotes equivalence of categories and $\rightarrow$ denotes a functor. The vertical chain on the left contains the actual steps of the proof; the diagram on the right is the justification for applying Theorem 18.

3 Adhesiveness

In this section we reformulate the core construction above, that of graph embeddings (Definition 14), in a more general way, getting away from the precise choice of graph category. For this, we adopt the setting of adhesive HLR categories, developed by Ehrig et al. [5] based on the adhesive categories of Lack and Sobociński [11]. One of the advantages is that, in this setting, many theorems come "for free;" an example is the *embedding theorem* used in the next section. We show that our embedding construction, generalised as the category of *reflected monos*, preserves adhesiveness, or can give rise to particular HLR adhesive categories. Among other things, this essentially constitutes an alternative proof strategy for the HLR adhesiveness of **SAttGraph**.

For lack of space, we have to omit the definitions of the basic categorical concepts. In addition we need the more involved concept of *Van Kampen squares*. A Van Kampen square in a given category is a commuting square which, if used as the bottom square in a "cube" diagram of which the back faces are pullbacks (see right), guarantees that the front faces are pullbacks if and only if the top square is a pushout.



Definition 21 (adhesive HLR category, [5]). *Let* $\mathbf{C}$ *be a category. A class of morphisms* $\mathcal{M}$ *in* $\mathbf{C}$ *is called* suitable *if it satisfies the following properties:*

- $\mathcal{M}$ *consists of monomorphisms;*
- $\mathcal{M}$ *is closed under isomorphisms and composition;*
- $\mathcal{M}$ *is closed under pushout and pullback.*

$\mathbf{C}$ *is called an* adhesive HLR category *for a suitable class of morphisms* $\mathcal{M}$ *if it satisfies the following properties for all* $f \in \mathcal{M}$*:*

- *Each cospan* $\bullet \xrightarrow{f} \bullet \leftarrow \bullet$ *has a pullback;*
- *Each span* $\bullet \xleftarrow{f} \bullet \rightarrow \bullet$ *has a pushout, such that the pushout diagram is a Van Kampen square.*

A category is *adhesive* in the sense of [11, Definition 5], if it is adhesive HLR for the class $\mathcal{M}$ of all monomorphisms, and moreover, all pullbacks exist. The conditions on adhesive categories essentially ensure that such categories are "set-like"; that is, the pushout is "union-like" and the pullback is "intersection-like".

For instance, our example category, **Graph**, is adhesive, as shown in [11, Proposition 8]; and so is $\mathbf{AlgGraph}^+$, due to the fact (not proved here) that $\mathbf{AlgGraph}^+$ is closed under **Graph**-pushouts and -pullbacks. On the other hand, **AlgGraph** is *not* adhesive, and indeed could not be, given that it is equivalent to **Alg** (see Theorem 9) which is well known not to be adhesive. Another observation is that in any category **C** the class of isomorphisms is suitable in the sense of Definition 21; since, moreover, pushouts and pullbacks over isomorphisms always trivially exist, the following is easy to show:

Proposition 22. *Every category is adhesive HLR for the class $\mathcal{M}$ of isomorphisms.*

3.1 Reflected Monos

We now define a categorical construction generalising reflected embeddings (Definition 14).

Definition 23 (reflected monos). *Let* **C** *be an arbitrary category. The category of* reflected monos *in* **C**, *denoted* $\mathbf{RMon}(\mathbf{C})$, *is defined as follows:*

- *Objects are monos $a\colon A \hookrightarrow B$ of* **C**; *we write a^- and a^+ for the* inner object *A and* outer object *B, respectively;*
- *Arrows $f\colon a \to b$ are pairs of arrows $(f^-\colon a^- \to b^-, f^+\colon a^+ \to b^+)$ from* **C** *such that the resulting square is a pullback diagram:*

$$\begin{array}{ccc} a^+ & \xrightarrow{f^+} & b^+ \\ {\scriptstyle a}\uparrow & PB & \uparrow{\scriptstyle b} \\ a^- & \xrightarrow{f^-} & b^- \end{array}$$

Identities and arrow composition are defined component-wise.

Note that this indeed gives rise to a category; in particular, arrow composition is correct due to the pullback composition property.

The intuition behind the definition of **RMon** is that monos a, in set-like categories, are essentially embeddings of the inner object a^- into the outer object a^+. We will refer to the part of a^+ that is "disjoint" from a^- as the *rim* of a; this may be thought of as the largest sub-object of a^+ which, when taking the coproduct with a^-, is still a sub-object of a^+. The pullback property of the morphisms $f\colon a \to b$ ensures that none of the rim of a "spills over" into the inner object b^-; or in other words, b^- is *reflected* in a^-. Some more observations:

- If **C** has an initial object 0, then monos $0 \hookrightarrow A$ have an "empty inner object"; essentially, the entire object A is rim. We call such objects *closed*.

- Intuitively, the outer object a^+ consists of the rim, the inner object, and some additional structure connecting the inner object to the rim. We informally refer to this connecting structure as "glue." For instance, in the case of attributed graphs, the glue is the set of attribute edges.
- In general, arrows f incorporate changes to both the rim, the inner object and the glue. Arrows f that completely preserve the inner object are characterised by the fact that f^- is an isomorphism; we call such arrows *inner isomorphisms*. Preservation of the rim, on the other hand, can be captured by requiring that the pullback diagram of f is also a pushout diagram (in $\mathbf{C}$). Finally, if $\mathbf{C}$ has an initial object, then the simultaneous preservation of the inner object and the glue can also be captured; see Definition 35.
- Due to the well-definedness of pullbacks up to isomorphism, every arrow $f\colon a \to b$ in $\mathbf{RMon}$ is essentially determined by its outer component, f^+.

The following is another core result of this paper. To prove it, we first need to establish that monos in $\mathbf{RMon}(\mathbf{C})$ are pairs of (outer and inner) monos in $\mathbf{C}$; pushouts over monos in $\mathbf{RMon}(\mathbf{C})$ consist of outer pushouts and inner VK squares in $\mathbf{C}$; and pullbacks in $\mathbf{RMon}(\mathbf{C})$ consist of outer and inner pullbacks in $\mathbf{C}$.

Theorem 24. *If* $\mathbf{C}$ *is an adhesive category, then so is* $\mathbf{RMon}(\mathbf{C})$.

Unfortunately, reflected monos do not yet capture the category $\mathbf{AttGraph}(\Sigma)$ defined in (1), since for $\mathbf{AttGraph}(\Sigma)$ we had the following further constraints:

- Inner graphs were restricted to the sub-category of algebra graphs over $\mathsf{flat}(\Sigma)$;
- Embeddings were restricted to those glued over a further sub-graph.

We will show how to lift the first kind of restriction to reflected monos, and very briefly hint on how to achieve the second. For a full sub-category $\mathbf{D}$ of $\mathbf{C}$, let $\mathbf{RMon}(\mathbf{D}, \mathbf{C})$ denote the full sub-category of $\mathbf{RMon}(\mathbf{C})$ such that all inner objects are in $\mathbf{D}$.

Proposition 25. *For any full subcategory* $\mathbf{G}$ *of* $\mathbf{Graph}$, $\mathbf{REmb}(\mathbf{G})$ *is equivalent with* $\mathbf{RMon}(\mathbf{G}, \mathbf{Graph})$.

For example, $\mathbf{REmb}(\mathbf{AlgGraph})$ is equivalent to $\mathbf{RMon}(\mathbf{AlgGraph}, \mathbf{Graph})$. The reason why this equivalence is not an isomorphism is that there are many monos that correspond to a single graph embedding. Now let us call $\mathbf{D}$ *closed under* $\mathcal{M}$-pushouts/pullbacks where $\mathcal{M}$ is a suitable class of morphisms if, for every [co]span in $\mathbf{D}$ with one of the morphisms in $\mathcal{M}$, the corresponding $\mathbf{C}$-pushout object [$\mathbf{C}$-pullback object] is also in $\mathbf{D}$.

Theorem 26. *If* $\mathbf{C}$ *is an adhesive category,* $\mathbf{D}$ *is a full sub-category of* $\mathbf{C}$, $\mathcal{M}$ *is a suitable class of morphisms in* $\mathbf{D}$, *and* $\mathbf{D}$ *is closed under* $\mathcal{M}$*-pushouts/pullbacks, then* $\mathbf{D}$ *is adhesive HLR for the class* $\mathcal{M}$, *and* $\mathbf{RMon}(\mathbf{D}, \mathbf{C})$ *is adhesive HLR for the class* $\mathcal{N}$ *of all monomorphisms with inner arrow in* $\mathcal{M}$.

Proof. This follows from the fact that the constructions of the pushouts and pullbacks in $\mathbf{RMon}(\mathbf{C})$ entirely rely on the corresponding $\mathbf{C}$-constructions over the inner and outer parts of the objects and arrows. We have assumed $\mathbf{D}$ to be closed under these constructions, hence the resulting objects are in $\mathbf{RMon}(\mathbf{D}, \mathbf{C})$; moreover, $\mathcal{D}$ is a full sub-category, hence the constructed objects also satisfy the necessary universal properties. It follows that all required pushouts and pullbacks exist.

An application of this result is the following.

Corollary 27. *Let Σ be an arbitrary signature.*

1. $\mathbf{REmb}(\mathbf{AlgGraph}^{+}(\Sigma|\pi))$ *is adhesive.*
2. $\mathbf{REmb}(\mathbf{AlgGraph}(\Sigma|\pi))$ *is adhesive HLR for inner isomorphic monos.*

To also lift the "gluing over"-construction of Definition 14 to reflected monos, instead of just a sub-category $\mathbf{D}$, we need a functor $\mathcal{E}\colon \mathbf{D} \to \mathbf{RMon}(\mathbf{E}, \mathbf{D})$, with $\mathbf{E}$ a further full sub-category of $\mathbf{D}$, such that $\mathcal{E}(G)^{+} = G$ and $\mathcal{E}(f)^{+} = f$ for all objects G and arrows f of $\mathbf{D}$. We can then define $\mathbf{RMon}(\mathcal{E}, \mathbf{C})$ as the full sub-category of $\mathbf{RMon}(\mathbf{D}, \mathbf{C})$ with objects a such that the diagram $\bullet \overset{\epsilon(a)}{\to} \bullet \overset{a}{\to} \bullet$ has a pushout complement.

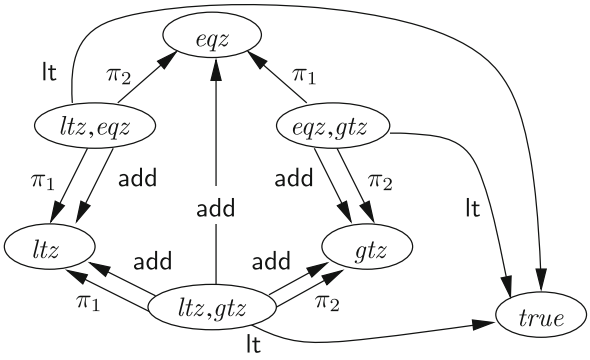


Fig. 3. Partial non-deterministic algebra graph for Prim of Example 2.

4 Data Abstraction

One of the most powerful analysis techniques for dynamic behaviour is *abstraction*. This involves discarding information from a model in order to make it more tractable, and over-approximating the original system by (where necessary) "guessing" what the discarded information may have been.

In a graph-based setting, a very natural kind of abstraction is obtained by taking a non-injective image of the start graph and applying the rules to that. The (standard) *embedding theorem* then implies that, under a certain consistency condition (Definition 30 below), all transformations on the original graph

can be applied to the abstract graph. (Other studies of abstraction for graph transformation are reported in [1,16,18].)

In this section, we show how *data abstraction*, i.e., where only the data domain and not the "proper" graph structure is abstracted, can be formulated in the framework of reflected monos. In this regard, our framework is more powerful than the standard attributed graph model, due to the ability to deal with non-determinism. The embedding theorem automatically holds due to adhesiveness; we show that this abstraction also automatically fulfills consistency, and that it is still valid in the presence of negative application conditions that only constrain the rim (i.e., the proper graph part).

Example 28. Figure 3 shows a partial abstract algebra graph G for $\mathsf{flat}(\mathsf{Prim})$, with Prim as in Example 2. There is a non-injective morphism h from the natural algebra graph H for $\mathsf{flat}(\mathsf{Prim})$ (partially displayed in Fig. 2) to G, with especially, for all $i \in N_H^{\mathsf{Int}}$,

$$h : i \mapsto \begin{cases} ltz & \text{if } i < 0 \\ eqz & \text{if } i = 0 \\ gtz & \text{if } i > 0. \end{cases}$$

As can be seen from Fig. 3, G is not deterministic: for instance, from the tuple element (ltz, gtz) there are three outgoing add-arrows, reflecting the fact that adding a negative to a positive number might give a negative, zero, or positive result.

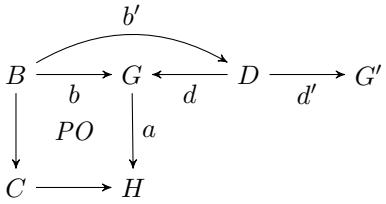
In contrast, the only non-injective algebra morphism from the natural algebra over Prim is to the point algebra, in which every sort has exactly one element. This abstraction loses all data distinctions and is therefore much too coarse for almost all uses.

Definition 29 (inner abstraction morphism). *An* inner abstraction morphism *is an arrow in* $\mathbf{RMon}(\mathbf{C})$ *that is a pushout in* $\mathbf{C}$.

As discussed in Sect. 3, an arrow in $\mathbf{RMon}(\mathbf{C})$ that is a pushout in $\mathbf{C}$ essentially does not modify the outer object — except to accommodate changes in the inner object.

To recall the embedding theorem, first we need the following *consistency condition.*

Definition 30 (consistency, cf. [4,6,12]). *A morphism* $a\colon G \to H$ *is called* consistent with *a span* $G \xleftarrow{d} D \xrightarrow{d'} G'$ *if a commuting diagram of the following shape exists:*



Intuitively, consistency comes down to the requirement that none of the items of G that are deleted by the span (meaning that they are not in d-image of D) are "modified" by a — where modification means (node or edge) merging or addition of incident edges. The embedding theorem refers to the derived span of a transformation sequence, which we will not formally define; however, in an adhesive HLR category with a class $\mathcal{M}$ of monos, the morphisms of derived spans are always in $\mathcal{M}$.

Theorem 31 (embedding, cf. [4,6,14]). *For any transformation $t : G_0 \stackrel{*}{\Longrightarrow} G_n$ and morphism $a_0 \colon G_0 \to H_0$ that is consistent with the derived span of t, there is a transformation $H_0 \stackrel{*}{\Longrightarrow} H_n$ consisting of the same rules as t, and a morphism $a_n \colon G_n \to H_n$.*

The following lemma implies a sufficient condition for consistency.

Lemma 32. *Let* **C** *be an adhesive category. If $G \stackrel{d}{\leftarrow} D \stackrel{d'}{\rightarrow} G'$ is a span of inner isomorphic monos and $a \colon G \to H$ is an inner abstraction in a category* **RMon**(**C**)*, then there is a diagram of the following shape, where e and a' are also inner abstractions:*

$$\begin{array}{ccccc} G & \xleftarrow{d} & D & \xrightarrow{d'} & G' \\ {\scriptstyle a}\downarrow & PO & {\scriptstyle e}\downarrow & PO & \downarrow{\scriptstyle a'} \\ H & \xleftarrow{c} & E & \xrightarrow{c'} & H' \end{array}$$

This means that, for categories where all the rule morphisms are inner isomorphic monos, inner abstractions are always consistent.

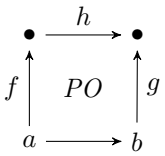
Corollary 33 (abstraction embedding). *Consider a sub-category of* **RMon** *which is adhesive HLR for a class $\mathcal{M}$ of inner isomorphisms. For any transformation $t : G_0 \stackrel{*}{\Longrightarrow} G_n$ and any inner abstraction $a_0 \colon G_0 \to H_0$, there is a transformation $H_0 \stackrel{*}{\Longrightarrow} H_n$ consisting of the same rules as t, with an inner abstraction $a_n \colon G_n \to H_n$.*

Negative application conditions. Negative application conditions (NACs) in combination with abstraction pose a problem: structures forbidden by a NAC may very well (appear to) exist on the abstract level, whereas they do not occur in the corresponding concrete graph. In general, to cope with this we can only "switch off" the evaluation of NACs on the abstract level; however, this makes the resulting over-approximation very coarse. The last result of this paper is to extend abstraction embedding to rules with NACs that do not constrain the inner objects. We first have to recall how NACs work.

Definition 34 (negative application condition). *A* negative application condition *is a morphism $n \colon L \to N$. n is said to be* satisfied *by a matching $m \colon L \to G$ if m does not factor through n, i.e., there is no $f \colon N \to G$ such that $m = f \circ n$.*

To avoid the problem of false positives after abstraction, it not enough to restrict the NACs to inner isomorphisms: they should also not introduce any new connections between the inner object and the rim. To formulate this as a general requirement, we have to assume that the base category has an initial object.

Definition 35. *Let* $\mathbf{C}$ *be an adhesive category with an initial object. A morphism* h *in* $\mathbf{RMon}(\mathbf{C})$ *is said to* avoid the inner object *if* h *is part of a pushout diagram of the following form, where* a *and* b *are closed objects (meaning that* a^- *and* b^- *are empty):*



The intuition is that a NAC avoids the inner object if it does not constrain the inner object itself, nor the glue between the inner object and the rim. If a NAC avoids the inner object, then inner abstractions do not cause false negatives.

Theorem 36. *Assume* $\mathbf{C}$ *is an adhesive category with an initial object; let* $n\colon L \to N$ *be a NAC in* $\mathbf{RMon}(\mathbf{C})$ *that avoids the inner object,* $m\colon L \to G$ *a matching, and* $a\colon G \to H$ *an inner abstraction. If* m *satisfies* n*, then* $a \circ m$ *satisfies* n*.*

It follows that Corollary 33 continues holding for rules with NACs that avoid the inner object.

5 Implementation

Here we show how the ideas exposed above have been partially implemented in the tool GROOVE (see [17]). GROOVE supports a basic signature Σ consisting of four sorts: whole (integer) numbers, floating point numbers, boolean and strings, with the typical operations found in programming languages.

As an example we take a graph transformation system that models the behaviour of an *indexed stack*, which is a stack modelled using an indexed list (rather than a linked list, as is more common for this particular data structure) for the elements. That is, elements on the stack have an *order*, which is 1 for the bottom element and increases for every next element on top of it. Figure 4 shows the graphs for an empty stack and a stack with three elements, using the natural algebra graphs for the sorts at hand (actually, in this example only integers). The node labels Stack and Cell are notational conventions for self-edges with those labels, which in practice serve as node types. The Stack-node has a length-edge to the number of elements currently contained in the stack; every Cell-node has an order-edge to its index. GROOVE supports single-pushout rules in general, but can also be restricted to double-pushout. Rules are thus spans of

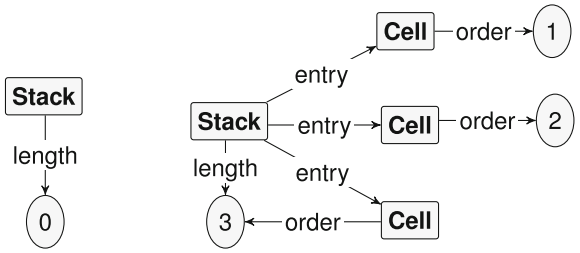


Fig. 4. Empty stack and 3-element stack

morphisms over rule graphs, in which the algebra subgraphs consist only of the constants from the signature and typed variable nodes for the four basic sorts; the values for the product sorts correspond to tuples of the above.

Typical operations on indexed stacks are pushing and popping elements. These are modelled by the rules shown in Fig. 5. The figure only shows the left hand side and right hand side graphs of both rules, leaving out the middle (interface) graph and suggesting the morphisms though the positioning of the nodes. For demonstration purposes, the *push* rule has been enriched with a condition that is satisfied only if the length of the stack is smaller than 5. The following graphical notational conventions are used:

- Only the relevant algebra graph nodes are shown in the figures. In particular, in host graphs, none of the auxiliary product nodes are ever included.
- Pure data nodes, i.e., elements of the data sets of the four basic sorts, are represented as ellipses labelled or by their values, by their types if they are variable nodes.
- Product nodes are represented as diamonds. The projection edges are labelled πi for index i starting at 0. The operator edges in Fig. 5 are add and lt in *push*, for addition and less-than, and sub in *pop* for subtraction.

In GROOVE, only part of the potential power of this paper's approach has been realised, in that non-deterministic algebra graphs such as the one in Fig. 3 are not supported. What is supported, on the other hand, are several (families) of algebras, namely

- *Point algebras*, where every value set consists of a single data value; i.e., all distinctions between data values are lost. If we interpret our indexed stacks under the point algebra, for instance, all order-edges point to the single integer representative, and rule *push* remains forever enabled because the lt-edge always points to the single Boolean value that represents both true and false.
- *Java algebras*, where every value set corresponds to its natural Java type, e.g., `int` for integers. This means that integer overflow is treated as Java does, by ignoring any significant bits above 31.
- *Big algebras*, where the most precise Java types available are chosen as value sets instead; e.g., `BigInteger` for integers.

– *Term algebras*, where every value set is given by the set of syntactic terms of the corresponding sort. Interpreted under the term algebra, for instance, rule *push* is not applicable to either of the graphs in Fig. 4, as in the term algebra graph the lt-edge leading from the tuple $\langle 0, 5 \rangle$ does not point to true but to the term $\mathsf{lt}(0, 5)$, which is (in that algebra) distinct from true.

6 Evaluation and Conclusion

In this paper we have proposed a new approach to model attributed graphs, which is more uniform than the standard model of [3] in that it stays entirely within a single (graph) category. Rather than resorting to a separate category of algebras to model the data, we encode the entire algebra structure into a sub-graph. This removes the need for additional algebraic equations specified outside the graph formalism and a corresponding satisfaction engine; thus, both tool implementers and users may benefit.

Contributions of the paper are:

– Equivalence of our model with the standard model (Corollary 20);
– An alternative proof of the adhesiveness of our construction (Theorem 26);
– Embedding theorems for data abstraction, without consistency condition (Corollary 33) and in the presence of negative application conditions (Theorem 36).

We have chosen a very common graph category in this paper: labelled binary graphs. The use of hyper-graphs instead would probably ease the encoding of the algebras. In particular, this would obviate the need for the product sorts, removing one important source of complexity. As a consequence, for instance, we would not have to flatten the signatures, and we would not have to resort to the "gluing over"-construction.

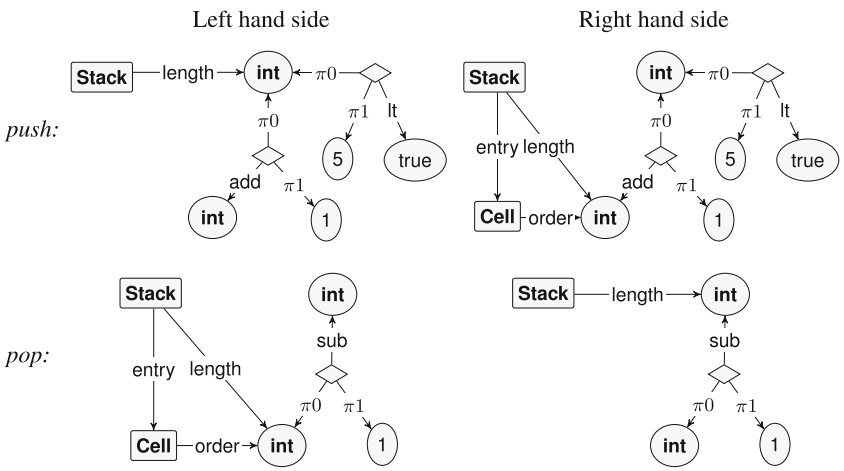


Fig. 5. Push and pop rules for the indexed stack

It should be noted that we have more or less silently restricted ourselves to node attributes. To support edge attributes as well, an extension of the standard notion of graph would be required in which (some) edges can have edges as their source, instead of nodes, just like in the standard model.

As we have briefly shown in Sect. 5, the setup described in this paper has been partially implemented in the tool GROOVE. It should be said, however, that the setup is not very appealing in terms of readability: for instance, already the fairly simple rules in Fig. 5 are non-trivial to read and write. In the newer versions of the tool, therefore, a lot of syntactic sugar has been added that allows the use of terms rather than product nodes, bringing it visually much closer to the standard model.

Related work. We have at several places referred to the "standard model" of representing attributes developed by Ehrig and al, but there are a number of other alternatives approaches. For instance, in the language *GP* for Graph Programs (e.g., [15]), attributes are encoded in labels: rules are able to compose and decompose such labels into their constituent values. In [2], the authors propose to associate exactly one attribute to every node and edge which may however be a tuple and so carry as many primitive values as one might wish. Morphisms have, apart from a structural backbone, a λ-term for each target graph element that expresses how its attribute is computed from the morphisms source. Refinements on the theme of adhesiveness that improve the way attributes fit have been studied and proposed in [6,14]. Another recent approach has been proposed in [13], using the *symbolic graphs* also studied in [12]. However, as the onderlying models are still algebras, and hence deterministic, we believe that symbolic graphs are not able to offer data abstraction in the sense of Sect. 4.

In related work of another type, an idea very similar to the one worked out in this paper has been used in [9] to extend a technique that was only available for graphs without attributes. This supports the point, made in the introduction, that there is a benefit to stick to the framework of graphs to encode the world of algebras.

Acknowledgement. For the proof of adhesiveness of **RMon**, we are very grateful for help from Andrea Corradini, Tobias Heindel, and Ulrike Prange.

References

1. Bauer, J., Wilhelm, R.: Static analysis of dynamic communication systems by partner abstraction. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_16
2. Boisvert, B., Féraud, L., Soloviev, S.: Typed lambda-terms in categorical attributed graph transformation. In: Durán, F., Rusu, V. (eds.) Algebraic Methods in Model-based Software Engineering (AMMSE). Electr. Notes Theor. Comput. Sci., vol. 56, pp. 33–47 (2011)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. Fund. Inf. **74**(1), 31–61 (2006)

4. 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
5. Ehrig, H., Padberg, J., Prange, U., Habel, A.: Adhesive high-level replacement systems: a new categorical framework for graph transformation. Fund. Inf. **74**(1), 1–29 (2006)
6. Golas, U.: A general attribution concept for models in $\mathcal{M}$-adhesive transformation systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 187–202. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_13
7. Kastenberg, H.: Towards attributed graphs in GROOVE: Work in progress. In: Heckel, R., König, B., Rensink, A. (eds.) Graph Transformation for Verification and Concurrency (GT-VC). Electr. Proc. Theor. Comput. Sci., vol. 154, pp. 47–54 (2006)
8. Kastenberg, H., Rensink, A.: Graph attribution through sub-graphs. CTIT Technical report TR-CTIT-12-27, Department of Computer Science, University of Twente (2012)
9. Kehrer, T., Alshanqiti, A., Heckel, R.: Automatic inference of rule-based specifications of complex in-place model transformations. In: Guerra, E., van den Brand, M. (eds.) ICMT 2017. LNCS, vol. 10374, pp. 92–107. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61473-1_7
10. König, B.: A general framework for types in graph rewriting. Acta Inf. **42**(4–5), 349–388 (2005)
11. Lack, S., Sobociński, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FoSSaCS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24727-2_20
12. Orejas, F.: Symbolic graphs for attributed graph constraints. J. Symb. Comput. **46**(3), 294–315 (2011)
13. Orejas, F., Lambers, L.: Symbolic attributed graphs for attributed graph transformation. In: Graph and Model Transformation (GraMoT). Electr. Comm. of the EASST., vol. 30 (2010)
14. Peuser, C., Habel, A.: Composition of m, n-adhesive categories with application to attribution of graphs. In: Plump, D. (ed.) Graph Computation Models (GCM). Electr. Comm. of the EASST, vol. 73 (2015)
15. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 128–143. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_11
16. Rensink, A.: Canonical graph shapes. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 401–415. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24725-8_28
17. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_40
18. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes Theor. Comput. Sci. **157**(1), 39–59 (2006)

On Normal Forms for Structured Specifications with Generating Constraints

Donald Sannella[1(✉)] and Andrzej Tarlecki[2]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, UK
dts@inf.ed.ac.uk
[2] Institute of Informatics, University of Warsaw, Warsaw, Poland
tarlecki@mimuw.edu.pl

Abstract. Hartmut Ehrig and others in [EWT83] studied normal form results for complex generating constraints imposed on basic specifications. Since then this work has been followed by subsequent results concerning normal forms for structured specifications, typically built from basic specifications using union, translation and hiding. We consider generating constraints as additional specification-building operations and follow and extend the results concerning normal forms for the resulting specifications with various forms of generating constraints.

1 Introduction

Hartmut Ehrig and others in [EWT83] studied normal form results for complex generating constraints imposed on basic specifications. Although from today's point of view the results were somewhat restricted in their generality, they spurred a line of work on normal forms of structured specifications, notably in [BHK90] and in the current general version in [Bor02], which turned out to be crucial in the study of proof systems for consequences of structured specifications, and in the analysis of completeness properties of such proof systems. But the more recent normal form results largely disregarded generating (or reachability) properties as imposed by the constraints studied in [EWT83]. Our aim here is to fill this gap, by generalising the results of [EWT83] as follows.

First, as has been standard since the introduction of institutions [GB84, GB92] to free algebraic specifications from dependency on a specific logical system, we abstract away from the specifics of the underlying logical system and present our results in the framework of a rather arbitrary logical system formalised as an institution with minimal extra structure and assumed properties.

Then, we consider a normal form of specifications that is somewhat more restrictive than the canonical constraints in [EWT83], giving a normal form result that is a bit sharper than the corresponding result in [EWT83].

This work has been partially supported by the (Polish) National Science Centre, grant 2013/11/B/ST6/01381 (AT).

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 266–284, 2018.
https://doi.org/10.1007/978-3-319-75396-6_15

Finally, and most crucially, Ehrig *et al.* [EWT83] studied generating constraints that impose generation requirements within models of a "flat" basic specification (presentation) only. We study generating constraints imposed by a more general specification building operation, which may be mixed with other specification-building operations in an arbitrary way, so that generating requirements may be imposed in multiple "layers" within models of an arbitrarily complex specification. This makes the study more delicate, and in fact the normal form result one might expect does not carry over to this more general case.

Dedication: *This study is dedicated to the memory of Hartmut Ehrig. We are grateful to Hartmut for his kindness and generosity over the years. We represented different "schools" of thought on algebraic specification, but Hartmut was always friendly and willing to explain his ideas and to try to understand our point of view.*

2 Constraints in the Standard Algebraic Framework

We begin with a summary of [EWT83].

As was usual at the time, the investigation was carried out in the context of the standard algebraic framework [EM85]. Specifications are *presentations* $\langle \Sigma, \Phi \rangle$ where Σ is a standard many-sorted signature and Φ is a set of Σ-axioms, usually equations. The category of Σ-algebras $\mathbf{Alg}(\Sigma)$ is defined as usual, with the notion of satisfaction between algebras and axioms yielding the obvious semantics of presentations: $[\![\langle \Sigma, \Phi \rangle]\!] = \{A \in |\mathbf{Alg}(\Sigma)| \mid A \models \Phi\}$.

The category of algebraic signatures **AlgSig** with standard signature morphisms $\sigma\colon \Sigma \to \Sigma'$ is cocomplete. For each signature morphism $\sigma\colon \Sigma \to \Sigma'$, we have a σ-reduct functor $U_\sigma\colon \mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$. A signature morphism $\sigma\colon \Sigma \to \Sigma'$ is a *presentation morphism* $\sigma\colon \langle \Sigma, \Phi \rangle \to \langle \Sigma', \Phi' \rangle$ if $U_\sigma([\![\langle \Sigma', \Phi' \rangle]\!]) \subseteq [\![\langle \Sigma, \Phi \rangle]\!]$. Colimits lift from the category of signatures to the category of presentations. The crucial satisfaction condition and amalgamation properties hold as expected—see Sect. 3 for more general formulations, or check for instance [ST12].

For each presentation $PRES$, the authors of [EWT83] introduce *constraints* on $PRES$, which are built from the empty constraint $\emptyset$, with semantics $[\![\emptyset]\!] = [\![PRES]\!]$, using *union*, with $[\![C_1 + C_2]\!] = [\![C_1]\!] \cap [\![C_2]\!]$, and the following constructors, for any presentation morphisms $\sigma\colon PRES' \to PRES$, $\sigma'\colon PRES \to PRES'$ and constraint C' on $PRES'$:

- translation TRA_σ: $[\![\mathrm{TRA}_\sigma(C')]\!] = \{A \in [\![PRES]\!] \mid U_\sigma(A) \in [\![C']\!]\}$
- reflection $\mathrm{REF}_{\sigma'}$: $[\![\mathrm{REF}_{\sigma'}(C')]\!] = \{U_{\sigma'}(A') \mid A' \in [\![C']\!]\}$
- generating constraint GEN_σ: $[\![\mathrm{GEN}_\sigma(C')]\!] = \{A \in [\![PRES]\!] \mid U_\sigma(A) \in [\![C']\!]$ and A is U_σ-generated in $[\![PRES]\!]\}$, where $A \in [\![PRES]\!]$ is *U_σ-generated in* $[\![PRES]\!]$ if in $[\![PRES]\!]$ there is no proper subalgebra of A with the same σ-reduct as A.

A constraint on *PRES* of the form $\mathrm{REF}_{\sigma_3}(\mathrm{TRA}_{\sigma_2}(\mathrm{GEN}_{\sigma_1}(\emptyset)))$ is called *canonical*. Such constraints might be easier to read in the diagrammatic notation of [EWT83]:

$$PRES_1 \xrightarrow[\mathrm{GEN}]{\sigma_1} PRES_2 \xrightarrow[\mathrm{TRA}]{\sigma_2} PRES_3 \xleftarrow[\mathrm{REF}]{\sigma_3} PRES$$

for presentations $PRES_i$ and presentation morphisms σ_i, $i = 1, 2, 3$.

The key result in [EWT83] is the following normal form theorem:

Theorem 2.1 ([EWT83]). *For each constraint on PRES an equivalent canonical constraint may be constructed.* □

We found the result very interesting and analysed it in detail already at the time [Tar83]. One observation was that the above form of canonical constraints is the only one possible for Theorem 2.1 to hold—that is, no other order of generating constraints, translation and reflection would work. Another was that some of the assumptions in [EWT83] are either misleading or unnecessary. For instance, the authors require reduct functors to lift isomorphisms, which in the standard algebraic framework excludes presentation morphisms that are not injective on sorts. Even if we could accept this as a reasonable restriction, it turns out that this property cannot be maintained under the constructions used in the proof of Theorem 2.1. Fortunately, such details did not prove to be crucial for the correctness of the proof, and the paper and the above theorem influenced subsequent developments, most notably the simpler normal form results for specifications without generating constraints in [BHK90, Bor02] and much work based in turn on those results.

3 Institutions with Model Inclusions

To capture in a very general way the concept of a submodel, we will require our model categories to come with *inclusions*.

A class of morphisms in a category is called a class of *inclusions* if it imposes a partial order on the objects of the category; to be precise, we require that

- all identities are inclusions
- inclusions are closed under composition
- between any two objects there is at most one inclusion (in either direction, i.e., for objects A, B, either there is no inclusion between A and B, or there is a unique inclusion from A to B, or from B to A, but not both unless A and B coincide).

In other words, a *category with inclusions* is a pair $\mathbb{C} = \langle \mathcal{C}, \mathcal{I} \rangle$ such that $\mathcal{C}$ is a category and $\mathcal{I}$ is a wide thin skeletal subcategory of $\mathcal{C}$; the morphisms of $\mathcal{I}$ are called *inclusions*. If there is an inclusion $\iota\colon A \to B$ then we say that A is a *subobject* of B and write $A \subseteq B$.

When no confusion may arise, we use the standard categorical terminology and notation in $\mathbb{C}$ to refer to the corresponding concepts in $\mathcal{C}$. So, for instance,

we write $|\mathbb{C}|$ for the class $|\mathcal{C}|$ of objects in $\mathcal{C}$, by a diagram in $\mathbb{C}$ we mean a diagram in $\mathcal{C}$, by (co)limits in $\mathbb{C}$ we mean (co)limits in $\mathcal{C}$, etc.

A functor between categories with inclusions $F\colon \langle \mathcal{C}, \mathcal{I}\rangle \to \langle \mathcal{C}', \mathcal{I}'\rangle$ is a functor $F\colon \mathcal{C} \to \mathcal{C}'$ that preserves inclusions, $F(\mathcal{I}) \subseteq \mathcal{I}'$. Clearly, such functors compose, and so this yields the (quasi-)category **ICat** of categories with inclusions.

These are extremely mild requirements concerning the subcategory of inclusions. For instance, in contrast to some other work, e.g. [DGS93, CR97, GR04, CMST17], we have no need to assume that inclusions form part of a (strict) factorisation system for $\mathcal{C}$, or that they admit intersections and unions, etc.

An *institution with model inclusions* **INS** consists of:

- a category $\mathbf{Sign}_{\mathbf{INS}}$ of *signatures*;
- a functor $\mathbf{Sen}_{\mathbf{INS}}\colon \mathbf{Sign}_{\mathbf{INS}} \to \mathbf{Set}$, giving a set $\mathbf{Sen}_{\mathbf{INS}}(\Sigma)$ of *Σ-sentences* for each signature $\Sigma \in |\mathbf{Sign}_{\mathbf{INS}}|$; and a function $\mathbf{Sen}_{\mathbf{INS}}(\sigma)\colon \mathbf{Sen}_{\mathbf{INS}}(\Sigma) \to \mathbf{Sen}_{\mathbf{INS}}(\Sigma')$ which translates Σ-sentences to Σ'-sentences for each signature morphism $\sigma\colon \Sigma \to \Sigma'$;
- a functor $\mathbf{Mod}_{\mathbf{INS}}\colon \mathbf{Sign}_{\mathbf{INS}}^{op} \to \mathbf{ICat}$, giving a category $\mathbf{Mod}_{\mathbf{INS}}(\Sigma)$ of *Σ-models with model inclusions* for each signature $\Sigma \in |\mathbf{Sign}_{\mathbf{INS}}|$; and a functor $\mathbf{Mod}_{\mathbf{INS}}(\sigma)\colon \mathbf{Mod}_{\mathbf{INS}}(\Sigma') \to \mathbf{Mod}_{\mathbf{INS}}(\Sigma)$ which translates Σ'-models to Σ-models and Σ'-morphisms to Σ-morphisms, preserving model inclusions, for each signature morphism $\sigma\colon \Sigma \to \Sigma'$; and
- a family $\langle \models_{\mathbf{INS},\Sigma} \subseteq |\mathbf{Mod}_{\mathbf{INS}}(\Sigma)| \times \mathbf{Sen}_{\mathbf{INS}}(\Sigma)\rangle_{\Sigma \in |\mathbf{Sign}_{\mathbf{INS}}|}$ of *satisfaction relations*

such that for any signature morphism $\sigma\colon \Sigma \to \Sigma'$ the translations $\mathbf{Mod}_{\mathbf{INS}}(\sigma)$ of models and $\mathbf{Sen}_{\mathbf{INS}}(\sigma)$ of sentences preserve the satisfaction relation, that is, for any $\varphi \in \mathbf{Sen}_{\mathbf{INS}}(\Sigma)$ and $M' \in |\mathbf{Mod}_{\mathbf{INS}}(\Sigma')|$ the following *satisfaction condition* holds:

$$M' \models_{\mathbf{INS},\Sigma'} \mathbf{Sen}_{\mathbf{INS}}(\sigma)(\varphi) \quad \text{iff} \quad \mathbf{Mod}_{\mathbf{INS}}(\sigma)(M') \models_{\mathbf{INS},\Sigma} \varphi$$

Note that institutions with model inclusions are not "inclusive institutions" in the sense of [DGS93, GR04, CMST17]: we require inclusion structure on models, not on signatures.

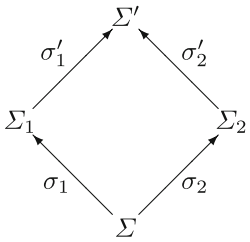
Examples of institutions with model inclusions abound. The institution **EQ** of equational logic has many-sorted algebraic signatures as signatures, many-sorted algebras as models with the usual notion of subalgebra determining the model inclusions and (explicitly quantified) equations as sentences. The institution **FOPEQ** of first-order predicate logic with equality has signatures that add predicate names to many-sorted algebraic signatures, models that extend algebras by interpreting predicate names as relations with inclusions that are required to preserve these relations, and sentences that are all closed (no free variables) formulae of first-order logic with equality. See [ST12] for detailed definitions of these and many other institutions, which often can be enriched with the obvious concept of a submodel, to yield institutions with model inclusions.

We will freely use standard terminology, and say that a Σ-model M *satisfies* a Σ-sentence φ, or that φ *holds* in M, whenever $M \models_{\mathbf{INS},\Sigma} \varphi$. We will omit the

subscript **INS**, writing $\mathbf{INS} = \langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|} \rangle$. Similarly, the subscript Σ on the satisfaction relations will often be omitted. For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, the translation function $\mathbf{Sen}(\sigma)\colon \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ will be denoted by $\sigma\colon \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$, the coimage function w.r.t. $\mathbf{Sen}(\sigma)$ by $\sigma^{-1}\colon \mathcal{P}(\mathbf{Sen}(\Sigma')) \to \mathcal{P}(\mathbf{Sen}(\Sigma))$, and the reduct functor $\mathbf{Mod}(\sigma)\colon \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ by $\_|_\sigma\colon \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$. Thus, the satisfaction condition may be re-stated as: $M' \models \sigma(\varphi)$ iff $M'|_\sigma \models \varphi$.

An institution with inclusions **INS** is *(finitely) exact* if its category **Sign** of signatures is (finitely) cocomplete, and the functor $\mathbf{Mod}\colon \mathbf{Sign}^{op} \to \mathbf{ICat}$ maps (finite) colimits of signatures to limits of model categories with inclusions in **ICat**. This adjusts the usual notion of exactness [ST12] to the framework where model inclusions are considered. In particular, the following stronger form of the amalgamation property [EM85] holds:

Lemma 3.1. *Given a finitely exact institution with model inclusions* **INS**, *consider a pushout of signatures*



Then, for any $M_1 \in |\mathbf{Mod}(\Sigma_1)|$ *and* $M_2 \in |\mathbf{Mod}(\Sigma_2)|$ *such that* $M_1|_{\sigma_1} = M_2|_{\sigma_2}$, *there exists a unique* $M' \in |\mathbf{Mod}(\Sigma')|$ *such that* $M'|_{\sigma'_1} = M_1$ *and* $M'|_{\sigma'_2} = M_2$. *Moreover, for any submodels* $N_1 \subseteq M_1$ *and* $N_2 \subseteq M_2$ *such that* $N_1|_{\sigma_1} = N_2|_{\sigma_2}$ *(hence* $N_1|_{\sigma_1} = N_2|_{\sigma_2} \subseteq M_1|_{\sigma_1} = M_2|_{\sigma_2}$*) the unique* $N' \in |\mathbf{Mod}(\Sigma')|$ *such that* $N'|_{\sigma'_1} = N_1$ *and* $N'|_{\sigma'_2} = N_2$ *is a submodel of* M', $N' \subseteq M'$. □

The standard institutions with model inclusions mentioned above (**EQ**, **FOPEQ**, etc.) are exact, hence enjoy the amalgamation property captured by Lemma 3.1.

Given a signature morphism $\sigma : \Sigma \to \Sigma'$ and class $\mathcal{M}' \subseteq |\mathbf{Mod}(\Sigma')|$ of models, we say that a model $M' \in |\mathbf{Mod}(\Sigma')|$ is *σ-generated in $\mathcal{M}'$* if $M' \in \mathcal{M}'$ and it has no proper submodels in $\mathcal{M}'$ with the same σ-reduct: for any submodel $M'' \subseteq M'$ if $M'' \in \mathcal{M}'$ and $M''|_\sigma = M'|_\sigma$ then $M'' = M'$. By taking $\mathcal{M}' = |\mathbf{Alg}(\Sigma')|$ we obtain the standard definition of σ-generated model, which requires M' to be generated by the set of all its elements in the carriers of $M'|_\sigma$. But note that when $\mathcal{M}' \subsetneq |\mathbf{Alg}(\Sigma')|$—in particular, when $\mathcal{M}'$ is not closed under submodels—models that are generated in $\mathcal{M}'$ need not be generated in the standard sense, exactly as in [EWT83]. For this reason a better terminology might be "minimal", as in [ST88], but we retain the terminology of [EWT83] to avoid confusion.

4 Structured Specifications in Institutions with Model Inclusions

Taking an institution as a starting point for talking about specifications, each signature Σ captures static information about the interface of a software system with each Σ-model representing a possible realisation of such a system, and with Σ-sentences used to describe properties that a realisation is required to satisfy. As a consequence, it is natural to regard the meaning of any specification SP built in an institution $\mathbf{INS} = \langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|} \rangle$ as given by its signature $Sig[SP] \in |\mathbf{Sign}|$ together with a class $Mod[SP]$ of $Sig[SP]$-models. Specifications SP with $Sig[SP] = \Sigma$ are referred to as *Σ-specifications*.

The semantics of specifications yields the obvious notion of specification equivalence: two specifications SP_1 and SP_2 are *equivalent*, written $SP_1 \equiv SP_2$, if $Sig[SP_1] = Sig[SP_2]$ and $Mod[SP_1] = Mod[SP_2]$.

Specifications we will consider are built from basic specifications (presentations in **INS**) using *specification-building operations* [ST12]. Specification formalisms differ in the choice of these operations, but typically all share a kernel introduced in ASL [SW83,ST88], where specifications are built from *basic specifications* using *union*, *translation*, and *hiding*. Following [EWT83], we add *generating constraints* to this repertoire, to capture constraints as studied there via a more general specification-building operation. We use a syntax inspired by that of CASL [BM04].

***basic specifications*:** For any signature $\Sigma \in |\mathbf{Sign}|$ and set $\Phi \subseteq \mathbf{Sen}(\Sigma)$ of Σ-sentences, a *basic specification* $\langle \Sigma, \Phi \rangle$ is a specification with:

$$Sig[\langle \Sigma, \Phi \rangle] = \Sigma$$
$$Mod[\langle \Sigma, \Phi \rangle] = \{M \in \mathbf{Mod}(\Sigma) \mid M \models \Phi\}$$

***union*:** For any signature $\Sigma \in |\mathbf{Sign}|$, given Σ-specifications SP_1 and SP_2, their *union* $SP_1 \cup SP_2$ is a specification with:

$$Sig[SP_1 \cup SP_2] = \Sigma$$
$$Mod[SP_1 \cup SP_2] = Mod[SP_1] \cap Mod[SP_2]$$

***translation*:** For any signature morphism $\sigma\colon \Sigma \to \Sigma'$ and Σ-specification SP, SP **with** σ is a specification with:

$$Sig[SP \textbf{ with } \sigma] = \Sigma'$$
$$Mod[SP \textbf{ with } \sigma] = \{M' \in |\mathbf{Mod}(\Sigma')| \mid M'|_\sigma \in Mod[SP]\}$$

***hiding*:** For any signature morphism $\sigma\colon \Sigma \to \Sigma'$ and Σ'-specification SP', SP' **hide via** σ is a specification with:

$$Sig[SP' \textbf{ hide via } \sigma] = \Sigma$$
$$Mod[SP' \textbf{ hide via } \sigma] = \{M'|_\sigma \mid M' \in Mod[SP']\}$$

***generating constraints*:** For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, Σ-specification SP and Σ'-specification SP', **generate by** σ **from** SP **in** SP' is a specification with:

$$Sig[\textbf{generate by } \sigma \textbf{ from } SP \textbf{ in } SP'] = \Sigma'$$
$$Mod[\textbf{generate by } \sigma \textbf{ from } SP \textbf{ in } SP'] =$$
$$\{M' \in |\mathbf{Mod}(\Sigma')| \mid M' \text{ is } \sigma\text{-generated in } Mod[SP'], M'|_\sigma \in Mod[SP]\}$$

The above specification-building operations may be arbitrarily combined to derive additional specification-building operations that capture some common patterns of their use. For instance:

***enrichment*:** For any specification SP and a set $\Phi \subseteq \mathbf{Sen}(Sig[SP])$ of sentences, SP **then** Φ abbreviates $SP \cup \langle Sig[SP], \Phi \rangle$.

5 Algebraic Properties of Specification-Building Operations

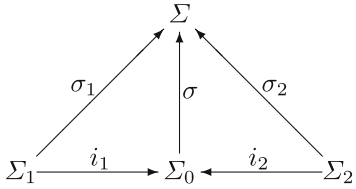
We start by recalling some easy facts concerning the standard specification-building operations [ST12]:

Proposition 5.1. *In any institution, under the obvious requirements on specification signatures, sentences and signature morphisms involved to ensure well-formedness of the specifications concerned:*

1. $\langle \Sigma, \Phi_1 \rangle \cup \langle \Sigma, \Phi_2 \rangle \equiv \langle \Sigma, \Phi_1 \cup \Phi_2 \rangle$
2. $\langle \Sigma, \Phi \rangle$ **with** $\sigma \colon \Sigma \to \Sigma' \equiv \langle \Sigma', \sigma(\Phi) \rangle$
3. SP **with** $id_{Sig[SP]} \equiv SP \equiv SP$ **hide via** $id_{Sig[SP]}$
4. (SP **with** σ) **with** $\sigma' \equiv SP$ **with** $\sigma;\sigma'$
5. (SP **hide via** σ) **hide via** $\sigma' \equiv SP$ **hide via** $\sigma;\sigma'$
6. $(SP \cup SP')$ **with** $\sigma \equiv (SP$ **with** $\sigma) \cup (SP'$ **with** $\sigma)$ □

The following easy fact follows directly from Proposition 5.1(4 and 6):

Proposition 5.2. *In any institution* **INS**, *consider the following commuting diagram of signatures:*

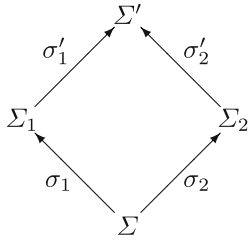


Then for any Σ_1-specification SP_1 and Σ_2-specification SP_2,

$(SP_1$ **with** $\sigma_1) \cup (SP_2$ **with** $\sigma_2) \equiv ((SP_1$ **with** $i_1) \cup (SP_2$ **with** $i_2))$ **with** σ

□

Proposition 5.3. *In any finitely exact institution* **INS**, *given a pushout of signatures*



1. $(SP_1$ **hide via** $\sigma_1)$ **with** $\sigma_2 \equiv (SP_1$ **with** $\sigma_1')$ **hide via** σ_2', *for any* Σ_1*-specification* SP_1, *and*
2. $(SP_1$ **hide via** $\sigma_1) \cup (SP_2$ **hide via** $\sigma_2) \equiv$
$((SP_1$ **with** $\sigma_1') \cup (SP_2$ **with** $\sigma_2'))$ **hide via** $\sigma_1;\sigma_1'$,
for any Σ_1*-specification* SP_1 *and* Σ_2*-specification* SP_2.

Proof. *See Propositions 5.6.5 and 5.6.7 in [ST12].* □

We can also derive algebraic properties for derived specification-building operations; for instance the following property follows directly from Proposition 5.1(6 and 2):

Proposition 5.4. *For any specification* SP, *set* $\Phi \subseteq$ **Sen**$(Sig[SP])$ *of sentences and signature morphism* $\sigma\colon Sig[SP] \to \Sigma'$,

$$(SP \textbf{ then } \Phi \textbf{ with } \sigma \equiv (SP \textbf{ with } \sigma) \textbf{ then } \sigma(\Phi)$$ □

In **generate by** σ **from** SP **in** SP' , both the "source" specification SP and the "target" specification SP' may be arbitrarily complex, built using any combination of specification-building operations, including generating constraints. This is considerably more general than the constraints considered in [EWT83], where in particular the "target" specification, within which we select the generated models, was taken to be a basic specification.

To begin with, let us note that the complexity of the source specification in a generating constraint may easily be removed. The source specification does not affect the generation property of the models within the target specification, and so it may be moved out of the scope of the constraint and imposed separately:

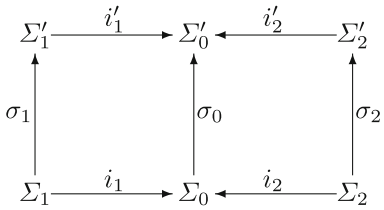
Proposition 5.5. *In any institution with model inclusions* **INS**, *for any signature morphism* $\sigma\colon \Sigma \to \Sigma'$, Σ*-specification* SP *and* Σ'*-specification* SP',

$$\textbf{generate by } \sigma \textbf{ from } SP \textbf{ in } SP' \equiv (\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } SP') \cup (SP \textbf{ with } \sigma)$$

Proof. *From the definition of the semantics of the specification-building operations involved.* □

The following property will be used to combine generating constraints:

Lemma 5.6. *Consider two constraints* **generate by** σ_1 **from** SP_1 **in** SP_1' *and* **generate by** σ_2 **from** SP_2 **in** SP_2' , *where* $Sig[SP_1] = \Sigma_1$, $Sig[SP_1'] = \Sigma_1'$, $Sig[SP_2] = \Sigma_2$, $Sig[SP_2'] = \Sigma_2'$. *Let* Σ_0 *be a coproduct of* Σ_1 *and* Σ_2 *with injections* $i_1\colon \Sigma_1 \to \Sigma_0$ *and* $i_2\colon \Sigma_2 \to \Sigma_0$, *and* Σ_0' *be a coproduct of* Σ_1' *and* Σ_2' *with injections* $i_1'\colon \Sigma_1' \to \Sigma_0'$ *and* $i_2'\colon \Sigma_2' \to \Sigma_0'$, *and let* $\sigma_0\colon \Sigma_0 \to \Sigma_0'$ *be the unique signature morphism such that* $i_1;\sigma_0 = \sigma_1;i_1'$ *and* $i_2;\sigma_0 = \sigma_2;i_2'$.



Then for any model $M' \in |\mathbf{Mod}(\Sigma'_0)|$, M' *is* σ_0*-generated in* $Mod[(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)]$ *iff both* $M'|_{i'_1}$ *is* σ_1*-generated in* $Mod[SP'_1]$ *and* $M'|_{i'_2}$ *is* σ_2*-generated in* $Mod[SP'_2]$.

Proof. For the "only if" part, consider a model $M' \in |\mathbf{Mod}(\Sigma'_0)|$ *such that* M' *is* σ_0*-generated in* $Mod[(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)]$. *Consider a submodel* $N'_1 \subseteq M'|_{i'_1}$ *such that* $N'_1 \in Mod[SP'_1]$ *and* $N'_1|_{\sigma_1} = (M'|_{i'_1})|_{\sigma_1}$. *Let* $N' \in |\mathbf{Mod}(\Sigma'_0)|$ *be (the unique model) such that* $N'|_{i'_1} = N'_1$ *and* $N'|_{i'_2} = M'|_{i'_2}$. *Then* $N' \subseteq M'$ *(by Lemma 3.1),* $N' \in Mod[(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)]$ *(from the definition of the semantics of structured specifications), and* $N'|_{\sigma_0} = M'|_{\sigma_0}$ *(since* $(N'|_{\sigma_0})|_{i_2} = (N'|_{i'_2})|_{\sigma_2} = (M'|_{i'_2})|_{\sigma_2} = (M'|_{\sigma_0})|_{i_2}$, *and* $(N'|_{\sigma_0})|_{i_1} = (N'|_{i'_1})|_{\sigma_1} = (M'|_{i'_1})|_{\sigma_1} = (M'|_{\sigma_0})|_{i_1}$). *Hence* $N' = M'$, *and so* $N'_1 = M'|_{i'_1}$, *which shows that* $M'|_{i'_1}$ *is* σ_1*-generated in* $Mod[SP'_1]$. *By symmetry,* $M'|_{i'_2}$ *is* σ_2*-generated in* $Mod[SP'_2]$.

For the opposite implication, suppose both $M'|_{i'_1}$ *is* σ_1*-generated in* $Mod[SP'_1]$ *and* $M'|_{i'_2}$ *is* σ_2*-generated in* $Mod[SP'_2]$. *Consider a submodel* $N' \subseteq M'$ *such that* $N' \in Mod[(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)]$ *and* $N'|_{\sigma_0} = M'|_{\sigma_0}$. *Then* $N'|_{i'_1} \in Mod[SP'_1]$ *is a submodel of* $M'|_{i'_1}$ *such that* $(N'|_{i'_1})|_{\sigma_1} = (N'|_{\sigma_0})|_{i_1} = (M'|_{\sigma_0})|_{i_1} = (M'|_{i'_1})|_{\sigma_1}$. *Therefore* $N'|_{i'_1} = M'|_{i'_1}$. *By symmetry,* $N'|_{i'_2} = M'|_{i'_2}$. *Hence* $N' = M'$, *which shows* M' *is indeed* σ_0*-generated in* $Mod[(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)]$. □

Corollary 5.7. *Under the notation of Lemma 5.6:*

$$\left(\begin{array}{l}((\textbf{generate by } \sigma_1 \textbf{ from } SP_1 \textbf{ in } SP'_1\ ) \textbf{ with } i'_1) \cup \\ \quad ((\textbf{generate by } \sigma_2 \textbf{ from } SP_2 \textbf{ in } SP'_2\ ) \textbf{ with } i'_2)\end{array}\right) \equiv$$
$$\textbf{generate by } \sigma_0 \textbf{ from } \left(\begin{array}{r}(SP_1 \textbf{ with } i_1) \cup \\ (SP_2 \textbf{ with } i_2)\end{array}\right) \textbf{ in } \left(\begin{array}{r}(SP'_1 \textbf{ with } i'_1) \cup \\ (SP'_2 \textbf{ with } i'_2)\end{array}\right)$$

Proof. Let SP_l *be the specification on the left-hand side of the equivalence, and let* SP_r *be the specification on its right-hand side.*

Consider $M'_0 \in Mod[SP_l]$. *Then* $M'_0|_{i'_1}$ *is* σ_1*-generated in* $Mod[SP'_1]$ *and* $M'_0|_{i'_2}$ *is* σ_2*-generated in* $Mod[SP'_2]$. *Hence, by Lemma 5.6,* M'_0 *is* σ_0*-generated in* $Mod[(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)]$. *Moreover,* $(M'_0|_{i'_1})|_{\sigma_1} = (M'_0|_{\sigma_0})|_{i_1} \in Mod[SP_1]$ *and* $(M'_0|_{i'_2})|_{\sigma_2} = (M'_0|_{\sigma_0})|_{i_2} \in Mod[SP_2]$, *hence we have* $M'_0|_{\sigma_0} \in Mod[(SP_1 \textbf{ with } i_1) \cup (SP_2 \textbf{ with } i_2)]$. *Thus,* $M'_0 \in Mod[SP_r]$.

Consider now $M'_0 \in Mod[SP_r]$. M'_0 *is* σ_0*-generated in* $Mod[(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)]$, *hence by Lemma 5.6,* $M'_0|_{i'_1}$ *is* σ_1*-generated in* $Mod[SP'_1]$ *and*

$M'_0|_{i'_2}$ *is* σ_2*-generated in* $Mod[SP'_2]$. *Moreover,* $M'_0|_{\sigma_0} \in Mod[(SP_1$ **with** $i_1) \cup (SP_2$ **with** $i_2)]$, *hence* $(M'_0|_{i'_1})|_{\sigma_1} = (M'_0|_{\sigma_0})|_{i_1} \in Mod[SP_1]$ *and* $(M'_0|_{i'_2})|_{\sigma_2} = (M'_0|_{\sigma_0})|_{i_2} \in Mod[SP_2]$. *Thus* $M'_0|_{i'_1} \in Mod[$**generate by** σ_1 **from** SP_1 **in** SP'_1 $]$ *and* $M'_0|_{i'_2} \in Mod[$**generate by** σ_2 **from** SP_2 **in** SP'_2 $]$, *which shows that* $M'_0 \in Mod[SP_l]$. □

6 Normal Form Results

We now come to the presentation of so-called normal-form results for structured specifications, whereby we show that all specifications of a certain kind are equivalent to a specification in a certain simple "normal form".

The first such result is easy:

Theorem 6.1. *In any institution* **INS**, *for every specification SP built from basic specifications using union and translation, there is an equivalent (basic) specification of the form* $\langle \Sigma, \Phi \rangle$ *(where* Φ *is finite provided the basic specifications involved in SP are finite).*

Proof. By induction on the structure of specifications, using Proposition 5.1 (1 and 2). □

Then, using Propositions 5.1 and 5.3, one can show the following normal form result by an easy induction on the structure of specifications:

Theorem 6.2. *Let* **INS** *be a finitely exact institution. For every specification SP built from flat specifications using union, translation and hiding, there is an equivalent specification of the form* $\langle \Sigma, \Phi \rangle$ **hide via** σ *(where* Φ *is finite provided the basic specifications involved in SP are finite).*

Proof. See Theorem 5.6.10 in [ST12]. □

The above key result may be derived from the normal form result for constraints in [EWT83] (see Theorem 2.1 above). It was given in a very similar form in [BHK90] for the standard institution of first-order logic, and then generalised in [Bor02] to an arbitrary exact institution. The result proved crucial for a number of further foundational developments, notably in the study of completeness of standard logical systems for proving consequences of structured specifications.

However, unlike the normal form result in [EWT83], Theorem 6.2 does not address specifications with generating constraints. The key problem is to reduce the complexity of the specifications involved in the generating constraints.

A generating constraint **generate by** σ **from** SP **in** SP' is *source-trivial* if SP is a basic specification with no axioms, i.e., is of the form $\langle Sig[SP], \emptyset \rangle$. A constraint **generate by** σ **from** SP **in** SP' is *basic* if SP' is a basic specification.

As already mentioned, the complexity of the source specifications in generating constraints is not a problem:

Corollary 6.3. *In any institution* **INS** *with model inclusions, any structured specification built from basic specifications using union, translation, hiding and generating constraints is equivalent to a specification built from basic specifications using union, translation, hiding and source-trivial generating constraints.*

Proof. Follows from Proposition 5.5 by induction on the structure of specifications. □

We are now ready for a direct generalisation of Theorem 2.1, the main normal form result in [EWT83].

As recalled in Sect. 2, the canonical constraints of [EWT83] are of the form:

$$\langle \Sigma_1, \Phi_1 \rangle \xrightarrow[\text{GEN}]{\sigma_1} \langle \Sigma_2, \Phi_2 \rangle \xrightarrow[\text{TRA}]{\sigma_2} \langle \Sigma_3, \Phi_3 \rangle \xleftarrow[\text{REF}]{\sigma_3} \langle \Sigma, \Phi \rangle$$

In terms of the specification-building operations introduced in Sect. 4, this may be written as follows:

$$((((\textbf{generate by } \sigma_1 \textbf{ from } \langle \Sigma_1, \Phi_1 \rangle \textbf{ in } \langle \Sigma_2, \Phi_2 \rangle \ ) \textbf{ with } \sigma_2) \textbf{ then } \Phi_3 \ ) \textbf{ hide via } \sigma_3) \textbf{ then } \Phi$$

with further requirements to ensure that the signature morphisms involved are in fact presentation morphisms. We will show below that the above form may be considerably simplified, by collecting all of the axioms involved in one place:

$$\langle \Sigma_1, \emptyset \rangle \xrightarrow[\text{GEN}]{\sigma_1} \langle \Sigma_2, \Phi_2 \rangle \xrightarrow[\text{TRA}]{\sigma_2} \langle \Sigma_3, \emptyset \rangle \xleftarrow[\text{REF}]{\sigma_3} \langle \Sigma, \emptyset \rangle$$

A specification is in *basic normal form* if it has the form:

$$((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi' \rangle \ ) \textbf{ with } \sigma') \textbf{ hide via } \delta$$

where the signatures and signature morphisms are as in the following diagram:

$$\Sigma \xrightarrow{\sigma} \Sigma' \xrightarrow{\sigma'} \Sigma'' \xleftarrow{\delta} \widehat{\Sigma}$$

This makes Theorem 6.4 below stronger, even in the standard algebraic framework, than Theorem 2.1.

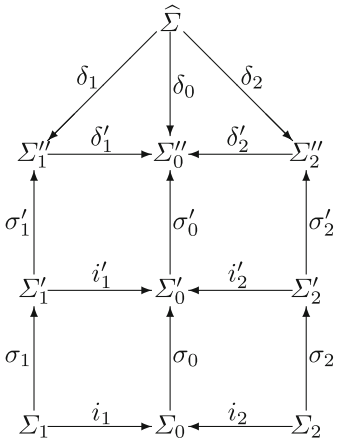
Theorem 6.4. *In any finitely exact institution* **INS** *with model inclusions, any structured specification built from basic specifications using union, translation, hiding and basic generating constraints is equivalent to a specification in basic normal form.*

Proof. First, by Corollary 6.3 (and Proposition 5.5) it is enough to consider structured specifications built from basic specifications using union, translation, hiding and source-trivial basic generating constraints. For those, we proceed by induction on the structure of specifications concerned, considering the last specification-building operation involved and assuming that its arguments, if any, are in basic normal form:

***basic specifications*:**

$\langle \Sigma, \Phi \rangle$
$\quad \equiv$ (*directly by the semantics*)
$((\textbf{generate by } id_\Sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma, \Phi \rangle\ )\ \textbf{with } id_\Sigma)\ \textbf{hide via } id_\Sigma$

***union*:** *The following diagram in* **Sign** *may help to follow the equivalences below:*



$(((\textbf{generate by } \sigma_1 \textbf{ from } \langle \Sigma_1, \emptyset \rangle \textbf{ in } \langle \Sigma_1', \Phi_1' \rangle\ )\ \textbf{with } \sigma_1')\ \textbf{hide via } \delta_1)$
$\cup$
$(((\textbf{generate by } \sigma_2 \textbf{ from } \langle \Sigma_2, \emptyset \rangle \textbf{ in } \langle \Sigma_2', \Phi_2' \rangle\ )\ \textbf{with } \sigma_2')\ \textbf{hide via } \delta_2)$

$\equiv$ (*by Proposition* 5.3(2), *taking a pushout* $\Sigma_1'' \xrightarrow{\delta_1'} \Sigma_0'' \xleftarrow{\delta_2'} \Sigma_2''$ *of the span* $\Sigma_1'' \xleftarrow{\delta_1} \widehat{\Sigma} \xrightarrow{\delta_2} \Sigma_2''$ *and* $\delta_0 = \delta_1;\delta_1' = \delta_2;\delta_2'$)

$$\left(\begin{array}{l}(((\textbf{generate by } \sigma_1 \textbf{ from } \langle \Sigma_1, \emptyset \rangle \textbf{ in } \langle \Sigma_1', \Phi_1' \rangle\ )\ \textbf{with } \sigma_1')\ \textbf{with } \delta_1') \\ \cup \\ (((\textbf{generate by } \sigma_2 \textbf{ from } \langle \Sigma_2, \emptyset \rangle \textbf{ in } \langle \Sigma_2', \Phi_2' \rangle\ )\ \textbf{with } \sigma_2')\ \textbf{with } \delta_2')\end{array}\right) \textbf{ hide via } \delta_0$$

$\equiv$ (*by Proposition 5.1(4)*)

$$\left(\begin{array}{l}((\textbf{generate by } \sigma_1 \textbf{ from } \langle \Sigma_1, \emptyset \rangle \textbf{ in } \langle \Sigma_1', \Phi_1' \rangle\ )\ \textbf{with } \sigma_1';\delta_1') \\ \cup \\ ((\textbf{generate by } \sigma_2 \textbf{ from } \langle \Sigma_2, \emptyset \rangle \textbf{ in } \langle \Sigma_2', \Phi_2' \rangle\ )\ \textbf{with } \sigma_2';\delta_2')\end{array}\right) \textbf{ hide via } \delta_0$$

$\equiv$ (*by Proposition* 5.2, *taking a coproduct* $\Sigma_1' \xrightarrow{i_1'} \Sigma_0' \xleftarrow{i_2'} \Sigma_2'$ *and* $\sigma_0' : \Sigma_0' \to \Sigma_0''$ *such that* $i_1';\sigma_0' = \sigma_1';\delta_1'$ *and* $i_2';\sigma_0' = \sigma_2';\delta_2'$)

$$\left(\left(\begin{array}{l}((\textbf{generate by } \sigma_1 \textbf{ from } \langle \Sigma_1, \emptyset \rangle \textbf{ in } \langle \Sigma_1', \Phi_1' \rangle\ )\ \textbf{with } i_1') \\ \cup \\ ((\textbf{generate by } \sigma_2 \textbf{ from } \langle \Sigma_2, \emptyset \rangle \textbf{ in } \langle \Sigma_2', \Phi_2' \rangle\ )\ \textbf{with } i_2')\end{array}\right) \textbf{ with } \sigma_0'\right) \textbf{ hide via } \delta_0$$

$\equiv$ (*by Corollary* 5.7, *taking a coproduct* $\Sigma_1 \xrightarrow{i_1} \Sigma_0 \xleftarrow{i_2} \Sigma_2$ *and* $\sigma_0 \colon \Sigma_0 \to \Sigma_0'$ *such that* $i_1;\sigma_0 = \sigma_1;i_1'$ *and* $i_2;\sigma_0 = \sigma_2;i_2'$, *and by Proposition* 5.1(2 *and* 1))

$$\big(\big(\textbf{generate by } \sigma_0 \textbf{ from } \langle \Sigma_0, \emptyset \rangle \textbf{ in } \langle \Sigma_0', i_1'(\Phi_1') \cup i_2'(\Phi_2') \rangle\ \big)\big)\ \textbf{with } \sigma_0')\ \textbf{hide via } \delta_0$$

***translation*:**

$$\begin{aligned}
&(((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi' \rangle\ ) \textbf{ with } \sigma')\ \textbf{hide via } \delta)\ \textbf{with } \tau \\
&\quad \equiv (\textit{by Proposition } 5.3(1), \textit{taking a pushout } \Sigma' \xrightarrow{\tau'} \bullet \xleftarrow{\delta'} \widehat{\Sigma'} \textit{ of the span} \\
&\qquad \Sigma' \xleftarrow{\delta} \widehat{\Sigma} \xrightarrow{\tau} \widehat{\Sigma'}) \\
&(((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi' \rangle\ ) \textbf{ with } \sigma')\ \textbf{with } \tau')\ \textbf{hide via } \delta' \\
&\quad \equiv (\textit{by Proposition } 5.1(4)) \\
&((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi' \rangle\ ) \textbf{ with } \sigma;\tau')\ \textbf{hide via } \delta'
\end{aligned}$$

***hiding*:**

$$\begin{aligned}
&(((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi' \rangle\ ) \textbf{ with } \sigma')\ \textbf{hide via } \delta)\ \textbf{hide via } \delta' \\
&\quad \equiv (\textit{by Proposition } 5.1(5)) \\
&((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi' \rangle\ ) \textbf{ with } \sigma')\ \textbf{hide via } \delta;\delta
\end{aligned}$$

***source-trivial basic generating constraints*:**

$$\begin{aligned}
&\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi' \rangle \\
&\quad \equiv (\textit{by Proposition } 5.1(3)) \\
&((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \Phi \rangle\ ) \textbf{ with } id_{\Sigma'})\ \textbf{hide via } id_{\Sigma'}
\end{aligned}$$

□

One might expect that Theorem 6.4 can be generalised to cover arbitrary specifications built from basic specifications using union, translation, hiding and (not necessarily basic) generating constraints. Unfortunately, in general this need not be the case, as the following counterexample shows.

Counterexample 6.5. Consider a very simple institution $\mathbf{INS}_0$ with three signatures $\Sigma_\emptyset$, Σ_1 and Σ_2, and signature morphisms $\sigma\colon \Sigma_\emptyset \to \Sigma_1$, $\delta\colon \Sigma_1 \to \Sigma_2$ (and identities and the composition $\sigma;\delta$). This defines the signature category, which is finitely cocomplete. Let $\mathbf{Mod}_0(\Sigma_\emptyset)$ be a singleton. Then let $\mathbf{Mod}_0(\Sigma_1)$ contain three distinct models K_1, N_1 and M_1 such that $K_1 \subseteq N_1 \subseteq M_1$. Let $\mathbf{Mod}_0(\Sigma_2)$ have two distinct models N_2 and M_2 with no inclusions other than identities. We also put $N_2|_\delta = N_1$ and $M_2|_\delta = M_1$. Finally, let $\mathbf{Sen}(\Sigma_\emptyset) = \mathbf{Sen}_0(\Sigma_1) = \mathbf{Sen}_0(\Sigma_2) = \emptyset$.

Consider

$$\textbf{generate by } \sigma \textbf{ from } \langle \Sigma_0, \emptyset \rangle \textbf{ in } (\langle \Sigma_2, \emptyset \rangle \textbf{ hide via } \delta)$$

Clearly, N_1 is the only model of the above specification. By a simple analysis of all possible cases, no Σ-specification in basic normal form has N_1 as the only model: $\{N_1\}$ cannot be the model class of a specification without generating constraints, since N_1 and M_1 cannot be distinguished in such a specification. All basic generating constraints here admit all models of their target specifications, except for the following one:

$$\textbf{generate by } \sigma \textbf{ from } \langle \Sigma_\emptyset, \emptyset \rangle \textbf{ in } \langle \Sigma_1, \emptyset \rangle$$

which does not have N_1 as a model, since it has a proper submodel K_1. Hence, no specification in basic normal form built on this constraint has N_1 as a model.

Unfortunately, the above construction is not quite right: the institution we defined does not satisfy our assumptions, since the model functor is not continuous. For instance, the coproduct of Σ_1 and Σ_2 is Σ_2, but the class of its models is not a product of the model classes of Σ_1 and Σ_2. The overall idea of the counterexample works, but the following more complex construction is needed.

Consider a category $\mathcal{C}_0$ with two objects Σ_1 and Σ_2 and a morphism $\delta\colon \Sigma_1 \to \Sigma_2$, with model categories, reduct functor, and sets of sentences defined as above.

Let the category of signatures in $\mathbf{INS}'_0$ be the category $\mathbf{Set}^{\to}$ of morphisms in **Set**, i.e. signatures now consist of two sets and a function between them, written $X_2 \xrightarrow{f} X_1$, and signature morphisms $\sigma\colon (X_2 \xrightarrow{f} X_1) \to (X'_2 \xrightarrow{f'} X'_1)$ are pairs of functions $\sigma_1\colon X_1 \to X'_1$ and $\sigma_2\colon X_2 \to X'_2$ such that $f;\sigma_1 = \sigma_2;f'$. It is well-known that $\mathbf{Set}^{\to}$ is cocomplete. In fact, this is a special case of the construction of a free cocomplete category generated by any category $\mathcal{C}$, given by the Yoneda embedding of $\mathcal{C}$ into the category $\mathbf{Set}^{\mathcal{C}^{op}}$ of presheaves on $\mathcal{C}$, see e.g. [Awo06] (Example 9.15 and Proposition 9.16). This also justifies the following choice of representation of the original signatures here: we identify Σ_1 with $\emptyset \to \{id_{\Sigma_1}\}$ and Σ_2 with $\{id_{\Sigma_2}\} \to \{\delta\}$, and δ with the unique morphism between them. We may write $\Sigma_\emptyset$ for the initial signature $\emptyset \to \emptyset$.

In $\mathbf{Set}^{\to}$, the signature $X_2 \xrightarrow{f} X_1$ is a colimit of a diagram with nodes $X_2 \uplus X_1$, where Σ_2 is the object in each node in X_2 and Σ_1 is the object in each node in X_1, with edges from $f(s)$ to s labelled by δ for each node $s \in X_2$. Now, extend the model functor so that the category $\mathbf{Mod}'_0(X_2 \xrightarrow{f} X_1)$ is the limit in **ICat** of the image of this diagram w.r.t. $\mathbf{Mod}_0$ (as defined above on Σ_1, Σ_2 and δ). This means in particular that a model P over a signature $X_2 \xrightarrow{f} X_1$ is a pair of functions $P_1\colon X_1 \to \{K_1, N_1, M_1\}$ and $P_2\colon X_2 \to \{N_2, M_2\}$ such that for $x_2 \in X_2$, $P_1(f(x_2)) = P_2(x_2)|_\delta$. Such a model P is a submodel of P' iff $P_2(x_2) = P'_2(x_2)$ for $x_2 \in X_2$, and $P_1(x_1) \subseteq P'_1(x_1)$ for $x_1 \in X_1$. Model reducts are given by (pre)composition with signature morphisms. Then for the sentence functor we set $\mathbf{Sen}_0(X_2 \xrightarrow{f} X_1) = \emptyset$.

Now, the institution so sketched is exact, and the counterexample works: the specification

$$\textbf{generate by } \sigma \textbf{ from } \langle \Sigma_\emptyset, \emptyset \rangle \textbf{ in } (\langle \Sigma_2, \emptyset \rangle \textbf{ hide via } \delta)$$

has no equivalent Σ_1-specification in basic normal form. To see this, just note that for any basic generating constraint C with $Sig[C] = (X_2 \xrightarrow{f} X_1)$, for any $P \in Mod[C]$, if for $x_1 \in (X_1 \setminus f(X_2))$, $P(x_1) = N_1$ then there are $P', P'' \in Mod[C]$ such that $P'(x_1) = K_1$ and $P''(x_1) = M_1$, and if for $x_2 \in X_2$, $P(x_2) = N_2$ then there is $P' \in Mod[C]$ such that $P'(x_2) = M_2$. Moreover, this property of specifications is preserved under translation w.r.t. any signature morphism. It follows then that no Σ_1-specification in basic normal form has N_1 (or rather, P such that $P_1(id_{\Sigma_1}) = N_1$) as its unique model. □

The source of the trouble is the use of hiding within the target specifications for generating constraints. One might suppose that when hiding is forbidden, the normal form result holds even if other specification-building operations are permitted within the target specifications used in generating constraints. Unfortunately, this is not the case: nested generating constraints may yield a similar effect as captured by Counterexample 6.5.

Counterexample 6.6. Consider a very simple institution $\mathbf{INS}_1$ with three signatures Σ, Σ_1 and Σ_2 and non-identity morphisms $\sigma_1 \colon \Sigma_1 \to \Sigma$ and $\sigma_2 \colon \Sigma_2 \to \Sigma$. $\mathbf{Mod}_1(\Sigma_1)$ has three distinct models K_1, LN_1 and M_1 with $K_1 \subseteq LN_1 \subseteq M_1$, $\mathbf{Mod}(\Sigma_2)$ has two distinct models KL_2 and NM_2 with $KL_2 \subseteq NM_2$, and $\mathbf{Mod}(\Sigma)$ has four distinct models K, L, N and M with $K \subseteq L \subseteq N \subseteq M$. We put $K|_{\sigma_1} = K_1$, $L|_{\sigma_1} = N|_{\sigma_1} = LN_1$ and $M|_{\sigma_1} = M_1$, and $K|_{\sigma_2} = L|_{\sigma_2} = KL_2$ and $N|_{\sigma_2} = M|_{\sigma_2} = NM_2$. Finally, we assume that there are no sentences in $\mathbf{INS}_1$, i.e., $\mathbf{Sen}_1(\Sigma) = \mathbf{Sen}_1(\Sigma_1) = \mathbf{Sen}_1(\Sigma_2) = \emptyset$.

Then $Mod[\mathbf{generate\ by}\ \sigma_1\ \mathbf{from}\ \langle \Sigma_1, \emptyset \rangle\ \mathbf{in}\ \langle \Sigma, \emptyset \rangle\ ] = \{K, L, M\}$, and so the constraint

$$\mathbf{generate\ by}\ \sigma_2\ \mathbf{from}\ \langle \Sigma_2, \emptyset \rangle\ \mathbf{in}\ (\mathbf{generate\ by}\ \sigma_1\ \mathbf{from}\ \langle \Sigma_1, \emptyset \rangle\ \mathbf{in}\ \langle \Sigma, \emptyset \rangle\ )$$

has K and M as its only models. It is easy to check though that no basic generating constraint, and no specification in basic normal form, has $\{K, M\}$ as its model class.

As in Counterexample 6.5, the above does not quite give a counterexample: the defined institution is not exact. Therefore, a construction of a new institution $\mathbf{INS}'_1$ analogous to that in Counterexample 6.5 has to be carried out.

Let $\mathcal{C}_1$ be the category with three signatures and morphisms as defined above. Take its free cocomplete closure via the Yoneda embedding into the category of presheaves over $\mathcal{C}_1$, $Y \colon \mathcal{C}_1 \to \mathbf{Set}^{\mathcal{C}_1^{op}}$. More explicitly, the resulting new category of signatures has objects of the form $X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2$, where X_1, X and X_2 are sets and $f_1 \colon X \to X_1$ and $f_2 \colon X \to X_2$ are functions. A morphism $h \colon (X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2) \to (X'_1 \xleftarrow{f'_1} X' \xrightarrow{f'_2} X'_2)$ consists of three functions $h_1 \colon X_1 \to X'_1$, $h_0 \colon X \to X'$ and $h_2 \colon X_2 \to X'_2$ such that $h_0; f'_1 = f_1; h_1$ and $h_0; f'_2 = f_2; h_2$. We identify Σ_1 with $\{id_{\Sigma_1}\} \leftarrow \emptyset \rightarrow \emptyset$, Σ_2 with $\emptyset \leftarrow \emptyset \rightarrow \{id_{\Sigma_2}\}$, Σ with $\{\sigma_1\} \leftarrow \{id_\Sigma\} \rightarrow \{\sigma_2\}$, and the morphisms $\sigma_1 \colon \Sigma_1 \to \Sigma$ and $\sigma_2 \colon \Sigma_2 \to \Sigma$ with unique morphisms between them. We do not add any sentences, so that the sentence functor $\mathbf{Sen}'_1$ yields the empty set on every signature.

A signature $X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2$ is a colimit of a diagram with nodes $X_1 \uplus X \uplus X_2$, where nodes in X_1 carry Σ_1, nodes in X carry Σ and nodes in X_2 carry Σ_2, and edges from $f_1(x)$ to x are labelled by σ_1 and from $f_2(x)$ to x are labelled by σ_2, for all $x \in X$. We define the model functor $\mathbf{Mod}'_1$ so that $\mathbf{Mod}'_1(X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2)$ is the limit in $\mathbf{ICat}$ of the image of this diagram under $\mathbf{Mod}_1$ (as defined above for $\mathcal{C}_1$). That is, any model $P \in |\mathbf{Mod}'_1(X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2)|$ consist of three functions $P_1 \colon X_1 \to \{K_1, LN_1, M_1\}$, $P_0 \colon X \to \{K, L, N, M\}$ and $P_2 \colon X_2 \to \{KL_2, NM_2\}$

such that for $x \in X$, $P_0(x)|_{\sigma_1} = P_1(f_1(x))$ and $P_0(x)|_{\sigma_2} = P_2(f_2(x))$. Such a model is a submodel of $P' \in |\mathbf{Mod}(X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2)|$ if for all $x_1 \in X_1$, $P_1(x_1) \subseteq P'_1(x_1)$, and similarly for X and X_2. Model reducts are given by (pre)composition with signature morphisms. For a class of models $\mathcal{P} \subseteq |\mathbf{Mod}'_1(X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2)|$ and $x \in X$, we write $\mathcal{P}(x) = \{P_0(x) \mid P \in \mathcal{P}\}$.

Consider now a signature morphism $h\colon (X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2) \to (X'_1 \xleftarrow{f'_1} X' \xrightarrow{f'_2} X'_2)$ and constraint $C' =$ **generate by** h **from** $\langle X_1 \xleftarrow{f_1} X \xrightarrow{f_2} X_2, \emptyset\rangle$ **in** $\langle X'_1 \xleftarrow{f'_1} X' \xrightarrow{f'_2} X'_2, \emptyset\rangle$. We analyse the class of the models of C'. Let $x' \in X'$.

- For some $x \in X$, $x' = h_0(x)$ (and so $f'_1(x') = h_1(f_1(x))$ and $f'_2(x') = h_2(f_2(x))$). Then $Mod[C'](x') = \{K, L, N, M\}$, since informally, the corresponding component of the signature morphism is the identity on this "occurrence" of Σ.
- x' is not in the image of h_0; then we have the following subcases.

- For some $x_1 \in X_1$ and $x_2 \in X_2$, $h_1(x_1) = f'_1(x')$ and $h_2(x_2) = f'_2(x')$. Then $Mod[C'](x') = \{K, L, N, M\}$, since informally, the corresponding component of the signature morphism is the map from the coproduct of Σ_1 and Σ_2 to this "occurrence" of Σ given by σ_1 and σ_2, and the reducts w.r.t. σ_1 and σ_2 do not jointly identify any models from $\{K, L, N, M\}$.
- For some $x_1 \in X_1$, $h_1(x_1) = f'_1(x')$ but $f'_2(x')$ is not in the image of h_2. Then $Mod[C'](x') = \{K, L, M\}$, since informally, the corresponding component of the signature morphism is σ_1, and the reduct w.r.t. σ_1 glues L and N together.
- For some $x_2 \in X_2$, $h_2(x_2) = f'_2(x')$ but $f'_1(x')$ is not in the image of h_1. Then $Mod[C'](x') = \{K, N\}$, since informally, the corresponding component of the signature morphism is σ_2, and the reduct w.r.t. σ_2 glues K and L as well as N and M together.
- Neither is $f'_1(x')$ in the image of h_1 nor is $f'_2(x')$ in the image of h_2. Then $Mod[C'](x') = \{K\}$, since informally, the corresponding component of the signature morphism is the unique morphism from the initial signature to Σ, and the reduct w.r.t. this morphism glues all models together.

Consequently, for any basic generation constraint C' as above, for $x' \in X'$, the class $Mod[C](x')$ is in the family $\mathcal{F} = \{\{K, L, N, M\}, \{K, L, M\}, \{K, N\}, \{K\}\}$. Moreover, this property is preserved under translation of specifications, since the family $\mathcal{F}$ is closed under intersection, and under hiding (reducts w.r.t. signature morphisms). Therefore, no specification in basic normal form may have $\{K, M\}$ is its class of models. □

The above counterexamples show that in general we cannot avoid nesting of structured specifications within generation constraints. We say that a specification is in *nested normal form* if either it is a basic specification, or it is built as follows:

$$((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset\rangle \textbf{ in } SP' \textbf{) with } \sigma') \textbf{ hide via } \delta$$

where SP' is a specification in nested normal form.

Corollary 6.7. *In any finitely exact institution* **INS** *with model inclusions, any structured specification built from basic specifications using union, translation, hiding and generating constraints is equivalent to a specification in nested normal form.*

Proof. *As in the proof of Theorem 6.4, we first use Corollary 6.3 to allow us to deal with source-trivial generating constraints only. Then the proof proceeds by double induction, on the maximal depth of nesting of generating constraints in the specifications, and then on the structure of specifications. When the depth of nesting is at most 1, the result follows by Theorem 6.4. Otherwise, we proceed by induction on the structure of specification, assuming the thesis for all specifications with a smaller depth of nesting of generating constraints. The case of basic specifications is trivial. The cases for translation and hiding follow much as in the proof of Theorem 6.4. For generating constraints, the thesis follows by the inductive assumption, since the specification used within the generating constraint has a smaller depth of nesting of generating constraints. For the case of union, we get by an argument analogous to that in the proof of Theorem 6.4 for the case of union:*

$$
\begin{array}{l}
(((\textbf{generate by } \sigma_1 \textbf{ from } \langle \Sigma_1, \emptyset \rangle \textbf{ in } SP'_1\ ) \textbf{ with } \sigma'_1) \textbf{ hide via } \delta_1) \\
\cup \\
(((\textbf{generate by } \sigma_2 \textbf{ from } \langle \Sigma_2, \emptyset \rangle \textbf{ in } SP'_2\ ) \textbf{ with } \sigma'_2) \textbf{ hide via } \delta_2) \\
\equiv \\
\big(\big((\textbf{generate by } \sigma_0 \textbf{ from } \langle \Sigma_0, \emptyset \rangle \textbf{ in } ((SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2))\ \big)\big) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{with } \sigma'_0)\ \textbf{hide via } \delta_0
\end{array}
$$

Now, the thesis follows by the inductive assumption, since the depth of nesting of generating constraints in $(SP'_1 \textbf{ with } i'_1) \cup (SP'_2 \textbf{ with } i'_2)$ *is lower than in the original specification.* □

7 Final Remarks

We started with the normal form result for constraints as studied in [EWT83] in the standard algebraic framework. We have shown that this result carries over to the more general setting of an arbitrary institution with some minimal extra structure: the notion of a submodel needed to capture the definition of generated model used in [EWT83]. Moreover, we sharpened the result somewhat via the use of a more restrictive definition of normal form.

We then considered the more general problem of normalising specifications where generating constraints are imposed in a class of models of an arbitrary specification, not just a presentation as in [EWT83]. Unfortunately, two counterexamples show that the normal form result does not carry over to this more general situation. Some nesting of generating constraints must be allowed, leading to a considerably weaker normal form result for this more general case.

The difficulties we encountered are linked to the definition of generated model in [EWT83], which we retained here. A standard alternative would be to free the concept of generated model from its dependency on the class of models of

the specification at hand, and consider generation in the class of all models over the given signature. In the standard algebraic framework this leads to the usual notion of generated algebra, where all elements are values of terms with variables taking values in the indicated carriers, with the usual connection to structural induction, as in CASL [BCH+04]. For specifications with generating constraints of this special form, by easy adaptation of Theorem 6.4 and its proof one can build an equivalent normal form of the following shape:

$$(((\textbf{generate by } \sigma \textbf{ from } \langle \Sigma, \emptyset \rangle \textbf{ in } \langle \Sigma', \emptyset \rangle \ ) \textbf{ then } \Phi \ ) \textbf{ with } \sigma') \textbf{ hide via } \delta$$

Restricting to this special case would considerably limit the power of generating constraints as considered here. For instance, in AI applications, McCarthy's notion of circumscription [McC80] used to impose a "closed world assumption" could not be captured in general, since no predicate ever holds in generated models over a first-order signature without any axioms or constraints imposed on the class of models considered. It is worth mentioning that a similarly general construct was introduced in DOL, the Distributed Ontology, Modeling and Specification Language [MCNK15].

One issue we did not touch on here at all is the development of proof systems for structured specifications. This is well-studied in the context of specifications built from basic specifications using union, translation and hiding, with a standard compositional proof system for consequences of specifications given in the framework of an arbitrary institution already in [ST88]. Completeness results follow under additional assumptions about the institution (most notably, interpolation is needed) where the proof of completeness heavily relies on the normal form result [Bor02]. It is well-known that once generating constraints are added, there is no hope for completeness [MS85]. However, in [ST14] we showed that the compositional proof system for structured specification built from basic specifications using union, translation and hiding is the best sound compositional proof system possible. It would be interesting to see how to carry this over to specifications with generating constraints, with some sound approximate techniques for proving consequences of generating constraints. Perhaps the normal form results studied here could be used to "concentrate" the necessary incompleteness at specific points in the structure of specifications, linked to the use of generating constraints in the normal forms.

Acknowledgements. Thanks to the anonymous referees for their constructive comments.

References

[Awo06] Awodey, S.: Category Theory. Oxford University Press, New York (2006)

[BCH+04] Baumeister, H., Cerioli, M., Haxthausen, A., Mossakowski, T., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL semantics. In: [Mos04] (2004)

[BHK90] Bergstra, J.A., Heering, J., Klint, P.: Module algebra. J. Assoc. Comput. Mach. **37**(2), 335–372 (1990)

[BM04] Bidoit, M., Mosses, P.D. (eds.): CASL User Manual. LNCS, vol. 2900. Springer, Heidelberg (2004). https://doi.org/10.1007/b11968. http://www.informatik.uni-bremen.de/cofi/index.php/CASL

[Bor02] Borzyszkowski, T.: Logical systems for structured specifications. Theor. Comput. Sci. **286**(2), 197–245 (2002)

[CMST17] Codescu, M., Mossakowski, T., Sannella, D., Tarlecki, A.: Specification refinements: calculi, tools, and applications. Sci. Comput. Program. **144**, 1–49 (2017)

[CR97] Căzănescu, V.E., Roşu, G.: Weak inclusion systems. Math. Struct. Comput. Sci. **7**(2), 195–206 (1997)

[DGS93] Diaconescu, R., Goguen, J.A., Stefaneas, P.: Logical support for modularisation. In: Huet, G., Plotkin, G. (eds.) Logical Environments, pp. 83–130. Cambridge University Press, Cambridge (1993)

[EM85] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69962-7

[EWT83] Ehrig, H., Wagner, E.G., Thatcher, J.W.: Algebraic specifications with generating constraints. In: Diaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 188–202. Springer, Heidelberg (1983). https://doi.org/10.1007/BFb0036909

[GB84] Goguen, J.A., Burstall, R.M.: Introducing institutions. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 221–256. Springer, Heidelberg (1984). https://doi.org/10.1007/3-540-12896-4_366. Many revised versions were widely circulated, with [GB92] as the endpoint

[GB92] Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. J. Assoc. Comput. Mach. **39**(1), 95–146 (1992)

[GR04] Goguen, J.A., Roşu, G.: Composing hidden information modules over inclusive institutions. In: Owe, O., Krogdahl, S., Lyche, T. (eds.) From Object-Orientation to Formal Methods. LNCS, vol. 2635, pp. 96–123. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-39993-3_7

[McC80] McCarthy, J.: Circumscription – a form of non-monotonic reasoning. Artif. Intell. **13**(1–2), 27–39 (1980)

[MCNK15] Mossakowski, T., Codescu, M., Neuhaus, F., Kutz, O.: The distributed ontology, modeling and specification language – DOL. In: Koslow, A., Buchsbaum, A. (eds.) The Road to Universal Logic. SUL, pp. 489–520. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15368-1_21

[Mos04] Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004). https://doi.org/10.1007/b96103

[MS85] MacQueen, D., Sannella, D.: Completeness of proof systems for equational specifications. IEEE Trans. Softw. Eng. **SE–11**(5), 454–461 (1985)

[ST88] Sannella, D., Tarlecki, A.: Specifications in an arbitrary institution. Inf. Comput. **76**(2–3), 165–210 (1988)

[ST12] Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2012)

[ST14] Sannella, D., Tarlecki, A.: Property-oriented semantics of structured specifications. Math. Struct. Comput. Sci. **24**(2), e240205 (2014)

[SW83] Sannella, D., Wirsing, M.: A kernel language for algebraic specification and implementation: extended abstract. In: Karpinski, M. (ed.) FCT 1983. LNCS, vol. 158, pp. 413–427. Springer, Heidelberg (1983). https://doi.org/10.1007/3-540-12689-9_122

[Tar83] Tarlecki, A.: Remarks on [EWT83]. Unpublished note, Department of Computer Science, University of Edinburgh (1983)

Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models

Dániel Varró[1,2,3], Oszkár Semeráth[1,2(✉)], Gábor Szárnyas[1,2], and Ákos Horváth[1,4]

[1] Budapest University of Technology and Economics, Budapest, Hungary
{varro,semerath,szarnyas,ahorvath}@mit.bme.hu
[2] MTA-BME Lendület Research Group on Cyber-Physical Systems, Budapest, Hungary
[3] Department of Electrical and Computer Engineering, McGill University, Montreal, Canada
[4] IncQuery Labs Ltd., Budapest, Hungary

Abstract. Automated model generation can be highly beneficial for various application scenarios including software tool certification, validation of cyber-physical systems or benchmarking graph databases to avoid tedious manual synthesis of models. In the paper, we present a long-term research challenge how to generate graph models specific to a domain which are consistent, diverse, scalable and realistic at the same time.

We provide foundations for a class of model generators along a refinement relation which operates over partial models with 3-valued representation and ensures that subsequently derived partial models preserve the truth evaluation of well-formedness constraints in the domain. We formally prove *completeness*, i.e. any finite instance model of a domain can be generated by model generator transformations in finite steps and *soundness*, i.e. any instance model retrieved as a solution satisfies all well-formedness constraints. An experimental evaluation is carried out in the context of a statechart modeling tool to evaluate the trade-off between different characteristics of model generators.

Keywords: Automated model generation · Partial models Refinement

1 Introduction

Smart and safe cyber-physical systems [16,54,69,93] are software-intensive autonomous systems that largely depend on the context in which they operate, and frequently rely upon intelligent algorithms to adapt to new contexts on-the-fly. However, adaptive techniques are currently avoided in many safety-critical systems due to major certification issues. Automated synthesis of prototypical

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 285–312, 2018.
https://doi.org/10.1007/978-3-319-75396-6_16

test contexts [58] aims to systematically derive previously unanticipated contexts for assurance of such smart systems in the form of graph models. Such prototype contexts need to be *consistent*, i.e. they need to fulfill certain well-formedness (consistency) constraints when synthesizing large and realistic environments.

In many design and verification tools used for engineering CPSs, system models are frequently represented as typed and attributed graphs. There has been an increasing interest in model generators to be used for validating, testing or benchmarking design tools with advanced support for queries and transformations [4,6,42,92]. Qualification of design and verification tools is necessitated by safety standards (like DO-178C [89], or ISO 26262 [43]) in order to assure that their output results can be trusted in safety-critical applications. However, tool qualification is extremely costly due to the lack of effective best practices for validating the design tools themselves. Additionally, design-space exploration [47,57,66] necessitates to automatically derive different solution candidates which are optimal w.r.t. certain objectives for complex allocation problems. For testing and DSE purposes, *diverse* models need to be synthesized where any pairs of models are structurally very different from each other in order to achieve high coverage or a diverse solution space.

Outside the systems engineering domain, many performance benchmarks for advanced relational databases [26], triple stores and graph databases [13,60,80], or biochemical applications [36,99] also rely on the availability of extremely large and *scalable* generators of graph models.

Since real models created by engineers are frequently unavailable due to the protection of intellectual property rights, there is an increasing need of *realistic* models which have similar characteristics to real models. However, these models should be domain-specific, i.e. graphs of biomedical systems are expected to be very different from graphs of social networks or software models. An engineer can easily distinguish an auto-generated model from a manually designed model by inspecting key attributes (e.g. names), but the same task becomes more challenging if we abstract from all attributes and inspect only the (typed) graph structure. While several graph metrics have been proposed [10,12,44,68], the characterization of *realistic* models is a major challenge [91].

As a long-term research challenge, we aim at automatically generating domain-specific graph models which are simultaneously scalable, realistic, consistent and diverse. In the paper, we precisely formulate the model generation challenge for the first time (Sect. 2). Then in Sect. 3, we revisit the formal foundations of partial models and well-formedness constraints captured by graph patterns. In Sect. 4, we propose a refinement calculus for partial models as theoretical foundation for graph model generation, and a set of specific refinement operations as novel contributions. Moreover, we precisely formulate certain soundness and completeness properties of this refinement calculus.[1] In addition, we carry out an experimental evaluation of some existing techniques and tools in Sect. 5

[1] The authors' copy of this paper is available at https://inf.mit.bme.hu/research/publications/towards-model-generation together with the proofs of theorems presented in Sect. 4.

to assess the trade-off between different characteristics (e.g. diverse vs. realistic, consistent vs. diverse, diverse vs. consistent and consistent vs. scalable) of model generation. Finally, related work is discussed w.r.t. the different properties required for model generation in Sect. 6.

2 The Graph Model Generation Challenge

A domain specification (or domain-specific language, DSL) is defined by a *metamodel MM* which captures the main concepts and relations in a domain, and specifies the basic graph structure of the models. In addition, a set of *well-formedness constraints* $WF = \{\phi_1, \ldots, \phi_n\}$ may further restrict valid domain models by extra structural restrictions. Furthermore, we assume that *editing operations* of the domain are also defined by a set of rules OP.

Informally, the *automated model generation challenge* is to derive a set of instance models where each M_i conforms to a metamodel MM. A model generator $Gen \mapsto \{M_i\}$ derives a set (or sequence) of models along a derivation sequence $M_0 \xrightarrow{op_1,\ldots,op_k} M_i$ starting from (a potentially empty) initial model M_0 by applying some operations op_j from OP at each step. Ideally, a single model M_i or a model generator Gen should satisfy the following requirements:

- **Consistent (CON)**: A model M_i is consistent if it satisfies all constraints in WF (denoted by $M_i \models WF$). A model generator Gen is consistent, if it is sound (i.e. if a model is derivable then it is consistent) and complete (i.e. all consistent models can be derived).
- **Diverse (DIV)**: The diversity of a model M_i is defined as the number of (direct) types used from its MM: M_i is more diverse than M_j if more types of MM are used in M_i than in M_j. A model generator Gen is diverse if there is a designated distance between each pairs of models M_i and M_j: $dist(M_i, M_j) > D$.
- **Scalable (SCA)**: A model generator Gen is scalable *in size* if the size of M_i is increasing exponentially $\#(M_{i+1}) \geq 2 \cdot \#(M_i)$, thus a single model M_i can be larger than a designated model size $\#(M_i) > S$. A model generator Gen is scalable *in quantity* if the generation of M_j (of similar size) does not take significantly longer than the generation of any previous model M_i: $time(M_j) < max_{0 \leq i < j}\{time(M_i)\} \cdot T$ (for some constant T).
- **Realistic (REA)**: A generated model is (structurally) realistic if it cannot be distinguished from the structure of a real model after all text and values are removed (by considering them irrelevant). A model generator is realistic w.r.t. some graph metrics [91] and a set of real models $\{RM_i\}$ if the evaluation of the metrics for the real and the generated set of models has similar values: $|metr(\{RM_i\}) - metr(\{M_i\})| < R$.

Note that we intentionally leave some metrics *metr* and distance functions *dist* open in the current paper as their precise definitions may either be domain-specific or there are no guidelines which ones are beneficial in practice.

Each property above is interesting in itself, i.e. it has been addressed in numerous papers, and used in at least one industrial application scenario. Moreover, similar properties might be defined in the future. However, the grand challenge is to develop an automated model generator which simultaneously satisfies multiple (ideally, all four) properties. For instance, a model generator for benchmarking purposes needs to be scalable, realistic and consistent, while a test model generator needs to be diverse, consistent (or intentionally faulty), and scalable in quantity. However, existing model generation approaches developed in different research areas usually support one (or rarely at most two) of these properties.

Such a multi-purpose model generator is out of scope also for the current paper. In fact, as a novel contribution, we provide precise theoretical foundations for a graph model generator that is scalable and consistent based on a refinement calculus. Our specific focus is motivated by a novel empirical evaluation to be reported in Sect. 5 which states that consistency is a prerequisite for the synthesis of both diverse and realistic models.

3 Preliminaries

We illustrate automated model generation in the context of Yakindu Statecharts Tools [101], which is an industrial DSL developed by Itemis AG for the development of reactive, event-driven systems using statecharts captured in a combined graphical and textual syntax. Yakindu supports validation of WF constraints, simulation and code generation from statechart models. We first revisit the formalization of the partial models and WF-constraints as defined in [85].

3.1 Metamodels and Instance Models

Formally, a *metamodel* defines a vocabulary $\Sigma = \{\mathtt{C}_1, \ldots, \mathtt{C}_n, \mathtt{R}_1, \ldots, \mathtt{R}_m, \sim\}$ where a unary predicate symbol $\mathtt{C}_i$ $(1 \leq i \leq n)$ is defined for each class (node type), and a binary predicate symbol $\mathtt{R}_j$ $(1 \leq j \leq m)$ is defined for each reference (edge type). The index of a predicate symbol refers to the corresponding metamodel element. The binary $\sim$ predicate is defined as an equivalence relation over objects (nodes) to denote if two objects can be merged. For space considerations, we omit the precise handling of attributes from this paper as none of the four key properties depend on attributes. For metamodels, we use the notations of the Eclipse Modeling Framework (EMF) [90], but our concepts could easily be adapted to other frameworks of typed and attributed graphs such as [21,28].

An *instance model* is a 2-valued logic structure $M = \langle Obj_M, \mathcal{I}_M \rangle$ over Σ where $Obj_M = \{o_1, \ldots, o_n\}$ $(n \in \mathbb{Z}^+)$ is a finite set of individuals (objects) in the model (where $\#(M) = |Obj_M| = n$ denotes the size of the model) and $\mathcal{I}_M$ is a 2-valued interpretation of predicate symbols in Σ defined as follows (where o_k and o_l are objects from Obj_M with $1 \leq k, l \leq n$):

- **Type predicates:** the 2-valued interpretation of a predicate symbol $\mathtt{C}_i$ in M (formally, $\mathcal{I}_M(\mathtt{C}_i) : Obj_M \to \{1, 0\}$) evaluates to 1 if object o_k is instance of class $\mathtt{C}_i$ (denoted by $[\![\mathtt{C}_i(o_k)]\!]^M = 1$), and evaluates to 0 otherwise.

- **Reference predicates:** the 2-valued interpretation of a predicate symbol $\mathtt{R}_j$ in M (formally, $\mathcal{I}_M(\mathtt{R}_j) : Obj_M \times Obj_M \rightarrow \{1,0\}$) evaluates to 1 if there exists an edge (link) of type $\mathtt{R}_j$ from o_k to o_l in M denoted as $[\![\mathtt{R}_j(o_k,o_l)]\!]^M = 1$, and evaluates to 0 otherwise.
- **Equivalence predicate:** the 2-valued interpretation of a predicate symbol $\sim$ in M (formally, $\mathcal{I}_M(\sim) : Obj_M \times Obj_M \rightarrow \{1,0\}$) evaluates to 1 for any object o_k, i.e. $[\![o_k \sim o_k]\!]^M = 1$, and evaluates to 0 for any different pairs of objects, i.e. $[\![o_k \sim o_l]\!]^M = 0$, if $o_k \neq o_l$. This equivalence predicate is rather trivial for instance models but it will be more relevant for partial models.

3.2 Partial Models

Partial models [31,46] represent uncertain (possible) elements in instance models, where one partial model represents a set of concrete instance models. In this paper, 3-valued logic [48] is used to explicitly represent unspecified or unknown properties of graph models with a third 1/2 value (beside 1 and 0 which stand for *true* and *false*) in accordance with [76,85].

A *partial model* is a 3-valued logic structure $P = \langle Obj_P, \mathcal{I}_P \rangle$ of Σ where $Obj_P = \{o_1, \ldots, o_n\}$ $(n \in \mathbb{Z}^+)$ is a finite set of individuals (objects) in the model, and $\mathcal{I}_P$ is a 3-valued interpretation for all predicate symbols in Σ defined below. The 3-valued truth evaluation of the predicates in a partial model P will be denoted respectively as $[\![\mathtt{C}_i(o_k)]\!]^P$, $[\![\mathtt{R}_j(o_k,o_l)]\!]^P$, $[\![o_k \sim o_l]\!]^P$.

- **Type predicates:** $\mathcal{I}_P$ gives a 3-valued interpretation for each class symbol $\mathtt{C}_i$ in Σ: $\mathcal{I}_P(\mathtt{C}_i) : Obj_P \rightarrow \{1, 0, 1/2\}$, where 1, 0 and 1/2 means that it is true, false or unspecified whether an object is an instance of a class $\mathtt{C}_i$.
- **Reference predicates:** $\mathcal{I}_P$ gives a 3-valued interpretation for each reference symbol $\mathtt{R}_j$ in Σ: $\mathcal{I}_P(\mathtt{R}_j) : Obj_P \times Obj_P \rightarrow \{1, 0, 1/2\}$, where 1, 0 and 1/2 means that it is true, false or unspecified whether there is a reference of type $\mathtt{R}_j$ between two objects.
- **Equivalence predicate:** $\mathcal{I}_P$ gives a 3-valued interpretation for the $\sim$ relation between the objects $\mathcal{I}_P(\sim) : Obj_P \times Obj_P \rightarrow \{1, 0, 1/2\}$.
  A predicate $o_k \sim o_l$ between two objects o_k and o_l is interpreted as follows:
  - If $[\![o_k \sim o_l]\!]^P = 1$ then o_k and o_l are equal and they can be merged;
  - If $[\![o_k \sim o_l]\!]^P = 1/2$ then o_k and o_l may be equal and may be merged;
  - If $[\![o_k \sim o_l]\!]^P = 0$ then o_k and o_l are different objects in the instance model, thus they cannot be merged.

  A predicate $o_k \sim o_k$ for any object o_k (as a self-edge) means the following:
  - If $[\![o_k \sim o_k]\!]^P = 1$ then o_k is a final object which cannot be further split to multiple objects;
  - If $[\![o_k \sim o_k]\!]^P = 1/2$ then o_k is a multi-object which may represent a set of objects.

  The traditional properties of the equivalence relation $\sim$ are interpreted as:
  - $\sim$ is a symmetric relation: $[\![o_k \sim o_l]\!]^P = [\![o_l \sim o_k]\!]^P$;
  - $\sim$ is a reflexive relation: $[\![o_k \sim o_k]\!]^P > 0$;

- $\sim$ is a transitive relation: $[\![o_k \sim o_l \wedge o_l \sim o_m \Rightarrow o_k \sim o_m]\!]^P > 0$ which prevents that $[\![o_k \sim o_l]\!]^P = 1$, $[\![o_l \sim o_m]\!]^P = 1$ but $[\![o_l \sim o_m]\!]^P = 0$.

Informally, this definition of partial models is very general, i.e. it does not impose any further restriction imposed by a particular underlying metamodeling technique. For instance, in case of EMF, each object may have a single direct type and needs to be arranged into a strict containment hierarchy while graphs of the semantic web may be flat and nodes may have multiple types. Such restrictions will be introduced later as structural constraints. Mathematically, partial models show close resemblance with graph shapes [75,76].

If a 3-valued partial model P only contains 1 and 0 values, and there is no $\sim$ relation between different objects (i.e. all equivalent nodes are merged), then P also represents a *concrete instance model* M.

Example 1. Figure 1 shows a metamodel extracted from Yakindu statecharts where `Regions` contain `Vertexes` and `Transitions` (leading from a `source` vertex to a `target` vertex). An abstract state `Vertex` is further refined into `States` and `Entry` states where `States` are further refined into `Regions`.

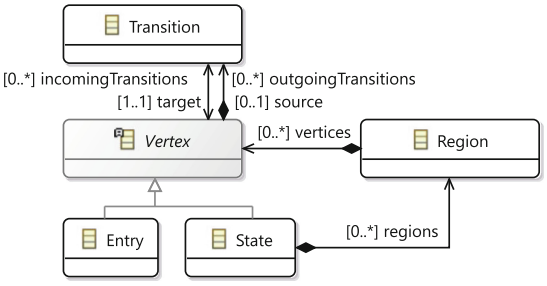


Fig. 1. Metamodel extract of Yakindu statecharts

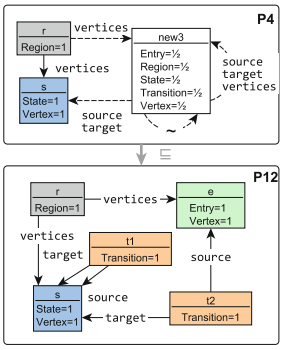


Fig. 2. Partial models

Figure 2 illustrates two partial models: P_4, P_{12} (to be derived by the refinement approach in Sect. 4). The truth value of the type predicates are denoted by labels on the nodes, where 0 values are omitted. Reference predicate values 1 and $1/2$ are visually represented by edges with solid and dashed lines, respectively, while missing edges between two objects represent 0 values for a predicate. Finally, uncertain $1/2$ equivalences are marked by dashed lines with an $\sim$ symbol, while certain equivalence self-loops on objects are omitted.

Partial model P_4 contains one (concrete) `Region` r, one `State` s, and some other objects collectively represented by a single node *new*3. Object s is both of type `State` and `Vertex`, while *new*3 represents objects with multiple possible types. Object s is linked from r via a `vertices` edge, and there are other possible references between r and *new*3. Partial model P_{12}, which is a refinement of P_4, has no uncertain elements, thus it is also a concrete instance model M.

3.3 Graph Patterns as Well-Formedness Constraints

In many industrial modeling tools, complex structural WF constraints are captured either by OCL constraints [70] or by graph patterns (GP) [11,49,67]. Here, we use a tool-independent first-order graph logic representation (which was influenced by [76,98] and is similar to [85]) that covers the key features of several existing graph pattern languages and a first-order logic (FOL) fragment of OCL.

Syntax. A graph pattern (or formula) is a first order logic (FOL) formula $\varphi(v_1, \ldots, v_n)$ over (object) variables. A graph pattern φ can be inductively constructed (see Fig. 3) by using atomic predicates of partial models: $\mathtt{C}(v)$, $\mathtt{R}(v_1, v_2)$, $v_1 \sim v_2$, standard FOL connectives $\neg$, $\vee$, $\wedge$, and quantifiers $\exists$ and $\forall$. A simple graph pattern only contains (a conjunction of) atomic predicates.

Semantics. A graph pattern $\varphi(v_1, \ldots, v_n)$ can be evaluated on partial model P along a variable binding Z, which is a mapping $Z : \{v_1, \ldots, v_n\} \to Obj_P$ from variables to objects in P. The truth value of φ can be evaluated over a partial model P and mapping Z (denoted by $[\![\varphi(v_1, \ldots, v_n)]\!]^P_Z$) in accordance with the semantic rules defined in Fig. 3. Note that *min* and *max* takes the numeric minimum and maximum values of 0, 1/2 and 1 with $0 \leq 1/2 \leq 1$, and the rules follow 3-valued interpretation of standard FOL formulae as defined in [76,85].

A variable binding Z is called a *match* if the pattern φ is evaluated to 1 over P, formally $[\![\varphi(v_1, \ldots, v_n)]\!]^P_Z = 1$. If there exists such a variable binding Z, then we may shortly write $[\![\varphi]\!]^P = 1$. Open formulae (with one or more unbound variables) are treated by introducing an (implicit) existential quantifier over unbound variables to handle them similarly to graph formulae for regular instance models. Thus, in the sequel, $[\![\varphi(v_1, \ldots, v_n)]\!]^P_Z = 1$ if $[\![\exists v_1, \ldots, \exists v_n : \varphi(v_1, \ldots, v_n)]\!]^P = 1$ where the latter is now a closed formula without unbound variables. Similarly, $[\![\varphi]\!]^P = 1/2$ means that there is a potential match where φ evaluates to 1/2, i.e. $[\![\exists v_1, \ldots, \exists v_n : \varphi(v_1, \ldots, v_n)]\!]^P = 1/2$, but there is no match with $[\![\varphi(v_1, \ldots, v_n)]\!]^P_Z = 1$. Finally, $[\![\varphi]\!]^P = 0$ means that there is surely no match, i.e. $[\![\exists v_1, \ldots, \exists v_n : \varphi(v_1, \ldots, v_n)]\!]^P = 0$ for all variable bindings. Here $\exists v_1, \ldots, \exists v_n : \varphi(v_1, \ldots, v_n)$ abbreviates $\exists v_1 : (\ldots, \exists v_n : \varphi(v_1, \ldots, v_n))$.

The formal semantics of graph patterns defined in Fig. 3 can also be evaluated on regular instance models with closed world assumption. Moreover, if a partial model is also a concrete instance model, the 3-valued and 2-valued truth evaluation of a graph pattern is unsurprisingly the same, as shown in [85].

Proposition 1. *Let P be a partial model which is simultaneously an instance model, i.e. $P = M$. Then the 3-valued evaluation of any φ on P and its 2-valued evaluation on M is identical, i.e. $[\![\varphi]\!]^P_Z = [\![\varphi]\!]^M_Z$ along any variable binding Z.*

$$
\begin{aligned}
[\![\mathtt{C}(v)]\!]_Z^P &:= \mathcal{I}_P(\mathtt{C})(Z(v)) \\
[\![\mathtt{R}(v_1,v_2)]\!]_Z^P &:= \mathcal{I}_P(\mathtt{R})(Z(v_1),Z(v_2)) \\
[\![v_1 \sim v_2]\!]_Z^P &:= \mathcal{I}_P(\sim)(Z(v_1),Z(v_2)) \\
[\![\varphi_1 \wedge \varphi_2]\!]_Z^P &:= min([\![\varphi_1]\!]_Z^P, [\![\varphi_2]\!]_Z^P) \\
[\![\varphi_1 \vee \varphi_2]\!]_Z^P &:= max([\![\varphi_1]\!]_Z^P, [\![\varphi_2]\!]_Z^P) \\
[\![\neg\varphi]\!]_Z^P &:= 1 - [\![\varphi]\!]_Z^P \\
[\![\exists v : \varphi]\!]_Z^P &:= max\{[\![\varphi]\!]_{Z,v\mapsto x}^P : x \in Obj_P\} \\
[\![\forall v : \varphi]\!]_Z^P &:= min\{[\![\varphi]\!]_{Z,v\mapsto x}^P : x \in Obj_P\}
\end{aligned}
$$

Fig. 3. Semantics of graph patterns (predicates)

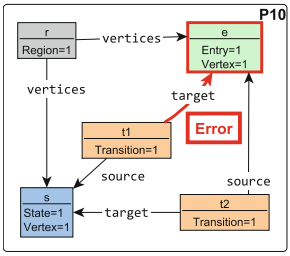


Fig. 4. Malformed model

Graph Patterns as WF Constraints. Graph patterns are frequently used for defining complex structural WF constraints and validation rules [96]. Those constraints are derived from two sources: the metamodel (or type graph) defines core structural constraints, and additional constraints of a domain can be defined by using nested graph conditions [40], OCL [70] or graph pattern languages [96].

When specifying a WF constraint ϕ by a graph pattern φ, pattern φ captures the malformed case by negating ϕ, i.e. $\varphi = \neg\phi$. Thus *a graph pattern match detects a constraint violation.* Given a set of graph patterns $\{\varphi_1, \ldots, \varphi_n\}$ constructed that way, a consistent instance model requires that no graph pattern φ_i has a match in M. Thus any match Z for any pattern φ_i with $[\![\varphi_i]\!]_Z^M = 1$ is a proof of inconsistency. In accordance with the consistency definition $M \models WF$ of Sect. 2, WF can defined by graph patterns as $WF = \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_n$.

Note that consistency is defined above only for instance models, but not for partial models. The refinement calculus to be introduced in Sect. 4 ensures that, by evaluating those graph patterns over partial models, the model generation will gradually converge towards a consistent instance model.

Example 2. The violating cases of two WF constraints checked by the Yakindu tool can be captured by graph patterns as follows:

- $incomingToEntry(v) : \mathtt{Entry}(v) \wedge \exists t : \mathtt{target}(t, v)$
- $noEntryInRegion(r) : \mathtt{Region}(r) \wedge \forall v : \neg(\mathtt{vertices}(r, v) \wedge \mathtt{Entry}(v))$

Both constraints are satisfied in instance model P_{12} as the corresponding graph patterns have no matches, thus P_{12} is a consistent result of model generation. On the other hand, P_{10} in Fig. 4 is a malformed instance model that violates constraint $incomingToEntry(v)$ along object e:

$$[\![incomingToEntry(v)]\!]_{v\mapsto e}^{P_{10}} = 1 \text{ and } [\![noEntryInRegion(r)]\!]^{P_{10}} = 0$$

While graph patterns can be easily evaluated on concrete instance models, checking them over a partial model is a challenging task, because one partial model may represent multiple concretizations. It is shown in [85] how a graph pattern φ can be evaluated on a partial model $\mathcal{P}$ with 3-valued logic and open-world semantics using a regular graph query engine by proposing a constraint

rewriting technique. Alternatively, a SAT-solver based approach can be used as in [24,31] or the general or initial satisfaction can be defined for positive nested graph constraints as in [41,82].

4 Refinement and Concretization of Partial Models

Model generation is intended to be carried out by a sequence of refinement steps which starts from a generic initial partial model and gradually derives a concrete instance model. Since our focus is to derive consistent models, we aim at continuously ensuring that each intermediate partial model can potentially be refined into a consistent model, thus a partial model should be immediately excluded if it cannot be extended to a well-formed instance model.

4.1 A Refinement Relation for Partial Model Generation

In our model generation, the level of uncertainty is aimed to be reduced step by step along a refinement relation which results in partial models that represent a fewer number of concrete instance models than before. In a refinement step, predicates with $1/2$ values can be refined to either 0 or 1, but predicates already fixed to 1 or 0 cannot be changed any more. This imposes an information ordering relation $X \sqsubseteq Y$ where either $X = 1/2$ and Y takes a more specific 1 or 0, or values of X and Y remain equal: $X \sqsubseteq Y := (X = 1/2) \vee (X = Y)$.

Refinement from partial model P to Q (denoted by $P \sqsubseteq Q$) is defined as a function $\mathit{refine} : \mathit{Obj}_P \to 2^{\mathit{Obj}_Q}$ which maps each object of a partial model P to a non-empty set of objects in the refined partial model Q. Refinement respects the information ordering of type, reference and equivalence predicates for each $p_1, p_2 \in \mathit{Obj}_P$ and any $q_1, q_2 \in \mathit{Obj}_Q$ with $q_1 \in \mathit{refine}(p_1)$, $q_2 \in \mathit{refine}(p_2)$:

- for each class $\mathtt{C}_i$: $[\![\mathtt{C}_i(p_1)]\!]^P \sqsubseteq [\![\mathtt{C}_i(q_1)]\!]^Q$;
- for each reference $\mathtt{R}_j$: $[\![\mathtt{R}_j(p_1, p_2)]\!]^P \sqsubseteq [\![\mathtt{R}_j(q_1, q_2)]\!]^Q$;
- $[\![p_1 \sim p_2]\!]^P \sqsubseteq [\![q_1 \sim q_2]\!]^Q$.

At any stage during refinement, a partial model P can be concretized into an instance model M by rewriting all class type and reference predicates of value $1/2$ to either 1 or 0, and setting all equivalence predicates with $1/2$ to 0 between different objects, and to 1 on a single object. But any concrete instance model will still remain a partial model as well.

Example 3. Figure 5 depicts two sequences of partial model refinement steps deriving two instance models P_{10} (identical to Fig. 4) and P_{12} (bottom of Fig. 2): $P_0 \sqsubseteq P_4 \sqsubseteq P_5 \sqsubseteq P_6 \sqsubseteq P_7 \sqsubseteq P_8 \sqsubseteq P_9 \sqsubseteq P_{10}$ and $P_0 \sqsubseteq P_4 \sqsubseteq P_5 \sqsubseteq P_6 \sqsubseteq P_7 \sqsubseteq P_8 \sqsubseteq P_{11} \sqsubseteq P_{12}$.

Taking refinement step $P_4 \sqsubseteq P_5$ as an illustration, object $\mathit{new}3$ (in P_4) is refined into e and $\mathit{new}4$ (in P_5) where $[\![e \sim \mathit{new}4]\!]^{P_5} = 0$ to represent two different objects in the concrete instance models. Moreover, all incoming and outgoing edges of $\mathit{new}3$ are copied in e and $\mathit{new}4$. The final refinement step $P_{11} \sqsubseteq P_{12}$ concretizes uncertain `source` and `target` references into concrete references.

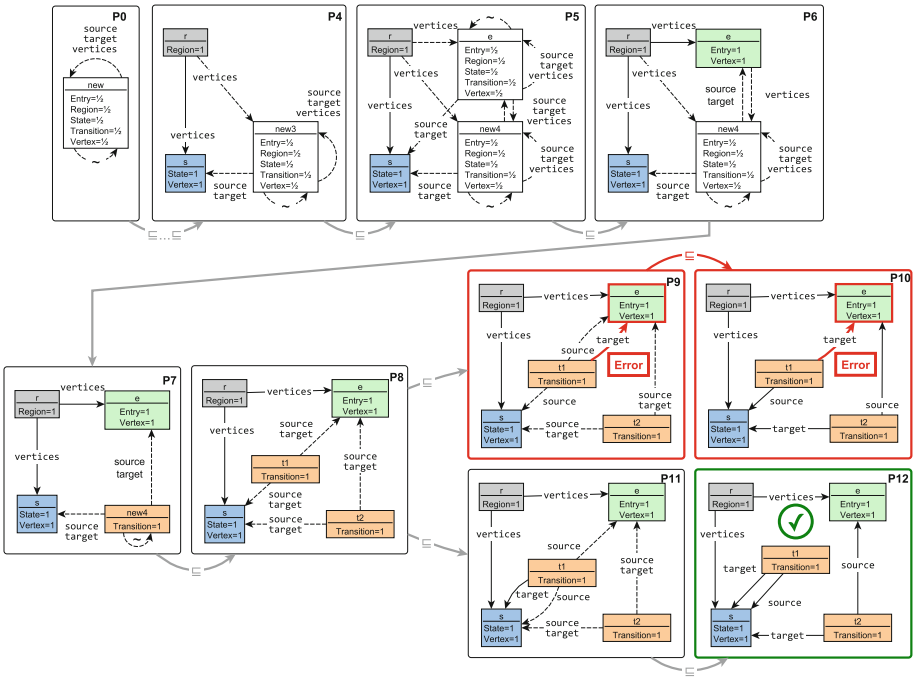


Fig. 5. Refinement of partial models

A model generation process can be initiated from an initial partial model provided by the user, or from the *most generic partial model* P_0 from which all possible instance models can be derived via refinement. Informally, this P_0 is more abstract than regular metamodels or type graphs as it only contains a single node as top-level class. P_0 contains one abstract object where all predicates are undefined, i.e. $P_0 = \langle Obj_{P_0}, \mathcal{I}_{P_0} \rangle$ where $Obj_{P_0} = \{new\}$ and $\mathcal{I}_{P_0}$ is defined as:

1. for all class predicates $\mathtt{C}_i$: $[\![\mathtt{C}_i(new)]\!]^{P_0} = 1/2$;
2. for all reference predicates $\mathtt{R}_j$: $[\![\mathtt{R}_j(new, new)]\!]^{P_0} = 1/2$;
3. $[\![new \sim new]\!]^{P_0} = 1/2$ to represent multiple objects of any instance model.

Our refinement relation ensures that if a predicate is evaluated to either 1 or 0 then its value will no longer change during further refinements as captured by the following approximation theorem.

Theorem 1. *Let P, Q be partial models with $P \sqsubseteq Q$ and φ be a graph pattern.*

- *If $[\![\varphi]\!]^P = 1$ then $[\![\varphi]\!]^Q = 1$; if $[\![\varphi]\!]^P = 0$ then $[\![\varphi]\!]^Q = 0$ (called **under-approximation**).*
- *If $[\![\varphi]\!]^Q = 0$ then $[\![\varphi]\!]^P \leq 1/2$; if $[\![\varphi]\!]^Q = 1$ then $[\![\varphi]\!]^P \geq 1/2$ (called **over-approximation**).*

If model generation is started from P_0 where all (atomic) graph patterns evaluate to $1/2$, this theorem ensures that if a WF constraint φ is violated in a partial model P then it can never be completed to a consistent instance model. Thus the model generation can terminate along this path and a new refinement can be explored after backtracking. This theorem also ensures that if we evaluate a constraint φ on a partial model P and on its refinement Q, the latter will be more precise. In other terms, if $[\![\varphi]\!]^P = 1$ (or 0) in a partial model P along some sequence of refinement steps, then under-approximation ensures that its evaluation will never change again along that (forward) refinement sequence, i.e. $[\![\varphi]\!]^Q = 1$ (or 0). Similarly, when proceeding backwards in a refinement chain, over-approximation ensures monotonicity of the 3-valued constraint evaluation along the entire chain. Altogether, we gradually converge to the 2-valued truth evaluation of the constraint on an instance model where less and less constraints take the $1/2$ value. However, a refinement step does not guarantee in itself that exploration is progressing towards a consistent model, i.e. there may be infinite chains of refinement steps which never derive a concrete instance model.

4.2 Refinement Operations for Partial Models

We define *refinement operations* Op to refine partial models by simultaneously growing the size of the models while reducing uncertainty in a way that each finite and consistent instance model is guaranteed to be derived in finite steps.

- *concretize*(p, val): if the atomic predicate p (which is either $\texttt{C}_i(o)$, $\texttt{R}_j(o_k, o_l)$ or $o_k \sim o_l$) has a $1/2$ value in the pre-state partial model P, then it can be refined in the post-state Q to *val* which is either a 1 or 0 value. As an effect of the rule, the level of uncertainty will be reduced.
- *splitAndConnect*$(o, mode)$: if o is an object with $[\![o \sim o]\!]^P = 1/2$ in the pre-state, then a new object *new* is introduced in the post state by splitting o in accordance with the semantics defined by the following two modes:
  - *at-least-two*: $[\![new \sim new]\!]^Q = 1/2$, $[\![o \sim o]\!]^Q = 1/2$, $[\![new \sim o]\!]^Q = 0$;
  - *at-most-two*: $[\![new \sim new]\!]^Q = 1$, $[\![o \sim o]\!]^Q = 1$, $[\![new \sim o]\!]^Q = 1/2$;

  In each case, $ObjQ = ObjP \cup \{new\}$, and we copy all incoming and outgoing binary relations of o to *new* in Q by keeping their original values in P. Furthermore, all class predicates remain unaltered.

On the technical level, these refinement operations could be easily captured by means of algebraic graph transformation rules [28] over typed graphs. However, for efficiency reasons, several elementary operations may need to be combined into compound rules. Therefore, specifying refinement operations by graph transformation rules will be investigated in a future paper.

Example 4. Refinement $P_4 \sqsubseteq P_5$ (in Fig. 5) is a result of applying refinement operation *splitAndConnect*$(o, mode)$ on object *new*3 and in *at-least-two* mode, splitting *new*3 to e and *new*4 copying all incoming and outgoing references. Next, in P_6, the type of object e is refined to `Entry` and `Vertex`, the $1/2$ equivalence is

refined to 1, and references incompatible with `Entry` or `Vertex` are refined to 0. Note that in P_6 it is ensured that `Region` r has an `Entry`, thus satisfying WF constraint *noEntryInRegion*. In P_7 the type of object *new*4 is refined to `Transition`, the incompatible references are removed similarly, but the $1/2$ self equivalence remain unchanged. Therefore, in P_8 object *new*4 can split into two separate `Transition`s: $t1$ and $t2$ with the same source and target options. Refinement $P_8 \sqsubseteq P_9 \sqsubseteq P_{10}$ denotes a possible refinement path, where the `target` of $t1$ is directed to an `Entry`, thus violating WF constraint *incomingToEntry*. Note that this violation can be detected earlier in an unfinished partial model P_9. Refinement $P_{11} \sqsubseteq P_{12}$ denotes the consecutive application of six *concretize*(p, val) operations on uncertain `source` and `target` edges leading out of $t1$ and $t2$ in P_{11}, resulting in a valid model.

Note that these refinement operations may result in a partial model that is unsatisfiable. For instance, if all class predicates evaluate to 0 for an object o of the partial model P, i.e. $[\![\texttt{C}(o)]\!]^P = 0$, then no instance models will correspond to it as most metamodeling techniques require that each element has exactly or at least one type. Similarly, if we violate the reflexivity of $\sim$, i.e. $[\![o \sim o]\!]^P = 0$, then the partial model cannot be concretized into a valid instance model. But at least, one can show that these refinement operations ensure a refinement relation between the partial models of its pre-state and post-state.

Theorem 2 (Refinement operations ensure refinement). *Let P be a partial model and op be a refinement operation. If Q is the partial model obtained by executing op on P (formally, $P \xrightarrow{op} Q$) then $P \sqsubseteq Q$.*

4.3 Consistency of Model Generation by Refinement Operations

Next we formulate and prove the consistency of model generation when it is carried out by a sequence of refinement steps from the most generic partial model P_0 using the previous refinement operations. We aim to show soundness (i.e. if a model is derivable along an open derivation sequence then it is consistent), finite completeness (i.e. each finite consistent model can be derived along some open derivation sequence), and a concept of incrementality.

Many tableaux based theorem provers build on the concept of closed branches with a contradictory set of formulae. We adapt an analogous concept for closed derivation sequences over graph derivations in [28]. Informally, refinement is not worth being continued as a WF constraint is surely violated due to a match of a graph pattern in case of a closed derivation sequence. Consequently, all consistent instance models will be derived along open derivation sequences.

Definition 1 (Closed vs. open derivation sequence). *A finite derivation sequence of refinement operations $op_1; \ldots; op_k$ leading from the most generic partial model P_0 to the partial model P_k (denoted as $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$) is closed w.r.t. a graph predicate φ if φ has a match in P_k, formally, $[\![\varphi]\!]^{P_k} = 1$.*

A derivation sequence is open if it is not closed, i.e. P_k is a partial model derived by a finite derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ with $[\![\varphi]\!]^{P_k} \leq 1/2$.

Note that a single match of φ makes a derivation sequence to be closed, while an open derivation sequence requires that $[\![\varphi]\!]^{P_k} \leq 1/2$ which, by definition, disallows a match with $[\![\varphi]\!]^{P_k} \leq 1$.

Example 5. Derivation sequence $P_0 \overset{\cdots}{\longrightarrow} P_9$ depicted in Fig. 5 is closed for $\varphi = \mathit{incomingToEntry}(v)$ as the corresponding graph pattern has a match in P_9, i.e. $[\![\mathit{incomingToEntry}(v)]\!]^{P_9}_{v \mapsto e} = 1$. Therefore, P_{10} can be avoided as the same match would still exist. On the other hand, derivation sequence $P_0 \overset{\cdots}{\longrightarrow} P_{11}$ is open for $\varphi = \mathit{incomingToEntry}(v)$ as $\mathit{incomingToEntry}(v)$ is evaluated to $1/2$ in all partial models $P_0, \ldots, P_{11}$.

As a consequence of Theorem 1, an open derivation sequence ensures that any prefix of the same derivation sequence is also open.

Corollary 1. *Let $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ be an open derivation sequence of refinement operations w.r.t. φ. Then for each $0 \leq i \leq k$, $[\![\varphi]\!]^{P_i} \leq 1/2$.*

The soundness of model generation informally states that if a concrete model M is derived along an open derivation sequence then M is consistent, i.e. no graph predicate of WF constraints has a match.

Corollary 2 (Soundness of model generation). *Let $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ be a finite and open derivation sequence of refinement operations w.r.t. φ. If P_k is a concrete instance model M (i.e. $P_k = M$) then M is consistent (i.e. $[\![\varphi]\!]^M = 0$).*

Effectively, once a concrete instance model M is reached during model generation along an open derivation sequence, checking the WF constraints on M by using traditional (2-valued) graph pattern matching techniques ensures the soundness of model generation as 3-valued and 2-valued evaluation of the same graph pattern should coincide due to Proposition 1 and Theorem 1.

Next, we show that any finite instance model can be derived by a finite derivation sequence.

Theorem 3 (Finiteness of model generation). *For any finite instance model M, there exists a finite derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ of refinement operations starting from the most generic partial model P_0 leading to $P_k = M$.*

Our completeness theorem states that any consistent instance model is derivable along open derivation sequences where no constraints are violated (under-approximation). Thus it allows to eliminate all derivation sequences where an graph predicate φ evaluates to 1 on any intermediate partial model P_i as such partial model cannot be further refined to a well-formed concrete instance model due to the properties of under-approximation. Moreover, a derivation sequence leading to a consistent model needs to be open w.r.t. all constraints, i.e. refinement can be terminated if any graph pattern has a match.

Theorem 4 (Completeness of model generation). *For any finite and consistent instance model M with $[\![\varphi]\!]^M = 0$, there exists a finite open derivation sequence $P_0 \xrightarrow{op_1;\ldots;op_k} P_k$ of refinement operations w.r.t. φ starting from the most generic partial model P_0 and leading to $P_k = M$.*

Unsurprisingly, graph model generation still remains undecidable in general as there is no guarantee that a derivation sequence leading to P_k where $[\![\varphi]\!]^{P_k} = 1/2$ can be refined later to a consistent instance model M. However, the graph model finding problem is decidable for a finite scope, which is an a priori upper bound on the size of the model. Informally, since the size of partial models is gradually growing during refinement, we can stop if the size of a partial model exceeds the target scope or if a constraint is already violated.

Theorem 5 (Decidability of model generation in finite scope). *Given a graph predicate φ and a scope $n \in \mathbb{N}$, it is decidable to check if a concrete instance model M exists with $|Obj_M| \leq n$ where $[\![\varphi]\!]^M = 0$.*

This finite decidability theorem is analogous with formal guarantees provided by the Alloy Analyzer [94] that is used by many mapping-based model generation approaches (see Sect. 6). Alloy aims to synthesize small counterexamples for a relational specification, while our refinement calculus provides the same for typed graphs without parallel edges for the given refinement operations.

However, our construction has extra benefits compared to Alloy (and other SAT-solver based techniques) when exceeding the target scope. First, all candidate partial models (with constraints evaluated to 1/2) derived up to a certain scope are reusable for finding consistent models of a larger scope, thus search can be incrementally continued. Moreover, if a constraint violation is found with a given scope, then no consistent models exist at all.

Corollary 3 (Incrementality of model generation). *Let us assume that no consistent models M^n exist for scope n, but there exists a larger consistent model M^m of size m (where $m > n$) with $[\![\varphi]\!]^{M^m} = 0$. Then M^m is derivable by a finite derivation sequence $P_i^n \xrightarrow{op_{i+1};\ldots;op_k} P_k^m$ where $P_k^m = M^m$ starting from a partial model P_i^n of size n.*

Corollary 4 (Completeness of refutation). *If all derivation sequences are closed for a given scope n, but no consistent model M^n exists for scope n for which $[\![\varphi]\!]^{M^n} = 0$, then no consistent models exist at all.*

While these theorems aim to establish the theoretical foundations of a model generator framework, it provides no direct practical insight on the exploration itself, i.e. how to efficiently provide derivation sequences that likely lead to consistent models. Nevertheless, we have an initial prototype implementation of such a model generator which is also used as part of the experimental evaluation.

5 Evaluation

As existing model generators have been dominantly focusing on a single challenge of Sect. 2, we carried out an initial experimental evaluation to investigate how popular strategies excelling in one challenge perform with respect to another challenge. More specifically, we carried out this evaluation in the domain of Yakindu statecharts to address four research questions:

RQ1 *Diverse vs. Realistic:* How realistic are the models which are derived by random generators that promise diversity?
RQ2 *Consistent vs. Realistic:* How realistic are the models which are derived by logic solvers that guarantee consistency?
RQ3 *Diverse vs. Consistent:* How consistent are the models which are derived by random generators?
RQ4 *Consistent vs. Scalable:* How scalable is it to evaluate consistency constraints on partial models?

Addressing these questions may help advancing future model generators by identifying some strength and weaknesses of different strategies.

5.1 Setup of Experiments

Target Domain. We conducted measurements in the context of Yakindu statecharts, see [2] for the complete measurement data. For that purpose, we extracted the statechart metamodel of Fig. 1 directly from the original Yakindu metamodel. Ten WF constraints were formalized as graph patterns based on the real validation rules of the Yakindu Statechart development environment.

Model Generator Approaches. For addressing *RQ1-3*, we used two different model generation approaches: (1) the popular *relational model finder* Alloy Analyzer [94] which uses Sat4j [53] as a back-end SAT-solver, and (2) the VIATRA Solver, *graph-based model generator* which uses the refinement calculus presented in the paper. We selected Alloy Analyzer as the primary target platform as it has been widely used in mapping based generators of consistent models (see Sect. 6).

We operated these solvers in two modes: in *consistent* mode (WF), all derived models need to satisfy all WF constraints of Yakindu statecharts, while in *metamodel-only* mode (MM), generated models need to be metamodel compliant, but then model elements are selected randomly. As such, we expect that this set of models is diverse, but the fulfillment of WF constraints is not guaranteed. To enforce diversity, we explicitly check that derived models are non-isomorphic.

Since mapping based approaches typically compile WF constraints into logic formulae in order to evaluate them on partial models, we set up a simple measurement to address *RQ4* which did not involve model generation but only constraint checking on existing instance models. This is a well-known setup for assessing scalability of graph query techniques used in a series of benchmarking papers [92,98]. So in our case, we encoded instance models as fully defined Alloy specifications using the mapping of [86], and checked if the constraints are satisfied (without extending or modifying the statechart). As a baseline of comparison, we checked the runtime of evaluating the same WF constraints on the same models using an industrial graph query engine [97] which is known to scale well for validation problems [92,98]. All measurements were executed on an average desktop computer[2].

[2] CPU: Intel Core-i5-m310M, MEM: 16 GB, OS: Windows 10 Pro.

Real Instance Models. To evaluate how realistic the synthetic model generators are in case of *RQ1-2*, we took 1253 statecharts as *real models* created by undergraduate students for a homework assignment. While they had to solve the same modeling problem, the size of their models varied from 50 to 200 objects. For *RQ4*, we randomly selected 300 statecharts from the homework assignments, and evaluated the original validation constraints. Real models were filtered by removing inverse edges that introduce significant noise to the metrics [91].

Generated Instance Models. To obtain comparable results, we generated four sets of statechart models with a timeout of 1 min for each model but without any manual domain-specific fine-tuning of the different solvers. We also check that the generated models are non-isomorphic to assure sufficient diversity.

- Alloy (MM): 100 metamodel-compliant models with 50 objects using Alloy.
- Alloy (WF): 100 metamodel- and WF-compliant models with 50 objects using Alloy (which was unable to synthesize larger models within 1 min).
- VIATRA Solver (MM): 100 metamodel-compliant instance models with 100 objects using VIATRA Solver.
- VIATRA Solver (WF): 100 Metamodel- and WF-compliant instance models with 100 objects using VIATRA Solver.

Two multi-dimensional graph metrics are used to evaluate how realistic a model generator is: (1) the *multiplex participation coefficient (MPC)* measures how the edges of nodes are distributed along the different edge types, while (2) *pairwise multiplexity (Q)* captures how often two different types of edges meet in an object. These metrics were recommended in [91] out of over 20 different graph metrics after a multi-domain experimental evaluation, for formal definitions of the metrics, see [91]. Moreover, we calculate the (3) number of WF constraints violated by a model as a numeric value to measure the degree of (in)consistency of a model (which value is zero in case of consistent models).

5.2 Evaluation of Measurement Results

We plot the distribution functions of the *multiple participation coefficient* metric in Fig. 6, and the *pairwise multiplexity* metric in Fig. 7. Each line depicts the characteristics of a single model and model sets (e.g. "Alloy (MM)", "VIATRA Solver (WF)") are grouped together in one of the facets including the characteristics of the real model set. For instance, the former metric tells that approximately 65% of nodes in real statechart models (right facet in Fig. 6) have only one or two types of incoming and outgoing edges while the remaining 35% of nodes have edges more evenly distributed among different types.

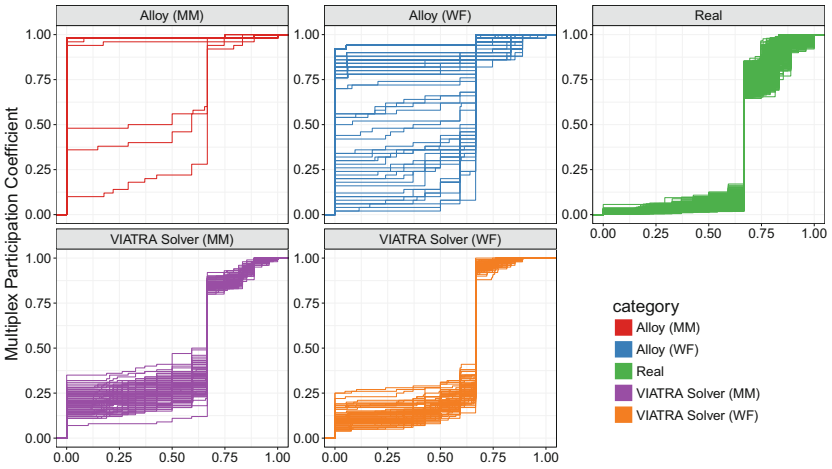


Fig. 6. Measurement results: Multiplex participation coefficient (MPC)

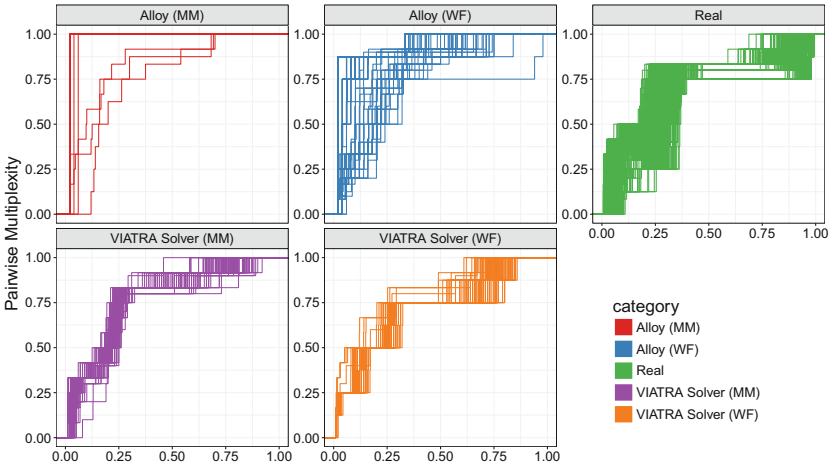


Fig. 7. Measurement results: Pairwise multiplexity (Q)

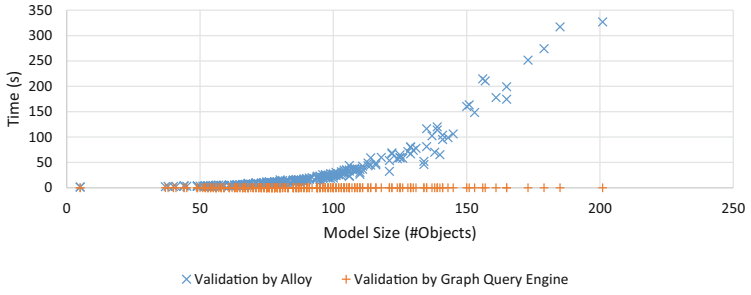


Fig. 8. Measurement results: Time of consistency checks: Alloy vs. VIATRA Solver

Comparison of Distribution Functions. We use visual data analytics techniques and the Kolmogorov-Smirnov statistic (*KS*) [55] as a distance measure of models (used in [91]) to judge how realistic an auto-generated model is by comparing the whole distributions of values (and not only their descriptive summary like mean or variance) in different cases to the characteristics of real models. The *KS* statistics quantifies the maximal difference between the distribution function lines at a given value. It is sensitive to both shape and location differences: it takes a 0 value only if the distributions are identical, while it is 1 if the values of models are in disjunct ranges (even if their shapes are identical). For comparing model generation techniques A and B we took the average value of the *KS* statistics between each (A, B) pair of models that were generated by technique A and B, respectively. The average *KS* values are shown in Fig. 9,[3] where a lower value denotes a more realistic model set.

Diverse vs. Realistic: For the models that are only metamodel-compliant, the characteristics of the metrics for "VIATRA Solver (MM)" are much closer to the "Real" model set than those of the "Alloy (MM)" model set, for both graph metrics (*KS* value of 0.27 vs. 0.95 for *MPC* and 0.38 vs. 0.88 for *Q*), thus more realistic results were generated in the "VIATRA Solver (MM)" case. However, these plots also highlight that the set of auto-generated metamodel-compliant models can be more easily distinguished from the set of real models as the plots of the latter show higher variability. Since the diversity of each model generation case is enforced (i.e. non-isomorphic models are dropped), we can draw as a conclusion that *a diverse metamodel-compliant model generator does not provide any guarantees in itself on the realistic nature of the output model set.* In fact, model generators that simultaneously ensure diversity and consistency always outperformed the random model generators for both solvers.

Consistent vs. Realistic: In case of models satisfying WF constraints "VIATRA Solver (WF)" generated more realistic results than "Alloy (WF)" according to both metrics. The plots show mixed results for differentiating between generated and realistic models. On the positive side, the shape of the plot of auto-generated models is very close to that of real models in case of the *MPC* metric (Fig. 6) – statistically, they have a relatively low average *KS* value of 0.24. However, for the *Q* metric (Fig. 7), real models are still substantially different from generated ones (average *KS* value of 0.3). Thus *further research is needed to investigate how to make consistent models more realistic.*

model set	**MPC**	**Q**
Alloy (MM)	0.95	0.88
Alloy (WF)	0.74	0.60
VIATRA Solver (MM)	0.27	0.37
VIATRA Solver (WF)	0.24	0.30

Fig. 9. Average Kolmogorov-Smirnov statistics between the *real* and generated model sets.

[3] Due to the excessive amount of homework models, we took a uniform random sample of 100 models from that model set.

Diverse vs. Consistent: We also calculated the average number of WF constraint violations, which was 3.1 for the "Alloy (MM)" case and 9.75 for the "VIATRA Solver (MM)" case, while only 0.07 for real models. We observe that a diverse set of randomly generated metamodel-compliant instance models do not yield consistent models as some constraints will always be violated – which is not the case for real statechart models. In other terms, the number of WF constraint violations is also an important characteristic of realistic models which is often overseen in practice. As a conclusion, a *model generator should ensure consistency prior to focusing on increasing diversity.* Since humans dominantly come up with consistent models, *ensuring consistency for realistic models is a key prerequisite.*

Consistent vs. Scalable: The soundness of consistent model generation inherently requires the evaluation of the WF constraints at least once for a candidate solution. Figure 8 depicts the validation time of randomly selected homework models using Alloy and the VIATRA graph query engine w.r.t. the size of the instance model (i.e. the number of the objects). For each model, the two validation techniques made the same judgment (as a test for their correctness). Surprisingly, the diagram shows that the Alloy Analyzer scales poorly for evaluating constraints on medium-size graphs, which makes it unsuitable for generating larger models. The runtime of the graph query engine was negligible at this scale as we expected based on detailed previous benchmarks for regular graph pattern matching and initial experiments for matching constraints over partial models [85].

While many existing performance benchmarks claim that they generate realistic models, most of them ignore WF constraints of the domain. According to our measurements, it is a major drawback since real models dominantly satisfy WF constraints while randomly generated models always violate some constraint. This way, those model generators can hardly be considered realistic.

Threats to Validity. We carried out experiments only in the domain of statecharts which limits the generalizability of our results. Since statecharts are a behavioral modeling language, the characteristics of models (and thus the graph metrics) would likely differ from e.g. static architectural DSLs. However, since many of our experimental results are negative, it is unlikely that the Alloy generator would behave any better for other domains. It is a major finding that while Alloy has been used as a back-end for mapping-based model generator approaches, its use is not justified from a scalability perspective due to the lack of efficient evaluation for complex structural graph constraints. It is also unlikely that randomly generated metamodel-compliant models would be more realistic, or more consistent in any other domains.

Concerning our real models, we included all the statecharts created by students, which may be a bias since students who obtained better grades likely produced better quality models. Thus, the variability of real statechart models created by engineers may actually be smaller. But this would actually increase the relative quality of models derived by VIATRA Solver which currently differs

Table 1. Characteristics of model generation approaches; +: feature provided, −: feature not provided, 0: feature provided in some tools/cases

		Logic solvers	Random generators	Network graphs	Performance benchmarks	Real dataset
CON	Model	+	−	−	+	+
	Complete	0	−	−	−	−
DIV	Model	−	+	−	−	−
	Set	−	+	−	−	−
SCA	In Size	−	+	+	+	+
	In Quantity	−	0	+	+	−
REA	Model	−	−	−	−	+
	Set	−	−	−	0	+

from real models by providing a lower level of diversity (i.e. plots of pairwise multiplicity are thicker for real models).

6 Related Work

We assess and compare how existing approaches address each challenge (Table 1).

Consistent Model Generators (CON): Consistent models can be synthesized as a side effect of a verification process when aiming to prove the consistency of a DSL specification. The metamodel and a set of WF constraints are captured in a high-level DSL and logic formulae are generated as input to back-end logic solvers. Approaches differ in the *language used for WF constraints*, OCL [18–20, 23,35,50–52,73,87,100], graph constraints [84,86], Java predicates [14] or custom DSLs like Formula [46], Clafer [8] or Alloy [45]. They also differ in *the solver used in the background*: graph transformation engines as in [100], SAT-solvers [53] are used in [51,52], model finders like Kodkod [94] are target formalisms in [5,23,50,87], first-order logic formulae are derived for SMT-solvers [65] in [73,84] while CSP-solvers like [1] are targeted in [18,19] or other techniques [59,74].

Solver-based approaches excel in finding inconsistencies in specifications, while the generated model is a proof of consistency. While SAT solvers can handle specifications with millions of Boolean variables, all these mapping-based techniques still suffer from severe scalability issues as the generated graphs may contain less than 50–100 nodes. This is partly due to the fact that a Boolean variable needs to be introduced for each *potential* edge in the generated model, which blows up the complexity. Moreover, the output models are highly similar to each other and lack diversity, thus they cannot directly be applied for testing or benchmarking purposes.

Diverse Model Generators (DIV): Diverse models play a key role in testing model transformations and code generators. Mutation-based approaches [6,25,61] take existing models and make random changes on them by applying mutation rules. A similar random model generator is used for experimentation purposes in [9]. Other automated techniques [15,29] generate models that only conform to the metamodel. While these techniques scale well for larger models, there is no guarantee whether the mutated models satisfy WF constraints.

There is a wide set of model generation techniques which provide certain promises for test effectiveness. White-box approaches [37,38,62,83] rely on the implementation of the transformation and dominantly use back-end logic solvers, which lack scalability when deriving graph models. Black-box approaches [17,34,39,56,63] can only exploit the specification of the language or the transformation, so they frequently rely upon contracts or model fragments. As a common theme, these techniques may generate a set of simple models, and while certain diversity can be achieved by using symmetry-breaking predicates, they fail to scale for larger model sizes. In fact, the effective diversity of models is also questionable since corresponding safety standards prescribe much stricter test coverage criteria for software certification and tool qualification than those currently offered by existing model transformation testing approaches.

Based on the logic-based Formula solver, the approach of [47] applies stochastic random sampling of output to achieve a diverse set of generated models by taking exactly one element from each equivalence class defined by graph isomorphism, which can be too restrictive for coverage purposes. Stochastic simulation is proposed for graph transformation systems in [95], where rule application is stochastic (and not the properties of models), but fulfillment of WF constraints can only be assured by a carefully constructed rule set.

Realistic Model Generators (REA): The igraph library [22] contains a set of randomized graph generators that produce one-dimensional (untyped) graphs that follow a particular distribution (e.g. Erdős-Rényi, Watts-Strogatz). The authors of [64] use Boltzmann samplers [27] to ensure efficient generation of uniform models. GSCALER [102] takes a graph as its input and generates a similar graph with a certain number of vertices and edges.

Scalable Model Generators (SCA): Several database benchmarks provide scalable graph generators with some degree of well-formedness or realism. The *Berlin SPARQL Benchmark (BSBM)* [13] uses a single dataset that scales in model size (10 million–150 billion tuples), but does not vary in structure. SP^2Bench [80] uses a data set, which is synthetically generated based on the real-world DBLP bibliography. This way, instance models of different sizes reflect the structure and complexity of the original real-world dataset.

The Linked Data Benchmark Council (LDBC) recently developed the Social Network Benchmark [30], which contains a social network generator module [88]. The generator is based on the S3G2 approach [72] that aims to generate non-uniform value distributions and structural correlations. gMark [7] generates

graphs driven by a pre-defined schema that allows users to specify vertex/edge types and degree distributions in the graph, which provides some level of realism.

The *Train Benchmark* [92] uses a domain-specific generator that is able to generate railway networks, scalable in size and satisfying a set of well-formedness constraints. The generator is also able to inject errors to the models during generation (thus intentionally violating the WF property).

Transformations of Partial Models. Uncertain models [31] document semantic variation points generically by annotations on a regular instance model. Potential concrete models compliant with an uncertain model can be synthesized by the Alloy Analyzer and its back-end SAT solvers [78,79], or refined by graph transformation rules [77].

Transformations over partial models [32,33] analyse possible matches and executions of model transformation rules on partial models by using a SAT solver (MathSAT4) or by automated graph approximation called "lifting", which inspects possible partitions of a finite concrete model, i.e. regular graph transformation rules are lifted, while in this paper, we attempt to introduce model generator rules directly on the level of partial models.

Regular graph transformation rules are used for model generation is carried out in [29,100] where output models are metamodel compliant, but they do not fulfill extra WF constraints of the domain [29] or (a restricted set of) constraints need to be translated first to rule application conditions [100].

Symbolic Model Generation. Certain techniques use abstract (or symbolic) graphs for analysis purposes. A tableau-based reasoning method is proposed for graph properties [3,71,81], which automatically refine solutions based on well-formedness constraints, and handle state space in the form of a resolution tree. As a key difference, our approach refines possible solutions in the form of partial models, while [71,81] resolves the graph constraints to a concrete solution.

7 Conclusion and Future Work

In this paper, we presented the challenge of automated graph model generation where models are consistent, diverse, scalable and realistic at the same time. In an experimental evaluation, we found that traditional model generation techniques which excel in one aspect perform poorly with respect to another aspect. Furthermore, consistent models turn out to be a prerequisite both for the realistic and diverse cases. As the main conceptual contribution of this paper, we presented a refinement calculus based on 3-valued logic evaluation of graph patterns that could drive the automated synthesis of consistent models. We proved soundness and completeness for this refinement approach, which also enables to incrementally generate instance models of a larger scope by reusing partial models traversed in a previous scope. As such, it offers stronger consistency guarantees than the popular Alloy Analyzer used as a back-end solver for many mapping-based model generation approaches.

While an initial version of a model generator operating that way was included in our experimental evaluation, our main ongoing work is to gradually address several model generation challenges at the same time. For instance, model generators which are simultaneously consistent, diverse and realistic could help in the systematic testing of the VIATRA transformation framework [97] or other industrial DSL tools.

Acknowledgements. The authors are really grateful for the anonymous reviewers and Szilvia Varró-Gyapay for the numerous constructive feedback to improve the current paper. This paper is partially supported by MTA-BME Lendület Research Group on Cyber-Physical Systems, and NSERC RGPIN-04573-16 project.

References

1. Choco. http://www.emn.fr/z-info/choco-solverp
2. Complete measurement setup and results of the paper (2017). https://github.com/FTSRG/publication-pages/wiki/Towards-the-Automated-Generation-of-Consistent,-Diverse,-Scalable,-and-Realistic-Graph-Models/
3. Al-Sibahi, A.S., Dimovski, A.S., Wasowski, A.: Symbolic execution of high-level transformations. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, 31 October–1 November 2016, pp. 207–220 (2016). http://dl.acm.org/citation.cfm?id=2997382
4. Ali, S., Iqbal, M.Z.Z., Arcuri, A., Briand, L.C.: Generating test data from OCL constraints with search techniques. IEEE Trans. Softw. Eng. **39**(10), 1376–1402 (2013)
5. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model. **9**(1), 69–86 (2010)
6. Aranega, V., Mottu, J.M., Etien, A., Degueule, T., Baudry, B., Dekeyser, J.L.: Towards an automation of the mutation analysis dedicated to model transformation. Softw. Test. Verif. Reliab. **25**(5–7), 653–683 (2015)
7. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gMark: schema-driven generation of graphs and queries. IEEE Trans. Knowl. Data Eng. **29**(4), 856–869 (2017)
8. Bak, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wasowski, A.: Clafer: unifying class and feature modeling. Softw. Syst. Model. **15**(3), 811–845 (2016)
9. Batot, E., Sahraoui, H.: A generic framework for model-set selection for the unification of testing and learning MDE tasks. In: MODELS. pp. 374–384. ACM Press (2016)
10. Battiston, F., Nicosia, V., Latora, V.: Structural measures for multiplex networks. Phys. Rev. E Stat. Nonlin. Soft Matter Phys. **89**(3), 032804 (2014)
11. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for EMF models. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 167–182. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21732-6_12
12. Berlingerio, M., et al.: Multidimensional networks: foundations of structural analysis. World Wide Web **16**(5–6), 567–593 (2013)
13. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Sem. Web Inf. Syst. **5**(2), 1–24 (2009)

14. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 123–133. ACM Press (2002)
15. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: ISSRE, pp. 85–94, November 2006
16. Bures, T., et al.: Software engineering for smart cyber-physical systems - towards a research agenda. ACM SIGSOFT Softw. Eng. Notes **40**(6), 28–32 (2015)
17. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 198–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_16
18. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. J. Syst. Softw. **93**, 1–23 (2014)
19. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. J. Syst. Softw. **82**(9), 1459–1478 (2009)
20. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. ECEASST, vol. 24 (2009)
21. 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
22. Csardi, G., Nepusz, T.: The igraph software package for complex network research. InterJournal Complex Syst. **1695** (2006). http://igraph.sf.net
23. Cunha, A., Garis, A., Riesco, D.: Translating between alloy specifications and UML class diagrams annotated with OCL. Softw. Syst. Model. **14**(1), 5–25 (2015)
24. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 211–220. ACM (2006)
25. Darabos, A., Pataricza, A., Varró, D.: Towards testing the implementation of graph transformations. In: GTVMT. ENTCS. Elsevier (2006)
26. DeWitt, D.J.: The Wisconsin benchmark: past, present, and future. In: The Benchmark Handbook, pp. 119–165 (1991)
27. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Boltzmann samplers for the random generation of combinatorial structures. Comb. Probab. Comput. **13**(4–5), 577–625 (2004)
28. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
29. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Softw. Syst. Model. **8**(4), 479–500 (2009)
30. Erling, O., et al.: The LDBC social network benchmark: interactive workload. In: SIGMOD, pp. 619–630 (2015)
31. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: ICSE, pp. 573–583. IEEE Press (2012)
32. Famelis, M., Salay, R., Chechik, M.: The semantics of partial model transformations. In: MiSE at ICSE, pp. 64–69. IEEE Press (2012)
33. Famelis, M., Salay, R., Di Sandro, A., Chechik, M.: Transformation of models containing uncertainty. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 673–689. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_41

34. Fleurey, F., Baudry, B., Muller, P.A., Le Traon, Y.: Towards dependable model transformations: qualifying input test data, appears to be published only in a technical report by INRIA (2007). https://hal.inria.fr/inria-00477567
35. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**(1–3), 27–34 (2007)
36. Goldberg, A.P., Chew, Y.H., Karr, J.R.: Toward scalable whole-cell modeling of human cells. In: SIGSIM-PADS, pp. 259–262. ACM Press (2016)
37. González, C.A., Cabot, J.: ATLTest: a white-box test generation approach for ATL transformations. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 449–464. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_29
38. González, C.A., Cabot, J.: Test data generation for model transformations combining partition and constraint analysis. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 25–41. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08789-4_3
39. Guerra, E., Soeken, M.: Specification-driven model transformation testing. Softw. Syst. Model. **14**(2), 623–644 (2015)
40. Habel, A., Pennemann, K.-H.: Nested constraints and application conditions for high-level structures. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 293–308. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31847-7_17
41. Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
42. Härtel, J., Härtel, L., Lämmel, R.: Test-data generation for Xtext. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 342–351. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_19
43. ISO: Road vehicles - functional safety (ISO 26262) (2011)
44. Izsó, B., Szatmári, Z., Bergmann, G., Horváth, Á., Ráth, I.: Towards precise metrics for predicting graph query performance. In: ASE, pp. 421–431 (2013)
45. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
46. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Automatically reasoning about metamodeling. Softw. Syst. Model. **14**(1), 271–285 (2015)
47. Jackson, E.K., Simko, G., Sztipanovits, J.: Diversely enumerating system-level architectures. In: EMSOFT, p. 11. IEEE Press (2013)
48. Kleene, S.C., De Bruijn, N., de Groot, J., Zaanen, A.C.: Introduction to Metamathematics, vol. 483. van Nostrand, New York (1952)
49. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 204–218. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-11447-2_13
50. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 415–431. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_27

51. Kuhlmann, M., Gogolla, M.: Strengthening SAT-based validation of UML/OCL models by representing collections as relations. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 32–48. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_5
52. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21952-8_21
53. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. J. Satisf. Boolean Model. Comput. **7**, 59–64 (2010)
54. Lee, E.A., et al.: The swarm at the edge of the cloud. IEEE Des. Test **31**(3), 8–20 (2014)
55. Lehmann, E.L., D'Abrera, H.J.: Nonparametrics: Statistical Methods Based on Ranks. Springer, New York (2006)
56. López-Fernández, J.J., Guerra, E., de Lara, J.: Combining unit and specification-based testing for meta-model validation and verification. Inf. Syst. **62**, 104–135 (2016)
57. Meedeniya, I., Aleti, A., Grunske, L.: Architecture-driven reliability optimization with uncertain model parameters. J. Syst. Softw. **85**(10), 2340–2355 (2012)
58. Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.: A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: Jezic, G., Kusek, M., Nguyen, N.-T., Howlett, R.J., Jain, L.C. (eds.) KES-AMSTA 2012. LNCS (LNAI), vol. 7327, pp. 504–513. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30947-2_55
59. Misailovic, S., Milicevic, A., Petrovic, N., Khurshid, S., Marinov, D.: Parallel test generation and execution with Korat. In: ESEC-FSE 2007, pp. 135–144. ACM (2007)
60. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_29
61. Mottu, J.-M., Baudry, B., Le Traon, Y.: Mutation analysis testing for model transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 376–390. Springer, Heidelberg (2006). https://doi.org/10.1007/11787044_28
62. Mottu, J.M., Sen, S., Tisi, M., Cabot, J.: Static analysis of model transformations for effective test generation. In: ISSRE, pp. 291–300. IEEE, November 2012
63. Mottu, J.M., Simula, S.S., Cadavid, J., Baudry, B.: Discovering model transformation pre-conditions using automatically generated test models. In: ISSRE, pp. 88–99. IEEE, November 2015
64. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 130–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02674-4_10
65. 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
66. Neema, S., Sztipanovits, J., Karsai, G., Butts, K.: Constraint-based design-space exploration and model synthesis. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 290–305. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45212-6_19

67. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: ICSE, pp. 742–745. ACM (2000)
68. Nicosia, V., Latora, V.: Measuring and modeling correlations in multiplex networks. Phys. Rev. E **92**, 032805 (2015)
69. Nielsen, C.B., Larsen, P.G., Fitzgerald, J.S., Woodcock, J., Peleska, J.: Systems of systems engineering: basic concepts, model-based techniques, and research directions. ACM Comput. Surv. **48**(2), 18 (2015)
70. The Object Management Group: Object Constraint Language, v2.0, May 2006
71. Pennemann, K.-H.: Resolution-like theorem proving for high-level conditions. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 289–304. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_20
72. Pham, M.-D., Boncz, P., Erling, O.: S3G2: a scalable structure-correlated social graph generator. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 156–172. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36727-4_11
73. Przigoda, N., Hilken, F., Peters, J., Wille, R., Gogolla, M., Drechsler, R.: Integrating an SMT-based ModelFinder into USE. In: Model-Driven Engineering, Verification and Validation (MoDeVVa) at MODELS, vol. 1713, pp. 40–45 (2016)
74. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: finite reasoning on UML/OCL conceptual schemas. Data Knowl. Eng. **73**, 1–22 (2012)
75. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes in Theoret. Comp. Sci. **157**(1), 39–59 (2006)
76. Reps, T.W., Sagiv, M., Wilhelm, R.: Static program analysis via 3-valued logic. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 15–30. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_2
77. Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A methodology for verifying refinements of partial models. J. Object Technol. **14**(3), 3:1–3:31 (2015)
78. Salay, R., Chechik, M., Gorzny, J.: Towards a methodology for verifying partial model refinements. In: ICST, pp. 938–945. IEEE (2012)
79. Salay, R., Famelis, M., Chechik, M.: Language independent refinement using partial modeling. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 224–239. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28872-2_16
80. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: a SPARQL performance benchmark. In: ICDE, pp. 222–233. IEEE (2009)
81. Schneider, S., Lambers, L., Orejas, F.: Symbolic model generation for graph properties. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 226–243. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_13
82. Schölzel, H., Ehrig, H., Maximova, M., Gabriel, K., Hermann, F.: Satisfaction, restriction and amalgamation of constraints in the framework of M-adhesive categories. In: Proceedings Seventh ACCAT Workshop on Applied and Computational Category Theory, ACCAT 2012, Tallinn, 1 April 2012. EPTCS, vol. 93, pp. 83–104 (2012)
83. Schonbock, J., Kappel, G., Wimmer, M., Kusel, A., Retschitzegger, W., Schwinger, W.: TETRABox - a generic white-box testing framework for model transformations. In: APSEC, pp. 75–82. IEEE, December 2013
84. Semeráth, O., Barta, Á., Horváth, Á., Szatmári, Z., Varró, D.: Formal validation of domain-specific languages with derived features and well-formedness constraints. Softw. Syst, Model. 16(2), 357–392 (2017)

85. Semeráth, O., Varró, D.: Graph constraint evaluation over partial models by constraint rewriting. In: Guerra, E., van den Brand, M. (eds.) ICMT 2017. LNCS, vol. 10374, pp. 138–154. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61473-1_10
86. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 87–103. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_6
87. Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select models for model transformation testing. In: ICST, pp. 328–337. IEEE (2008)
88. Spasic, M., Jovanovik, M., Prat-Pérez, A.: An RDF dataset generator for the social network benchmark with real-world coherence. In: BLINK (2016)
89. RTCA: DO-178C, software considerations in airborne systems and equipment certification (2012). Technical report
90. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, Reading (2009)
91. Szárnyas, G., Kővári, Z., Salánki, Á., Varró, D.: Towards the characterization of realistic models: evaluation of multidisciplinary graph metrics. In: MODELS, 87–94 (2016)
92. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The train benchmark: cross-technology performance evaluation of continuous model queries. Softw. Syst. Model. (2017). https://doi.org/10.1007/s10270-016-0571-8
93. Sztipanovits, J., Koutsoukos, X., Karsai, G., Kottenstette, N., Antsaklis, P., Gupta, V., Goodwine, B., Baras, J.: Toward a science of cyber-physical system integration. Proc. IEEE **100**(1), 29–44 (2012)
94. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
95. Torrini, P., Heckel, R., Ráth, I.: Stochastic simulation of graph transformation systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12029-9_11
96. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries. Sci. Comput. Program. **98**, 80–99 (2015)
97. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. Softw. Syst. Model. **15**(3), 609–629 (2016)
98. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3), 214–234 (2007)
99. Waltemath, D., et al.: Toward community standards and software for whole-cell modeling. IEEE Trans. Bio-med. Eng. **63**(10), 2007–2014 (2016)
100. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. Electr. Notes Theor. Comput. Sci. **211**, 159–170 (2008)
101. Yakindu Statechart Tools: Yakindu. http://statecharts.org/
102. Zhang, J.W., Tay, Y.C.: GSCALER: synthetically scaling a given graph. In: EDBT, pp. 53–64 (2016). https://doi.org/10.5441/002/edbt.2016.08

Graph Operations and Free Graph Algebras

Uwe Wolter[1(✉)], Zinovy Diskin[2], and Harald König[3]

[1] University of Bergen, Bergen, Norway
Uwe.Wolter@uib.no
[2] McMaster University, Hamilton, Canada
diskinz@mcmaster.ca
[3] FHDW Hannover, Hannover, Germany
Harald.Koenig@fhdw.de

Abstract. We introduce a concept of *graph algebra* that generalizes the traditional concept of algebra in the sense that (1) we use graphs rather than sets as carriers, and (2) we generalize algebraic operations to diagrammatic operations over graphs, which we call *graph operations*.

Our main objective is to extend the construction of term algebras, i.e., free algebras, for the new setting. The key mechanism for the construction of free graph algebras are pushout-based graph transformations for non-deleting injective rules. The application of rules, however, has to be controlled in such a way that "no confusion" arises. For this, we introduce *graph terms* and present a concrete construction of free graph algebras as graph term algebras.

As the main result of the paper, we obtain for any graph signature Γ an adjunction between the category Graph of graphs and the category $\mathsf{GAlg}(\Gamma)$ of graph Γ-algebras. In such a way, we establish an "integrating link" between the two areas Hartmut Ehrig contributed most: algebraic specifications with initial/free semantics and pushout-based graph transformations.

Keywords: Universal Algebra · Term · Term algebra
Free algebra · Graph operation · Graph algebra · Graph term
Graph term algebra · Free graph algebra · Kleisli morphism

1 Introduction

Graph operations have been a key ingredient of the generalized sketch framework, developed in the 90s by a group around the second author and motivated by applications in databases and data modeling [1,3,5]. What was missing, until now, is a proper formal substantiation of the "Kleisli mapping" construct heavily employed in those papers. When we re-launched, ten years later, generalized sketches under the name Diagrammatic Predicate Framework (DPF) [6,12–14], we dropped operations due to the lack of a proper formalization appropriate for our applications in Model Driven Software Engineering. Finally, during his period in Hartmut's group in 1991-00, the first author had always been wondering if one

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 313–331, 2018.
https://doi.org/10.1007/978-3-319-75396-6_17

should look for a uniform mechanism to create names for new items produced by injective graph transformation rules via pushouts.

In the paper, we present a concept of graph algebra that generalizes the traditional concept of algebra in the sense that (1) we use graphs as carriers, instead of sets, and (2) we generalize algebraic operations to graph operations. We introduce graph terms and present a concrete construction of free graph algebras as graph term algebras. As a side effect, graph terms provide a uniform mechanism for the new names problem mentioned above.

As the main result of the paper we obtain for any graph signature Γ an adjunction between the category Graph of graphs and the category $\mathsf{GAlg}(\Gamma)$ of graph Γ-algebras. These adjunctions generalize the traditional adjunctions between the category Set and categories $\mathsf{Alg}(\Sigma)$ of Σ-algebras. The Kleisli categories of the new adjunctions provide the necessary substantiation of the idea of "Kleisli morphisms" of the second author, we have been looking for.

As a pleasant surprise, we realized that the new setting of graph algebras establishes an "integrating link" between the two areas Hartmut Ehrig contributed most - algebraic specifications with initial semantics and graph transformations.

To keep technicalities simple, and to meet the space limitations, we only consider unsorted/untyped signatures and algebras, and leave the straightforward generalization for the many-sorted/typed case for future work. As a running example for a graph signature Γ, we have chosen Γ consisting of arrow composition, identity, initial object, and pullback, which hopefully most of the readers are familiar with.

The paper is organized as follows. In Sect. 2 we recapitulate the basic algebraic concepts signature, operation, algebra, variable, term and term algebra, and we discuss the characterization of term algebras as free algebras. In Sect. 3 we analyze algebraic operations and "diagrammatic" operations, like composition and pullbacks, in the light of graphs, and develop the new concepts graph signature, graph operation and graph algebra. We define corresponding categories $\mathsf{GAlg}(\Gamma)$ of graph algebras for given graph signatures Γ. In Sect. 4, we analyse the construction of terms in the light of graph transformations, and develop the new concepts of a graph term and a graph term algebra. We show that graph term algebras are free graph algebras and discuss applications of this main result. Finally, we discuss related work in Sect. 5 and Sect. 6 outlines different dimensions of further research.

2 Background: Algebras and Term Algebras

An *(algebraic) signature* $\Sigma = (F, ar)$ is given by a finite set F of *operation symbols* and an *arity function* $ar : F \to \mathbb{N}$. A *Σ-algebra* $\mathcal{A} = (A, F^{\mathcal{A}})$ is provided by a *(carrier) set* A, also denoted by $|\mathcal{A}|$, and a family $F^{\mathcal{A}} = (\omega^{\mathcal{A}} : A^{ar(\omega)} \to A \mid \omega \in F)$ of operations. For $n \in \mathbb{N}$ we denote by A^n the set of all n-tuples $\bar{a} = (a_1, \ldots, a_n)$ of elements in A. For $n = 0$ we obtain, in such a way, the singleton set $A^0 = \{()\}$ containing the empty tuple. A symbol $c \in F$ with $ar(c) = 0$ is

also called a *constant symbol*. The corresponding operation $c^{\mathcal{A}} : A^0 \to A$ in a Σ-algebra $\mathcal{A}$ is a "pointer" with the only element () in A^0 pointing to the element $c^{\mathcal{A}}()$ in A.

A Σ-algebra $\mathcal{A}$ is a subalgebra of a Σ-algebra $\mathcal{B}$ if, and only if, $A \subseteq B$ and $\omega^{\mathcal{A}}(\bar{a}) = \omega^{\mathcal{B}}(\bar{a})$ for all $\omega \in F$ and all $\bar{a} \in A^{ar(\omega)} \subseteq B^{ar(\omega)}$. This means that the subset A of the carrier of $\mathcal{B}$ has to be closed under the operations in $\mathcal{B}$. Specifically, A has to contain all the constants $c^{\mathcal{B}}()$ from $\mathcal{B}$.

A *Σ-homomorphism* $f : \mathcal{A} \to \mathcal{B}$ between two Σ-algebras $\mathcal{A}$ and $\mathcal{B}$ is a map $f : A \to B$ such that for every $\omega \in F$, $ar(\omega) = n$ we have $f \circ \omega^{\mathcal{A}} = \omega^{\mathcal{B}} \circ f^n$ where the n'th power $f^n : A^n \to B^n$ of the map f is defined by $f^n(\bar{a}) = (f(a_1), \ldots, f(a_n))$ for all $\bar{a} = (a_1, \ldots, a_n) \in A^n$. That is, for each $\omega \in F$, $ar(\omega) = n$ we require

$$f(\omega^{\mathcal{A}}(a_1, \ldots, a_n)) = \omega^{\mathcal{B}}(f(a_1), \ldots, f(a_n)) \text{ for all } (a_1, \ldots, a_n) \in A^n. \quad (1)$$

$f^0 : A^0 \to B^0$ is the identity on $\{()\}$. Note that requirement (1) for constants $c \in F$ means that constants are mapped to constants: $f(c^{\mathcal{A}}()) = c^{\mathcal{B}}(f^0()) = c^{\mathcal{B}}()$.

The composition $g \circ f : \mathcal{A} \to \mathcal{C}$ of two Σ-homomorphisms $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B} \to \mathcal{C}$ is given by the composition $g \circ f : A \to C$ of the underlying maps $f : A \to B$ and $g : B \to C$. In such a way, Σ-algebras and Σ-homomorphisms constitute a category $\mathsf{Alg}(\Sigma)$, and the assignments $\mathcal{A} \mapsto |\mathcal{A}|$ and $(f : \mathcal{A} \to \mathcal{B}) \mapsto (f : |\mathcal{A}| \to |\mathcal{B}|)$ define a forgetful functor $|\_| : \mathsf{Alg}(\Sigma) \to \mathsf{Set}$.

Example 1 (Natural numbers). We consider the signature $\Sigma = (F, ar)$ with $F = \{\mathsf{z}, \mathsf{s}, \mathsf{p}\}$ consisting of a constant symbol z, $ar(\mathsf{z}) = 0$, a unary operation symbol s, $ar(\mathsf{s}) = 1$, and a binary operation symbol p, $ar(\mathsf{p}) = 2$. As sample Σ-algebra $\mathcal{N} = (\mathbb{N}, F^{\mathcal{N}})$ we consider the natural numbers with a zero, a successor, and a plus operation: $\mathsf{z}^{\mathcal{N}}() = 0$, $\mathsf{s}^{\mathcal{N}} = \_ + 1 : \mathbb{N} \to \mathbb{N}$, $\mathsf{p}^{\mathcal{N}} = \_ + \_ : \mathbb{N}^2 \to \mathbb{N}$.

Let be given an algebraic signature Σ and a set X of variables. Σ-terms on X are strings build of three kinds of symbols: operation symbols from F, variables from X and three auxiliary symbols ",", "$\langle$", "$\rangle$". The inductive definition of terms goes traditionally as follows (compare [8], p. 18):

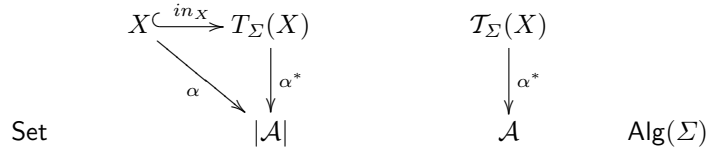
Definition 1 (Terms). *The set $T_\Sigma(X)$ of all Σ-terms on a set X of variables is the smallest set of strings of symbols such that:*
(Variables). $X \subseteq T_\Sigma(X)$,
(Constants). $c\langle\rangle' \in T_\Sigma(X)$ *for all* $c \in F$ *with* $ar(c) = 0$,
(Operations). $\omega\langle t_1, \ldots, t_n\rangle \in T_\Sigma(X)$ *for all operation symbols* $\omega \in F$ *with* $ar(\omega) = n \geq 1$ *and all Σ-terms* $t_1, \ldots, t_n \in T_\Sigma(X)$.

A simple, but crucial, observation is, that the generation of terms can be interpreted as operations in special Σ-algebras (compare [8], p. 67):

Definition 2 (Term algebra). *For a given set X of variables we denote by $\mathcal{T}_\Sigma(X) = (T_\Sigma(X), F^{\mathcal{T}_\Sigma(X)})$ the Σ-algebra of Σ-terms on X with:*
(Constants). $c^{\mathcal{T}_\Sigma(X)}() = c\langle\rangle \in T_\Sigma(X)$ *for all* $c \in F$ *with* $ar(c) = 0$,
(Operations). $\omega^{\mathcal{T}_\Sigma(X)}(t_1, \ldots, t_n) = \omega\langle t_1, \ldots, t_n\rangle \in T_\Sigma(X)$ *for all operation symbols* $\omega \in F$ *with* $ar(\omega) = n \geq 1$ *and all n-tuples* $(t_1, \ldots, t_n) \in T_\Sigma(X)^n$.

That any term is generated in a unique way, is abstractly reflected by the characterization of term algebras as free algebras (compare [8], p. 68):

Proposition 1 (Free algebras). *For each set X of variables the Σ-algebra $\mathcal{T}_\Sigma(X) = (T_\Sigma(X), F^{\mathcal{T}_\Sigma(X)})$ has the following universal property: For any Σ-algebra $\mathcal{A}$ and any variable assignment $\alpha : X \to |\mathcal{A}|$ there exists a unique Σ-homomorphism $\alpha^* : \mathcal{T}_\Sigma(X) \to \mathcal{A}$ such that: $\alpha^* \circ in_X = \alpha$.*



Proposition 1 can be shown by structural induction according to the inductive definition of terms in Definition 1: For the basic case of variables the defining condition forces $\alpha^*(x) = \alpha(x)$ for all $x \in X$. For the basic case of constant symbols the definition of operations in $\mathcal{T}_\Sigma(X)$ and the homomorphism condition entail $\alpha^*(c\langle\rangle) = \alpha^*(c^{\mathcal{T}_\Sigma(X)}()) = c^{\mathcal{A}}()$ for all $c \in F$ with $ar(c) = 0$. And, for the induction step the definition of operations in $\mathcal{T}_\Sigma(X)$ and the homomorphism condition provide the necessary induction/recursion scheme

$$\alpha^*(\omega\langle t_1, \ldots, t_n\rangle) = \alpha^*(\omega^{\mathcal{T}_\Sigma(X)}(t_1, \ldots, t_n)) = \omega^{\mathcal{A}}(\alpha^*(t_1), \ldots, \alpha^*(t_n)) \quad (2)$$

for all operation symbols $\omega \in F$ with $ar(\omega) = n \geq 1$ and all $t_1, \ldots, t_n \in T_\Sigma(X)$.

The universal property determines $\mathcal{T}_\Sigma(X)$ up to isomorphism in $\mathsf{Alg}(\Sigma)$. A Σ-algebra $\mathcal{A}$ is isomorphic to $\mathcal{T}_\Sigma(X)$ iff the following conditions are satisfied:

1. **Generators:** There is an injective mapping $em : X \to |\mathcal{A}|$.
2. **No confusion:** $em(x) \neq \omega^{\mathcal{A}}(\bar{a})$ for any $x \in X$, any operation symbol $\omega \in F$ and any tuple $\bar{a} \in A^{ar(\omega)}$. For any operation symbols $\omega_1, \omega_2 \in F$ and any tuples $\bar{a}_1 \in A^{ar(\omega_1)}$, $\bar{a}_2 \in A^{ar(\omega_2)}$ we have

$$\omega_1^{\mathcal{A}}(\bar{a}_1) \neq \omega_2^{\mathcal{A}}(\bar{a}_2) \quad \text{iff} \quad \omega_1 \neq \omega_2 \text{ or } \bar{a}_1 \neq \bar{a}_2.$$

3. **No junk:** $\mathcal{A}$ has no proper Σ-subalgebra containing $em(X)$.

As any free construction [10], the universal property in Proposition 1 ensures that we can extend the assignments $X \mapsto \mathcal{T}_\Sigma(X)$ to a functor $\mathcal{T}_\Sigma(\_) : \mathsf{Set} \to \mathsf{Alg}(\Sigma)$ that is left-adjoint to the forgetful functor $|\_| : \mathsf{Alg}(\Sigma) \to \mathsf{Set}$. The adjunction

$$\mathsf{Set} \underset{|\_|}{\overset{\mathcal{T}_\Sigma(\_)}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathsf{Alg}(\Sigma)$$

is the fundament for the area of algebraic specifications as the two volumes [8,9] exemplify. Just to mention, that any variant of equational and/or first-order specifications is syntactically based on terms while the semantics relies on the

uniqueness of the evaluation of terms w.r.t. variable assignments. And, not to forget, the Kleisli category of this adjunction provides us a substitution calculus: A substitution of terms for variables is a morphism in the Kleisli category, i.e., a map $\sigma : X \to T_\Sigma(Y)$. The corresponding extended map $\sigma^* : T_\Sigma(X) \to T_\Sigma(Y)$ describes the simultaneous substitution of all variables x in Σ-terms on X by the corresponding terms $\sigma(x) \in T_\Sigma(Y)$. The composition of two substitutions $\sigma : X \to T_\Sigma(Y)$ and $\delta : Y \to T_\Sigma(Z)$ is given by $\delta^* \circ \sigma : X \to T_\Sigma(Z)$.

3 From Algebras to Graph Algebras

As graphs we consider "directed multigraphs" [7]. A graph $G = (G_V, G_E, \mathsf{sr}^G, \mathsf{tg}^G)$ consists of a set G_V of vertices, a set G_E of edges, and two maps $\mathsf{sr}^G, \mathsf{tg}^G : G_E \to G_V$. A homomorphism $\varphi = (\varphi_V, \varphi_E)$ between two graphs $G = (G_V, G_E, \mathsf{sr}^G, \mathsf{tg}^G)$ and $H = (H_V, H_E, \mathsf{sr}^H, \mathsf{tg}^H)$ consists of two maps $\varphi_V : H_V \to G_V$ and $\varphi_E : H_E \to G_E$ such that $\varphi_V \circ \mathsf{sr}^G = \mathsf{sr}^H \circ \varphi_E$ and $\varphi_V \circ \mathsf{tg}^G = \mathsf{tg}^H \circ \varphi_E$.

The identity graph homomorphism on a graph G is the pair $id_G = (id_{G_V}, id_{G_E})$ of identity maps and graph homomorphisms are composed componentwise. By Graph we denote the category with graphs as objects and graph homomorphisms as morphisms. To establish the concept of graph algebras, we need, first, an adequate concept of signature.

Definition 3 (Graph signature). *A* graph signature $\Gamma = (OP, I, R)$ *is given by a finite set OP of operation symbols and two maps I and R assigning to each operation symbol $\omega \in OP$ a finite graph $I(\omega)$, its* input arity*, and a finite graph $R(\omega)$, its* result arity*, respectively. Moreover, we assume that there is an inclusion $\iota_\omega : I(\omega) \hookrightarrow R(\omega)$ between the two arity graphs.*

To substantiate this definition, we discuss, in more detail, the transition from algebraic signatures to graph signatures.

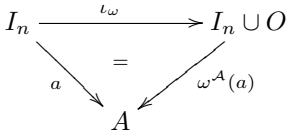
Let $I_n = \{in_1, \ldots, in_n\}$ be a set of indices. We can consider an n-tuple $\bar{a} = (a_1, \ldots, a_n)$ of elements from a set A as a representation of a map $a : I_n \to A$ where $a_i = a(in_i)$ for all $1 \leq i \leq n$. The empty tuple () represents, in this view, the unique map from the empty set $I_0 = \emptyset$ into A.

For any $n \in \mathbb{N}$ there is a bijection between A^n and the set A^{I_n} of all maps from I_n into A, thus we can consider maps $a \in A^{I_n}$ as inputs for operations in a Σ-algebra $\mathcal{A}$. What about the output? Algebraic operations give only a single value as output thus we can consider the codomain of an operation in $\mathcal{A}$ as the set A^O of all maps from a singleton $O = \{out\}$ into A. From this viewpoint, we can consider the declaration of an operation symbol ω with $ar(\omega) = n$ as declaring a span $I_n \hookleftarrow \emptyset \hookrightarrow O$ of set inclusions, where the corresponding operation in a Σ-algebra $\mathcal{A}$ would be a map from A^{I_n} into A^O.

Operations are assumed, however, to have no side effects. This means that the input is neither deleted nor changed. In such a way, we can consider the declaration of the arity of an operation symbol ω as declaring a set inclusion

$\iota_\omega : I_n \hookrightarrow I_n \cup O$ (obtained by pushing out the above span of inclusions). The corresponding operation in $\mathcal{A}$ becomes then a map

$$\omega^{\mathcal{A}} : A^{I_n} \longrightarrow A^{I_n \cup O} \quad \text{such that } a = \omega^{\mathcal{A}}(a) \circ \iota_\omega \text{ for all } a \in A^{I_n}. \tag{3}$$

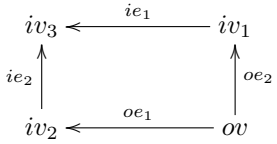


We can recognize the same pattern in "graph operations", like composition of morphisms and limit/colimit constructions in categories, for example. There are, however, three essential differences to the case of algebraic operations:

1. There are two different kinds of input items, namely vertices and edges.
2. Operations can produce arbitrary many output items instead of exactly one.
3. To relate output edges in an appropriate way to the input, we have to work with non-empty "boundary graphs" instead of just the empty set.

As an example we consider the construction of pullbacks. Let a category C with pullbacks be given and let $|C|$ denote the underlying graph of C. To turn the existence of pullbacks into an operation, we have to choose for any cospan $A \xrightarrow{f} C \xleftarrow{g} B$ in C one of the existing pullbacks, i.e., we have to choose an object D and morphisms $g^* : D \to A$, $f^* : D \to B$ such that the resulting square is a pullback in C.

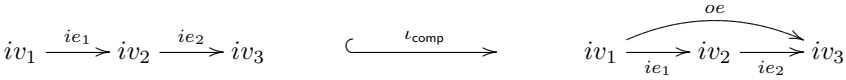
The input arity of a corresponding operation symbol pb can be described by the graph $\mathsf{Cospan} = (iv_1 \xrightarrow{ie_1} iv_3 \xleftarrow{ie_2} iv_2)$, i.e., a cospan in C is a graph homomorphism from Cospan into $|C|$. Here, "iv" stands for *input vertex* while "ie" refers to *input edge*. We will often use the term *binding* for these graph homomorphisms. The output arity of the operation could be described by the graph $\mathsf{Span} = (iv_1 \xleftarrow{oe_2} ov \xrightarrow{oe_1} iv_2)$, where "$ov$" stands for *output vertex* while "oe" refers to *output edge*. The "boundary graph" $\underline{2}$, consisting of the two vertices iv_1 and iv_2, connects the output items with the input items. Instead of a span of graph inclusions $\mathsf{Cospan} \hookleftarrow \underline{2} \hookrightarrow \mathsf{Span}$ we consider, however, the inclusion ι_{pb} of the graph Cospan into the graph Square,



obtained by pushing out the above span of graph inclusions, as the declaration of the arity of the operation symbol pb.

Convention 4 (Graph signature). *For notational convenience we use canonical names for input vertices and edges. For a given operation symbol $\omega \in OP$, we denote the elements of $I(\omega)_V$ by $\{iv_1, \ldots, iv_{nv_\omega}\}$ and the elements of $I(\omega)_E$ by $\{ie_1, \ldots, ie_{ne_\omega}\}$, where nv_ω and ne_ω are the numbers of vertices and edges in $I(\omega)$ resp. Output vertices and edges will be denoted by ov_i and oe_j.*

Example 2 (Graph signature). We consider a graph signature $\Gamma = (OP, I, R)$ with $OP = \{\mathsf{pb}, \mathsf{comp}, \mathsf{id}, \mathsf{ini}\}$. For the operation symbol pb we declare $I(\mathsf{pb}) = \mathsf{Cospan}$ and $R(\mathsf{pb}) = \mathsf{Square}$. The arity of the composition operation symbol comp is given by the following inclusion of graphs

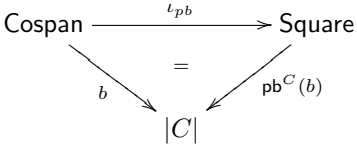


The input arity of id is the graph $\underline{1}$ with exactly one vertex iv and the result arity is the graph Loop with exactly one vertex iv and one edge oe. Finally, the input arity of ini is the *empty graph* $\underline{\emptyset}$, and the result arity is a graph $\underline{1}$ with exactly one vertex ov. That is, ini is a constant symbol with a trivial result arity, but in general the result arity of a constant could be any finite graph!

For graphs G and H we denote by G^H the set of all graph homomorphisms from H into G. A fixed choice of pullbacks in category C gives rise to a map

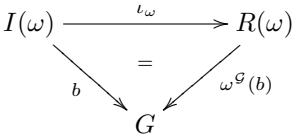
$$\mathsf{pb}^C : |C|^{\mathsf{Cospan}} \longrightarrow |C|^{\mathsf{Square}}$$

such that $b = pb^C(b) \circ \iota_{pb}$ for all bindings $b : \mathsf{Cospan} \to |C|$.



Generalizing the pullback example, we coin now the new concept of graph algebra.

Definition 5 (Graph algebra). *For a graph signature $\Gamma = (OP, I, R)$ a Γ-*algebra $\mathcal{G} = (G, OP^{\mathcal{G}})$ *is given by a* (carrier) graph $G = (G_V, G_E, \mathsf{sr}^G, \mathsf{tg}^G)$, *also denoted by $|\mathcal{G}|$, and a family $OP^{\mathcal{G}} = (\omega^{\mathcal{G}} : G^{I(\omega)} \to G^{R(\omega)} \mid \omega \in OP)$ of maps such that $b = \omega^{\mathcal{G}}(b) \circ \iota_\omega$ for all $\omega \in OP$ and all $b \in G^{I(\omega)}$. These maps will be called* graph operations.



For a vertex $v \in R(\omega)_V$ and edge $e \in R(\omega)_E$, we write $\omega^{\mathcal{G}}_V(b)(v)$ and $\omega^{\mathcal{G}}_E(b)(e)$ rather than $\omega^{\mathcal{G}}(b)_V(v)$ and $\omega^{\mathcal{G}}(b)_E(e)$. This eases reading formulas with b defined by long tuples. We will also omit V, E subindices if it eases reading formulas.

For any graph G there is exactly one graph homomorphism $\underline{\emptyset}_G : \underline{\emptyset} \to G$, i.e., $G^{\underline{\emptyset}} = \{\underline{\emptyset}_G\}$ is a singleton, thus for any constant symbol $c \in OP$, i.e., $I(c) = \underline{\emptyset}$, the corresponding graph operation in a graph Γ-algebra $\mathcal{G}$ just points at a subgraph of G, namely the image of $R(c)$ w.r.t. $c^{\mathcal{G}}(\underline{\emptyset}_G)$.

Example 3 (Graph algebra). The composition in a category C can be presented as a graph operation $\mathsf{comp}^C : |C|^{I(\mathsf{comp})} \to |C|^{R(\mathsf{comp})}$ where the only output of the graph operation is given by $\mathsf{comp}^C_E(b)(oe) = b(ie_2) \circ b(ie_1)$ for all bindings $b : I(\mathsf{comp}) \to |C|$. Note, that comp^C is a total operation. The identity in C gives another graph operation $\mathsf{id}^C : |C|^{\underline{1}} \to |C|^{\mathsf{Loop}}$ such that $\mathsf{id}^C_E(b)(oe) = id_{b(iv)}$.

If C has pullbacks, we can define a graph operation $\mathsf{pb}^C : |C|^{\mathsf{Cospan}} \to |C|^{\mathsf{Square}}$, in such a way, that for any binding $b : \mathsf{Cospan} \to |C|$ the result $\mathsf{pb}^C(b) : \mathsf{Square} \to |C|$ is a chosen pullback diagram. And, if C has initial objects, we can define a constant $\mathsf{ini}^C : |C|^{\underline{\emptyset}} \to |C|^{\underline{1}}$, in such a way, that $\mathsf{ini}^C_V(\underline{\emptyset}_G)(ov)$ is a (chosen) initial object in C.

A Γ-algebra $\mathcal{G}$ is a subalgebra of a Γ-algebra $\mathcal{H}$ if G is a subgraph of H and $in \circ \omega^{\mathcal{G}}(b) = \omega^{\mathcal{H}}(in \circ b)$ for all $\omega \in OP$ and all $b \in G^{I(\omega)}$, where $in : G \to H$ is the corresponding inclusion graph homomorphism (compare Definition 6). This means that the subgraph G of the carrier of $\mathcal{H}$ has to be closed under the operations in $\mathcal{H}$ in the sense that for all $\omega \in OP$ and all $b \in G^{I(\omega)}$ the image $\omega^{\mathcal{H}}(in \circ b)(R(\omega))$ is a subgraph of G. Especially, G has to contain the image graph $c^{\mathcal{H}}(\underline{\emptyset}_H)(R(c))$ for any constant symbol c in OP.

A functor $\mathfrak{F} : C \to D$ between two categories C and D is a graph homomorphism $\mathfrak{F} : |C| \to |D|$ compatible with composition, i.e., for all morphisms $f : A \to B$, $g : B \to C$ in C we have $\mathfrak{F}(g \circ f) = \mathfrak{F}(g) \circ \mathfrak{F}(f)$. We can reformulate this condition in terms of the corresponding graph operations $comp^C$ and $comp^D$ by requiring that the following diagram commutes:

$$\begin{array}{ccc} |C|^{I(\mathsf{comp})} & \xrightarrow{\mathsf{comp}^C} & |C|^{R(\mathsf{comp})} \\ {\scriptstyle \mathfrak{F}\circ\_}\downarrow & = & \downarrow{\scriptstyle \mathfrak{F}\circ\_} \\ |D|^{I(\mathsf{comp})} & \xrightarrow{\mathsf{comp}^D} & |D|^{R(\mathsf{comp})} \end{array}$$

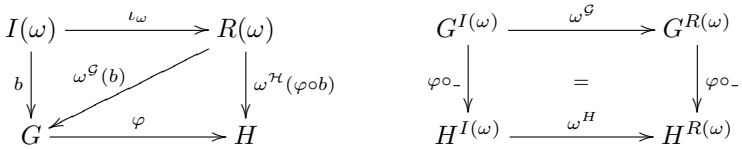
That is, for any binding $b : I(\mathsf{comp}) \to |C|$ we require (compare (1))

$$\mathfrak{F} \circ \mathsf{comp}^C(b) = \mathsf{comp}^D(\mathfrak{F} \circ b).$$

This example motivates our concept of homomorphisms between graph algebras.

Definition 6. *A* Γ-homomorphism $\varphi : \mathcal{G} \to \mathcal{H}$ *between two* Γ*-algebras* $\mathcal{G} = (G, OP^{\mathcal{G}})$ *and* $\mathcal{H} = (H, OP^{\mathcal{H}})$ *is a graph homomorphism* $\varphi : G \to H$ *such that*

$$\varphi \circ \omega^{\mathcal{G}}(b) = \omega^{\mathcal{H}}(\varphi \circ b) \quad \textit{for all } \omega \in OP \textit{ and all } b \in G^{I(\omega)}. \tag{4}$$



Example 4. Given two categories C and D with pullback operations, a functor $\mathfrak{F} : C \to D$, that preserves pullbacks, establishes a graph algebra homomorphism only if it maps chosen pullbacks in C to chosen pullbacks in D.

The composition $\psi \circ \varphi : \mathcal{G} \to \mathcal{K}$ of Γ-homomorphisms $\varphi : \mathcal{G} \to \mathcal{H}$ and $\psi : \mathcal{H} \to \mathcal{K}$ is given by the composition $\psi \circ \varphi : G \to K$ of the underlying graph homomorphisms $\varphi : G \to H$ and $\psi : H \to K$. For any graph Γ-algebra $\mathcal{G}$ the identity Γ-homomorphism $id_{\mathcal{G}} : \mathcal{G} \to \mathcal{G}$ is given by the identity graph homomorphism $id_G : G \to G$. In such a way, Γ-algebras and Γ-homomorphisms constitute a category $\mathsf{GAlg}(\Gamma)$ where the assignments $\mathcal{G} \mapsto |\mathcal{G}|$ and $(\varphi : \mathcal{G} \to \mathcal{H}) \mapsto (\varphi : |\mathcal{G}| \to |\mathcal{H}|)$ define a forgetful functor $|\_| : \mathsf{GAlg}(\Gamma) \to \mathsf{Graph}$.

We conclude this section with a discussion how algebras can be transformed into corresponding graph algebras. Any set A can be transformed into a graph $\mathfrak{V}(A)$ with an empty set of edges, and any map $f : A \to B$ provides trivially a graph homomorphism $\mathfrak{V}(f) : \mathfrak{V}(A) \to \mathfrak{V}(B)$ thus the assignments $A \mapsto \mathfrak{V}(A)$ and $(f : A \to B) \mapsto (\mathfrak{V}(f) : \mathfrak{V}(A) \to \mathfrak{V}(B))$ define a functor $\mathfrak{V} : \mathsf{Set} \to \mathsf{Graph}$.

Turning back to the discussion, at the beginning of this section, it becomes obvious that we can transform any algebraic signature $\Sigma = (F, ar)$ into a graph signature $\Gamma_\Sigma = (F, I_\Sigma, R_\Sigma)$ with $I_\Sigma(\omega) = \mathfrak{V}(I_{ar(\omega)})$ and $R_\Sigma(\omega) = \mathfrak{V}(I_{ar(\omega)} \cup \{ov\})$. It is easy to see that any Σ-algebra can be transformed into a Γ_Σ algebra $\mathfrak{G}(\mathcal{A})$, and any Σ-algebra homomorphism $f : \mathcal{A} \to \mathcal{B}$ gives rise to a Γ_Σ-homomorphism $\mathfrak{V}(f) : \mathfrak{G}(\mathcal{A}) \to \mathfrak{G}(\mathcal{B})$.

Finally, the assignments $\mathcal{A} \mapsto \mathfrak{G}(\mathcal{A})$ and $(f : \mathcal{A} \to \mathcal{B}) \mapsto (\mathfrak{V}(f) : \mathfrak{G}(\mathcal{A}) \to \mathfrak{G}(\mathcal{B}))$ define an embedding $\mathfrak{G} : \mathsf{Alg}(\Sigma) \to \mathsf{GAlg}(\Gamma_\Sigma)$ where we have, by construction, that $|\_| \circ \mathfrak{G} = \mathfrak{V} \circ |\_|$ (see Fig. 1). Note, that $\mathsf{Alg}(\Sigma)$ and $\mathsf{GAlg}(\Gamma_\Sigma)$ are, in general, neither isomorphic nor equivalent since the carrier of a Γ_Σ-algebra can have edges even if the operations only work on vertices.

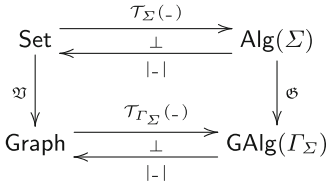


Fig. 1. Two compatible adjunctions

In the next section we will discuss the construction of free Γ-algebras for arbitrary graph signatures Γ providing a functor $\mathcal{T}_\Gamma(\_) : \mathsf{Graph} \to \mathsf{GAlg}(\Gamma)$ to be shown to be left adjoint to the forgetful functor $|\_| : \mathsf{GAlg}(\Gamma) \to \mathsf{Graph}$. This construction should generalize the construction of term algebras, in the sense, that for any algebraic signature Σ there is a natural isomorphism between the two functors $\mathfrak{G} \circ \mathcal{T}_\Sigma(\_)$ and $\mathcal{T}_{\Gamma_\Sigma}(\_) \circ \mathfrak{V}$ from Set into $\mathsf{GAlg}(\Gamma_\Sigma)$.

4 From Terms to Graph Terms

To have a guideline how to define terms in the setting of graph algebras, we analyze the construction of terms in Definition 1 in the light a graph signatures. As example we consider the algebraic signature Σ in Example 1.

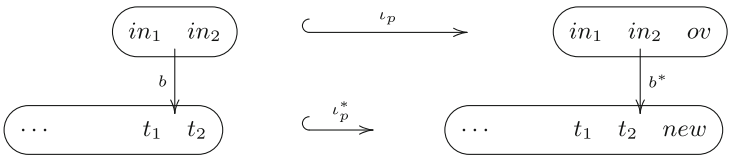


Fig. 2. Term construction as pushout

For the corresponding graph signature Γ_Σ the graph inclusion $\iota_\mathsf{p} : I_{\Gamma_\Sigma}(\mathsf{p}) \to R_{\Gamma_\Sigma}(\mathsf{p})$ is depicted in the upper part of the diagram in Fig. 2. In the left lower corner we depict the set of terms that have been constructed until now. Applying rule 3 in Definition 1 for $\omega = \mathsf{p}$ and two already constructed terms t_1, t_2 means to apply ι_p, considered as a graph transformation rule, for the binding $b = (in_1 \mapsto t_1, in_2 \mapsto t_2)$ and to construct a pushout, i.e., to produce exactly one new vertex new, as depicted in the lower right corner in Fig. 2. Denoting this new item by the term $\mathsf{p}\langle t_1, t_2\rangle$, solves two problems:

1. The term notation provides a uniform mechanism to create identifiers for new graph items introduced by applying non-deleting injective graph transformation rules (at least for graphs without edges). This problem is seldom addressed in the graph transformation literature.
2. The term $\mathsf{p}\langle t_1, t_2\rangle$ codes all the information about the pushout that has been creating the new item:
   (a) The symbol "p" identifies the rule that has been applied and, by consulting the signature, we find the necessary information about the input and result arity, respectively.
   (b) The string "t_1, t_2" codes the actual binding (match) $b = (in_1 \mapsto t_1, in_2 \mapsto t_2)$ for the input arity.
   (c) Since there is exactly one new item, we do have all information to identify uniquely the new item, and thus to define the resulting binding b^*.

In such a way, the term notation offers two possibilities to deal with the problem of applying the same rule twice for the same binding:

1. *A priori:* Before we apply a rule for a certain binding, we check the term denotations of all the items that have already been constructed. In such a way, we can find out, if the rule had already been applied for this binding. If this is the case, we do not apply the rule. Note, that the idea to use a rule as its own negative application condition [7], would be too rigid here. In case of the operation p, e.g., we couldn't apply ι_{p} to any graph with more than two vertices.
2. *A posteriori:* After applying a rule another time for a certain binding we repair the mistake silently by identifying the newly generated items with the "same" already existing items by the assumption that sets are extensional.

We like to adapt the silent a posteriori reparation mechanism and extend the term notation correspondingly. To deal with the rules arising from declaring arities of graph operations (see Definition 3) we have to address two problems: (1) An item can be of two different kinds - vertex or edge, and (2) there can be any finite number of output items instead of exactly one. To tackle problem (1), we will use for each kind a separate string of given terms, instead of just one string. And, by using output items as additional parts of terms, we solve problem (2).

In the context of graph algebras, we consider a collection of variables to be a graph rather than a set. Given a graph signature $\Gamma = (OP, I, R)$ and a graph X of variables, we will define (graph) Γ-terms over X using the following symbols/names: operation symbols from OP, names of output vertices and edges in $R(\omega) \setminus I(\omega)$ for all $\omega \in OP$, and auxiliary symbols like commas and brackets.

Convention 7. *For a graph G, operation symbol $\omega \in OP$, and binding $b : I(\omega) \to G$, we write the strings "$b(iv_1) \dots b(iv_{nv_\omega})$" and "$b(ie_1) \dots b(ie_{ne_\omega})$" of, resp., vertices and edges in G without commas and brackets, denote them by $\overline{b}_V$ and $\overline{b}_E$ resp., and write $\overline{b}$ for $\overline{b}_V; \overline{b}_E$. Extensionality ensures that $b1 = b2$ iff $\overline{b1} = \overline{b2}$ so that we can omit the overline bar.*

Now we are prepared to define graph terms in parallel to the traditional definition of terms in Definition 1.

Definition 8 (Graph terms). *Let be given a graph signature $\Gamma = (OP, I, R)$ and a graph X of variables. The graph $T_\Gamma(X)$ of all* graph Γ-terms *on X is the smallest graph, which satisfies the following three conditions:*
(Variables). *$T_\Gamma(X)$ contains the graph of variables, $X \sqsubseteq T_\Gamma(X)$;*
(Constants). *For all $c \in OP$ with $I(c) = \underline{\emptyset}$, graph $T_\Gamma(X)$ contains*

- *for each $ov \in R(c)_V$, tuple $\langle ov, c, \langle\rangle\rangle$ as a vertex,*
- *for each $oe \in R(c)_E$, tuple $\langle oe, c, \langle\rangle\rangle$ as an edge, where*

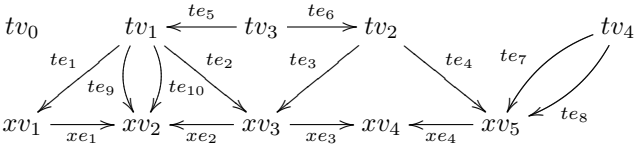
$sc^{T_\Gamma(X)}(\langle oe, c, \langle\rangle\rangle) = \langle sc^{R(c)}(oe), c, \langle\rangle\rangle$ and $tg^{T_\Gamma(X)}(\langle oe, c, \langle\rangle\rangle) = \langle tg^{R(c)}(oe) c \langle\rangle\rangle$;
(Operations) *For all $\omega \in OP$ with $I(\omega) \neq \underline{\emptyset}$ and any $b : I(\omega) \to T_\Gamma(X)$, graph $T_\Gamma(X)$ contains*

– *for each* $ov \in R(\omega)_V \setminus I(\omega)_V$, *tuple* $\langle ov, \omega, b\rangle$ *as a vertex*[1],
– *for each* $oe \in R(\omega)_E \setminus I(\omega)_E$, *tuple* $\langle oe, \omega, b\rangle$ *as an edge*[2], *whose source and target vertices are defined as follows:*

$$\mathsf{sr}^{T_\Gamma(X)}(\langle oe, \omega, b\rangle) = \begin{cases} b(\mathsf{sr}^{R(\omega)}(oe)) & \text{if } \mathsf{sr}^{R(\omega)}(oe) \in I(\omega)_V \\ \langle \mathsf{sr}^{R(\omega)}(oe), \omega, b\rangle & \text{if } \mathsf{sr}^{R(\omega)}(oe) \notin I(\omega)_V \end{cases}$$

$$\mathsf{tg}^{T_\Gamma(X)}(\langle oe, \omega, b\rangle) = \begin{cases} b(\mathsf{tg}^{R(\omega)}(oe)) & \text{if } \mathsf{tg}^{R(\omega)}(oe) \in I(\omega)_V \\ \langle \mathsf{tg}^{R(\omega)}(oe), \omega, b\rangle & \text{if } \mathsf{tg}^{R(\omega)}(oe) \notin I(\omega)_V \end{cases}$$

Example 5. As an example we consider the graph signature Γ from Example 2 and the graph X depicted in the last line in the following diagram.



Vertex tv_0 is generated by the rule ι_{ini}, i.e., $tv_0 = \langle ov, ini, \langle\rangle\rangle$. Vertices $tv_1, \ldots, tv_4$ and edges $te_1, \ldots, te_8$ are generated by the following four applications bi, $i = 1..4$ of the rule ι_{pb} (as there are no isolated vertices in the arity of pb, it's sufficient to specify the values of bindings on edges):

	$b1$	$b2$	$b3$	$b4$
ie_1	xe_1	xe_3	te_2	xe_4
ie_2	xe_2	xe_4	te_3	xe_4

which produce

$tv_1 = \langle ov, \mathsf{pb}, b1\rangle$ $\quad te_1 = \langle oe_2, \mathsf{pb}, b1\rangle$ $\quad te_2 = \langle oe_1, \mathsf{pb}, b1\rangle$
$tv_2 = \langle ov, \mathsf{pb}, b2\rangle$ $\quad te_3 = \langle oe_2, \mathsf{pb}, b2\rangle$ $\quad te_4 = \langle oe_1, \mathsf{pb}, b2\rangle$
$tv_3 = \langle ov, \mathsf{pb}, b3\rangle$ $\quad te_5 = \langle oe_2, \mathsf{pb}, b3\rangle$ $\quad te_6 = \langle oe_1, \mathsf{pb}, b3\rangle$
$tv_4 = \langle ov, \mathsf{pb}, b4\rangle$ $\quad te_7 = \langle oe_2, \mathsf{pb}, b4\rangle$ $\quad te_8 = \langle oe_1, \mathsf{pb}, b4\rangle$

Note, that the edge pair te_7 and te_8 could be declared as kernel of edge xe_4. Finally, edges te_9 and te_{10} are obtained by two applications of rule ι_{comp}: $b5(ie_1) = te_1$, $b5(ie_2) = xe_1$, and $b6(ie_1) = te_2$, $b5(ie_2) = xe_2$, which produce $te_9 = \langle oe, \mathsf{comp}, b5\rangle$, $te_{10} = \langle oe, \mathsf{comp}, b6\rangle$.

Analogously, to the case of terms, we can interpret the construction of graph terms as operations in special Γ-algebras:

[1] To show analogy with Definition 1 clearer, we could denote such tuples as $\omega_{ov}\langle b(iv_1), \ldots, b(iv_{nv_\omega});\ b(ie_1), \ldots, b(ie_{ne_\omega})\rangle$.
[2] Dito for $\omega_{oe}\langle b(iv_1), \ldots, b(iv_{nv_\omega});\ b(ie_1), \ldots, b(ie_{ne_\omega})\rangle$.

Definition 9 (Graph term algebra). *For a graph X of variables we denote by $\mathcal{T}_\Gamma(X) = (T_\Gamma(X), OP^{\mathcal{T}_\Gamma(X)})$ the Γ-algebra of graph Γ-terms on X with:*
(Constants). *For all $c \in OP$ with $I(c) = \underline{\emptyset}$*

$$c_V^{\mathcal{T}_\Gamma(X)}(\underline{\emptyset}_G)(ov) = \langle ov, c, \langle\rangle\rangle \quad \text{for all } ov \in R(c)_V,$$

$$c_E^{\mathcal{T}_\Gamma(X)}(\underline{\emptyset}_G)(oe) = \langle oe, c, \langle\rangle\rangle \quad \text{for all } oe \in R(c)_E;$$

(Operations). *For all $\omega \in OP$ with $I(\omega) \neq \underline{\emptyset}$ and all $b \in T_\Gamma(X)^{I(\omega)}$*

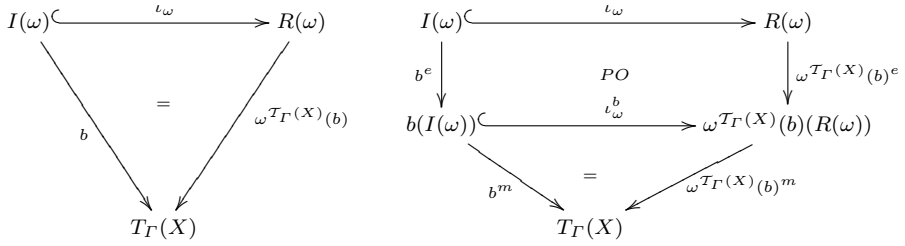
$$\omega_V^{\mathcal{T}_\Gamma(X)}(b)(v) = \begin{cases} b_V(v) & , \text{if } v \in I(\omega)_V \\ \langle v, \omega, b\rangle & , \text{if } v \in R(\omega)_V \setminus I(\omega)_V \end{cases}$$

$$\omega_E^{\mathcal{T}_\Gamma(X)}(b)(e) = \begin{cases} b_E(e) & , \text{if } e \in I(\omega)_E \\ \langle e, \omega, b\rangle & , \text{if } e \in R(\omega)_E \setminus I(\omega)_E \end{cases}$$

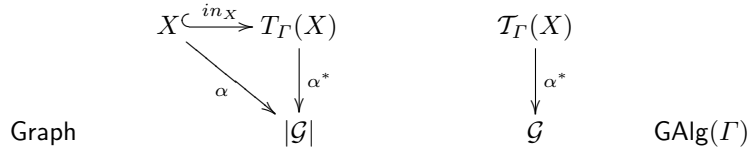
The definitions ensure that all resulting bindings $\omega^{\mathcal{T}_\Gamma(X)}(b) \in T_\Gamma(X)^{R(\omega)}$ are indeed graph homomorphisms and that $b = \omega^{\mathcal{T}_\Gamma(X)}(b) \circ \iota_\omega$, as required.

The condition "the smallest graph" in Definition 8 ensures that $\mathcal{T}_\Gamma(X)$ has "no junk", i.e., no proper subalgebra containing X, and the graph term notation ensures that there is "no confusion", i.e., variables are not identified with items introduced by operation applications. Moreover, items, introduced by different operation applications, are not identified either.

More structurally, "no confusion" means, especially, that for any $\omega \in OP$ and any $b \in T_\Gamma(X)^{I(\omega)}$, the commutative triangle below (on the left) factorizes, by epi-mono-factorization $b = b^m \circ b^e$ and $\omega^{\mathcal{T}_\Gamma(X)}(b) = \omega^{\mathcal{T}_\Gamma(X)}(b)^m \circ \omega^{\mathcal{T}_\Gamma(X)}(b)^e$, into a pushout square and a commutative triangle (below on the right).



Proposition 2 (Free graph algebras). *For each graph X the graph term Γ-algebra $\mathcal{T}_\Gamma(X) = (T_\Gamma(X), OP^{\mathcal{T}_\Gamma(X)})$ has the following universal property: For any Γ-algebra $\mathcal{G}$ and any variable assignment $\alpha : X \to |\mathcal{G}|$ there exists a unique Γ-homomorphism $\alpha^* : \mathcal{T}_\Gamma(X) \to \mathcal{G}$ such that: $\alpha^* \circ in_X = \alpha$.*



Proof. We prove by structural induction according to Definition 8:
(Variables). In this basic case the defining condition forces $\alpha^*_V(xv) = \alpha_V(xv)$ for all $xv \in X_V$ and $\alpha^*_E(xe) = \alpha_E(xe)$ for all $xe \in X_E$.
(Constants). In this basic case the definition of operations in $\mathcal{T}_\Gamma(X)$ and the desired homomorphism condition for α^* forces for all $ov \in R(c)$

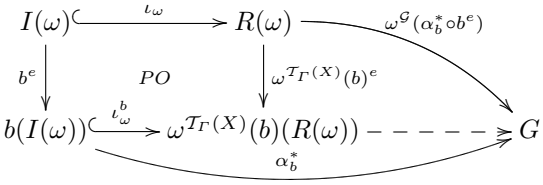
$$\alpha^*_V(\langle ov, c, \langle\rangle\rangle) = \alpha^*_V(c_V^{\mathcal{T}_\Gamma(X)}(\underline{\emptyset}_{T_\Gamma(X)})(ov)) = c_V^{\mathcal{G}}(\alpha^* \circ \underline{\emptyset}_{T_\Gamma(X)})(ov) = c_V^{\mathcal{G}}(\underline{\emptyset}_{\mathcal{G}})(ov)$$

and for all $oe \in R(c)_E$ we get, analogously, $\alpha^*_E(\langle oe, c, \langle\rangle\rangle) = c_E^{\mathcal{G}}(\underline{\emptyset}_{\mathcal{G}})(oe)$.
(Operations). The definition of operations in $\mathcal{T}_\Gamma(X)$ and the desired homomorphism condition forces α^* to be defined according to a corresponding recursion scheme for all $\omega \in OP$ with $I(\omega) \neq \underline{\emptyset}$ and all $b \in T_\Gamma(X)^{I(\omega)}$: The induction hypothesis is that α^* is already defined on the subgraph $b(I(\omega)) \sqsubseteq T_\Gamma(X)$. We denote the restriction of α^* to $b(I(\omega))$ by α^*_b. In the induction step we extend α^* to the subgraph $\omega^{\mathcal{T}_\Gamma(X)}(b)(R(\omega)) \sqsubseteq T_\Gamma(X)$ (that contains $b(I(\omega))$), i.e., to all graph terms that have been constructed exactly by applying rule ι_ω for the binding b: For all $ov \in R(\omega)_V$ we get

$$\alpha^*_V(\langle ov, \omega, b\rangle) = \alpha^*_V(\omega_V^{\mathcal{T}_\Gamma(X)}(b)(ov)) = \omega_V^{\mathcal{G}}(\alpha^*_b \circ b^e)(ov)$$

and for all $oe \in R(\omega)_E$ we get $\alpha^*_E(\langle oe, \omega, b\rangle) = \omega_E^{\mathcal{G}}(\alpha^*_b \circ b^e)(oe)$.

More structurally considered, the induction step constructs the unique mediating morphism from $\omega^{\mathcal{T}_\Gamma(X)}(b)(R(\omega))$ into G in the following diagram (Keep in mind that $\alpha^* \circ b^e = \omega^{\mathcal{G}}(\alpha^*_b \circ b^e) \circ \iota_\omega$ since $\omega^{\mathcal{G}}$ is a graph operation.):



A more traditional presentation of the induction step, analogously to (2), can be given if we use for the binding $b \in T_\Gamma(X)^{I(\omega)}$ the abbreviations $tv_j = b(iv_j)$, $1 \leq j \leq nv_\omega$ and $te_k = b(ie_k)$, $1 \leq k \leq ne_\omega$ (compare Convention 7), consider tuples as presentations of finite maps, as discussed at the beginning of Sect. 3, and represent the two maps constituting a binding for $I(\omega)$ in G as a sequence of vertices and edges of length $nv_\omega + ne_\omega$ in G: For all $ov \in R(\omega)_V$, we get

$$\begin{aligned}&\alpha^*_V(\langle ov, \omega, tv_1 \ldots tv_{nv_\omega} te_1 \ldots te_{ne_\omega}\rangle)\\ &= \omega_V^{\mathcal{G}}(\alpha^*_V(tv_1) \ldots \alpha^*_V(tv_{nv_\omega})\alpha^*_E(te_1) \ldots \alpha^*_E(te_{ne_\omega}))(ov).\end{aligned}$$

The universal property in Proposition 2 ensures that we can extend the assignments $X \mapsto \mathcal{T}_\Gamma(X)$ to a functor $\mathcal{T}_\Gamma(\_) : \mathsf{Graph} \to \mathsf{GAlg}(\Gamma)$ that is left-adjoint to the forgetful functor $|\_| : \mathsf{GAlg}(\Gamma) \to \mathsf{Graph}$ (see Fig. 1). That the adjunction $\mathcal{T}_\Gamma(\_) \dashv |\_|$ generalizes the construction of term algebras, in the sense, that for any algebraic signature Σ there is a natural isomorphism between the

two functors $\mathfrak{G} \circ \mathcal{T}_\Sigma(\_)$ and $\mathcal{T}_{\Gamma_\Sigma}(\_) \circ \mathfrak{V}$ from Set into $\mathsf{GAlg}(\Gamma_\Sigma)$(see Fig. 1) can be shown straightforwardly.

Establishing the adjunctions $\mathcal{T}_\Gamma(\_) \dashv |\_|$ is the main result of the paper. Since the new adjunctions generalize the adjunctions $\mathcal{T}_\Sigma(\_) \dashv |\_|$, we will be able to transfer smoothly many concepts, constructions, and results from the area of algebraic specifications to the new setting of graph algebras (see Sect. 6).

Equations, for example, can be defined as pairs of graph terms and can be used to formulate properties of graph operations. Associativity of composition, e.g., can be expressed by the equation (we recall Convention 7 about denotations of binding mappings)

$$\langle oe, \mathsf{comp}, \langle oe, \mathsf{comp}, xe_1 xe_2 \rangle xe_3 \rangle = \langle oe, \mathsf{comp}, xe_1 \langle oe, \mathsf{comp}, xe_2 xe_3 \rangle \rangle$$

where X is the graph $(xv_1 \overset{xe_1}{\rightarrow} xv_2 \overset{xe_2}{\rightarrow} xv_3 \overset{xe_3}{\rightarrow} xv_4)$. Since there are no isolated vertices in the arities of our sample operations we list only edge variables in the sample equations. We may also require that our choice of pullbacks is symmetric in the sense that the following equations are satisfied:

$$\begin{aligned} \langle ov, \mathsf{pb}, xe_1 xe_2 \rangle &= \langle ov, \mathsf{pb}, xe_2 xe_1 \rangle \\ \langle oe_1, \mathsf{pb}, xe_1 xe_2 \rangle &= \langle oe_2, \mathsf{pb}, xe_2 xe_1 \rangle \\ \langle oe_2, \mathsf{pb}, xe_1 xe_2 \rangle &= \langle oe_1, \mathsf{pb}, xe_2 xe_1 \rangle, \end{aligned}$$

where X is the graph $(xv_1 \overset{xe_1}{\rightarrow} xv_3 \overset{xe_2}{\leftarrow} xv_2)$. Note, that we can not summarize the three equations by a single (hypothetical) equation between bindings

$$\mathsf{pb}\langle xe_1 xe_2 \rangle = \mathsf{pb}\langle xe_2 xe_1 \rangle$$

since oe_1 and oe_2 are interchanged in the last two equations above.

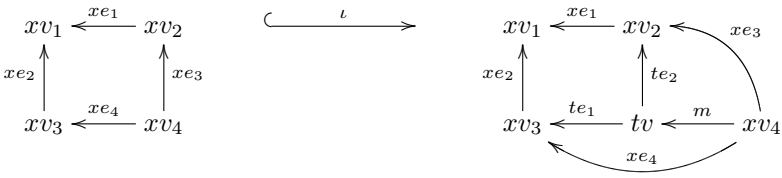
The Kleisli category of the new adjunction provides an appropriate substitution calculus. A substitution is an arrow in the Kleisli category, i.e., a graph homomorphism $\sigma : X \rightarrow T_\Gamma(Y)$. The corresponding extended graph homomorphism $\sigma^* : T_\Gamma(X) \rightarrow T_\Gamma(Y)$ describes the simultaneous substitution of all variable vertices xv and variable edges xe in graph Γ-terms on X by the corresponding graph terms $\sigma_V(xv) \in T_\Gamma(Y)_V$ or $\sigma_E(xe) \in T_\Gamma(Y)_E$, respectively. The composition of two substitutions $\sigma : X \rightarrow T_\Gamma(Y)$ and $\delta : Y \rightarrow T_\Gamma(Z)$ is given by $\delta^* \circ \sigma : X \rightarrow T_\Gamma(Z)$. Substitutions $\sigma : X \rightarrow T_\Gamma(Y)$ allow us, for example, to formalize the concepts of queries and views in databases [3].

Remark 1 (Universal properties). For a categorically minded reader, considering such operations as pullback and pushout without their universal properties does not make too much sense. Below we will show how to include universal properties into our framework of diagram operations. We will consider universality of pullbacks, but the method is quite general and applicable for any limit/colimit operation over graphs.

Commutativity of a pullback square can be expressed by the following equation

$$\langle oe, \mathsf{comp}, \langle oe_1, \mathsf{pb}, ie_1 ie_2 \rangle ie_2 \rangle = \langle oe, \mathsf{comp}, \langle oe_2, \mathsf{pb}, ie_1 ie_2 \rangle ie_1 \rangle$$

where operations pb and comp are defined in Example 3. Universal properties, however, are conditional statements thus we need a kind of implication to express them. The implications, we are looking for, are a further development of the *sketch axioms* in [11] (see also Sect. 5). Those implications are based on graph homomorphisms. To express the existence of mediating morphisms we consider, in case of pullbacks, the following inclusion of graphs:

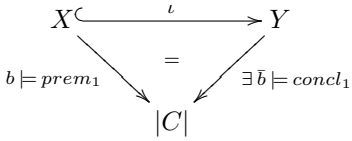


where $tv = \langle ov, \mathsf{pb}, xe_1xe_2\rangle$, $te_1 = \langle oe_2, \mathsf{pb}, xe_1xe_2\rangle$ and $te_2 = \langle oe_1, \mathsf{pb}, xe_1xe_2\rangle$. We denote the graph on the left-hand side by X and the graph on the right-hand side by Y. Since, m is the only item in Y, that is not in X or generated by X, respectively, the inclusion homomorphism allows us to formulate a conditional existence statement of the form

$$\forall\, X.(prem_1 \overset{\iota}{\Rightarrow} \exists\, m.\ concl_1) \quad \text{where}$$

$$prem_1 := \langle oe, \mathsf{comp}, xe_4xe_2\rangle = \langle oe, \mathsf{comp}, xe_3xe_1\rangle$$
$$concl_1 := \langle oe, \mathsf{comp}, m\, te_1\rangle = xe_4 \wedge \langle oe, \mathsf{comp}, m\, te_2\rangle = xe_3.$$

A graph operation $\mathsf{pb}^C : |C|^{\mathsf{Cospan}} \to |C|^{\mathsf{Square}}$, as in Example 3, satisfies this implication iff every binding $b : X \to |C|$, that makes the premise $prem_1$ true, can be extended to a binding $\bar{b} : Y \to |C|$ with $\bar{b} \circ \iota = b$ such that the following two conditions hold: (a) $\bar{b}(te_1) = \langle oe_1, \mathsf{pb}, xe_1xe_2\rangle$ and $\bar{b}(te_2) = \langle oe_2, \mathsf{pb}, xe_1xe_2\rangle$ and (b) the conclusion $concl_1$ becomes true. Note, that condition (a) ensures that only an appropriate match for the edge m needs to be found.



To express the uniqueness of mediating morphisms we exploit a non-injective but surjective graph homomorphism $\varphi : Y' \to Y$ where Y' is Y plus an additional edge m' from xv_4 to tv. φ is the identity except that it maps m and m' in Y' to m in Y. The premise $prem_2$ is given by two corresponding copies of $concl_1$ above and the conclusion $concl_2$ is just *true*. A graph operation pb^C satisfies the implication $\forall\, Y'.(prem_2 \overset{\varphi}{\Rightarrow} true)$ iff every binding $b : Y' \to |C|$, that makes the premise $prem_2$ true, can be extended to a binding $\bar{b} : Y \to |C|$ with $\bar{b} \circ \varphi = b$. In other words, there is no binding $b : Y' \to |C|$ with $\bar{b}(m) \neq \bar{b}(m)$ that makes the premise $prem_2$ true.

5 Related Work

An abstract diagrammatic approach to logic, including a general notion of diagram predicates and their models (generalized sketches), and implications between diagram predicates (sketch axioms), was pioneered by Makkai in [11] (see also historical remarks in our paper [6]). However, Makkai did not work with diagram operations and algebras. Formal definitions of a (diagrammatic) graph operation and a graph algebra were introduced by the second author in [4], and many examples and discussions in the database context can be found in [3]. The latter paper also describes the construction of what they call *sketch parsing*. The idea is that any operation signature Γ gives rise to a predicate signature Γ^* by forgetting the input arity parts in the entire operation arities. Then any Γ-term becomes a Γ^*-sketch [6]. Parsing does the inverse: given an Γ^*-sketch, it tries to convert it into an Γ-term. In these papers, graph terms are defined as trees labeled by diagrams respecting operation arities. In the present paper, we are more interested in the entire object of graph term algebra and its universal properties rather than in the notion of a single graph term. Neither of the papers above formally defined the graph term algebra and proved its universal properties.

Injective graph transformation rules have been studied extensively by Hartmut Ehrig and his co-authors (see [7]). The special feature of injective rules, elucidated in the paper, may shed new light on the "nature" of those rules.

6 Conclusion and Future Work

In the paper, we extended the classical construction of term algebra for operations over sets to the case of diagrammatic operations over graphs. We showed that any graph term algebra freely generated by applying graph operations to a given graph of variables is indeed free: it possesses the respective universal property in the category of graph algebras. This basic result shows that our definitions of graph operations and graph algebras work as we wanted, i.e., in parallel with the ordinary algebra case. Moreover, this result hopefully paves a way to a wider generalization of the core Universal Algebra framework for graph operations and graph algebras. In more detail, we aim at defining congruence relations, quotients and epi-mono factorizations for graph algebras, thus building what we could call *Graph-based Universal Algebra*. More abstractly, it would also be interesting to extend our result in [6] concerning institutions of generalized sketches to any of the envisaged logical extensions.

We see other interesting and useful extensions of the framework.

Typing. The step from unsorted to many-sorted algebras is relatively straightforward. In the same way, we see no principle problems to extend the framework, presented in this paper, to typed graphs [7]. This extension will be necessary to meet the situations in applications (compare [3,12–14]).

Term Language. In the paper we considered two roles of ordinary terms and their extension for graph algebras. These two roles are (a) to denote elements in

free algebras and (b) to provide induction/recursion schemes for evaluating the elements of free algebras in arbitrary algebras. However, terms are also used (c) to denote composed/derived operations in algebras. This role provides the foundation for functorial semantics and thus for a categorical approach to Universal Algebra. By extending the approach in [2], we plan to specify this role in the setting of graph algebras too.

From Graphs to Presheaf Toposes. To meet the spirit of the Festschrift, in the paper we focused on graph-based structures, which have been the basis for research on graph transformations in Hartmut's group for decades [7]. The category of graphs, however, is a very simple instance of a quite general concept of a presheaf topos that encompasses 2-graphs, Petri nets, attributed graphs [7], and many other structures employed in computer science. There should be no principle problems to extend the definitions and results of the paper to the broader class of presheaf topoi.

References

1. Cadish, B., Diskin, Z.: Heterogeneous view integration via sketches and equations. In: Raś, Z.W., Michalewicz, M. (eds.) ISMIS 1996. LNCS, vol. 1079, pp. 603–612. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61286-6_184
2. Claßen, I., Große-Rhode, M., Wolter, U.: Categorical concepts for parameterized partial specifications. Math. Struct. Comput. Sci. **5**(2), 153–188 (1995). https://doi.org/10.1017/S0960129500000700
3. Diskin, Z.: Databases as diagram algebras: specifying queries and views via the graph-based logic of sketches. Technical report 9602, Frame Inform Systems, Riga, Latvia (1996).http://www.cs.toronto.edu/zdiskin/Pubs/TR-9602.pdf
4. Diskin, Z.: Towards algebraic graph-based model theory for computer science. Bull. Symb. Log. **3**, 144–145 (1997)
5. Diskin, Z., Cadish, B.: A graphical yet formalized framework for specifying view systems. In: First East-European Symposium on Advances in Databases and Information Systems, pp. 123–132. Nevsky Dialect (1997)
6. Diskin, Z., Wolter, U.: A diagrammatic logic for object-oriented visual modeling. ENTCS **203**(6), 19–41 (2008). https://doi.org/10.1016/j.entcs.2008.10.041
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformations. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
8. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69962-7
9. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 2: Module Specifications and Constraints. EATCS Monographs on Theoretical Computer Science, vol. 21. Springer, Heidelberg (1990). https://doi.org/10.1007/978-3-642-61284-8
10. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Springer, New York (1978). https://doi.org/10.1007/978-1-4757-4721-8
11. Makkai, M.: Generalized sketches as a framework for completeness theorems. J. Pure Appl. Algebra **115**, 49–79, 179–212, 214–274 (1997)

12. Mantz, F., Taentzer, G., Lamo, Y., Wolter, U.: Co-evolving meta-models and their instance models: a formal approach based on graph transformation. Sci. Comput. Program. **104**, 2–43 (2015). https://doi.org/10.1016/j.scico.2015.01.002
13. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in MDE. J. Log. Algebraic Programm. **79**(7), 636–658 (2010). https://doi.org/10.1016/j.jlap.2009.10.003
14. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A formal approach to the specification and transformation of constraints in MDE. J. Log. Algebraic Programm. **81**(4), 422–457 (2012). https://doi.org/10.1016/j.jlap.2012.03.006

Author Index