Hartmut Ehrig
Reiko Heckel
Grzegorz Rozenberg
Gabriele Taentzer (Eds.)

LNCS 5214

Graph Transformations

4th International Conference, ICGT 2008
Leicester, United Kingdom, September 2008
Proceedings

Springer

Lecture Notes in Computer Science 5214

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

Hartmut Ehrig Reiko Heckel
Grzegorz Rozenberg Gabriele Taentzer (Eds.)

Graph Transformations

4th International Conference, ICGT 2008
Leicester, United Kingdom, September 7-13, 2008
Proceedings

Springer

Volume Editors

Hartmut Ehrig
Technical University of Berlin
Berlin, Germany
E-mail: ehrig@cs.tu-berlin.de

Reiko Heckel
Department of Computer Science
University of Leicester
Leicester, UK
E-mail: reiko@mcs.le.ac.uk

Grzegorz Rozenberg
Leiden Center for Natural Computing
Leiden University
Leiden, The Netherlands
E-mail: rozenber@liacs.nl

Gabriele Taentzer
Department of Mathematics and Computer Science
Philipps-University Marburg
Marburg, Germany
E-mail: taentzer@mathematik.uni-marburg.de

Library of Congress Control Number: 2008934309

CR Subject Classification (1998): E.1, G.2.2, D.2.4, F.1, F.2.2, F.3, F.4.2-3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-87404-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-87404-1 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12512355 06/3180 5 4 3 2 1 0

Preface

Graphs are among the simplest and most universal models for a variety of systems, not just in computer science, but throughout engineering and the life sciences. When systems evolve we are interested in the way they change, to predict, support, or react to their evolution. Graph transformation combines the idea of graphs as a universal modelling paradigm with a rule-based approach to specify their evolution. The area is concerned with both the theory of graph transformation and their application to a variety of domains.

The International Conferences on Graph Transformation aim at bringing together researchers and practitioners interested in the foundations and applications of graph transformation. The 4th International Conference on Graph Transformation (ICGT 2008) was held in Leicester (UK) in the second week of September 2008, along with several satellite events. It continued the line of conferences previously held in Barcelona (Spain) in 2002, Rome (Italy) 2004, and Natal (Brazil) in 2006 as well as a series of six International Workshops on Graph Transformation with Applications in Computer Science between 1978 to 1998. Also, ICGT alternates with the workshop series on Application of Graph Transformation with Industrial Relevance (AGTIVE). The conference was held under the auspices of EATCS, EASST, and IFIP WG 1.3.

Responding to the call for papers, 57 papers were submitted. The papers were reviewed thoroughly by program committee members and their co-reviewers. The committee selected 27 papers for presentation at the conference and publication in the proceedings. These papers mirror well the wide-ranged ongoing research activities in the theory and application of graph transformation. They are concerned with different kinds of graph transformation approaches, compositional systems, validation and verification as well as various applications, mainly to model transformation and distributed systems. Paper submission and reviewing were supported by the free conference management system EasyChair.

In addition to the presentation of technical papers the conference featured three invited speakers, a doctoral symposium, a tutorial, and four workshops.

Invited Speakers. The invited talk by Perdita Stevens introduced an algebraic approach of bidirectional transformations exploring the group theory of the lens framework for bidirectional transformations. In his invited talk, Wil van der Aalst presented the Petri net formalism as a natural candidate for modeling and analysis of workflow processes. The third invited speaker was Heiko Dörr, who presented an approach for model-based engineering in automotive systems and discussed the role of rule-based transformations in that context.

Satellite Events. For the first time at ICGT, a Doctoral Symposium took place – it was organized by Andrea Corradini and Emilio Tuosto. 16 young researchers had the opportunity to present their work and interact with established researchers of the graph transformation community. A tutorial by Reiko Heckel gave newcomers to the field an opportunity to get a general introduction to graph transformation. In addition four workshops were organized where participants of the ICGT could exchange ideas and views on some subareas of graph transformation:

- *Workshop on Graph Computation Models* by Mohamed Mosbah and Annegret Habel,
- *4th Workshop on Graph-Based Tools* by Arend Rensink and Pieter Van Gorp,
- *3rd Workshop on Petri Nets and Graph Transformations* by Paolo Baldan and Barbara König, and
- *Workshop on Natural Computing and Graph Transformations* by Ion Petre and Grzegorz Rozenberg.

We would like to thank Dénes Bisztray, Karsten Ehrig, Stefan Jurack, Paolo Torrini, and Gerd Wierse who provided their valuable help throughout the preparation and organization of the conference and the proceedings. Last but not least, we are grateful to Springer for their helpful collaboration and quick publication.

July 2008

Hartmut Ehrig
Reiko Heckel
Grzegorz Rozenberg
Gabriele Taentzer

Organization

Program Committee

Paolo Baldan	Padova (Italy)
Luciano Baresi	Milano (Italy)
Michel Bauderon	Bordeaux (France)
Andrea Corradini	Pisa (Italy)
Hartmut Ehrig	Berlin (Germany)
Gregor Engels	Paderborn (Germany)
Annegret Habel	Oldenburg (Germany)
Reiko Heckel (Co-chair)	Leicester (UK)
Dirk Janssens	Antwerp (Belgium)
Garbor Karsai	Nashville (USA)
Barbara König	Duisburg-Essen (Germany)
Hans-Jörg Kreowski	Bremen (Germany)
Juan de Lara	Madrid (Spain)
Tom Mens	Mons (Belgium)
Mark Minas	München (Germany)
Ugo Montanari	Pisa (Italy)
Mohamed Mosbah	Bordeaux (France)
Manfred Nagl	Aachen (Germany)
Fernando Orejas	Barcelona (Spain)
Francesco Parisi-Presicce	Rome (Italy)
Mauro Pezzè	Milan (Italy)
John Pfaltz	Charlottesville (Virginia USA)
Rinus Plasmeijer	Nijmegen (The Netherlands)
Detlef Plump	York (UK)
Arend Rensink	Twente (The Netherlands)
Leila Ribeiro	Porto Alegre (Brasil)
Grzegorz Rozenberg	Leiden (The Netherlands)
Andy Schürr	Darmstadt (Germany)
Gabriele Taentzer (Co-chair)	Marburg (Germany)
Hans Vangheluwe	Montreal (Canada)
Dániel Varró	Budapest (Hungary)
Albert Zündorf	Kassel (Germany)

Subreviewers

Sergio Antoy
Nina Aschenbrenner
Karl Azab
Dénés Bisztray
Ivoka Boneva
Florian Brieler
Sander Bruggink
Roberto Bruni
Simone Costa

Bilel Derbel
Ira Diethelm
Frank Drewes
Jörn Dreyer
Claudia Ermel
Fabio Gadducci
Leif Geiger
Joel Greenyer
Stefan Gruner
Esther Guerra
Tero Harju
Thomas Heer
Tobias Heindel
Frank Hermann
Berthold Hoffmann
Felix Klar
Renate Klempien-Hinrichs
Pieter Koopman
Vitali Kozioura
Anne-Therese Kürtgen
Elodie Legros
Sylvain Lippi
Alberto Lluch Lafuente
Rodrigo Machado
Sonja Maier
Thomas Maier
Steffen Mazanek
Damiano Mazza
Andrea Mocci
Olaf Muliawan
Karl-Heinz Pennemann
Ulrike Prange
István Ráth
Guilherme Rangel
Carsten Reckord
Daniel Retkowitz
Hans Schippers
Maria Semenyak
Christian Soltenborn
Paola Spoletini
Paolo Torrini
Gergely Varro
Erhard Weinell

Sponsoring Institutions

The European Association for Theoretical Computer Science (EATCS)
The European Association of Software Science and Technology (EASST)
The IFIP Working Group 1.3, Foundations of Systems Specification

Table of Contents

Invited Papers

Towards an Algebraic Theory of Bidirectional Transformations 1
Perdita Stevens

Discovery, Verification and Conformance of Workflows with Cancellation 18
W.M.P. van der Aalst

The AUTOSAR Way of Model-Based Engineering of Automotive Systems 38
Heiko Dörr

Hypergraphs and Termgraph Rewriting

An Algorithm for Hypergraph Completion According to Hyperedge Replacement Grammars 39
Steffen Mazanek, Sonja Maier, and Mark Minas

Labelled (Hyper)Graphs, Negotiations and the Naming Problem 54
Jérémie Chalopin, Antoni Mazurkiewicz, and Yves Métivier

Abstracting Complex Data Structures by Hyperedge Replacement 69
Stefan Rieger and Thomas Noll

Inductively Sequential Term-Graph Rewrite Systems 84
Rachid Echahed

Applications of Graph Transformation

Mobile Agents Implementing Local Computations in Graphs 99
Bilel Derbel, Mohamed Mosbah, and Stefan Gruner

A Decentralized Implementation of Mobile Ambients 115
Fabio Gadducci and Giacoma Valentina Monreale

Network Applications of Graph Bisimulation 131
Pietro Cenciarelli, Daniele Gorla, and Emilio Tuosto

Graph Transformation for Topology Modelling 147
Mathieu Poudret, Agnès Arnould, Jean-Paul Comet, and Pascale Le Gall

Execution of Graph Transformations

Embedding and Confluence of Graph Transformations with Negative Application Conditions ... 162
Leen Lambers, Hartmut Ehrig, Ulrike Prange, and Fernando Orejas

Formal Analysis of Model Transformations Based on Triple Graph Rules with Kernels ... 178
Hartmut Ehrig and Ulrike Prange

Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation ... 194
Hartmut Ehrig and Claudia Ermel

High-Level Programs and Program Conditions ... 211
Karl Azab and Annegret Habel

Compositional Systems

Parallel and Sequential Independence for Borrowed Contexts ... 226
Filippo Bonchi, Fabio Gadducci, and Tobias Heindel

Behavior Preservation in Model Refactoring Using DPO Transformations with Borrowed Contexts ... 242
Guilherme Rangel, Leen Lambers, Barbara König, Hartmut Ehrig, and Paolo Baldan

Open Petri Nets: Non-deterministic Processes and Compositionality ... 257
Paolo Baldan, Andrea Corradini, Hartmut Ehrig, and Barbara König

Validation and Verification

Attributed Graph Constraints ... 274
Fernando Orejas

Resolution-Like Theorem Proving for High-Level Conditions ... 289
Karl-Heinz Pennemann

Towards the Verification of Attributed Graph Transformation Systems ... 305
Barbara König and Vitali Kozioura

A Modal-Logic Based Graph Abstraction ... 321
Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink

Graph Languages and Special Transformation Concepts

On the Recognizability of Arrow and Graph Languages ... 336
H.J. Sander Bruggink and Barbara König

Graph Multiset Transformation as a Framework for Massively Parallel Computation 351
Hans-Jörg Kreowski and Sabine Kuske

Pullback Grammars Are Context-Free 366
Michel Bauderon, Rui Chen, and Olivier Ly

On Switching to H-Free Graphs 379
Eva Jelínková and Jan Kratochvíl

Patterns and Model Transformations

A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation 396
Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró

15 Years of Triple Graph Grammars: Research Challenges, New Contributions, Open Problems 411
Andy Schürr and Felix Klar

Pattern-Based Model-to-Model Transformation 426
Juan de Lara and Esther Guerra

Adaptive Star Grammars for Graph Models 442
Frank Drewes, Berthold Hoffmann, and Mark Minas

Tutorial and Workshops

Tutorial Introduction to Graph Transformation 458
Reiko Heckel

Workshop on Graph Computation Models 460
Mohamed Mosbah and Annegret Habel

Graph-Based Tools: The Contest 463
Arend Rensink and Pieter Van Gorp

Workshop on Petri Nets and Graph Transformations 467
Paolo Baldan and Barbara König

Summary of the Workshop on Natural Computing and Graph Transformations 470
Ion Petre and Grzegorz Rozenberg

Doctoral Symposium

ICGT 2008 Doctoral Symposium 473
Andrea Corradini and Emilio Tuosto

Verification of Architectural Refactorings: Rule Extraction and Tool Support ... 475
Dénes Bisztray

Abstract Semantics by Observable Contexts ... 478
Filippo Bonchi

Triple Graph Grammars and OCL for Validating System Behavior ... 481
Duc-Hanh Dang

From Separation Logic to Hyperedge Replacement and Back (Extended Abstract) ... 484
Mike Dodds

Security, Cryptography and Directed Bigraphs ... 487
Davide Grohmann

Modelling Clustering of Sensor Networks with Synchronised Hyperedge Replacement ... 490
Mohammad Hammoudeh

Grammar Morphisms and Weakly Adhesive Categories ... 493
Tobias Heindel

Process Construction and Analysis for Workflows Modelled by Adhesive HLR Systems with Application Conditions ... 496
Frank Hermann

Towards a Two Layered Verification Approach for Compiled Graph Transformation ... 499
Ákos Horváth

Model-Based Analysis of Network Reconfigurations Using Graph Transformation Systems ... 502
Ajab Khan

Service Extraction from Legacy Systems ... 505
Carlos Matos

Development of Correct Graph Transformation Systems ... 508
Karl-Heinz Pennemann

Graph Transformation for the Semantic Web: Queries and Inference Rules ... 511
Hong Qing Yu and Yi Hong

Using a Triple Graph Grammar for State Machine Implementations ... 514
Michael Striewe

Model-Driven Development of Model Transformations 517
Pieter Van Gorp

Transformation-Based Operationalization of Graph Languages 520
Erhard Weinell

Author Index 523

Towards an Algebraic Theory of Bidirectional Transformations

Perdita Stevens

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. Bidirectional transformations are important for model-driven development, and are also of wide interest in computer science. In this paper we present early work on an algebraic presentation of bidirectional transformations. In general, a bidirectional transformation must maintain consistency between two models, either of which may be edited, and each of which may incorporate information not represented in the other. Our main focus here is on lenses [2,1,3] which provide a particularly well-understood special case, in which one model is an abstraction of the other, and either the abstraction or the full model may be edited. We show that there is a correspondence between lenses and short exact sequences of monoids of edits. We go on to show that if we restrict attention to invertible edits, *very well-behaved* lenses correspond to split short exact sequences of groups; this helps to elucidate the structure of the edit groups.

1 Introduction

Fundamental to the idea of graph transformations is the idea that a change in one structure can correspond to a change in another in a precise sense. This fundamental idea appears in different guises in many areas of informatics; the guise most familiar to the present author is that of bidirectional model transformations, as they appear in the OMG's Model-Driven Architecture (or, as it is now usually more suggestively called, Model-Driven Development) initiative. A bidirectional transformation R between two classes of models, say M and N, incorporates a precise notion of what it is for $m \in M$ to be consistent with $n \in N$:

$$R \subseteq M \times N$$

It also specifies how, if one model is changed, the other can be changed so as to restore consistency. The forward transformation

$$\overrightarrow{R} : M \times N \longrightarrow N$$

takes a pair of models (m, n) which are not (necessarily) consistent. Leaving m alone, it calculates how to modify n so as to restore consistency. It returns this calculated n' such that $R(m, n')$. Symmetrically,

$$\overleftarrow{R} : M \times N \longrightarrow M$$

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 1–17, 2008.

explains how to roll changes made to a model from N back to a change to make to a model from M.

For practical reasons, it is preferable that all three elements of the transformation – R, $\overrightarrow{R}$ and $\overleftarrow{R}$ – be expressed in one text; but this will not be essential to our semantic treatment here.

This basic framework is flexible enough to explain a wide range of languages for bidirectional transformations, including for example the OMG's QVT-R (Queries Views and Transformations – Relations) language. That language is discussed in [5], as are the postulates that a bidirectional transformation may be expected to satisfy. The reader is referred to that paper for details. In brief, the two main postulates are *correctness* and *hippocraticness*. Correctness has already been mentioned: it states that the forwards and backwards transformations really do restore consistency, e.g. that the returned n' above really does satisfy $R(m, n')$. Hippocraticness ("first do no harm") states that the transformation must not modify a pair of models which is already consistent (not even by returning a different consistent model). Correctness and hippocraticness go a long way to ruling out "silly" transformations, but something else still seems to be required. In [5] a third postulate, *undoability* is proposed, but this is arguably too strong.

The crucial point to notice is that there may be a genuine choice about how consistency is restored. In the absence of defensible way to define which is the "best" option, we want that choice to be in the hands of the person who designs the transformation. Given $m \in M$, there may be many $n' \in N$ such that $R(m, n')$. Given a model n such that $R(m, n)$ does not hold, the designer of the transformation $\overrightarrow{R}$ should be able to choose which of the possible n' will be returned. Although it may be that our transformation language imposes some limitations, for it to be practically useful it will have to permit considerable choice.

Thus far, our framework, like those typically used in graph transformations, is completely symmetric in M, N. Neither model is necessarily an abstraction of the other: each may contain information which is not contained in the other. We will begin with this general situation, but later we shall specialise to the particular case where N is an abstraction of M. This is the situation studied by the Harmony group and reported on in a series of papers including [2,1]. Much of the present paper can reasonably be seen as "just" a translation into algebraic language of that work, sometimes with generalisation, sometimes with restriction. At the end of the paper we will discuss why this may be a useful undertaking; at the very least, it is hoped that it may amuse the algebraically-inclined reader.

The rest of this paper is structured as follows. In Section 2 we introduce some important equivalence relations that a bidirectional transformation imposes on the sets of models it relates. In Section 3 we discuss *edits* and introduce some basic algebraic ideas. Section 4 shows how to construct a short exact sequence of monoids (or groups) from appropriate lenses, while Section 5 shows how to go the other way, from a suitable sequence to a lens. Finally Section 6 concludes and briefly mentions future work.

A recent survey of bidirectional transformation approaches is found in [6]; these are many, so this paper will not attempt to summarise again, but will stick to the technical focus.

2 Transformations and Equivalences

Let R (comprising, by abuse of notation, a consistency relation R, a forward transformation $\overrightarrow{R}$ and a backward transformation $\overleftarrow{R}$) be a transformation which is correct and hippocratic.

We will always assume that there is a trivial or content-free element of each set of models; for example, we will write the trivial element of M as Ω_M. If M is a set of models defined by a metamodel, this might be the model containing no model elements, if that is a member of M, i.e. permitted according to the metamodel. However, it might not be literally empty; if for example all models in M are required to contain some basic model elements, then Ω_M will contain these and nothing else. We will assume that $R(\Omega_M, \Omega_N)$.

Definition 1. *The equivalence relations $\overrightarrow{B}$ and $\overleftarrow{B}$ on M, and $\overrightarrow{F}$ and $\overleftarrow{F}$ on N, are defined as follows:*

- $m \sim_{\overrightarrow{B}} m' \Leftrightarrow \forall n \in N. \overrightarrow{R}(m,n) = \overrightarrow{R}(m',n)$
  (Intuitively, this says "m and m' do not differ in any way that is visible on the N side". The reader familiar with lenses will recognise that this generalises $\sim_g$.)
- $m \sim_{\overleftarrow{B}} m' \Leftrightarrow \forall n \in N. \overleftarrow{R}(m,n) = \overleftarrow{R}(m',n)$
  (Intuitively, "the only *differences between m and m' are those visible on the N side, so that they become indisinguishable after any synchronisation with an element of N". The reader familiar with [1] will recognise that this generalises $\sim_{max}$, the* coarsest *equivalence with respect to which a lens is quasi-oblivious.)*

and dually,

- $n \sim_{\overrightarrow{F}} n' \Leftrightarrow \forall m \in M. \overrightarrow{R}(m,n) = \overrightarrow{R}(m,n')$
- $n \sim_{\overleftarrow{F}} n' \Leftrightarrow \forall m \in M. \overleftarrow{R}(m,n) = \overleftarrow{R}(m,n')$

We can also, in the obvious way give versions of these definitions which are parameterised on subsets of M, N, respectively, the above then being given by plugging in the largest available set, getting the finest available equivalence relations. We do not need any of the coarser equivalences in this paper, however.

Thus, the transformation defines two different equivalences on M (and dually on N). Of course, any element $m \in M$ can then be viewed as a representative of its equivalence class $[m]_{\sim_{\overrightarrow{B}}}$, or as a representative of its other equivalence class $[m]_{\sim_{\overleftarrow{B}}}$. These are the co-ordinates of m in the sense that m is uniquely defined by its two classes; this was already remarked in the case of lenses in [1]:

Lemma 1. [1] *Let $m_1, m_2 \in M$. If both $m_1 \sim_{\overrightarrow{B}} m_2$ and $m_1 \sim_{\overleftarrow{B}} m_2$ then $m_1 = m_2$.*

A useful picture to bear in mind – although, of course, since M need not be finite or even countable, it is only an informal idea – is of the elements of M laid out on a grid whose columns represent $\sim_{\overrightarrow{B}}$-equivalence classes and whose rows represent $\sim_{\overleftarrow{B}}$-equivalence classes. We have just shown that no square on the grid can contain more than one element of M. In general, not every square need be occupied; indeed, the equivalence classes might have different cardinalities.

The *closure* of M with respect to transformation R, denoted by $\bar{M}$, is the cartesian product of the two sets of equivalence classes, which "contains" M: informally, the set of all squares in the grid. We will have $M = \bar{M}$ in the special case that R is an *undoable* transformation (again, this corresponds to a remark in [1] for lenses).

Since we are, so far, in the completely symmetric case of general bidirectional transformations, the same remarks and result apply to N. In the special case of lenses, which we shall come to, the grid for N, which in that setting is a strict abstraction of M, is degenerate, since then $\sim_{\overrightarrow{F}}$ is universal and $\sim_{\overleftarrow{F}}$ is trivial. In the even more special case of a bijective transformation (or oblivious lens, in the terminology of [2]), the grid for M is also degenerate.

3 Edits and Algebraic Basics

We will assume that the reader is familiar with the standard notions of group, monoid, mono-, epi- and isomorphisms of groups and monoids, subgroup, submonoid, and normal subgroup. Other definitions from algebra will be reproduced, marked (Standard).

In order to discuss how transformations behave it is useful to have a notion of an *edit*: a way in which a model is changed by its user. When an edit has been done on a model, restoring consistency between it and another model is a matter of performing the "corresponding" edit on the other model. The task of a transformation is then to specify what it means for an edit on one structure to correspond to an edit on the other structure.

The notion of an edit, though, is a little trickier than at first appears. What is an "edit" on a model? Intuitively, it is a thing you can do to an model, changing it into another model. Doing nothing is certainly an edit; edits can be undone; two edits can be done in sucession. Can the same edit be done on *any* model from a given model set; in other words, if we model an edit as an endofunction

$$g : M \longrightarrow M$$

should it be total? It is easy to come up with reasonable examples we might want to model that are ("add a class with a new name in the top level package") but also easy to come up with examples that at first sight are not ("delete the class called `Customer`"). We can get around the problem of edits which are not

[1] All proofs – generally easy – are omitted. A version of the paper which does contain proofs is available from the author's web page.

obviously total by decreeing that if an edit is not naturalistically applicable to a given model, then it should leave the model unchanged ("delete the class called `Customer` if there is one, otherwise do nothing"). In this way, we can model only total edits without imposing any real restriction.

To say that doing nothing is an edit is simply to say that the identity function is an edit. Then to say that edits can be composed is to say that the set of edits is a *monoid*. We will give the definition in order to set up some notation.

Definition 2. *(Standard) A set G provided with an operation*

$$* : G \times G \longrightarrow G$$

(written infix, e.g. $g_1 * g_2$, and in practice normally omitted: $g_1 g_2$) is a monoid if

1. *G contains an identity element, written 1_G, such that for any $g \in G$*

$$1_G * g = g * 1_G = g$$

2. *$*$ is associative: that is, for any g_1, g_2, $g_3 \in G$, we have*

$$(g_1 * g_2) * g_3 = g_1 * (g_2 * g_3)$$

Given a set S, we will often be interested in the monoid of all endofunctions $S \longrightarrow S$, in which the operation is function composition and the identity is the identity, "do nothing" function. We will write this $M(S)$.

What about the fact that edits can be undone? It is tempting to say that this means we have a group of edits, but this is premature. To say that an edit can be undone in the sense that a modelling tool will allow simply means that the tool will retain enough information to reverse any change that the user makes. It does not mean that there will necessarily be an edit g^{-1} which always undoes the effect of edit g, regardless of which model it was applied to. For example, in the case of our edit "delete the class called `Customer` if there is one, otherwise do nothing", there is no inverse, because the edit is not injective.

Definition 3. *(Standard) Let G be a monoid. If in addition G has inverses; that is, for any $g \in G$ there is an element $g^{-1} \in G$ such that*

$$g^{-1} * g = g * g^{-1} = 1_G$$

then G is a group. In that case, inverses are necessarily unique.

Let us pause to observe that an edit can be total without being invertible, and vice versa. For example,

- "delete everything" is total, but not invertible
- "delete package `P` and everything in it" is neither total nor invertible
- "add 1 to constant `MAX`" is not total, but it is invertible where defined
- "swap `true` and `false` wherever they occur" is both total and invertible (as it happens, it is self-inverse).

Given any monoid M of endofunctions on a set S, we will sometimes be interested in the set of all invertible – that is, bijective – elements of M, which of course forms a group. We will write this $G(M)$. Then in particular $G(M(S))$ is the full permutation group on S.

Let us also note that if our transformation engine only sees models, before and after edits, it does not have access to information about what edit the user was doing, in the sense that we do not find out what s/he would have done on a different model; we only see what was done in one instance. Since the user *may* be thinking of a permutation, the transformation certainly has to behave sensibly in that case. Thus let us proceed, for now not committing ourselves to whether we have a group or only a monoid of edits.

Definition 4. *(Standard) Let G be a group or a monoid. The action of G on a set M is a function*

$$\cdot : G \times M \longrightarrow M$$

such that for any g, h, m

1. $1_G \cdot m = m$
2. $(gh) \cdot m = g(h \cdot m)$

We normally omit the dot and just write gm.

If G is a group, i.e. has inverses, it is easy to see that $g^{-1}n = m$ iff $gm = n$. This is why any group action on a set is a permutation action.

3.1 Lenses

We now switch to the restricted setting of lenses, in which one of the models is a strict abstraction of the other. We will use the notation of [1].

The basic premise is that we have two (maybe structured) sets, C and A, connected by an abstraction function $get \ : C \longrightarrow A$. We consider c and a to be consistent iff $get\ c = a$.

The *get* function, as well as specifying consistency, also provides the forwards transformation. Because of the restricted framework there is no choice, in the sense that the forward transformation is completely determined by the consistency relation: given c, there is a unique consistent a. Thus a lens l corresponds to a special bidirectional transformation R in which $R(c, a)$ holds iff $a = get\ c$, and $\overrightarrow{R}(c, a) = get\ c$ (note that in this special case $\overrightarrow{R}$ ignores a).

We will also need the two equivalence relations on C denoted $\sim_{\overrightarrow{B}}$ and $\sim_{\overleftarrow{B}}$ above, which as remarked are called $\sim_g$ and $\sim_{\max}$ in [1]. In the special case of lenses, we will refer to these equivalences as $\sim_A$ and $\sim_L$ respectively, for reasons which will become apparent. Thus $c_1 \sim_A c_2$ iff $get\ c_1 = get\ c_2$, while $c_1 \sim_L c_2$ iff for every $a \in A$ we have $put\ a\ c_1 = put\ a\ c_2$.

Where the lens designer has a genuine choice is in the *put* function, which corresponds to the backward transformation. A lens also provides

$$put \quad : A \longrightarrow C \longrightarrow C$$

Note that in [2] lenses are for technical reasons not required to be total on their domains, in order that a language of lenses can be defined using recursion; the lenses eventually written by a lens programmer will be total. In this paper, where we consider only semantic issues and do not concern ourselves with the language in which lenses are defined, we are only considering total lenses.

We will, as remarked in the symmetric setting, always assume that there is a trivial or content-free element of C, written Ω_C, and similarly for A. We require $get\ \Omega_C = \Omega_A$ (thus ensuring that Ω_C and Ω_A are consistent, as required) and we derive a function

$$\begin{aligned} create\ &: A \longrightarrow C \\ &\quad a \longmapsto put\ a\ \Omega_C \end{aligned}$$

To complete the definition, lenses are (in this paper) required to satisfy two basic *lens laws*, as follows.

Definition 5. *(adapted from [2]) Let C and A be sets, containing trivial elements Ω_C and Ω_A respectively. A lens from C to A consists of a pair of functions, $get : C \rightarrow A$ and $put : A \rightarrow C \rightarrow C$, such that the following conditions hold:*

$$\begin{aligned} get\ \Omega_C &= \Omega_A \\ put\ (get\ c)\ c &= c \qquad\qquad \textsc{GetPut} \\ get\ (put\ a\ c) &= a \qquad\qquad \textsc{PutGet} \end{aligned}$$

Note that the lens law CreateGet from [1], viz that for any $a \in A$ we have $get\ (create\ a) = a$, follows from the definition and PutGet.

In general $create\ (get\ c)$ need not of course be c (it could be something else in the same $\sim_A$ equivalence class), but we do have (and will later use):

Lemma 2. *Create $\Omega_A = \Omega_C$*

This framework is equivalent to a restricted version of the model transformation framework in which the right hand model is required to be an abstraction of the left hand model, and transformations are required to be correct and hippocratic but not undoable. The curious thing is how little those two conditions alone actually restrict the transformation writer: there is an enormous amount of choice about what the put function should do, and many such choices will be in no way defensible as "sensible" behaviour. Formally:

Lemma 3. *Let $get\ : C \longrightarrow A$ be a surjective function, and let $f_c : A \longrightarrow C$ be a family of injective functions, one for each $c \in C$. Then provided only that $f_c(get\ c) = c$ for each $c \in C$, get together with the function put defined by $put\ a\ c =_{def} f_c(a)$ is a lens.*

Basically, the lens laws force *put* to behave correctly if putting back an abstract element against the concrete element of which it is an abstraction – it is not allowed to break things if nothing has changed – but once any modification has been made in the abstract view, all bets are off. The corresponding issue in the model transformation framework is that hippocraticness requires a transformation not to fix something that isn't broken, but as soon as it is broken in even a trivial detail, the transformation is allowed to do whatever it wants. This is intuitively all wrong: we generally want a tiny change to one model to cause a tiny change to another, or at the very least, only certain enormous changes will seem reasonable! The question of how this should best be captured in a language framework is still open. As discussed in [5], we currently have no entirely satisfactory candidate condition. See also the discussion in [2]. Most convincing, although for some applications too strong, is the law called PutPut in [2]: it states (modulo totality) that for any $a, a' \in A$ and $c \in C$,

$$put\ a'\ (put\ a\ c) = put\ a'\ c \qquad \textsc{PutPut}$$

Definition 6. *(from [2]) A lens is called* very well-behaved *if it satisfies* PutPut*.*

4 Building Sequences from Lenses

Suppose we are given a lens: that is, sets C and A, each with their trivial element, with functions *get* and *put* satisfying the lens laws (and derived function *create*). In this section we will show how to represent this lens algebraically.

Now, fundamentally what we want to do is to say what edit on one model corresponds to an edit on the other, and we want to do this in such a way that composition of edits is respected and obviously so that doing nothing on one model corresponds to doing nothing on the other.

Lenses, however, do not come equipped with a notion of edit: we have to add that. What should the edits on C be? Our first thought might be to use the whole monoid of functions from C to itself: but in fact, we will need a *compatibility condition* in order for *get*, which is supposed to be an abstraction function, to work as one. The condition is that for any g in our monoid of edits, and for any $c, c' \in C$:

$$get\ c = get\ c' \Rightarrow get\ gc = get\ gc' \qquad \textsc{Compat}$$

– in other words, an edit should *act* on $C/\sim_A$. Let $\mathit{\Pi}_C \subseteq M(C)$ be the set of all functions from C to itself that satisfy this compatibility condition. It is easy to see that $\mathit{\Pi}_C$ is itself a monoid, and that it acts transitively on C, which reassures us that it is expressive enough to model anything the user does to an actual model. In fact, an element g of $\mathit{\Pi}_C$ is defined by:

1. a function $\bar{g} : C/\sim_A \longrightarrow C/\sim_A$, together with
2. for each $[c] \in C/\sim_A$, a function $g_{[c]} : [c] \longrightarrow [gc]$.

Essentially the compatibility condition says that the abstraction embodied in *get* respects the edits which are allowed, in the sense that if two concrete states look the same in the abstract view before a concrete edit, they will also look the same in the abstract view after the edit. Although this may repay further study, it seems a plausible requirement, in order for there to *be* a notion of edit on the abstract domain which is compatible with the notion of edit on the concrete domain.

Lemma 4. *Any lens induces a monoid homomorphism*

$$\mu : \Pi_C \longrightarrow M(A)$$

defined as

$$(\mu g)(a) = get\ (g(create\ a))$$

Let us write K and H for the kernel and image of μ, respectively.

Lemma 5. *If* $g_1\Omega_C = g_2\Omega_C$ *then* $\mu g_1\Omega_A = \mu g_2\Omega_A$

Next, consider the *function* (in the absence of PutPut it is not necessarily a homomorphism, as we shall discuss):

$$\lambda : H \longrightarrow \Pi_C$$

given by

$$(\lambda h)(c) = put\ h(get\ c)\ c$$

This is the function that captures how to "put back" information introduced by a user editing an abstract model, to give a corresponding edit on the concrete model.

Lemma 6. λ *is well-defined.*

Lemma 7. $\mu\lambda$ *is the identity on* H*.*

Thus, the function λ is a right inverse for the epimorphism μ.

Although in general λ may not be a monoid homomorphism, it does behave as such on the identity:

Lemma 8. $(\lambda 1_H) = 1_{\Pi_C}$

Later, we shall want:

Lemma 9. *For all* $g \in \Pi_C$ *and for all* $h \in H$*, we have*

$$h(\mu g)\Omega_A = (\mu g)\Omega_A \Rightarrow (\lambda h)g\Omega_C = g\Omega_C$$

To sum up what we have done so far, we need two more standard definitions:

Definition 7. *(Standard) A sequence of groups or monoids*

$$\ldots \rightarrow G_{i-1} \overset{\lambda_i}{\rightarrow} G_i \overset{\lambda_{i+1}}{\rightarrow} G_{i+1} \rightarrow \ldots$$

is exact if for each i*,*

$$img\ \lambda_i = ker\ \lambda_{i+1}$$

– that is, the elements of G_i *which are the images under* λ_i *of elements of* G_{i-1} *are exactly those elements of* G_i *which are mapped by* λ_{i+1} *to the identity element of* G_{i+1}*.*

Definition 8. *(Standard) A short exact sequence is an exact sequence of length 5, whose ends are trivial:*

$$1 \to K \to G \to H \to 1$$

Therefore we may rephrase what we have shown so far as

Proposition 1. *Let l be a lens from C to A, consisting of functions put and get. Let Π_C be the monoid of endofunctions on C which satisfy* Compat. *Then*

$$1 \to K \to \Pi_C \xrightarrow{\mu} H \to 1$$

is a short exact sequence of monoids, where the monoid homomorphism μ is defined by

$$(\mu g)(a) = get\ (g(create\ a))$$

Moreover, the function $\lambda : H \longrightarrow \Pi_C$ defined by $(\lambda h)(c) = put\ h(get\ c)\ c$ is a right inverse for μ.

However, the usual reason in algebra for considering short exact sequences is that they often encode useful information about the structures in them; unfortunately, in the case of general monoids, they are not so informative. The rephrasing above is suggestive, but not yet very useful. In order to go further, we have to restrict the setting. There are two obvious ways to do this: we can consider only very well-behaved lenses (those which satisfy PutPut), and/or we can restrict attention to invertible edits. Let us consider the first of these restrictions first.

4.1 Very Well-Behaved Lenses

It turns out that insisting that the lens be very well-behaved corresponds exactly to insisting that λ be a monoid homomorphism.

Lemma 10. *If* PutPut *holds then λ is a monoid homomorphism.*

Lemma 11. *If λ is a homomorphism then* PutPut *holds.*

This is a very interesting correspondence, particularly in view of the difficulty, mentioned earlier, in choosing an appropriate condition to complement the basic lens laws and ensure "sensible" behaviour. The fact that PutPut corresponds to so basic an algebraic phenomenon as homomorphism is encouraging. Let us now consider the restriction to invertible edits.

4.2 Invertible Edits

Recall that for any monoid M, $G(M)$ is the collection of invertible elements of M, which forms a group. Using exactly the same definition as above, we can define $\mu : G(\Pi_C) \longrightarrow G(M(A))$.

However, it turns out that the development we did for monoids will fail in two ways if we try to use an arbitrary lens in conjunction with considering only invertible edits. Firstly, the action of $G(\Pi_C)$ will not necessarily be transitive on C, because if c_1 and c_2 are in $\sim_A$ equivalence classes of different cardinalities, then no invertible element of Π_C can map c_1 to c_2. A consequence of this is that, if we restrict to invertible edits but still consider an arbitrary lens, there might be cases were we could not handle the situation in which a user modified a model c_1, turning it into c_2, and the changes were rolled through to a corresponding model. Since the original lens, which is independent of any notion of edit, can roll through any change a user might make, our algebraic framework would then be failing to describe the full behaviour of the lens. Secondly, our function λ might not be well-defined, since if it is not a monoid homomorphism, it might map an invertible edit to one which is not invertible.

Both of these problems are solved if we assume, for the remainder of the section, that l is a very well-behaved lens, so that PutPut holds. (This may not be the only way to proceed, however.) The development done for monoids now goes through smoothly, using

Lemma 12. *(from [1]) If l is a very well-behaved lens, then there is a bijection between C and the cartesian product $C/{\sim_L} \times C/{\sim_A}$.*

In particular, all the equivalence classes in $C/{\sim_A}$ have the same cardinality: according to the informal grid picture we suggested before, there is exactly one element of C occupying every square of the rectangular grid whose columns are labelled by elements of $C/{\sim_A}$ and whose columns are labelled by elements of $C/{\sim_L}$.

Thus an element g of $G(\Pi_C)$ is defined by:

1. a permutation $\bar{g} : C/{\sim_A} \longrightarrow C/{\sim_A}$, together with
2. for each $[c] \in C/{\sim_A}$, a bijection $g_{[c]} : [c] \longrightarrow [gc]$.

First, we need to check that for any $g \in G(\Pi_C)$, the endofunction μg is indeed invertible. This is immediate from the fact that μ is a monoid homomorphism. Recall that any monoid homomorphism between groups is a group homomorphism. Finally we have to check that λ remains well-defined when restricted. Since λ is a monoid homomorphism, the image of any invertible element is invertible, so it is a group homomorphism.

We will now write G instead of $G(\Pi_C)$.

Now that we are considering groups rather than just monoids, let us return to our short exact sequence. The crucial standard result is

Lemma 13. *(Standard) Let*

$$1 \to K \to G \to H \to 1$$

be a short exact sequence of groups. Then $K \trianglelefteq G$ and $G/K \simeq H$; we say that G is an extension of H by K.

That is, the short exact sequence tells you how G is in a certain sense built from its substructure K together with the extending structure H. Notice, though, that H need not embed in G, i.e., there need not be any group monomorphism from H to G; if these are edit structures, there need not be a systematic way to regard an edit done on an abstract model as an edit done on the concrete model. That is, we cannot necessarily express the edits that can be done on the concrete domain, in terms of edits done on the abstract domain together with other information. Algebraically, this is because – *in general* – a short exact sequence does not necessarily *split*.

Definition 9. *(Standard) A short exact sequence of groups*

$$1 \to K \to G \overset{\sigma}{\to} H \to 1$$

is said to split if there exists a group monomorphism $\lambda : H \to G$ which composes with σ to the identity on H:

$$\forall h \in H \sigma(\lambda h) = h$$

In that case, λ is said to split the sequence, and $G \simeq K \rtimes H$.

Definition 10. *(Standard) Let G be a group, with subgroups $K \trianglelefteq G$ and $H \leq G$. G is the (internal) semi-direct product $K \rtimes H$ if:*

- $KH = G$
- $K \cap H = 1_G$

In this case, we observe that

- *every element g of G can be written uniquely as the product $g = kh$ of elements $k \in K$ and $h \in H$;*
- $(kh)^{-1} = (h^{-1}k^{-1}h)h^{-1}$ *(note that $h^{-1}k^{-1}h \in K$ by normality of K);*
- $(k_1h_1)(k_2h_2) = (k_1h_1k_2h_1^{-1})(h_1h_2)$ *(noting again that this is the product of an element of K and one of H, by normality).*

The product is direct if in addition H and K commute.

Our restriction to very well-behaved lenses gives us that λ is a monoid and hence a group homomorphism, which is exactly what is needed to ensure that the short exact sequence splits. That is, we have (summarising)

Theorem 1. *Let l be a very well-behaved lens from C to A, consisting of functions put and get. Let G be the group of invertible endofunctions on C which satisfy* Compat. *Then*

$$1 \to K \to G \overset{\mu}{\to} H \to 1$$

is a short exact sequence of groups, where the group homomorphism μ is defined by

$$(\mu g)(a) = get\ (g(create\ a))$$

Moreover, the function $\lambda : H \longrightarrow G$ defined by $(\lambda h)(c) = put\ h(get\ c)\ c$ is a right inverse for μ, and a group homomorphism. Therefore it splits the sequence and we have an isomorphism

$$K \rtimes H \simeq G$$

We can now discuss the action of G on C in terms of what K, H do to $C/{\sim_L}$ and $C/{\sim_A}$.

Lemma 14. *The subgroup λH of G acts, trivially, on $C/{\sim_L}$: that is, for any $c \in C$ and $h \in H$, $(\lambda h)c \sim_L c$.*

To put it another way, $\lambda(H)$ stabilises the $\sim_L$-equivalence classes.

In particular, $\lambda(H) \leq G$ acts on *create* $A \subseteq C$ just as H acts on A.

Let us identify A with the set *create* A and take this as the transversal of $\sim_A$; and let L be the set *put* Ω_A C, and take these elements as the transversal of $\sim_L$. Observe that (in this restricted setting) *create* $a \sim_L$ *create* b for any $a, b \in A$, and also *get* (*put* Ω_A c) $=$ *get* (*put* Ω_A d)($= \Omega_A$) for any $c, d \in C$, so we can picture the elements of *create* A laid out as the bottom row and the elements of L in the left-hand column of our grid, respectively. We can identify C with $L \times A$ via the bijection $c \mapsto$ (*put* Ω_A c, *create* (*get* c)).

Let us from now on elide λ and regard H as a subgroup of G via λ.[2]

In terms of our informal grid, elements of H stabilise the rows, permuting the elements of each row. Each row is permuted identically. Formally:

Lemma 15. *For any $h \in H$ and $(l, a) \in L \times A$ we have $h(l, a) = (l, ha)$.*

Next we consider the role of K.

Lemma 16. *The normal subgroup K of G acts, trivially, on $C/{\sim_A}$: that is, for any $c \in C$ and $k \in K$, $kc \sim_A c$.*

To put it another way, K stabilises the $\sim_A$-equivalence classes. In particular, K acts on L. In terms of our informal grid, elements of K stabilise the columns, possibly permuting the elements of each column individually. Unlike H acting on rows, however, K does not necessarily do the same permutation on each column. Suppose for a moment that we are not given G with its action on $L \times A = C$, but instead are given just H and K, together with K's action on L (that is, on the left-hand column of the informal grid only) and H's action on A (that is, on the bottom row of the grid). We may ask, does this information determine the full action of G on $L \times A$? If not, to what extent does it constrain it? Since we know that H acts in the same way on every row, so its action on C is determined by its action on A, the interesting part is how K can act on a general element.

Lemma 17. *Let $k \in K$ and $(l, a) \in L \times A$. Then $k(l, a)$ can be written as $((hkh^{-1})l, a)$ where $h \in H$ satisfies $ha = \Omega_A$.*

Putting these calculations together, we see that the action of G on C can be composed from the actions of H on A and of K on L, thus:

$$(kh)(l, a) = ((h_1 k h_1^{-1})l, ha)$$

where $h_1 a = \Omega_A$.

[2] That is, as usual we can safely elide the distinction between internal and external semi-direct products.

In general there is a genuine choice of element of H – or equivalently, a genuine choice of semidirect product, that is, of homomorphisms in our short exact sequence – so that the actions of H on A and K on L, devoid of information about how the two groups are connected, do not completely determine G with its action on C. We also need an oracle to make the necessary choices, or, equivalently, to be given the homomorphism μ which determines which of the various semidirect products of H with K is intended.

We should, however, observe two special cases. First, if the action of H on A is such that there is always a unique element h such that $ha = \Omega_A$, then each choice is unique, and the actions of H and K on A and L respectively will completely determine the action of G on C. Second, if G is actually the *direct* product $K \times H$ – that is, elements of K commute with elements of H – then the action of G on C simplifies to the pointwise action

$$(k,h)(l,a) = (kl,ha)$$

as expected, there is no choice to be made, and again the actions of K and H on L and A do completely determine the action of G on C. This means, informally, that in this special case an edit acts independently on the part of the concrete model from C that's retained in the abstract view A and on the part which is discarded by the abstraction.

5 Building Lenses from Sequences

To show that we really do have an alternative way of looking at this world, we now need to consider the other direction.

Suppose we are given a short exact sequence of monoids

$$1 \to K \to G \overset{\mu}{\to} H \to 1$$

in which G acts on a set C (equipped with a trivial element Ω_C) and H acts on a set A (equipped with trivial element Ω_A). We can read this as telling us how to translate edits on C to edits on A: that is, it already gives us a (unidirectional) model transformation.

If we are given, additionally, an injective function $\lambda : H \longrightarrow G$ such that $\mu\lambda$ is the identity function on H, we can regard this as a bidirectional transformation: it tells us how to translate edits in both directions.

However, it does not necessarily correspond to a lens. Fundamentally the issue is this. Lenses work in the absence of any intentional information about the edits a user has made to the models: the lens only sees the modified models. In principle, there is no reason why we should not define a different kind of bidirectional transformation that does take notice of *how* the user achieved their changes. Two different edits might have the same effect on a model in C (rsp. A), but their images under μ (rsp. λ) might legitimately have different effects on a corresponding model in A (rsp. C).

We will always require that the action of G on C and the action of H on A are transitive, so that there always is some edit that will take the current model to the desired modified model. Beyond this, different choices might represent different means of editing models.

For the rest of this section we restrict attention to those sequences from which lenses can be defined:

Definition 11. *A sequence of monoids as described above is lens-like if it satisfies the following two conditions:*

LL1: if $g_1\Omega_C = g_2\Omega_C$ *then* $\mu g_1\Omega_A = \mu g_2\Omega_A$
LL2: for all $g \in G$ *and for all* $h \in H$*, we have*

$$h(\mu g)\Omega_A = (\mu g)\Omega_A \Rightarrow (\lambda h)g\Omega_C = g\Omega_C$$

Note that any sequence which arises from a lens by the construction in Section 4 is lens-like, as expected, by Lemmas 5 and 9.

Given a lens-like sequence, we can define a get function as follows. Given $c \in C$, let $g \in G$ be any element such that $g\Omega_C = c$; then

$$get\ c =_{\text{def}} \mu g\Omega_A$$

Lemma 18. *get is well-defined*

Lemma 19. *get and the group action of* G *on* C *satisfy the original compatibility condition* Compat.

Our definition of *put* involves making a choice, for each pair $a \in A$ and $c \in C$, of an element of the group H which has the desired effect; different choices may give different *put* functions, so our definition is parameterised on an oracle. This is not surprising, in the light of the many choices of put function discussed earlier (Lemma 3).

Suppose we have an oracle which, given arguments a and c, returns $h \in H$ such that $a = h(get\ c)$. (At least one such element is always guaranteed to exist by transitivity of the action of H on A.) Then

$$put\ a\ c =_{\text{def}} \lambda hc$$

Theorem 2. *The put and get functions defined above comprise a lens (for any oracle).*

Theorem 3. *If this sequence was in fact constructed from a lens* l *as described in Section 4, then the lens we construct from the sequence is exactly* l *(and in particular, it does not then depend on our choice of oracle).*

Lemma 20. *If, further,* λ *is a group homomorphism, so that it splits the sequence, and in addition* either *of the following holds,*

1. G *and* H *are groups; or*
2. *we have the property that* $h(get\ c) = h'(get\ c) \Rightarrow (\lambda h)c = (\lambda h')c$

then the lens is very well-behaved.

6 Conclusions and Further Work

In this paper we have described an algebraic framework in which to think about bidirectional transformations between sets of models. We have focused on an important special case, where one of the models is an abstraction of the other, and we have shown how to translate key elements of the body of work on lenses into algebraic terms. The lens framework was invented with the pragmatic needs of transformation programmers in mind: yet, that it fits so neatly into the algebraic framework suggests that the choice of laws it embodies are canonical within its region of the transformation language design space.

Much remains to be done, especially in exploiting the algebraic framework to give new (and/or easier) insight into how edit structures and transformations can be composed, and to explore beyond the boundaries of the lens framework. On the other hand, within those boundaries, it would be interesting to incorporate the work on dictionary and skeleton lenses from [1] (where wreath products clearly have a role to play) and on lenses up to equivalences from [3]. Looking more widely, it is to be hoped that the algebraic approach will also be useful in integrating different approaches to bidirectional transformations, including those from the graph transformation community; this may shed light on the design space of bidirectional transformation languages and thus contribute, ultimately, to the development of more useful languages for model-driven development.

From a theoretical point of view, it would be interesting to widen the search for connections into the fields of topology and category theory, and to understand the connections with earlier work such as [4] better. Finally, a major area of future work is to understand the connections with graph grammars, especially triple graph grammars.

Acknowledgements. I thank the PC chairs of ICGT, Reiko Heckel and Gabriele Taentzer, for the invitation to write this paper and for comments on a draft. I thank Benjamin Pierce for many helpful discussions, and the many people who have commented on my earlier work on transformations.

References

1. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California (January 2008)
2. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems 29(3), 17 (2007); Preliminary version presented at the Workshop on Programming Language Technologies for XML (PLAN-X) (2004); extended abstract presented at Principles of Programming Languages (POPL) (2005)
3. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: Proceedings of International Conference on Functional Programming (to appear, 2008), http://www.cis.upenn.edu/~jnfoster/papers/quotient-lenses.pdf

4. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Trans. Database Syst. 13(4), 486–524 (1988)
5. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
6. Stevens, P.: A landscape of bidirectional model transformations. In: Post-proceedings of GTTSE 2007 (to appear, 2008)

Discovery, Verification and Conformance of Workflows with Cancellation

W.M.P. van der Aalst

Department of Mathematics and Computer Science,
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
w.m.p.v.d.aalst@tue.nl

Abstract. Petri nets are frequently used for the modeling and analysis of workflows. Their graphical nature, well-defined semantics, and analysis techniques are attractive as information systems become more "process-aware". Unfortunately, the classical Petri net has problems modeling cancellation in a succinct and direct manner. Modeling cancellation regions in a classical net is impossible or results in a "spaghetti-like" net. Cancellation regions are supported by many workflow management systems, but these systems do not support advanced analysis techniques (process mining, verification, performance analysis, etc.). This paper proposes to use *reset workflow nets* (RWF-nets) and discusses (1) the *discovery* of RWF-nets (i.e., extracting information from event logs to construct such models), (2) the *verification* of RWF-nets (i.e., checking whether a workflow process has deadlocks, livelocks, etc.), and (3) the *conformance* of an event log with respect to a RWF-net (i.e., comparing real with modeled behavior).

Keywords: Petri Nets, Reset Nets, Soundness, Verification, and Process Mining.

1 Introduction

Information systems have become "process-aware", i.e., they are driven by process models [26]. Often the goal is to automatically configure systems based on process models rather than coding the control-flow logic using some conventional programming language. Early examples of process-aware information systems were called WorkFlow Management (WFM) systems [5,30,36,50]. In more recent years, vendors prefer the term Business Process Management (BPM) systems. BPM systems have a wider scope than the classical WFM systems and are not just focusing on process automation. BPM systems tend to provide more support for various forms of analysis and management support. Both WFM and BPM aim to support operational processes that we refer to as "workflow processes" or simply "workflows".

The flow-oriented nature of workflow processes makes the Petri net formalism a natural candidate for the modeling and analysis of workflows. Figure 1 shows

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 18–37, 2008.

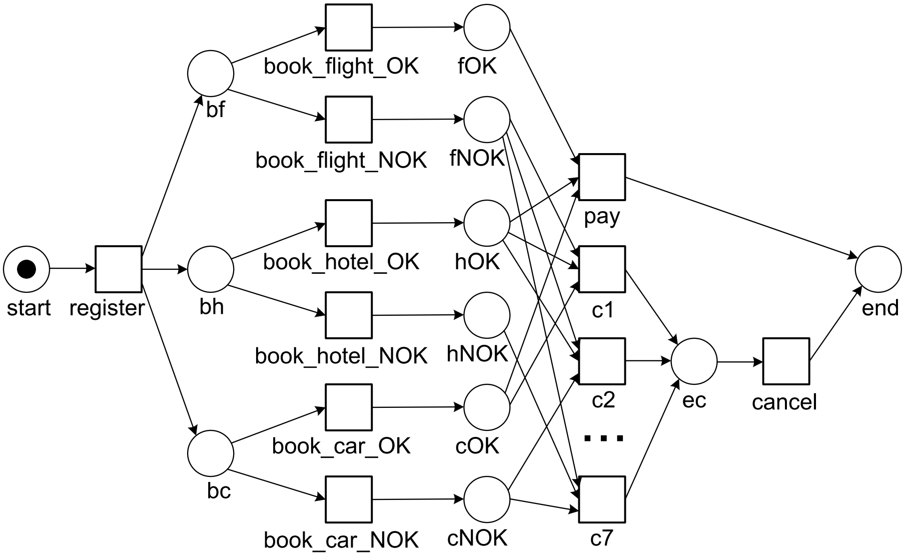


Fig. 1. A WF-net N_1 modeling the booking of trips. Note that after one "NOK" the trip will be cancelled eventually. There are $2^3 - 1 = 7$ situations modeled by transitions $c1, c2, \ldots, , c7$. Although the model is already complicated it fails to model that there should not be any booking activities after the first "NOK", because the trip will be cancelled anyway.

a so-called *workflow net* (WF-net), i.e., a Petri net with a start place and an end place such that all nodes are on some path from start to end. WF-nets were introduced in [1,2].[1]

The WF-net in Figure 1 models the booking of trips. After registration a flight, a hotel, and a car are booked. Each of these booking activities can succeed ("OK") or fail ("NOK"). For reasons of simplicity, only the successful or unsuccessful completion of these activities is shown in Figure 1 (i.e., activities are considered to be atomic). If all booking activities succeed, then a payment follows. If one of them fails, a cancellation activity follows. Since each of the 3 booking activities can succeed or fail, there are $2^3 = 8$ scenarios. Only for one of these eight scenarios, payment is executed. For all other seven scenarios, the trip is cancelled.

Figure 1 is already rather complex but fails to model that there should not be any unnecessary work, i.e., after the first failure ("NOK"), no other booking activities should be executed as cancellation is inevitable. To model this, the seven c-transitions $(c1, c2, \ldots, , c7)$ are not adequate as there are $3^3 - 2^3 = 19$ possible states after registration and before payment/cancellation in which there is at least one "NOK". Hence, $19 - 7 = 12$ additional c-transitions are needed to

[1] According to Google Scholar (visited on April 23rd, 2008), [2] got more than thousand references illustrating the interest in the topic. In fact, [2] is the second most cited workflow paper after [30].

capture this. Moreover, we simplified the model by assuming that the activities are atomic. This is not realistic because the booking of the flight, the hotel, and the car may happen in parallel and if one of them fails the other ones need to be withdrawn. If we incorporate this in the model, there there are $4^3 - 3^3 = 37$ states after registration and before payment/cancellation in which there is at least one "NOK".[2] This implies that to fully model the example 37 c-transitions are needed to remove the tokens from the right places. This illustrates that cancellation is difficult to model in WF-nets. Therefore, we propose to use *reset arcs* [24,25,29]. A reset arc removes tokens from a place but does not block the corresponding transition if the place is empty. This is a very useful construct that allows for the modeling of various cancellation operations supported by contemporary workflow languages, e.g., the withdraw construct of Staffware, the cancellation region of YAWL, the cancel event of BPMN, etc. In our example, the 37 c-transitions that are needed to remove the tokens from the right places, can be replaced by a single transition with reset arcs. This illustrates the usefulness and relevance of reset arcs. Therefore, we introduce the so-called *Reset WorkFlow nets* (RWF-nets) as an extension to the classical WF-nets [1,2].

Taking RWF-nets as a starting point we explore a spectrum of analysis questions. Concretely, we investigate the following three challenges:

- *Discovering RWF-Nets.* Given an event log extracted from some database, transaction log, or set of use cases/audit trails, we want to automatically infer a process model. Since cancellation is important when modeling workflows, it is also important to discover cancellations when observing systems and processes.
- *Verification of RWF-Nets.* Many of the modern workflow languages offer some form of cancellation. Hence, it is important to be able to verify such models and point out deadlocks, livelocks, etc.
- *Conformance with respect to a RWF-Net.* The alignment of model and systems on the one hand and real-life processes on the other often leaves much to be desired. Therefore, it is important to be able to compare event logs with models. Since real-live processes and their models exhibit cancellation, it is important to take this into account when checking conformance of event logs and models.

The remainder of this paper is organized as follows. First, we present reset workflow nets and introduce three challenges (discovery, verification, and conformance), followed by Section 3 which introduces the concept of event logs. Section 4 focuses on the verification of workflows with cancellation. Section 5 shows that conformance can be checked by "playing the token game" based on the event log. Section 6 presents the challenge to discover reset workflow nets. An overview of related work is provided in Section 7 and Section 8 concludes the paper.

[2] Each of the booking activities has 4 states: enabled, running, succeeded (i.e., "OK"), and failed (i.e., "NOK"). Therefore, there are $4^3 = 64$ possible states and $3^3 = 27$ of these states are non-failure states.

2 Reset Workflow Nets

The WF-net in Figure 1 is a nice illustration of the inability of classical Petri nets to model cancellation. Therefore, we use *reset nets*, i.e., classical Petri net extended with reset arcs.

Definition 1 (Reset net). *A* reset net *is a tuple* (P, T, F, W, R)*, where:*

- (P, T, F) *is a classical Petri net with places* P*, transitions* T*, and flow relation* $F \subseteq (P \times T) \cup (T \times P)$*,*
- $W \in F \rightarrow I\!N \setminus \{0\}$ *is an (arc) weight function, and*
- $R \in T \rightarrow 2^P$ *is a function defining reset arcs.*

A reset net extends the classical Petri net with reset arcs. These are used to remove all tokens from a place independent of the number of tokens. $R(t)$ is the set of places that are emptied when firing t. Also note that we are using arc weights. Arc weights specify the number of tokens to be consumed or produced. $W(p,t)$ is the number of tokens transition t consumes from input place p and $W(t,p)$ is the number of tokens transition t produces for output place p.

Figure 2 shows a reset net. In this example all arc weights are 1, i.e., $W(x,y) = 1$ for $(x,y) \in F$. Transition c has seven reset arcs connected to it. When c fires all tokens are removed from places *fOK*, *hOK*, *cOK*, *NOK*, *bf*, *bh*, and *bc*. For the enabling of c these reset arcs is irrelevant, i.e., c is enabled if and only if there is a token in place *NOK*.

The state of a reset net, also referred to as *marking*, is described as a multiset. Therefore, we introduce some notation. Let A be a set, e.g., the set of places P. $\mathbb{B}(A) = A \rightarrow \mathbb{N}$ is the *set of multi-sets* (bags) over A, i.e., $X \in \mathbb{B}(A)$ is a multi-set where for each $a \in A$: $X(a)$ denotes the number of times a is included in the multi-set. The sum of two multi-sets $(X+Y)$, the difference $(X-Y)$, the presence of an element in a multi-set $(x \in X)$, and the notion of sub-multi-set $(X \leq Y)$ are defined in a straightforward way and they can handle a mixture of sets and multi-sets. $\pi_{A'}(X)$ is the projection of X onto $A' \subseteq A$, i.e., $(\pi_{A'}(X))(a) = X(a)$ if $a \in A'$ and $(\pi_{A'}(X))(a) = 0$ if $a \notin A'$.

To represent a concrete multi-set we use square brackets, e.g., [*fOK*, *hOK*, *cOK*] is the marking with a token in each of the “OK places” and $[NOK^3]$ is the marking with three tokens in place *NOK*.

Because of the arc weights the classical preset and postset operators return bags rather than sets: $\bullet a = [x^{W(x,y)} \mid (x,y) \in F \ \wedge \ a = y]$ and $a\bullet = [y^{W(x,y)} \mid (x,y) \in F \ \wedge \ a = x]$. For example, $\bullet pay = [fOK, hOK, cOK]$ is the bag of input places of *pay* and $pay\bullet = [end]$ is the bag of output places of *pay*.

Now we can formalize the notions of enabling and firing.

Definition 2 (Firing rule). *Let* $N = (P, T, F, W, R)$ *be a reset net and* $M \in I\!B(P)$ *be a marking.*

- *A transition* $t \in T$ *is enabled, notation* $(N, M)[t\rangle$*, if and only if,* $M \geq \bullet t$*.*
- *An enabled transition* t *can fire while changing the state to* M'*, notation* $(N, M)[t\rangle(N, M')$*, if and only if,* $M' = \pi_{P \setminus R(t)}(M - \bullet t) + t\bullet$*.*

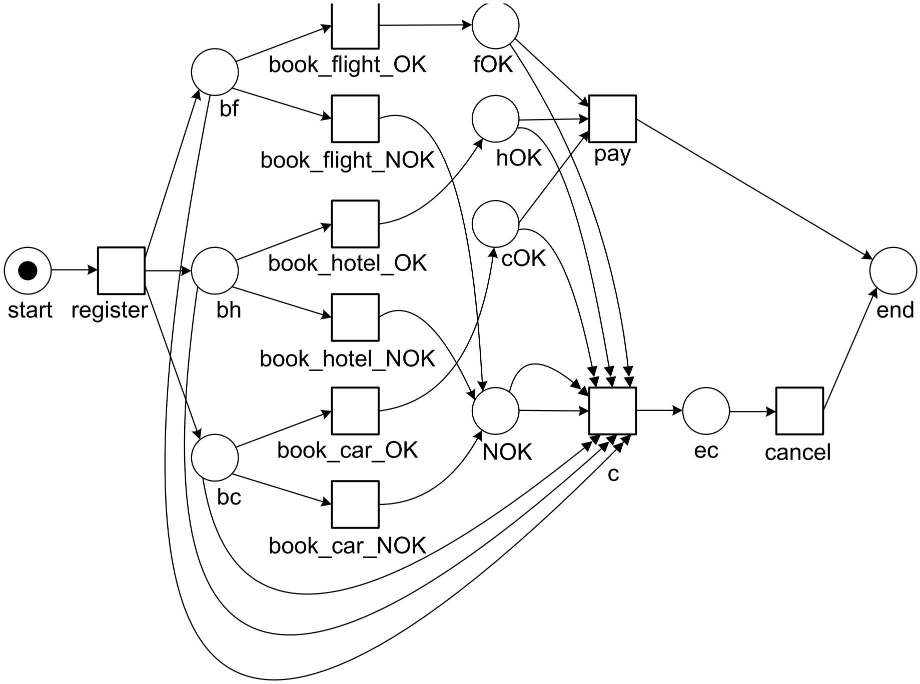


Fig. 2. A RWF-net N_2 modeling the booking of trips. Note that unlike the WF-net in Figure 1, unnecessary work is avoided. Moreover, the number of nodes to handle the cancellation is constant and the number of arcs is linear in the number of booking activities (flight, hotel, car, etc.) and activity states (enabled, running, succeeded, failed, etc.).

The resulting marking $M' = \pi_{P \setminus R(t)}(M - \bullet t) + t\bullet$ is obtained by first removing the tokens required for enabling: $M - \bullet t$. Then all tokens are removed from the reset places of t using projection. Applying function $\pi_{P \setminus R(t)}$ removes all tokens except the ones in the non-reset places $P \setminus R(t)$. Finally, the specified numbers of tokens are added to the output places. Note that $t\bullet$ is a *bag* of tokens.

$(N, M)[t\rangle(N, M')$ defines how a Petri net can move from one marking to another by firing a transition. We can extend this notion to firing sequences. Suppose $\sigma = \langle t_1, t_2, \ldots, t_n \rangle$ is a sequence of transitions present in some Petri net N with initial marking M. $(N, M)[\sigma\rangle(N, M')$ means that there exists a sequence of markings $\langle M_0, M_1, \ldots, M_n \rangle$ where $M_0 = M$, $M_n = M'$, such that for any $0 \leq i < n$: $(N, M_i)[t_{i+1}\rangle(N, M_{i+1})$. Using this notation we define the set of reachable markings $R(N, M)$ as follows: $R(N, M) = \{M' \in \mathbb{B}(P) \mid \exists_\sigma (N, M)[\sigma\rangle(N, M')\}$. Note that by definition $M \in R(N, M)$ because the initial marking M is trivially reachable via the empty sequence ($n = 0$).

We would like to emphasize that any reset net with arc weights can be transformed into a reset net without arc weights, i.e., all arcs have weight 1. Therefore, in proofs can assume arc weights of 1 when convenient and still use them in constructs. See [6] for a construction.

The idea of a workflow process is that many *cases* (also called *process instances*) are handled in a uniform manner. The workflow definition describes the ordering of *activities* to be executed for each case including a clear *start state* and *end state*. These basic assumptions lead to the notion of a *WorkFlow net* (WF-net) [1,2] which can easily be extended in the presence of reset arcs.

Definition 3 (RWF-net). *An reset net* $N = (P, T, F, W, R)$ *is a Reset WorkFlow net (RWF-net) if and only if*

- *There is a single source place* i*, i.e.,* $\{p \in P \mid \bullet p = \emptyset\} = \{i\}$*.*
- *There is a single sink place* o*, i.e.,* $\{p \in P \mid p\bullet = \emptyset\} = \{o\}$*.*
- *Every node is on a path from* i *to* o*, i.e., for any* $n \in P \cup T$*:* $(i, n) \in F^*$ *and* $(n, o) \in F^*$*.*
- *There is no reset arc connected to the sink place, i.e.,* $\forall_{t \in T}\ o \notin R(t)$*.*

Figures 1 and 2 both show a RWF-net. The requirement that $\forall_{t \in T}\ o \notin R(t)$ has been added to emphasize that termination should be irreversible, i.e., it is not allowed to complete (put a token in o) and then undo this completion (remove the token from o).

Let us now compare figures 1 and 2 showing RWF-nets N_1 and N_2 respectively. In the the original net without reset arcs (N_1) the number of cancellation transitions is exponential in the number of bookings while in the second net (N_2) there is just one cancellation transition and the number of reset arcs is linear in the number of bookings. Also note that in N_2 tokens are also removed from the input places of the booking activities to make sure than no unnecessary work is conducted. Extending N_1 to obtain the same behavior requires the addition of 12 more cancellation transitions. This clearly shows the benefits of using reset arcs. Moreover, figures 1 and 2 also illustrate the need for the modeling of cancellations in real-life workflow processes.

3 Event Logs

Traditionally, the focus of workflow analysis at design-time has been on model-based verification and simulation while at run-time the focus has been on measuring simple key performance indicators such as flow times, service levels, etc. Because more and more information about processes is recorded by information systems in the form of so-called “event logs”, it seems vital to also use this information while analyzing processes. A wide variety of process-aware information systems [26] is recording excellent data on actual events taking place. ERP (Enterprise Resource Planning), WFM (WorkFlow Management), CRM (Customer Relationship Management), SCM (Supply Chain Management), and PDM (Product Data Management) systems are examples of such systems. Despite the omnipresence and richness of these event logs, most software vendors have been focusing on relatively simple questions *under the assumption that the process is fixed and known*, e.g., the calculation of simple performance metrics like utilization and flow time. However, in many domains processes are evolving and people typically have an oversimplified and incorrect view of the actual business processes. Therefore, *process mining* techniques attempt to extract non-trivial and useful information from event logs. One aspect of process mining is *control-flow discovery*, i.e., automatically constructing a process model (e.g., a Petri net) describing the causal dependencies between activities [11,12,17,20].

Later in this paper, we discuss process discovery and conformance checking using RWF-nets. These are particular process mining techniques that require event logs as input. Therefore, we define the notion of an event log.

Definition 4 (Event log). *Let A be a set of activities. A* trace *σ can be described as a sequence of activities, i.e., $\sigma \in A^*$. An event log L is a multiset of traces, i.e., $L \in \mathbb{B}(A^*)$.*

A trace can be considered as the execution path of a single process instance (case). Note that this is a rather simplified view, i.e., in real life events have timestamps (When did the activity happen?), resource information (Who executed the activity?), data (What information was used and produced?), etc. However, for this paper we focus on the control-flow only. A trace possible according to Figure 2 is $\sigma = \langle \mathit{register}, \mathit{book\_flight\_NOK}, c, \mathit{cancel} \rangle$. An example event log consisting of 5 traces is $L = [\langle \mathit{register}, \mathit{book\_hotel\_NOK}, c, \mathit{cancel} \rangle^3, \langle \mathit{register}, \mathit{book\_hotel\_OK}, \mathit{book\_car\_OK}, \mathit{book\_flight\_OK}, \mathit{pay} \rangle^2]$.

As already indicated in Section 1, this paper focuses on three challenges:

- *Discovering RWF-Nets.* Given an event log L, we want to infer a RWF-net N.
- *Verification of RWF-Nets.* Given a RWF-net N, we want to discover errors such as deadlocks, etc.
- *Conformance with respect to a RWF-Net.* Given an event log L and a RWF-net N, we want to discover discrepancies between L and N.

The remainder of this paper will focus on these three challenges. We start by elaborating on the verification of RWF-nets.

4 Verification of RWF-Nets

Based on the notion of RWF-nets we now investigate the fundamental question: “Is the workflow correct?”. If one has domain knowledge, this question can be answered in many different ways. However, without domain knowledge one can only resort to generic questions such as: “Does the workflow terminate?”, “Are there any deadlocks?”, “Is it possible to execute activity A?”, etc. Such kinds of generic questions triggered the definition of *soundness* [1,2]. Different soundness notions have been proposed, e.g., k-soundness [32,33], weak soundness [39], generalized soundness [32,33], relaxed soundness [21], etc. However, here we focus on the original definition given in [1].

Definition 5 (Classical soundness [1,2]). *Let $N = (P, T, F, W, R)$ be a RWF-net. N is sound if and only if the following three requirements are satisfied:*

- *Option to complete:* $\forall_{M \in R(N,[i])}\ [o] \in R(N, M)$.
- *Proper completion:* $\forall_{M \in R(N,[i])}\ (M \geq [o]) \Rightarrow (M = [o])$.
- *No dead transitions:* $\forall_{t \in T}\ \exists_{M \in R(N,[i])}\ (N, M)[t\rangle$.

A RWF-net such as the one sketched in Figure 1 is sound if and only if the following three requirements are satisfied: (1) *option to complete*: for each case it is always still possible to reach the state which just marks place *end*, (2) *proper completion*: if place *end* is marked all other places are empty for a given case, and (3) *no dead transitions*: it should be possible to execute an arbitrary activity by following the appropriate route through the RWF-net. It is easy to see that N_1 and N_2 (figures 1 and 2) are sound.

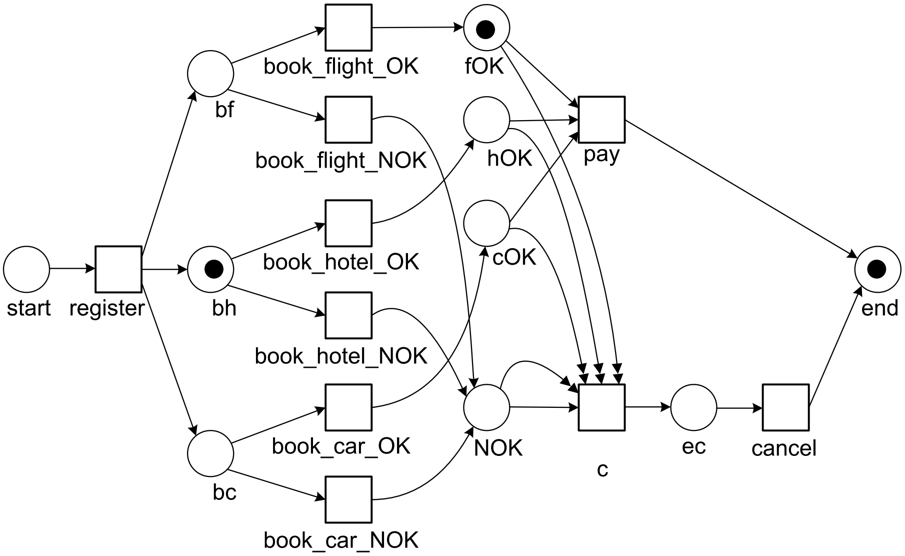


Fig. 3. A RWF-net N_3 that is not sound. From the initial marking $[start]$ e.g. the marking shown (i.e., $[bh, fOK, end]$) is reachable. This shows that the first two requirements stated in Definition 5 do not hold.

RWF-net N_3 shown in Figure 3 is an example of a workflow that is not sound. Since *c* does not remove tokens from the places before the booking activities, tokens may be left behind. In fact, it is still possible to book a hotel after transition *cancel* has put a token in *end* (cf. Figure 3). This example shows that it is easy to make errors when modeling workflows with cancellation.

In [1,2] it was shown that soundness is decidable for WF-nets, i.e., RWF-nets without reset arcs. A WF-net $N = (P, T, F)$ (without reset arcs and arc weights) is sound if and only if the short-circuited net $(\overline{N}, [i])$ with $\overline{N} = (P, T \cup \{t^*\}, F \cup \{(o, t^*), (t^*, i)\})$ is live and bounded. Since liveness and boundedness are both decidable, soundness is also decidable. For some subclasses (e.g., free-choice nets), this is even decidable in polynomial time [1,2].

Since the mid-nineties many people have been looking at the verification of workflows. These papers all assume some underlying model (e.g., WF-nets) and some correctness criterion (e.g., soundness). However, in many cases a rather simple model is used (WF-nets or even less expressive) and practical features such as *cancellation* are missing. Many practical languages have a cancellation feature, e.g., Staffware has a withdraw construct, YAWL has a cancellation region, BPMN has cancel, compensate, and error events, etc. Therefore, it is interesting to investigate the notion of soundness in the context of RWF-nets, i.e., WF-nets with reset arcs [24,25,29]. Unfortunately, soundness is *not* decidable for RWF-nets with reset arcs.

Theorem 1 (Undecidability of soundness). *Soundness is undecidable for RWF-nets.*

For a proof we refer to [6]. Although far from trivial, it is possible to construct a RWF-net N' given an arbitrary reset net N such that N' is sound if and only if M' is *not* reachable from M in N. Since reachability is undecidable for reset nets [24,25,29], this implies that soundness is also undecidable for RWF-nets.

Theorem 1 is non-trivial because properties such as coverability (Is it possible to reach a marking M' that covers M, i.e., $M' \geq M$?) are decidable for reset nets.

Note that although soundness is undecidable for RWF-nets, for many representatives of this class, it may still be possible to conclude soundness or non-soundness. There may be rules of the form "If WF-net N has property X, then N is sound" or "If WF-net N has property Y, then N is not sound". As shown in [41] it is possible to find many errors using such an approach. In [41] a set of more than 2000 process models from practice (including more than 600 processes from the SAP reference model) was analyzed. It could be shown that at least 10 percent of these models is not sound. These examples show that *even if soundness is undecidable, errors can be discovered*. Similarly, for many models it is still possible to guarantee soundness even if the general verification problem is undecidable.

In the related work section, we provide some pointers to analysis techniques using brute force (e.g. coverability graphs), structural techniques (invariants), and/or reduction rules. For example, RWF-nets can be reduced using the reduction rules presented in [48] to speed-up analysis and improve diagnostics.

5 Conformance with Respect to a RWF-Net

As indicated in Section 3, lion's share of analysis efforts has been devoted to model-based analysis (verification, simulation, etc.) and measuring simple performance indicators. However, given the abundance of event logs it is interesting to "discover models" based on event logs (see Section 6) or to measure the conformance of existing models based on the real behavior recorded in logs. In this section, we focus on the latter question, i.e., "Do the model and the log *conform* to each other?". Conformance checking aims at the detection of inconsistencies between a process model N and its corresponding execution log L, and their quantification by the formation of metrics. In [4,42,43] two basic conformance notions have been identified (fitness and appropriateness). First of all, the *fitness* between the log and the model is measured (i.e., "Does the observed process comply with the control flow specified by the process model?"). Second, the *appropriateness* of the model can be analyzed with respect to the log (i.e., "Does the model describe the observed process in a suitable way?"). Appropriateness can be evaluated from both a *structural* and a *behavioral* perspective [43].

In this paper, we only consider fitness. However, it is important to stress that a model with good fitness may not be appropriate. For example, the model with just a single place that serves as a self-loop for all transitions T is able to parse any trace in T^* [4,42,43].

One way to measure the fit between event logs and process models is to "replay" the log in the model and somehow measure the mismatch. The replay of every trace starts with the marking of the initial place in the model, e.g., $[start]$ in Figure 1. Then, the transitions that belong to the logged events in the trace are fired one after another. While replay progresses, we count the number of tokens that had to be created artificially (i.e., the transition belonging to the logged event was not enabled and therefore could not be *successfully executed*) and the number of tokens that were left in the model, which indicate that the process was not *properly completed*. Based on counting the number of missing tokens during replay and the number of remaining tokens after replay, we define a function f that has a value between 0 (=poor fitness) and 1 (=good fitness).

Definition 6 (Fitness). *Let $N = (P, T, F, W, R)$ be a RWF-net and let $L \in \mathbb{B}(A^*)$ be an event log where we assume that $T = A$. Let k be the number of traces in event log L. For each log trace i ($1 \leq i \leq k$), m_i is the number of missing tokens, r_i is the number of remaining tokens, c_i is the number of consumed tokens, and p_i is the number of produced tokens during log replay of trace i. The token-based fitness metric f is defined as follows:*

$$f(N, L) = \frac{1}{2}(1 - \frac{\sum_{i=1}^{k} m_i}{\sum_{i=1}^{k} c_i}) + \frac{1}{2}(1 - \frac{\sum_{i=1}^{k} r_i}{\sum_{i=1}^{k} p_i})$$

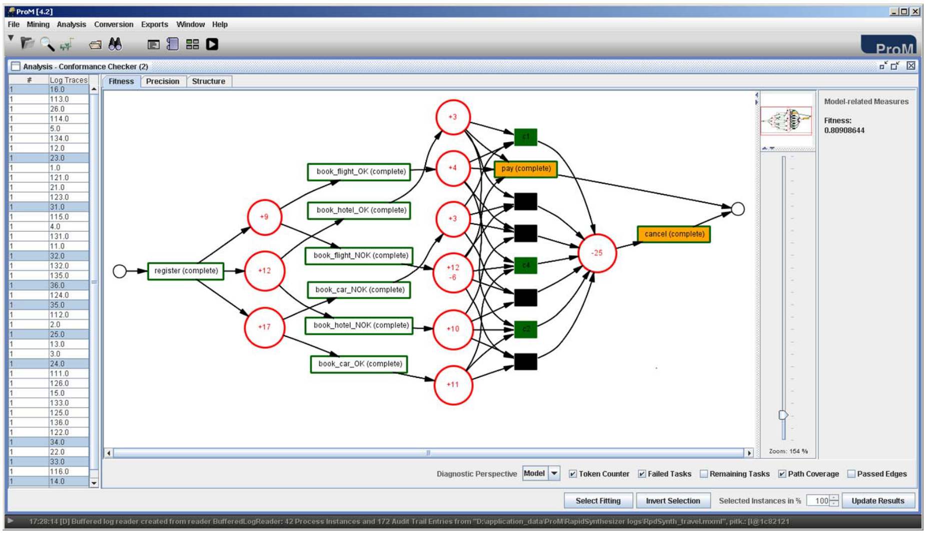
Note that the above definition is rather informal. In [42,43] this metric was defined for WF-nets, i.e., workflows without cancellation. However, as shown here the metric can easily be extended for RWF-nets. Let us consider some trace i consisting of a sequence of n events $\sigma_i = \langle e_1, e_2, \ldots, e_n\rangle \in T^*$. The initial state of N is $[i]$ and the desired end state is $[o]$. If $(N, [i])[\sigma_i\rangle(N, [o])$, then there is a perfect fit, i.e., the trace can be replayed without leaving tokens behind. So, $(N, [i])[\sigma_i\rangle(N, [o])$ if and only if $f(N, [\sigma_i]) = 1$. Let $\sigma_1 = \langle \mathit{register}, \mathit{book\_hotel\_NOK}, c, \mathit{cancel}\rangle$. It is easy to see that $f(N_2, [\sigma_1]) = 1$, i.e., the trace can be replayed without missing or remaining tokens. (Recall that N_2 is the RWF-net shown in Figure 2.) Now consider $\sigma_2 = \langle \mathit{register}, \mathit{book\_hotel\_OK}, c, \mathit{pay}\rangle$. It is possible to execute the partial trace $\langle \mathit{register}, \mathit{book\_hotel\_OK}\rangle$. However, to execute c, there has to be a token in NOK (i.e., one token is missing). If we force c to fire anyway, the resulting state is $[ec]$. In this state, we cannot fire *pay* as there are three missing tokens. This brings the number of missing tokens to 4. After forcing *pay* to fire, the resulting state is $[ec, end]$. Hence one token remains in place *ec*.

We can show the calculation of the values m_2, r_2, c_2, and p_2 step-by-step using four temporary variables. Initially, $m = 0$ (no missing tokens), $r = 0$ (no remaining tokens), $c = 0$ (no consumed tokens), and $p = 1$ (prior to the execution of *register* the net is in state $[start]$, so the environment already put a token in the initial place). After *start* fires state $[bf, bh, bc]$ is obtained and $m = 0$, $c = 0+1 = 1$, and $p = 1+3 = 4$. After *book_hotel_OK* fires marking $[bf, bc, hOK]$ is reached and $m = 0$, $c = 1 + 1 = 2$, and $p = 4 + 1 = 5$. After c fires state $[ec]$ is obtained and $m = 0 + 1 = 1$ (missing token in *NOK*), $c = 2 + 4 = 6$ (four tokens are removed, one by a "normal" arc and three by reset arcs), and $p = 5 + 1 = 6$. After *pay* fires state $[ec, end]$ is reached and $m = 1 + 3 = 4$, $c = 6 + 3 = 9$, and $p = 6 + 1 = 7$. Finally, the token is removed from *end* and the remaining token is recorded, i.e., $c = 9 + 1 = 10$ and $r = 1$. Note that in the calculation the marking of the source place is considered to be a production step while the removal of the token from the sink place is considered to be a consumption step. Also note that the removal of tokens through reset arcs is calculated as a consumption step. Hence, $m_2 = 4$, $r_2 = 1$, $c_2 = 10$, and $p_2 = 7$. Therefore, $f(N_2, [\sigma_2]) = \frac{1}{2}(1 - \frac{4}{10}) + \frac{1}{2}(1 - \frac{1}{7}) = \frac{51}{70} \cong 0.73$. The fitness of $f(N_2, [\sigma_1, \sigma_2]) = \frac{1}{2}(1 - \frac{0+4}{7+10}) + \frac{1}{2}(1 - \frac{0+1}{7+7}) = \frac{403}{476} \cong 0.85$.

Several definitions of fitness are possible. For example, Definition 6 gives equal weights to missing tokens and remaining tokens. By replacing the weights $\frac{1}{2}$ by e.g. weight $\frac{3}{4}$ and weight $\frac{1}{4}$ in Definition 6, more emphasis is put on problems related to missing tokens and less on proper termination.

Several metrics for fitness and various appropriateness notions have been implemented in ProM for WF-nets [42,43]. As shown in this section, these metrics can be adapted for RWF-nets in a straightforward manner. Figure 4 illustrates the current functionality of ProM. For a specific log, the fitness is calculated with respect to the WF-net shown in Figure 1. As Figure 4 shows the fitness is 0.80908644 (metric is shown in top right corner). Several places are annotated with one or two numbers. Positive numbers refer to remaining tokens (i.e., $\sum_{i=1}^{k} r_i$ for a particular place) and negative tokens refer to missing tokens (i.e., $\sum_{i=1}^{k} m_i$ for a particular place). The input place of *cancel* (i.e., place *ec* in Figure 1) has the number "-25" indicating that in the whole log there were 25 situations where according to the log *cancel* had to fire while according to the model this was not possible because *ec* was empty. As shown in Figure 4, nice diagnostics can be given showing were in the model mismatches occur and how severe they are.

Note that the transitions $c1, c2, \ldots, , c7$ in Figure 1 are depicted differently in the conformance checker (cf. Figure 4). The reason is that there are no events related to $c1, c2, \ldots, , c7$ in the log, i.e., these are "silent transitions" and cannot be observed. The conformance checker in ProM can deal with such situations using state-space analysis. The same technique is used to deal with "duplicates", i.e., two transitions having the same label. See [43] for details. Interestingly, all ideas from [43] can be generalized to workflows with cancellation.



#	Log Traces
1	16.0
1	113.0
1	26.0
1	114.0
1	5.0
1	134.0
1	12.0
1	23.0
1	1.0
1	121.0
1	21.0
1	123.0
1	31.0
1	115.0
1	4.0
1	131.0
1	11.0
1	32.0
1	132.0
1	135.0
1	36.0
1	124.0
1	35.0
1	112.0
1	2.0
1	25.0
1	13.0
1	3.0
1	24.0
1	111.0
1	126.0
1	15.0
1	133.0
1	125.0
1	136.0
1	122.0
1	34.0
1	22.0
1	33.0
1	116.0
1	14.0

Model-related Measures

Fitness:
0.80908644

Diagnostic Perspective: Model ☑ Token Counter ☑ Failed Tasks □ Remaining Tasks ☑ Path Coverage □ Passed Edges

Selected Instances in % 100

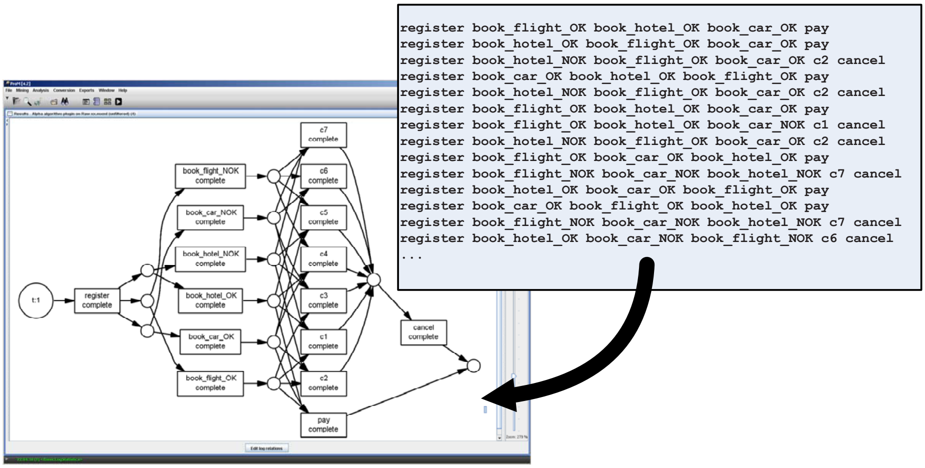
```
17:28:14 [D] Buffered log reader created from reader BufferedLogReader: 42 Process Instances and 172 Audit Trail Entries from "D:\application_data\ProM\RapidSynthesizer logs\RpdSynth_travel.mxml", pitk.: [I@1c82121
```

Fig. 4. The conformance checker in ProM applied to the WF-net shown in Figure 1 and an event log containing 42 cases and 172 events

6 Discovering RWF-Nets

The last challenge addressed in this paper is the discovery of workflows with cancellation, i.e., based on some event log L we want to automatically construct a RWF-net N that "captures" the behavior seen in the log. Many techniques have been proposed in literature [11,8,12,17,20,22,23,49]. However, *none of these techniques discovers workflow models with cancellation features.*

Figure 5 illustrates the concept of process discovery. Given an event log without any explicit process information, we want to discover a process model. On the right-hand side of Figure 5, a fragment of a larger event log is shown. As discussed in Section 3, event logs tend to have more information (e.g., timestamps, data, etc.), but here was assume that an event log is simply a multiset of traces. For example, the first trace in Figure 5 refers to a scenario were all bookings succeeded. The α-algorithm [11] is a very basic process mining algorithm that is able to discover the model shown in Figure 5. Since the traces shown correspond to possible traces of the WF-net N_1 shown in Figure 1, it is nice to see that the α-algorithm is actually able to discover N_1 (modulo renaming of places). The α-algorithm [11] is very simple but not very suitable for real-life applications. The algorithm makes strong assumptions about the routing constructs to be used and the completeness of the log. For example, it is not realistic that one actually observes the routing transitions $c1, c2, \ldots, , c7$. Unlike transition *cancel* which is a real activity, $c1, c2, \ldots, , c7$ have only been added for routing purposes. Fortunately, better process mining algorithms are available today (see Section 7). However, these *do not capture cancellation as the underlying models do not allow for a direct representation of such constructs.*



```
register book_flight_OK book_hotel_OK book_car_OK pay
register book_hotel_OK book_flight_OK book_car_OK pay
register book_hotel_NOK book_flight_OK book_car_OK c2 cancel
register book_car_OK book_hotel_OK book_flight_OK pay
register book_hotel_NOK book_flight_OK book_car_OK c2 cancel
register book_flight_OK book_hotel_OK book_car_OK pay
register book_flight_OK book_hotel_OK book_car_NOK c1 cancel
register book_hotel_NOK book_flight_OK book_car_OK c2 cancel
register book_flight_OK book_car_OK book_hotel_OK pay
register book_flight_NOK book_car_NOK book_hotel_NOK c7 cancel
register book_hotel_OK book_car_OK book_flight_OK pay
register book_car_OK book_flight_OK book_hotel_OK pay
register book_flight_NOK book_car_NOK book_hotel_NOK c7 cancel
register book_hotel_OK book_car_NOK book_flight_NOK c6 cancel
...
```

Fig. 5. Based on a complete event log, the α-algorithm [11] can discover the WF-net shown in Figure 1

The goal is to develop process mining algorithms that discover cancellations in event logs and treat cancellation as a basic construct. Concretely, we want to *discover a RWF-net N with "suitable" reset arcs based on some event log L*. Since we do not what to develop a process mining algorithm from scratch, we try to extend existing techniques.

The basic idea behind most of the existing process mining algorithms is to add a causal dependency based on an analysis of the log. For example, $a >_L b$ iff there is a trace in L where a is directly followed by b and $a \rightarrow_W b$ iff $a >_W b$ and $b \not>_W a$. Using such information places are added, e.g., a and b are connected through some place if $a \rightarrow_W b$. Hence, the places provide information about one activity triggering another activity. However, there is no explicit information in the log on disabling events (i.e., there is no "negative information" in the log). Therefore, we suggest to use existing algorithms and do some post-processing using reset arcs as a cleanup. Below is an informal sketch of the basic idea:

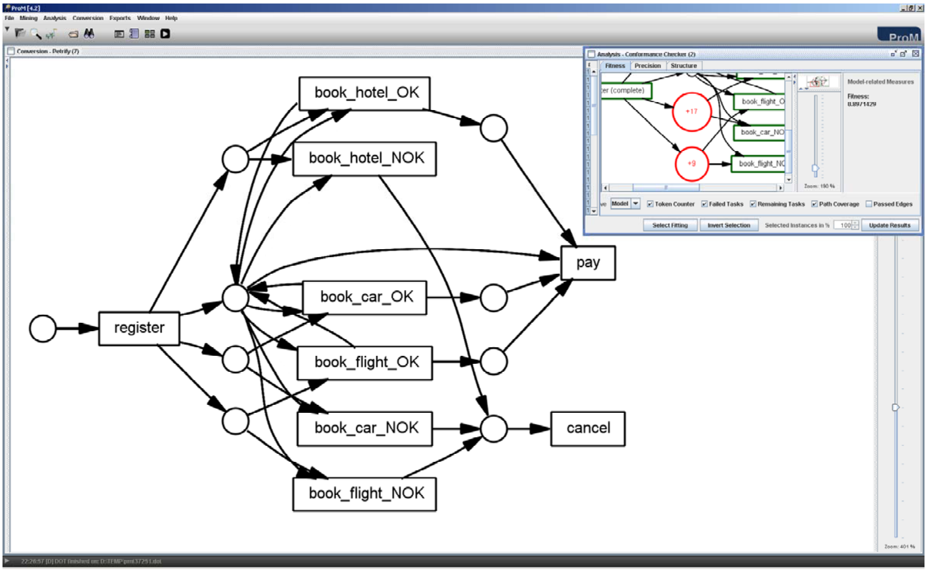
Step 1. Given an event log L construct a RWF-net $N = (P, T, F, W, R)$ using conventional process mining techniques (initially $R(t) = \emptyset$ for all $t \in T$). It is best to use a technique that avoids blocking transitions, i.e., no missing tokens (m_i) in the sense of Definition 6.

Step 2. Construct a relation $\not\gg_L \subseteq T \times T$ such that $a \not\gg_L b$ if and only if a is never followed by b.

Step 3. Replay the log in $N = (P, T, F, W, R)$ and record places with remaining tokens and calculate the fitness. If there are no remaining tokens or all alternatives below have been tried, return N.

Step 4. Pick a place p that has the most remaining tokens. Let T_p be the set of output transitions of p, i.e., $T_p = p\bullet$.

Step 5. $T' = \{t' \in T \setminus T_p \mid \forall_{t \in T_P}\ t' \not\gg_L t \ \wedge\ p \notin R(t')\}$, i.e., transitions that "seem" to disable T_p transitions but do not actually disable these transitions



Fitness: 0.8971429

Model ☑ Token Counter ☑ Failed Tasks ☑ Remaining Tasks ☑ Path Coverage □ Passed Edges

Selected Instances in % 100

22:28:57 [0] DOT finished on: D:\TEMP\prm3729.dot

Fig. 6. Using regions, a Petri net is discovered that captures the behavior but where tokens are left behind. The fitness is 0.897 as shown in top right corner.

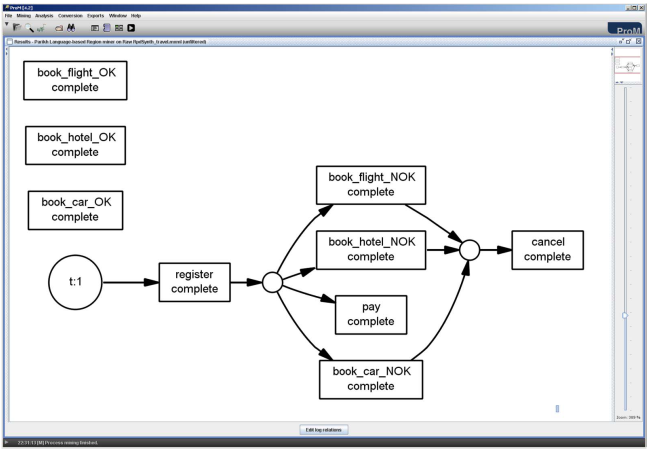
yet. If $T' = \emptyset$, then go to Step 3, otherwise pick a $t_r \in T'$. Take the "earliest" transition in T', e.g., using a relation similar to $\gg_L$.

Step 6. Add a reset arc to N connecting p and t_r, i.e., $N' = (P, T, F, W, R')$ where $R'(t_r) = R(t_r) \cup \{p\}$ and $R'(t) = R(t)$ for all other t.

Step 7. Return to Step 3 using $N = N'$.

Note that the above is *not* indented to be a concrete algorithm. It is merely a solution approach that needs to be made specific in the context of a concrete process mining algorithm. To illustrate this let us use a log L_2 that contains all possible behaviors of the RWF-net shown in Figure 2. In the log transition c is not visible as it is just there for routing purposes, i.e., an example trace in L_2 is $\langle$*register*, *book_flight_NOK*, *cancel*$\rangle$. Applying the α-algorithm to L_2 gives an incorrect and rather meaningless result because of the invisible routing activity c and the cancellation construct. If we apply the region-based approach presented in [9,44], then we obtain the Petri net shown in Figure 6. The region-based approach guarantees that it is possible to replay all traces in the log without missing tokens, but there may be remaining tokens. In terms of Definition 6, this means $m_i = 0$ and $r_i \geq 0$ for any trace i. This makes the technique of [9,44] suitable for the post-processing mentioned above. Note that there are six places where tokens may remain.

The Petri net in Figure 6 is able to execute the sequence $\langle$*register*, *book_flight_NOK*, *cancel*$\rangle$ but leaves two tokens behind. Note that the central place in-between *register* and *pay* acts as a mutex place blocking all activities after the first "NOK". Also note that there is a not a sink place in Figure 6, i.e., it is not a WF-net. This is due to the implementation of the plug-in in ProM and is merely a technicality that can be resolved easily (e.g., by adding a dummy end transition). Using the 7 steps described above reset arcs are added from the six places with remaining tokens to transition *cancel*. However, also superfluous reset arcs are added, e.g., from some of places with remaining tokens to *pay*. This can be optimized in various ways. First of all, additions that do not improve fitness can be discarded. Second, improving T' to filter out transitions that do not appear in traces that have problems (i.e., if there is no problem related to sequences where transition t' appears, then no reset of t' on p is needed). Both optimizations would get rid of the superfluous reset arcs. The resulting model nicely captures the behavior recorded in the log including the cancellation after the first "NOK" result.



Results - Parikh Language-based Region miner on Raw RpdSynth_travel.mxml (unfiltered)

22:31:13 [M] Process mining finished.

Fig. 7. A Petri net constructed using language-based regions theory [45]

There are alternatives to the post-processing approach described above. First of all, it would be relatively easy to extend the genetic miner in ProM [7,40] to deal with reset arcs. For genetic mining basically only a good representation (RWF-nets) and fitness function (Definition 6) are needed [7,40]. Second, it would be interesting to extend mining approaches based on language-based regions [45] to come up with special arcs. Figure 7 shows the application of the language-based region miner in ProM using a very conservative setting for log L_2, i.e., the same log as used to construct Figure 6. Because of the conservative setting just a few places were added, however, a correct characterization of the behavior is given. This shows the basic principle that a Petri net without any places can parse any log and that adding places corresponds to adding constraints. Since this approach uses integer linear programming as a basis, it is versatile and seems to be a good platform to add special types of arcs such as reset arcs, inhibitor arcs, etc.

7 Related Work

Since the mid nineties, many researchers have been working on workflow verification techniques. It is impossible to give a complete overview here. Moreover, most of the papers on workflow verification focus on rather simple languages, e.g., AND/XOR-graphs which are even less expressive than classical Petri nets. Therefore, we only mention the work directly relevant to this paper.

The use of Petri nets in workflow verification has been studied extensively. In [1,2] the foundational notions of WF-nets and soundness are introduced. In [32,33] two alterative notions of soundness are introduced: k-soundness and generalized soundness. These notions allow for dead parts in the workflow but address problems related to multiple instantiation. In [39] the notion of weak soundness is proposed. This notion allows for dead transitions. The notion of relaxed soundness is introduced in [21]. This notion allows for potential deadlocks and livelocks, however, for each transition there should be at least one proper execution.

Most soundness notions (except generalized soundness [32,33]) can be investigated using classical model checking techniques that explore the state space. However, such approaches can be intractable or even impossible because the state-space may be infinite. Therefore, alternative approaches that avoid constructing the (full) state space have been proposed. [3] describes how structural properties of a workflow net can be used to detect the soundness property. [46,47] presents an alternative approach for deciding relaxed soundness in the presence of OR-joins using invariants. The approach taken results in the approximation of OR-join semantics and transformation of YAWL nets into Petri nets with inhibitor arcs. In [51] it is shown that the backward reachability graph can be used to determine the enabling of OR-joins in the context of cancellation. In the general area of reset nets, Dufourd et al.'s work has provided valuable insights into the decidability status of various properties of reset nets including reachability, boundedness and coverability [24,25,29]. Moreover, in [48] it is shown that reduction rules can be applied to reset nets (and even to inhibitor nets) to speed-up analysis and improve diagnostics.

Since the mid-nineties several groups have been working on techniques for process mining [11,8,12,17,20,22,23,49], i.e., discovering process models based on observed events. In [10] an overview is given of the early work in this domain. The idea to apply process mining in the context of workflow management systems was introduced in [12]. In parallel, Datta [20] looked at the discovery of business process models. Cook et al. investigated similar issues in the context of software engineering processes [17]. Herbst [34] was one of the first to tackle more complicated processes, e.g., processes containing duplicate tasks.

Most of the classical approaches have problems dealing with concurrency. The α-algorithm [11] is an example of a simple technique that takes concurrency as a starting point. However, this simple algorithm has problems dealing with complicated routing constructs and noise (like most of the other approaches described in literature). In [22,23] a more robust but less precise approach is presented. The classical "theory of regions" [13,14,18,19,27] can also be used to discover Petri-net-based models as shown in [9,44]. Recently, some work on language-based regions theory appeared [16,45,37,38]. In [16,45] it is shown how this can be applied to process mining.

In this paper we do not consider issues such as noise. Heuristics [49] or genetic algorithms [7,40] have been proposed to deal with issues such as noise.

For an overview of related work with respect to conformance checking we refer to [4,42,43]. Note that so far no process mining techniques (discovery and/or conformance) have been proposed for models with cancellation such as RWF-nets.

To conclude this related work section, we provide some pointers to the relationships between Petri nets and graph grammars/transformations [28,31]. The relationships between Petri nets and graph grammars have been well known for quite some time [35]. In fact, graph grammars can be seen as a proper generalization of Petri nets. The firing of a transition corresponds to applying a "production" to a graph while firing sequences correspond to graph derivations. Reset arcs can easily be encoded in terms of graph grammars. In Section 3.2 of [31], extensions of graph rewriting using multi objects are discussed, i.e., universally quantified operations are used to remove all objects of a particular type in one go. Such ideas directly apply to reset nets. In [15] the relation between "extended Petri nets" and graph rewriting is investigated in detail. In this paper, Petri nets having read, inhibitor and reset arcs are mapped onto graph grammars. Thus far little work has been done on the relation between graph grammars/transformations on the one hand and workflow verification and process discovery on the other. It would be interesting to explore this further.

8 Conclusion

In this paper we explored various analysis questions related to workflows with cancellations. As a modeling language we used *reset workflow nets* (RWF-nets). Taking RWF-nets as a starting point we explored challenges related to *discovery* (process mining), *verification*, and *conformance*. For example, it was shown that soundness is undecidable for RWF-nets. However, despite this result, analysis is possible in most cases using e.g. reduction rules and structural techniques as shown in [48]. Conformance checking can be done in a straightforward manner by adapting the techniques described in [42,43] to RWF-nets. From a process mining viewpoint, no prior work has been done on the discovery of processes with cancellations. In this paper we made some initial suggestions to develop process discovery algorithms for RWF-nets. Given the importance of cancellation in workflows, it is interesting to develop techniques and tools to further address the challenges mentioned in this paper.

References

1. van der Aalst, W.M.P.: Verification of Workflow Nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
2. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)

3. van der Aalst, W.M.P.: Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 161–183. Springer, Heidelberg (2000)
4. van der Aalst, W.M.P.: Business Alignment: Using Process Mining as a Tool for Delta Analysis and Conformance Testing. Requirements Engineering Journal 10(3), 198–211 (2005)
5. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT press, Cambridge (2004)
6. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. BPM Center Report BPM-08-02, BPMcenter.org (2008)
7. van der Aalst, W.M.P., de Medeiros, A.K.A., Weijters, A.J.M.M.: Genetic Process Mining. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 48–69. Springer, Heidelberg (2005)
8. van der Aalst, W.M.P., Reijers, H.A., Weijters, A.J.M.M., van Dongen, B.F., de Medeiros, A.K.A., Song, M., Verbeek, H.M.W.: Business Process Mining: An Industrial Application. Information Systems 32(5), 713–732 (2007)
9. van der Aalst, W.M.P., Rubin, V., van Dongen, B.F., Kindler, E., Günther, C.W.: Process Mining: A Two-Step Approach using Transition Systems and Regions. BPM Center Report BPM-06-30, BPMcenter.org (2006)
10. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow Mining: A Survey of Issues and Approaches. Data and Knowledge Engineering 47(2), 237–267 (2003)
11. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
12. Agrawal, R., Gunopulos, D., Leymann, F.: Mining Process Models from Workflow Logs. In: Sixth International Conference on Extending Database Technology, pp. 469–483 (1998)
13. Badouel, E., Bernardinello, L., Darondeau, P.: The Synthesis Problem for Elementary Net Systems is NP-complete. Theoretical Computer Science 186(1-2), 107–134 (1997)
14. Badouel, E., Darondeau, P.: Theory of regions. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)
15. Baldan, P., Corradini, A., Montanari, U.: Relating SPO and DPO Graph Rewriting with Petri nets having Read, Inhibitor and Reset Arcs. Electronic Notes in Theoretical Computer Science 127(2), 5–28 (2005)
16. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007)
17. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology 7(3), 215–249 (1998)
18. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Synthesizing Petri Nets from State-Based Models. In: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1995), pp. 164–171. IEEE Computer Society Press, Los Alamitos (1995)
19. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri Nets from Finite Transition Systems. IEEE Transactions on Computers 47(8), 859–882 (1998)

20. Datta, A.: Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. Information Systems Research 9(3), 275–301 (1998)
21. Dehnert, J., van der Aalst, W.M.P.: Bridging the Gap Between Business Models and Workflow Specifications. International Journal of Cooperative Information Systems 13(3), 289–332 (2004)
22. van Dongen, B.F., van der Aalst, W.M.P.: Multi-Phase Process Mining: Building Instance Graphs. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 362–376. Springer, Heidelberg (2004)
23. van Dongen, B.F., van der Aalst, W.M.P.: Multi-Phase Mining: Aggregating Instances Graphs into EPCs and Petri Nets. In: Marinescu, D. (ed.) Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, pp. 35–58. Florida International University, Miami, Florida, USA (2005)
24. Dufourd, C., Finkel, A., Schnoebelen, Ph.: Reset Nets Between Decidability and Undecidability. In: Larsen, K., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)
25. Dufourd, C., Jančar, P., Schnoebelen, Ph.: Boundedness of Reset P/T Nets. In: Wiedermann, J., van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 301–310. Springer, Heidelberg (1999)
26. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software through Process Technology. Wiley & Sons, Chichester (2005)
27. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures - Part 1 and Part 2. Acta Informatica 27(4), 315–368 (1989)
28. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Concurrency, Parallelism, and Distribution, vol. 3. World Scientific, Singapore (1999)
29. Finkel, A., Schnoebelen, P.: Well-structured Transition Systems everywhere! Theoretical Computer Science 256(1–2), 63–92 (2001)
30. Georgakopoulos, D., Hornick, M., Sheth, A.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases 3, 119–153 (1995)
31. Heckel, R.: Graph Transformation in a Nutshell. Electronic Notes in Theoretical Computer Science 148(1), 187–198 (2006)
32. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 335–354. Springer, Heidelberg (2003)
33. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Generalised Soundness of Workflow Nets Is Decidable. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 197–215. Springer, Heidelberg (2004)
34. Herbst, J.: A Machine Learning Approach to Workflow Management. In: López de Mántaras, R., Plaza, E. (eds.) ECML 2000. LNCS (LNAI), vol. 1810, pp. 183–194. Springer, Heidelberg (2000)
35. 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)
36. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice-Hall PTR, Upper Saddle River (1999)

37. Lorenz, R., Bergenthum, R., Desel, J., Mauser, S.: Synthesis of Petri Nets from Finite Partial Languages. In: Basten, T., Juhás, G., Shukla, S.K. (eds.) International Conference on Application of Concurrency to System Design (ACSD 2007), pp. 157–166. IEEE Computer Society Press, Los Alamitos (2007)
38. Lorenz, R., Juhás, G.: How to Synthesize Nets from Languages: A Survey. In: Henderson, S.G., Biller, B., Hsieh, M., Shortle, J., Tew, J.D., Barton, R.R. (eds.) Proceedings of the Wintersimulation Conference (WSC 2007), pp. 637–647. IEEE Computer Society Press, Los Alamitos (2007)
39. Martens, A.: Analyzing Web Service Based Business Processes. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 19–33. Springer, Heidelberg (2005)
40. de Medeiros, A.K.A.: Genetic Process Mining. PhD thesis, Eindhoven University of Technology, Eindhoven (2006)
41. Mendling, J., Neumann, G., van der Aalst, W.M.P.: Understanding the Occurrence of Errors in Process Models Based on Metrics. In: Curbera, F., Leymann, F., Weske, M. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 113–130. Springer, Heidelberg (2007)
42. Rozinat, A., van der Aalst, W.M.P.: Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In: Bussler, C., et al. (eds.) BPM 2005. LNCS, vol. 3812, pp. 163–176. Springer, Heidelberg (2006)
43. Rozinat, A., van der Aalst, W.M.P.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems 33(1), 64–95 (2008)
44. Rubin, V., Günther, C.W., van der Aalst, W.M.P., Kindler, E., van Dongen, B.F., Schäfer, W.: Process Mining Framework for Software Processes. In: Wang, Q., Pfahl, D., Raffo, D.M. (eds.) ICSP 2007. LNCS, vol. 4470, pp. 169–181. Springer, Heidelberg (2007)
45. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Computer Science Report (08-04), Eindhoven University of Technology, Eindhoven, The Netherlands (2008)
46. Verbeek, H.M.W., van der Aalst, W.M.P.: Analyzing BPEL Processes using Petri Nets. In: Marinescu, D. (ed.) Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management. Florida International University, Miami, Florida, USA, pp. 59–78 (2005)
47. Verbeek, H.M.W., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. The Computer Journal 50(3), 294–314 (2007)
48. Verbeek, H.M.W., Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Reduction Rules for Reset/Inhibitor Nets. BPM Center Report BPM-07-13, BPMcenter.org (2007)
49. Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. Integrated Computer-Aided Engineering 10(2), 151–162 (2003)
50. Weske, M.: Business Process Management: Concepts, Languages, Architectures . Springer, Berlin (2007)
51. Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Reset Nets and Reachability Analysis. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 389–394. Springer, Heidelberg (2006)

The AUTOSAR Way of Model-Based Engineering of Automotive Systems

Heiko Dörr

Carmeq, Carnotstr. 4, 10587 Berlin, Germany
Heiko.Doerr@carmeq.com

Abstract. The international development partnership AUTOSAR strives for reuse and exchange of SW components along various dimensions: between OEM and supplier, between vehicle platforms, between electronic networks, and between individual electronic control units (ECUs). To enable the exchange an abstraction layer – the AUTOSAR runtime environment (RTE) – has been defined as a set of services provided to applications. Typical services are communication via buses, memory access, or diagnostic support. The services are implemented by several stacks of basic SW-modules being highly configurable to support the large variety of ECUs. By configuration, the same SW module can be used for those ECUs with a given micro-controller abstraction.

To exploit the variability in a structured approach, AUTOSAR defined a methodology which heavily makes use of so-called templates capturing all information of an electronic vehicle network. The data-structures are defined by a meta-model allowing for a consistent definition of the templates.

The development approach adopted by AUTOSAR is related to model-driven development. At the first step in the methodology, an overall system description is defined showing the logical architecture of the SW system. Each SW component is described by an instance of the SW-template. When the SW system shall be applied to an electronic network, the dedicated HW resources must be described as well. So, the properties and capabilities of ECUs and buses are captured by an ECU resource template. During the next step, a mapping of the logical to the physical architecture has to be found. This task is of high importance, since it determines the overall system performance. Once the mapping is established, configuration files for individual ECUs of the network are generated. Based on the configuration the appropriate SW-components can be loaded onto the ECU and the required communication channels are established such that the SW application can be successfully executed.

Currently, there is no strong tool-support for mapping besides plain editors. Rule-based transformations may be applied to define and realize (semi-) automatic mapping strategies for the different areas in the system and ECU resource template.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, p. 38, 2008.

An Algorithm for Hypergraph Completion According to Hyperedge Replacement Grammars

Steffen Mazanek, Sonja Maier, and Mark Minas

Universität der Bundeswehr München, Germany
{steffen.mazanek,sonja.maier,mark.minas}@unibw.de

Abstract. The algorithm of Cocke, Younger, and Kasami is a dynamic programming technique well-known from string parsing. It has been adopted to hypergraphs successfully by Lautemann. Therewith, many practically relevant hypergraph languages generated by hyperedge replacement can be parsed in an acceptable time. In this paper we extend this algorithm by hypergraph completion: If necessary, appropriate fresh hyperedges are inserted in order to construct a derivation. The resulting algorithm is reasonably efficient and can be directly used, among other things, for auto-completion in the context of diagram editors.

Keywords: hypergraph completion, hyperedge replacement, parsing.

1 Introduction

Hypergraphs are an extension of graphs where edges are allowed to visit an arbitrary number of nodes. A well-known way of describing hypergraph languages are hyperedge replacement grammars HRG [1]. Although restricted in power, this formalism comprises several beneficial properties: It is context-free and still quite powerful. Grammars are comprehensible, and reasonably efficient parsers can be defined for practical languages. In general, parsing is NP-complete though.

Lautemann has provided a detailed discussion of the complexity of hyperedge replacement [2]. He also has suggested a hypergraph parsing algorithm straightforwardly adopting the dynamic programming approach proposed by Cocke, Younger, and Kasami CYK [3] for string parsing. Given a string $s = a_1...a_n$, the CYK algorithm computes a table where the cell in row i and column j contains derivation trees that derive the substring $a_i...a_{i+j-1}$. This table can be computed bottom up by joining two appropriate entries at a time – provided that the grammar is in Chomsky normalform CNF, which is no restriction.

An extended version of Lautemann's CYK-style algorithm for hypergraphs has been proposed by the third author and incorporated into the diagram editor generator DiaGen [4]. In analogy to the string setting, HRGs are transformed to a kind of CNF first. However, the parsing algorithm does not need to compute a table but rather n layers where n is the number of hyperedges in the hypergraph. Thereby, layer k is computed by combining two "compatible" derivation trees from layers i and j at a time where $i + j = k$.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 39–53, 2008.

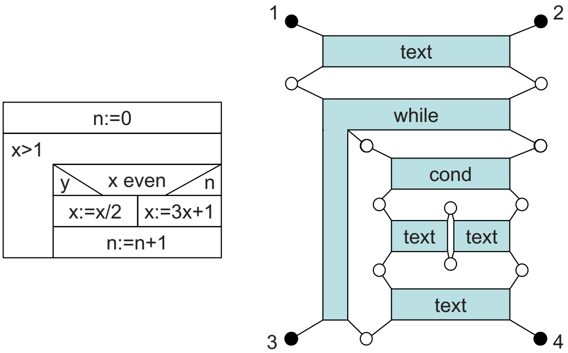


Fig. 1. Hypergraphs as a model for diagrams

Diagram editors are an important area of application for hypergraph parsing since hypergraphs have appeared to be well-suited as a model for diagrams. For instance, in DiaGen the abstract syntax of a diagram language is defined using HRGs, and the parser checks which parts of a freely drawn diagram are correct. In Fig. 1 an example Nassi-Shneiderman-Diagram NSD and a corresponding hypergraph model are shown. Hyperedges are represented by boxes with the particular label inside. Nodes are represented as circles; the so-called external ones as black dots. Lines (tentacles) indicate that a hyperedge visits a node. The hypergraph language of NSDs can be defined using an HRG as we see later.

In this paper we propose an algorithm for hypergraph completion with respect to HRGs. Hypergraph completion can be used, among other things, as a powerful and flexible base for diagram completion. Indeed, *content assist* in diagram editors is just as valuable as conventional content assist as known from modern text editors and integrated development environments. The editor user normally does not only want to be notified when his diagram is incorrect, but is also interested in the particular problem and possible solutions. For instance, the insertion of a simple statement at the right place is already sufficient in order to repair the diagram shown in Fig. 2. Such suggestions are particularly important for free-hand editors like the ones generated with DiaGen. By providing assistance we can effectively combine the advantages of structured and free-hand editing: The user is allowed to draw his diagram with maximal freedom, but guidance is provided if needed.

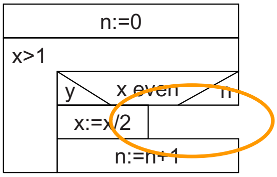


Fig. 2. Incomplete NSD

The information required for some assistance can be gathered by the parser. It is possible to infer places where new hyperedges may be added to complete a given hypergraph, although those might not be uniquely determined. We have discussed a first, logic-based approach in [5]. The proposed framework of *graph parser combinators* follows a top-down approach with backtracking, thus partial results might be computed several times.

In this paper we substantially improve efficiency by using dynamic programming techniques. We basically extend Lautemann's CYK-style algorithm to support the computation of hypergraph completions. Our key idea is to pretend a (limited) number of hyperedges while parsing. Therefore, several fresh hyperedges are introduced initially, which visit fresh, special nodes. Later, these nodes can be glued with nodes already occurring in the input hypergraph.

We proceed as follows: First, we introduce hypergraphs and HRGs in Sect. 2 using NSDs as a running example. Thereafter, we describe our parser and how it computes so-called complement hypergraphs (Sect. 3). This is the main contribution of this paper. In Sect. 4 we discuss related work. Finally, we sketch future prospects and conclude the paper (Sect. 5).

2 Hypergraphs and Hyperedge Replacement Grammars

In this section the formal basics are introduced. Most definitions are close to [1] and [2]. We just recapitulate them to make this paper self-contained.

Let C be an arbitrary, but fixed set of *labels* and let *type*: $C \to \mathbb{N}$ be a *typing* function for C. Let $\mathcal{V}$ denote a universe of nodes. A *hypergraph* H over C is a tuple $(V_H, E_H, att_H, lab_H, ext_H)$ where $V_H \subset \mathcal{V}$ is a finite set of *nodes*, E_H is a finite set of *hyperedges*, $att_H : E_H \to V_H^*$ is a mapping assigning a sequence of pairwise distinct *attachment nodes* $att_H(e)$ to each $e \in E_H$, $lab_H : E_H \to C$ is a mapping that *labels* each hyperedge such that $type(lab_H(e)) = |att_H(e)|$ (length of sequence), and $ext_H \in V_H^*$ is a sequence of pairwise distinct *external nodes* (in pictures numbers represent a node's position in *ext*). We further define $type(H) := |ext_H|$, $|H| := |E_H|$, and $H_0 := (V_H, E_H, att_H, lab_H, \epsilon)$ (the underlying *type*-0 hypergraph of a hypergraph H). We denote the empty hypergraph as $\emptyset_G$. The set of all hypergraphs over $\Sigma \subseteq C$ is denoted by $\mathcal{H}_\Sigma$. We occasionally call hypergraphs just graphs and hyperedges edges.

A hypergraph H is called *elementary* if it is induced by a single edge e, i.e., $E_H = \{e\}$, $V_H = [att_H(e)]$[1], and $ext_H = att_H(e)$. In this case, we define $lab(H) := lab_H(e)$ and $edge(H) := e$. Given a set of hypergraphs $\mathcal{H}$, the set of elementary hypergraphs in $\mathcal{H}$ is denoted by $elem(\mathcal{H})$.

Given hypergraphs $H, H' \in \mathcal{H}_C$ with $E_H \cap E_{H'} = \emptyset$, $e \in E_H$ such that $V_H \cap V_{H'} = [att_H(e)]$ and $ext_{H'} = att_H(e)$, the hypergraph $H[e/H']$ resulting from the replacement of e by H' is defined as $H[e/H'] := (V_H \cup V_{H'}, (E_H \backslash \{e\}) \cup E_{H'}, att_H \cup att_{H'}, lab_H \cup lab_{H'}, ext_H)$. Let $B \subseteq E_H$ be a set of hyperedges to be replaced and let $repl : B \to \mathcal{H}_C$ be a mapping such that $H[e/repl(e)]$ is defined for all $e \in B$. We denote the replacement of all edges contained in B by $H[repl]$ (the order of their replacement does not matter [1]).

Let $H, H' \in \mathcal{H}_C$. Then H is a *sub-hypergraph* of H', denoted $H \subseteq H'$, if $V_H \subseteq V_{H'}$, $E_H \subseteq E_{H'}$, $att_H(e) = att_{H'}(e)$, and $lab_H(e) = lab_{H'}(e)$ for all $e \in E_H$. The *ext-union* of H and H', denoted $H \cup_{ext} H'$ where $ext \in (V_H \cup V_{H'})^*$, is defined as $(V_H \cup V_{H'}, E_H \cup E_{H'}, att_H \cup att_{H'}, lab_H \cup lab_{H'}, ext)$ provided $E_H \cap E_{H'} = \emptyset$. Two hypergraphs H and H' are isomorphic, denoted $H \cong H'$, if

[1] $[a_1...a_n] := \{a_1, ..., a_n\}$.

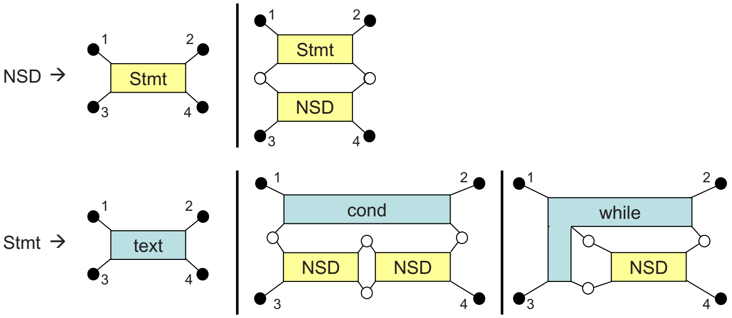


Fig. 3. Productions P_{NSD} of HRG G_{NSD}

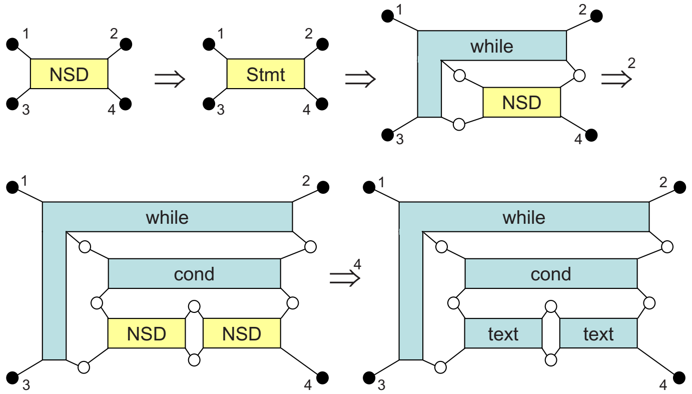


Fig. 4. A possible derivation wrt G_{NSD}

there are bijections $\phi_V : V_H \to V_{H'}$ and $\phi_E : E_H \to E_{H'}$ such that for all $e \in E_H$ $lab_{H'}(\phi_E(e)) = lab_H(e)$, $att_{H'}(\phi_E(e)) = \phi_V^*(att_H(e))$ and $ext_{H'} = \phi_V^*(ext_H)$.[2]

A *production* $A \to R$ over $N \subseteq C$ consists of a label $A \in N$ and a hypergraph $R \in \mathcal{H}_C$ such that $type(A) = type(R)$. Let P be a set of productions, $H \in \mathcal{H}_C$, $e \in E_H$, $(lab_H(e) \to R) \in P$, $R' \in \mathcal{H}_C$ such that $R' \cong R$ and $H[e/R']$ is defined. Then H *directly derives* $H' = H[e/R']$, denoted $H \Longrightarrow_P H'$.

A *hyperedge replacement grammar* is a system $G = (N, T, P, S)$ where $N \subset C$ is a set of *nonterminals*, $T \subset C$ with $T \cap N = \emptyset$ is a set of *terminals*, P is a finite set of *productions* over N, and $S \in N$ is the *start symbol*. Let $\mathcal{L}_A(G) := \{H \in \mathcal{H}_T \mid \exists L \in elem(\mathcal{H}_{\{A\}}) : L \Longrightarrow_P^* H\}$. The *hypergraph language* $\mathcal{L}(G)$ *generated by* G is defined as $\mathcal{L}(G) := \mathcal{L}_S(G)$.

As an example consider the grammar of NSD graphs $G_{\text{NSD}} = (\{\text{NSD, Stmt}\}, \{\text{text, cond, while}\}, P_{\text{NSD}}, \text{NSD})$ where the set of productions P_{NSD} is shown in Fig. 3. A possible derivation is given in Fig. 4. Note, that in the following we omit the "while"-production for the sake of brevity.

[2] $f^*(a_1 ... a_n) := f(a_1) ... f(a_n)$.

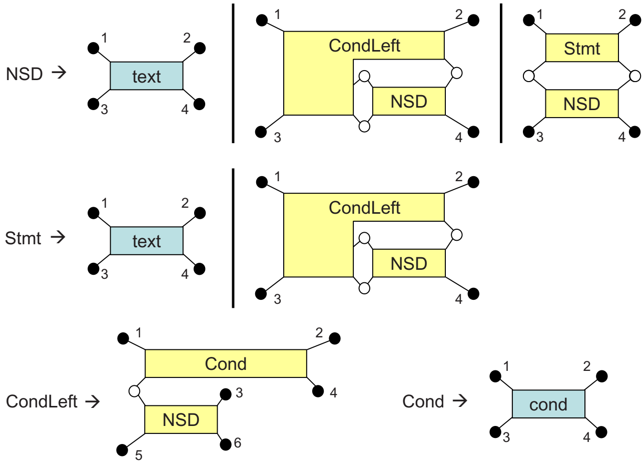


Fig. 5. Chomsky normalform of Fig. 3 (without "while")

In the string setting the CYK-algorithm requires grammars to be in so-called Chomsky normalform CNF. This is no restriction, since every context-free grammar (that does not generate the empty string) can be transformed to a grammar that defines the very same language and is in CNF. A similar notion can be defined for HRGs:

Definition 1 (Chomsky normalform). *An HRG is in Chomsky normalform CNF, iff for every production $A \to R$ holds: R does not contain isolated nodes, and either $R \in \mathcal{H}_T \wedge |R| = 1$ (terminal production) or $R \in \mathcal{H}_N \wedge |R| = 2$ (expansion production).*

Every HRG whose language does not contain hypergraphs with isolated nodes or the empty graph $\emptyset_G$ can be transformed to an equivalent grammar in CNF. A constructive proof of this proposition is given in [6]. The productions of an HRG in CNF equivalent to G_{NSD} (without "while") are shown in Fig. 5.[3] Note, that isolated nodes do not occur in the context of visual language specifications. However, if they cannot be avoided in a particular situation, it might be sufficient to add unary dummy edges.

Since HRGs are context-free (cf. *context-freeness lemma* in [1]) the notion of a *derivation tree* is meaningful. Let $G = (N, T, P, S)$ be an HRG. The set of derivation

[3] Basically, productions with more than two hyperedges at their right-hand side are split successively (therefore, in our example the new nonterminal "CondLeft" and the corresponding production are added). Where necessary, special nonterminal labels $\hat{l}$ are introduced that only derive an elementary graph labeled l, see, e.g., "Cond". And finally, so-called chain productions like the derivation of a single "Stmt" nonterminal from "NSD" are eliminated by adding the productions over "Stmt" also to "NSD". This transformation is very similar to the corresponding transformation in the string setting and can be performed automatically (as realized in DiaGen).

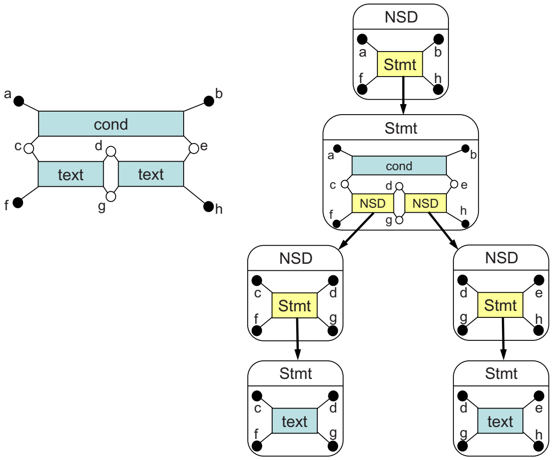


Fig. 6. A hypergraph and its derivation tree wrt G_{NSD}

trees $\mathrm{TREE}(G)$ is recursively defined: Consider a triple $t = (L, R, \mathit{branch})$ where $L \in \mathit{elem}(\mathcal{H}_N), R \in \mathcal{H}_C, L \Longrightarrow_P R, \mathit{branch} : E_R^N \to \mathrm{TREE}(G)$[4]. Let $\mathit{root}(t) := L$. The triple t is in $\mathrm{TREE}(G)$ iff $\mathit{lab}_R(e) = \mathit{lab}(\mathit{root}(\mathit{branch}(e)))$ for all $e \in E_R^N$ and $\mathit{result}(t) := R[\{e \mapsto \mathit{result}(\mathit{branch}(e)) \mid e \in E_R^N\}]$ (the graph spanned by this tree) is defined. Note, that according to this definition the leaves of a derivation tree are triples where $R \in \mathcal{H}_T$.

An example graph and its derivation tree are shown in Fig. 6. The arrows represent the mapping determined by *branch*. The elementary *root* graphs L can be represented by their labels $\mathit{lab}(L)$, since the nodes visited by $\mathit{edge}(L)$ are just the external nodes of R. Numbers of external nodes are omitted. All nodes are marked with letters to make them distinguishable. The notions derivation and derivation tree are indeed equivalent as, e.g., proven in [1].

Next, we introduce the concept of a *complement hypergraph*. Informally, this is a graph such that its union with the given input graph can be derived from a particular start graph.

Definition 2 (complement hypergraph). *Given an HRG $G = (N, T, P, S)$, type-0 hypergraphs $H, H_c \in \mathcal{H}_T$, and $L \in \mathit{elem}(\mathcal{H}_N)$. H_c is a complement hypergraph of H with respect to G and L iff $E_{H_c} \cap E_H = \emptyset$ and $L \Longrightarrow_P^* H \cup_{\mathit{ext}_L} H_c$.*

Note, that we do not assume anything about the correctness of the given graph H wrt G. If H is incorrect, H_c is a completion. Otherwise, it is a correctness-preserving extension of H, just as needed, among other things, for the realization of situation-dependent structured editing operations in diagram editors.

This notion is illustrated by example in Fig. 7. Given the *type*-0 hypergraph H surrounded by the box. In the figure several complement hypergraphs of H are shown (in union with H) – each with respect to the elementary graph labeled

[4] $E_R^N := \{e \in E_R \mid \mathit{lab}_R(e) \in N\}$.

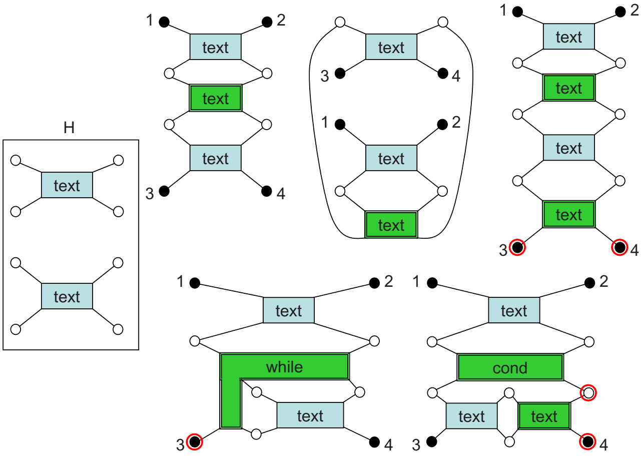


Fig. 7. Some complement graphs of the given graph H

NSD whose external nodes are just the ones marked in the particular image. Note, that our definition does not relate V_H and V_{H_c}. Indeed those sets may either overlap or be disjoint. Nodes of the complement graphs that do not already belong to the original hypergraph H are surrounded by an extra circle in the figure. In general, the number and size of complement graphs is not restricted, although a practical implementation surely has to impose meaningful bounds.

3 A CYK-Style Complementing Parser

Given a *type*-0 hypergraph H, a conventional parser analyzes H with respect to an HRG G. If possible, a sequence of external nodes is established such that the resulting graph is in $\mathcal{L}(G)$. The corresponding derivation tree can be constructed, e.g., by using dynamic programming similar to the CYK algorithm. So the hypergraph parser of the DiaGen system computes $n = |H|$ layers. In layer k, $1 \leq k \leq n$, all derivation trees are contained whose *result* is a graph $H' \subseteq H$ such that $|H'| = k$. Since HRGs are transformed to CNF in advance, for $k > 1$ layer k can be computed by combining two derivation trees from layers i and j at a time where $i + j = k$. Thereby, expansion productions are applied reversely.

The complementing parser proposed in this section does not only check if H is a member of the language. It also computes complement hypergraphs up to a particular size *max*. If the parameter *max* is set to zero it simply is a conventional parser. Our key idea is to introduce fresh edges that can be embedded into the input graph in a flexible manner. If these edges are actually used in a particular derivation tree they constitute the complement graph. For every terminal symbol t of the language, *max* fresh edges with label t are introduced. Each of these edges visits *type*(t) nodes, which are also fresh and special in the sense, that – in the process of parsing – they might be identified with nodes from H or even with other fresh nodes (similar to logic variables).

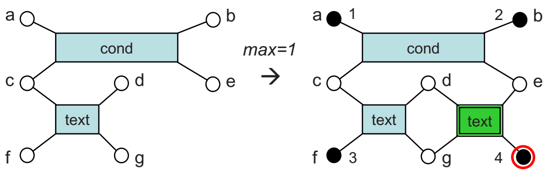


Fig. 8. Example input/output of the algorithm

Before we define the algorithm more formally we provide an example run with $max = 1$ to clarify its basic principle. Fig. 8 shows a possible input hypergraph and, resulting from the algorithm, its union with the only complement graph wrt NSD of size up to 1. The corresponding external nodes are also marked.

Fig. 9 illustrates how the layers are filled according to this example. The number enclosed by a circle in the upper right of a derivation tree indicates, how many of the fresh edges have already been used in this derivation. The capital letters A, B, C and D are just shorthands for the particular trees to avoid cluttering the figure by too many arrows. We further simplify the figure by joining derivation trees that only differ in the labels of their *root* graphs. Those mainly appear due to the elimination of chain productions. In this case, trees are marked with all possible *root* labels, here NSD and Stmt.

Furthermore, all "imaginable" derivation trees in layer 2 are shown, although most of them are invalid for some reason and, thus, disregarded by the parser. Such invalid trees are shaded in the figure and the particular problem is marked with a lightning symbol. When constructing derivation trees, we have to ensure that at most *max* of the fresh edges are used at a time. The derivation tree in the lower right of layer two, for instance, consumes two fresh edges, which is prevented by the restriction *max=1*. The others violate the so-called gluing condition. Here, a node is reduced, i.e., it is non-external within the right-hand side of an "instantiated" production, although there still is an edge in the remaining graph visiting this node. It is important to filter invalid derivation trees as soon as possible to reduce the number of combinations in the layers above.

We compute an equivalence relation $\sim$ between nodes to realize the gluing of fresh nodes with nodes actually occurring in the input graph. The "significant" subset of this relation is shown in the figure. A simple mapping indeed is not sufficient, since an arbitrary number of fresh nodes may coincide.

Complete derivation trees are those consuming the whole input graph. They are surrounded by thicker lines in the figure and can, of course, only occur in the layers between $|H|$ and $|H| + max$. In the example, there is only one tree with complete coverage of the input graph whose *root* is labeled with the start symbol NSD at the same time: the one whose *result* is shown in Fig. 8.

Next, we define the parsing algorithm more formally. Let $G = (N, T, P, S)$ be a HRG in CNF, $H \in \mathcal{H}_T$ a *type*-0 hypergraph, and $max \in I\!N$. The parser successively constructs layers $L_i \subseteq \mathcal{H}_T \times \mathcal{H}_T \times 2^{\mathcal{V} \times \mathcal{V}} \times \mathrm{TREE}(G)$ for $1 \leq i \leq |H| + max$. We first provide a lemma which states the properties holding for the elements of the computed layers. This simplifies the understanding of the algorithm defined afterwards.

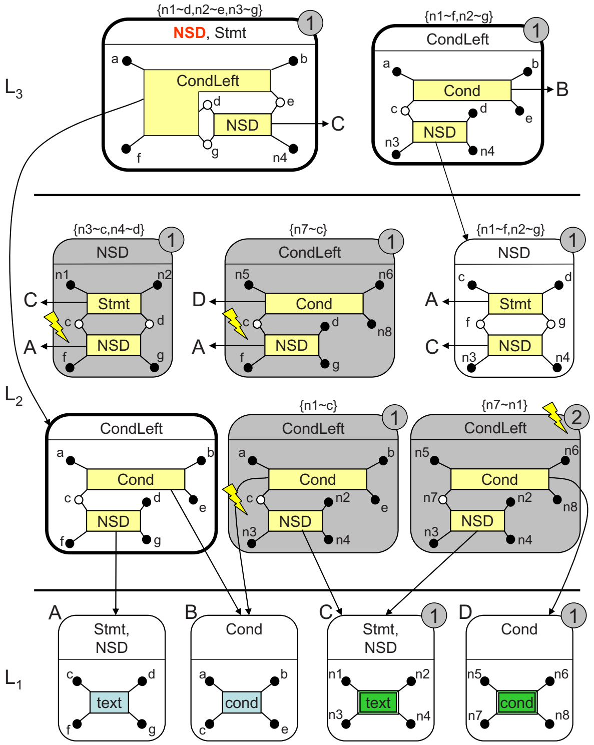


Fig. 9. Illustration of the CYK-style graph parser with completion

Lemma 1. *The following properties hold for the elements* $(H', H_c, \sim, t) \in L_i$ *if we identify nodes equivalent wrt* $equi(\sim)$[5]*:*

1. $|H'| + |H_c| = i$,
2. $H' \subseteq H$,
3. $result(t) = H' \cup_{ext_{result(t)}} H_c$,
4. $|H_c| \leq max$,
5. H_c *is a complement hypergraph of* H' *with respect to* G *and* $root(t)$.

A proof sketch of this lemma is given in Appendix A. After processing the layers we are mainly interested in entries $(H', H_c, \sim, t)$ where $lab(root(t)) = S$ and $|H'| = |H|$, i.e., the whole input graph is covered. To simplify the definitions of the layers let us define an auxiliary predicate for ensuring the gluing condition:

$$\text{For a graph } R,\ \ gc_H(R) :\Leftrightarrow \forall e \in E_H \backslash E_R : [att_H(e)] \cap V_R \subseteq [ext_R]$$

[5] $equi(\sim) := (\sim \cup \sim^T)^*$ denotes the smallest equivalence relation containing $\sim$. Formally, the identification of equivalent nodes means to deal with the corresponding quotient graph (whose nodes actually are equivalence classes of nodes), but to avoid cluttering we just assume the identification of equivalent nodes implicitly.

Thus, $gc_H(R)$ holds if and only if no edge of H that is not already in R visits a non-external node of R. Next, we define the layers recursively. Layer 1 only contains derivation trees resulting from the reverse-application of terminal productions. Thereby, either a single, terminal edge of H is derived, or one of $max \cdot |T|$ fresh edges. This possibly large number of fresh edges is necessary, since edges should keep their label and at this early stage we cannot know how many edges with a particular label we will eventually need:

$$\begin{aligned}
L_1 := \big\{ & (R_0, \emptyset_G, \emptyset, (L, R, \emptyset)) \mid \\
& R \subseteq H, |R| = 1, L \in elem(\mathcal{H}_N), L \Longrightarrow_P R, gc_H(R) \big\} \\
\bigcup \; & \{(\emptyset_G, R_0, \emptyset, (L, R, \emptyset)) \mid t \in T, k \in \{1, ..., max\}, \\
& e_k \text{ fresh edge}, ns_k \text{ sequence of } type(t) \text{ fresh nodes}, \\
& L \in elem(\mathcal{H}_N), R = ([ns_k], \{e_k\}, \{e_k \mapsto ns_k\}, \{e_k \mapsto t\}, ext_L), L \Longrightarrow_P R \big\}
\end{aligned}$$

As illustrated in Fig. 9, from layer 2 on the derivation trees are composed by combining two compatible derivation trees t_1 and t_2 of lower layers at a time by reverse-applying expansion productions. Thus, layer $L_i, i > 1$ is constructed from already computed layers L_j and L_{i-j} where $j < i$. For $1 < i \leq |H| + max$ we define:

$$L_i := \bigcup_{j=1}^{\lfloor i/2 \rfloor} (L_j \oplus L_{i-j})$$

Thereby, the combination of two sets $M, N \subseteq \mathcal{H}_T \times \mathcal{H}_T \times 2^{\mathcal{V} \times \mathcal{V}} \times \text{TREE}(G)$ is defined as follows:

$$\begin{aligned}
M \oplus N := \big\{ & (H_1' \cup_\epsilon H_2', H_{1_c} \cup_\epsilon H_{2_c}, \sim_n, (L, R, \{e_1 \mapsto t_1, e_2 \mapsto t_2\})) \mid \\
& (H_1', H_{1_c}, \sim_1, t_1) \in M, (H_2', H_{2_c}, \sim_2, t_2) \in N, \\
& E_{H_1'} \cap E_{H_2'} = \emptyset, E_{H_{1_c}} \cap E_{H_{2_c}} = \emptyset, |H_{1_c}| + |H_{2_c}| \leq max, \\
& \text{let } L_k := root(t_k), e_k := edge(L_k) \text{ for } k \in \{1, 2\}, \\
& \exists \sim_n \text{ minimal relation in } 2^{\mathcal{V} \times \mathcal{V}} \text{ such that} \\
& \sim_1 \subseteq \sim_n, \; \sim_2 \subseteq \sim_n, \; \sim := equi(\sim_n), preserves_{V_H}(\sim), \\
& \text{when identifying nodes equivalent wrt to } \sim \\
& \exists L \in elem(\mathcal{H}_N), R := L_1 \cup_{ext_L} L_2, L \Longrightarrow_P R, gc_H(R) \big\}
\end{aligned}$$

When combining derivation trees the fresh nodes can be glued to other nodes. For this purpose an equivalence relation between nodes is established such that the union of the roots of t_1 and t_2 is isomorphic to the right-hand side of the particular production. It must not happen, however, that nodes of the input graph V_H are identified among each other. Rather their identities have to be preserved. This restriction is ensured by using the predicate $preserves_V(\sim) :\Leftrightarrow \forall n_1, n_2 \in V : n_1 \sim n_2 \Rightarrow n_1 = n_2$. Thus, the relation $\{(n_1, n_2) \in \sim \mid n_1, n_2 \in V_H\}$ has to be the identity. Since the layers are recursively defined Lemma 1 now can be proven by induction as sketched in Appendix A.

Discussion

This algorithm also ensures that indeed all *structurally different* complement graphs up to size *max* are computed, because all possible embeddings of up to *max* arbitrarily labeled, fresh edges into H are constructed. At the end, those equivalence classes of nodes not containing a node of the input graph can be considered as new nodes contributed by a complement graph.

Performance. Although the algorithm is correct and complete, it suffers from an inherent problem. If the bound *max* is increased, we get a lot of redundant derivation trees, since new derivation trees in layer 1 (where fresh edges have the same label) can be embedded at different places *interchangeably*. Unsurprisingly, this effect has a negative impact on the performance of the algorithm.

It is possible to avoid this problem though. In our current implementation, layer 1 only contains derivation trees for edges originally occurring in the input graph. In addition, for each terminal production a special leaf derivation tree is constructed that can be *cloned* if required. This approach is more syntax-driven and, thus, yields only structurally different solutions. Therefore, it can even be used as a reasonably efficient language generator. However, the formulation given in this paper is less technical. It has been preferred for the sake of presentation.

Indeed our current implementation has turned out to be sufficiently efficient even for interactive applications. In Fig. 10 we provide performance data for different values of *max*. The input graphs have been simple sequences of statements, i.e., $|H|$ is the length of the sequence. Thus, the corresponding completions are extensions of correct graphs (incorrect graphs show a similar behavior, but are more difficult to construct in a homogeneous way). The algorithm shows a polynomial runtime behavior. Note, that our implementation has not been optimized with respect to performance yet. Further enhancements can be achieved by a prior analysis of the grammar such that compatible derivation trees can be found more efficiently. We also work on an incremental version of the algorithm where additional fresh edges can be efficiently incorporated on demand.

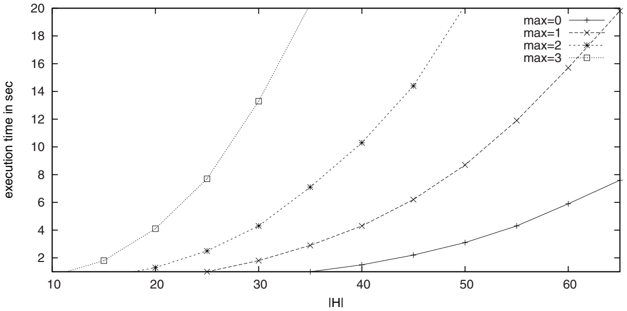


Fig. 10. Execution time of our implementation applied to NSD graphs with different values of *max* on standard hardware

There is one factor particularly known for its strong impact on the performance: the degree of connectedness. We already know that parsing performance suffers if graphs of a language are highly disconnected, cf. [1,2]. This effect unfortunately becomes worse with our approach, since the gluing condition cannot be used as effectively anymore to exclude derivation trees at an early stage.

Gluing Nodes. Per definition the algorithm presented in this section preserves the nodes of the input graph. However, we have noted that it is sometimes convenient to relax this condition. Incorrect input graphs often can already be corrected by gluing some nodes appropriately.

For instance, reconsider the example graph given in Fig. 7. It can be corrected by inserting an artificial extra edge. However, there also is a more lightweight way: The two isolated edges can be joined together by gluing some nodes. There are two ways to do this both shown in Fig. 11. Support for this kind of repair action can be achieved with a very little adaptation of the given algorithm. However, formally this approach would have to rely on a broader notion of completion.

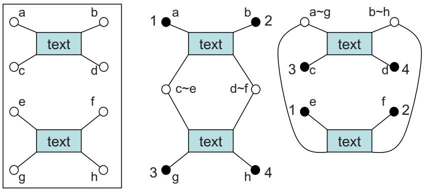


Fig. 11. Gluing original nodes

4 Related Work

To the best of our knowledge graph completion wrt hyperedge replacement grammars has not been considered yet. Due to their logic-based approach *graph parser combinators* [5] provide some support for completion as a nice side effect. However, from a performance point of view this framework cannot be used for practical, interactive applications. In contrast, the algorithm presented in this paper does not suffer from this problem.

In the more general context of graphs, several approaches exist to error correction and inexact matching, respectively. Most practical approaches propose either particular restrictions on the graph grammar formalism, incorporate application-specific knowledge or even make use of heuristics [7] in order to solve the particular problem with an acceptable performance.

For instance, Kaul proposed a fast parser for the computation of a minimum error distance [8]. It depends on so-called *precedence graph grammars* and runs in $\mathcal{O}(n^3)$. Sánchez et al. add special error correcting rules to the graph grammar [9], which they define in terms of *region adjacency graphs*. Their approach appears to be beneficial in the domain of texture symbol recognition.

In the context of diagram editors, the grammar-based system VlDesk [10] provides support for so-called *symbol prompting*. Here, the parsing table of a Yacc-based parser is exploited to extract information about possible contexts of a particular symbol. Following this approach, local suggestions can be made that may, however, be misleading from an overall perspective (although local context in general can provide more suggestions and can be computed more efficiently). In the tool AToM^3 [11] model completion can be realized by solving a *constraint-logic program* that can be generated from the metamodel of the particular language [12].

5 Conclusion and Further Work

We have presented an approach for hypergraph completion with respect to hyperedge replacement grammars. The practicability of the proposed algorithm has been validated by incorporating it in the diagram editor generator DiaGen. The algorithm appears to be widely applicable and sufficiently efficient even for interactive applications. It can be directly used for the realization of content assist in the domain of diagram editors.

Our algorithm is quite beneficial to correct errors in a graph. However, we do not require the given graph to be incomplete. Whereas incomplete graphs can be completed, we can further compute powerful structured editing operations from the complement graphs of already complete graphs. In both DiaGen and also Tiger [13] complex editing commands can be specified by means of graph transformation rules. While this is a powerful way to specify editing operations, it is also quite tedious and error-prone. With our approach a set of applicable commands, which automatically preserve or even establish the syntactical correctness of the resulting graph/diagram, can be computed on the fly.

We already have specified several, prototypical diagram editors with completion support. Nevertheless, in the future we have to study in depth how hypergraph completions can be translated back to diagram completions in a systematic way. We further have to realize more sophisticated user interaction mechanisms which provide maximal benefit of the different completions.

Our algorithm can be extended in a variety of ways. A quite severe restriction is that it can only be applied to context-free languages. Unfortunately, many graph/diagram languages are not context-free. A restrictive embedding mechanism all practical diagram languages can be defined with has been proposed by the third author to address this issue. The adopted parser (as incorporated in DiaGen) is still efficient. In the future we want to investigate how context-sensitive embedding rules can be supported by our parser to further widen its range of applicability.

Acknowledgment

We greatly appreciate the discussions with Birgit Elbl and Wolfram Kahl. Their feedback has been very helpful. Furthermore, the first author is especially grateful to Lothar Schmitz for polishing the wording.

References

1. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Foundations, vol. I, pp. 95–162. World Scientific, Singapore (1997)
2. Lautemann, C.: The complexity of graph languages generated by hyperedge replacement. Acta Inf. 27(5), 399–421 (1989)
3. Kasami, T.: An efficient recognition and syntax analysis algorithm for context free languages. Scientific Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachussetts (1965)
4. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (2002)
5. Mazanek, S., Minas, M.: Functional-logic graph parser combinators. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 261–275. Springer, Heidelberg (2008)
6. Minas, M.: Spezifikation und Generierung graphischer Diagrammeditoren. Shaker-Verlag, Aachen (2001) zugl. Habilitationsschrift Universität Erlangen-Nürnberg (2000)
7. Bengoetxea, E., Pedro Larra, n., Bloch, I., Perchant, A.: Estimation of distribution algorithms: A new evolutionary computation approach for graph matching problems. In: Figueiredo, M., Zerubia, J., Jain, A.K. (eds.) EMMCVPR 2001. LNCS, vol. 2134, pp. 454–468. Springer, Heidelberg (2001)
8. Kaul, M.: Specification of error distances for graphs by precedence graph grammars and fast recognition of similarity. In: Tinhofer, G., Schmidt, G. (eds.) WG 1986. LNCS, vol. 246, pp. 29–40. Springer, Heidelberg (1987)
9. Sánchez, G., Lladós, J., Tombre, K.: An error-correction graph grammar to recognize texture symbols. In: Blostein, D., Kwon, Y.-B. (eds.) GREC 2001. LNCS, vol. 2390, pp. 128–138. Springer, Heidelberg (2002)
10. Costagliola, G., Deufemia, V., Polese, G., Risi, M.: Building syntax-aware editors for visual languages. Journal of Visual Languages and Computing 16(6), 508–540 (2005)
11. de Lara, J., Vangheluwe, H.: Atom3: 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)
12. Sen, S., Baudry, B., Vangheluwe, H.: Domain-specific model editors with model completion. In: Multi-paradigm Modelling Workshop at MoDELS 2007 (2007)
13. Taentzer, G., Crema, A., Schmutzler, R., Ermel, C.: Generating domain-specific model editors with complex editing commands. In: Proc. Third Intl. Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007) (2007)

A Proof Sketch of Lemma 1

Proof. By induction on i. For $i = 1$ layer i is defined as the union of two sets. It can be easily checked that both satisfy the given properties.

Induction step: Let $r = (H', H_c, \sim_n, t) \in L_i$ where $t = (L, R, \mathit{branch})$. Then there has to be a j, $1 \leq j \leq \lfloor i/2 \rfloor$, such that $r \in L_j \oplus L_{i-j}$. This means, that there are two tuples $(H_1', H_{1_c}, \sim_1, t_1) \in L_j$, $(H_2', H_{2_c}, \sim_2, t_2) \in L_{i-j}$ with $E_{H_1'} \cap E_{H_2'} = \emptyset, E_{H_{1_c}} \cap E_{H_{2_c}} = \emptyset, |H_{1_c}| + |H_{2_c}| \leq \mathit{max}$, $\sim_n$ is a minimal relation such that

$\sim_1 \subseteq \sim_n$, $\sim_2 \subseteq \sim_n$, $\mathit{preserves}_{V_H}(\sim)$ and, when identifying nodes equivalent wrt $\sim$, $L \Longrightarrow_P R$ and $\mathit{gc}_H(R)$. Thereby, $L_k := \mathit{root}(t_k), e_k := \mathit{edge}(L_k), k \in \{1,2\}, \sim := \mathit{equi}(\sim_n)$ and $R := L_1 \cup_{\mathrm{ext}_L} L_2$.

1. $|H'| + |H_c| = i$: follows by induction hypothesis and the fact that only graphs with disjoint edge sets are combined.
2. $H' \subseteq H$: this statement holds, since, by induction hypothesis, both H_1' and H_2' are subgraphs of H.
3.

$$\begin{aligned}
\mathit{result}(t) &= R[\{e_k \mapsto \mathit{result}(\mathit{branch}(e_k)) \mid k \in \{1,2\}\}] && (\text{def. } \mathit{result})\\
&= R[\{e_k \mapsto \mathit{result}(t_k) \mid k \in \{1,2\}\}] && (\text{constr. } \mathit{branch})\\
&= R[\{e_k \mapsto H_k' \cup_{\mathrm{ext}_{\mathit{result}(t_k)}} H_{k_c} \mid k \in \{1,2\}\}] && (\text{ind. hypothesis})\\
&= L_1[\{e_1 \mapsto H_1' \cup_{\mathrm{ext}_{\mathit{result}(t_1)}} H_{1_c}\}] \cup_{\mathrm{ext}_L} \\
&\quad\; L_2[\{e_2 \mapsto H_2' \cup_{\mathrm{ext}_{\mathit{result}(t_2)}} H_{2_c}\}] && (e_k \in E_{L_k}, \text{def. } R)\\
&= (H_1' \cup_{\mathrm{ext}_{\mathit{result}(t_1)}} H_{1_c}) \cup_{\mathrm{ext}_L} (H_2' \cup_{\mathrm{ext}_{\mathit{result}(t_2)}} H_{2_c}) && (L_k \text{ elementary})\\
&= (H_1' \cup_\epsilon H_2') \cup_{\mathrm{ext}_L} (H_{1_c} \cup_\epsilon H_{2_c}) && (\text{outermost } \mathit{ext})\\
&= H' \cup_{\mathrm{ext}_L} H_c && (\text{constr. } H', H_c)\\
&= H' \cup_{\mathrm{ext}_{\mathit{result}(t)}} H_c && (L \Longrightarrow_P R)
\end{aligned}$$

4. $|H_c| \leq \mathit{max}$: holds by definition of the layer ($|H_c| = |H_{1_c}| + |H_{2_c}| \leq \mathit{max}$).
5. H_c is complement hypergraph of H' with respect to G and $\mathit{root}(t)$: We just argue here, since otherwise we would need to formally introduce quotient graphs. The prerequisites for the definition of complement graphs are satisfied. Disjointness of edge sets is maintained while constructing the layers. So the only statement to prove is $\mathit{root}(t) = L \Longrightarrow_P^* H' \cup_{\mathrm{ext}_L} H_c = \mathit{result}(t)$. Since by construction t is a proper derivation tree this statement normally holds. However, we have implicitly dealt with equivalence classes of nodes, so that two issues have to be clarified. Firstly, with *preserve* we have prevented nodes of the input graph H to coincide via $\sim$. Thus, H is isomorphic with its quotient graph. Second, it must not happen that by joining nodes the gluing condition ensured at a particular layer can be hurt afterwards. This cannot happen though, since the gluing condition only prevents non-external nodes of the right-hand side of an instantiated production to occur in the remaining graph. In the following, however, only external nodes are joined due to the minimality of $\sim_n$.

Labelled (Hyper)Graphs, Negotiations and the Naming Problem*

Jérémie Chalopin[1], Antoni Mazurkiewicz[2], and Yves Métivier[3]

[1] LIF, Aix-Marseille Université
39 rue Joliot-Curie
13453 Marseille France
jeremie.chalopin@lif.univ-mrs.fr
[2] Institue of Computer Science of PAS
Ordona 21, PL-01-237
Warsaw, Poland
amaz@ipipan.waw.pl
[3] Université de Bordeaux, LaBRI
351 Cours de la Libération
33405 Talence France
metivier@labri.fr

Abstract. We consider four different models of process interactions that unify and generalise models introduced and studied by Angluin et al. [2] and models introduced and studied by Mazurkiewicz [17,18]. We encode these models by labelled (hyper)graphs and relabelling rules on this labelled (hyper)graphs called negotiations. Then for these models, we give complete characterisations of labelled graphs in which the naming problem can be solved. Our characterizations are expressed in terms of locally constrained homomorphisms that are generalisations of known graph homomorphisms.

1 Introduction

Three major process influence (interaction) models in distributed systems have principally been considered: the message passing model, the shared memory model, the local computation model. In the three models the processes are represented by vertices of a graph and the interactions are represented by edges or hyperedges. In the message passing model processes interact by messages: they can be sent along edges. In the shared memory model atomic read/write operations can be performed on registers associated with hyperedges. In the local computation model, interactions are defined by labelled graph relabelling rules; supports of rules (graphs used for the description of labellings) are edges or stars. These models (and their sub-models) reflect different system architectures, different levels of synchronization and different levels of abstraction. The

* This work was supported by grant No ANR-06-SETI-015-03 awarded by Agence Natioanle de la Recherche.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 54–68, 2008.

structure of the communication or of the interaction subsystem is represented as a graph. In general this graph is static: it means that it remains fixed during the distributed computation. Some works consider dynamic graphs: some links may fail and recover and some nodes may fail and recover.

In this paper we consider local computations on another kind of dynamic distributed system: processes are mobile and they interact when they are sufficiently close to each other or when some localisation conditions are verified. In [2], Angluin et al. consider a distributed system where a set of moving sensors can interact when they are sufficiently close to each other. They assume that every pair of sensors repeatedly come sufficiently close to each other for being able to communicate, i.e., the interaction graph is the complete graph. In their work, they consider finite-state sensors (each sensor has a constant number of bits of memory) and they study the computational power of such a system.

In [17,18] the distributed system is presented in the following way. There is a number of individuals, each of them brings an integer as a label. They are grouped into associations: within an association they can communicate, exchange information, and modify their labels; there is no possibility of direct communication between individuals that do not belong to the same association. However, since some individuals can be affiliated to more than one association, indirect communication between remote individuals is possible using individuals with multiple affiliations as go between. Such systems are called communication structures. Associations act by their assemblies that take place from time to time; an association is active during its assembly, and passive out of it. The purpose of an assembly is to exchange information among participants and to update the states of the participants.

In this work, we consider a system where the processes have an unbounded number of states and where a computation step can involve an arbitrary number of processes. Moreover, we do not assume that each process can interact with any other process: we just assume that the communication structure is connected. We study the computational power of such systems through the naming problem.

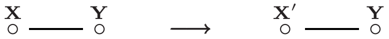
The naming problem. We focus on the naming problem, that is a classical problem to highlight differences between different models of distributed computing. A distributed algorithm $\mathcal{A}$ is a naming algorithm if each execution of $\mathcal{A}$ terminates and leads to a final configuration where all processes have unique identities. Being able to give dynamically and in a distributed way unique identities to all processes is very important since many distributed algorithms work correctly only under the assumption that all processes can be unambiguously identified. In this paper naming is done using a distributed enumeration algorithm. A distributed enumeration algorithm assigns to each network vertex a unique integer ; in such a way we obtain a bijection between the set $V(G)$ of vertices and $\{1, 2, \ldots, |V(G)|\}$.

The study of the naming problem makes it possible to highlight combinatorial tools useful for other problems like termination detection or recognition (see [20,21,5,7]).

Formal Models. A *communication structure* is defined by a set of *individuals* that belong to different *associations*. Some individuals can be affiliated to more than one association. A communication structure is represented as an undirected hypergraph: vertices represent individuals and hyperedges define associations. In the particular case where all associations have exactly two members, a communication structure can be seen as a simple graph. Labels (states) are attached to individuals and associations thus we consider labelled (hyper)graphs which are defined by a labelling function λ which associates to a vertex or an (hyper)edge a label. In the more general model of computation called *labelled negotiations*, a computation step enables to modify the states of the vertices of a hyperedge and the label of the hyperedge itself according only to their previous states. In other words, in one computation step, the members of an association synchronize, exchange their labels and modify them. We consider communication structures where hyperedges cannot be labelled and then vertices cannot always distinguish the different hyperedges they belong to. This restriction leads us to study *unlabelled negotiations* We also consider models of computations where in one computation step, one vertex observes the states of the vertices of a hyperedge it belongs to and the state of the hyperedge (if available) and modifies only its state and the state of the hyperedge (if this one is available). Such a model of computation will be called *cellular*. Thus, we study *cellular (un)labelled negotiations*.

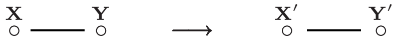
Our results. We characterize labelled (hyper)graphs where the naming problem can be solved in the four different models we consider. We first show that cellular labelled negotiations have the same computational power as labelled negotiations (Proposition 1). To give our characterization, we generalize locally constrained graph homomorphisms to hypergraphs (Section 2). This enables us to formulate conveniently necessary conditions (Lemma 1) inspired by Angluin's lifting lemma [1]. It turns out that the necessary conditions are also sufficient. Then we present algorithms that solve the naming problem (Theorems 1, 2 and 3) (Theorem 1 is another formulation of a result presented in [17]).

Related Work. In [2,4,3] Angluin et al. study the power of models of computation by pairwise interactions of identical finite-state agents. The general question is of characterising what computations are possible. They prove in particular that all predicates stably computable are semilinear, in the model in which finite state agents compute a predicate of their inputs via two-way interactions in the all-pairs family of communication networks [3]. This kind of computations may be encoded by local computations on edges of labelled graphs. The case of one-way communication between two agents corresponds to cellular local computations on edges, thus of the form:



where X, Y and X' are labels (states) attached to vertices, $X' = f(X, Y)$ and f is a transition function. In [11], a complete characterization of labelled graphs for which enumeration and election are possible is presented.

The case of two-way communication corresponds to local computations on edges of labelled graphs, thus of the form:



where X, Y, X' and Y' are labels (states) attached to vertices, $X' = f_1(X, Y)$, $Y' = f_2(Y, X)$ and f_1, f_2 are transition functions. Graphs for which the naming problem is solvable are characterized in [8].

All-pairs family of communication network is captured by our model by considering the case where each association has no name and has exactly two members, and the set of associations defines the complete graph. The two-way interaction model of [3] corresponds to our general model of computation. The one-way population protocol defined in [4] corresponds to the cellular computation model. In [17,18] associations are labelled and cellular relabellings are not considered.

Overview. The structure of this paper is as follows. Section 2 reviews basic definitions of communication structures and negotiations. In Section 3 first we prove that labelled negotiations can be simulated by cellular labelled negotiations, then we present characterisations of communication structures which admit a naming algorithm using (cellular) labelled negotiations. Section 4 presents characterisations of communication structures which admit a naming algorithm using (cellular) unlabelled negotiations. Section 5 presents final remarks.

2 Preliminaries

2.1 Communication Structures and Labelled Graphs

A *communication structure* C is defined by a set $B(C)$ of *individuals* and a set $A(C)$ of *associations*: each association is a set of individuals. Each individual $b \in B(C)$ belongs to one or more associations $a \in A(C)$ and it will be denoted by $b \in a$; one will say that b is a *member* of a and that *a contains* b. Each association may contain an arbitrary number of elements and two distinct associations can have the same members. A communication structure is connected if for any associations $a, a' \in A(C)$, there exists a sequence $a_0, a_1, \ldots, a_n$ such that $a_0 = a$, $a_n = a'$ and for any $i \in [1, n]$, there exists an individual that belongs to a_{i-1} and a_i. In the following, we will only consider connected communication structures.

A communication structure C is bilateral if each association contains exactly two elements. In this case C can be represented by a graph where vertices are individual and edges are associations.

A communication structure C may be viewed as a hypergraph where vertices denote individuals and hyperedge denote associations. It will be represented by a simple bipartite graph G_C that is a classical representation of hypergraphs. The set of vertices $V(G_C)$ contains two disjoint subsets $V_A(G_C)$ and $V_B(G_C)$. Each association a (resp. individual b) of C corresponds to a vertex $v_a \in V_A(G_C)$ (resp. $v_b \in V_B(G_C)$). If an individual b belongs to an association a, then there is an edge $\{v_a, v_b\}$ in $E(G_C)$. Given a vertex $v \in V_A(G_C) \cup V_B(G_C)$, $N_{G_C}(v)$

denotes the set of neighbours of $v \in G_C$, i.e., the set $\{v' \mid \{v, v'\} \in E(G_C)\}$. A graph homomorphism φ from G to G' is a mapping from $V(G)$ to $V(G')$ such that if $\{v, w\} \in E(G)$, then $\{\varphi(v), \varphi(w)\} \in E(G')$.

We want to extend to communication structures the definitions of coverings, pseudo-coverings and submersions that are used in [8,9,10,14] for graphs, that are bilateral communication structures studied in [17]. We give a definition of homomorphism between communication structures: it is a generalization of graph homomorphisms.

Definition 1. *Given two communications structures C and C', a mapping φ from $B(C)$ to $B(C')$ and from $A(C)$ to $A(C')$ is a homomorphism from C to C' if it induces a graph homomorphism of G_C to $G_{C'}$ such that for each vertex $v_a \in V_A(G_C)$, the following holds: (1) $|N_{G_C}(v_a)| = |N_{G_{C'}}(\varphi(v_a))|$, and (2) $\varphi(N_{G_C}(v_a)) = N_{G_{C'}}(\varphi(v_a))$.*

Throughout the paper we will consider communication structures where individuals and associations are labelled with labels from a recursive label set L that admits a total order $<_L$. A labelled communication structure will be denoted by $\mathbf{C} = (C, \lambda)$ where C is the underlying unlabelled communication structure and where $\lambda : B(C) \cup A(C) \to L$ is a labelling function. A mapping φ from $\mathbf{C} = (C, \lambda)$ to $\mathbf{C}' = (C', \lambda')$ is a homomorphism if φ is a homomorphism from C to C' that preserves the labelling, i.e., for each $x \in B(C) \cup A(C), \lambda(x) = \lambda'(\varphi(x))$.

For any set S, $|S|$ denotes the cardinality of S while $\mathcal{P}_{\text{fin}}(S)$ is the set of finite subsets of S.

2.2 Locally Constrained Homomorphisms

We now define submersions, coverings and pseudo-coverings of communication structures that are just generalizations of existing definitions for graphs. A communication structure $\mathbf{C}$ is a submersion of $\mathbf{C}'$ if there exists a locally surjective homomorphism from $\mathbf{C}$ to $\mathbf{C}'$.

Definition 2. *Given two communication structures $\mathbf{C}$=(C, λ) and $\mathbf{C}'$=(C', λ'), $\mathbf{C}$ is a* submersion *of $\mathbf{C}'$ via a homomorphism φ if for each vertex $v_b \in V_B(G_C)$, $\varphi(N_{G_C}(b)) = N_{G_{C'}}(\varphi(b))$. In this case, we say that φ is a locally surjective homomorphism from $\mathbf{C}$ to $\mathbf{C}'$.*

In other words, a homomorphism φ from $\mathbf{C}$ to $\mathbf{C}'$ is locally surjective if for each individual $b \in B(C)$, the associations that contain $\varphi(b)$ are the images of the associations that contain b. A communication structure $\mathbf{C}$ will be called *submersion-minimal* if for any $\mathbf{C}'$ with $|B(C')| < |B(C)|$, $\mathbf{C}$ is not a submersion.

A communication structure $\mathbf{C}$ is a covering of $\mathbf{C}'$ if there exists a locally bijective homomorphism from $\mathbf{C}$ to $\mathbf{C}'$.

Definition 3. *Given two communication structures $\mathbf{C}$ and $\mathbf{C}'$, $\mathbf{C}$ is a* covering *of $\mathbf{C}'$ via a homomorphism φ if for each vertex $v_b \in V_B(G_C)$, $|N_{G_C}(v_b)| = |N_{G_{C'}}(\varphi(v_b))|$ and $\varphi(N_{G_C}(v_b)) = N_{G_{C'}}(\varphi(v_b))$. In this case, we say that φ is a locally bijective homomorphism from C to C'.*

In other words, a homomorphism φ from $\mathbf{C}$ to $\mathbf{C}'$ is locally bijective if for each individual $b \in B(C)$ φ induces a bijection between the associations that contain b and the associations that contain $\varphi(b)$. A communication structure $\mathbf{C}$ will be called *covering-minimal* if $\mathbf{C}$ is not a covering of any $\mathbf{C}'$ such that $|B(C')| < |B(C)|$.

We now define pseudo-coverings that generalize pseudo-coverings of graphs introduced in [8].

Definition 4. *Given two communication structures* $\mathbf{C}=(C,\lambda)$ *and* $\mathbf{C}'=(C',\lambda')$, $\mathbf{C}$ *is a* pseudo-covering *of* $\mathbf{C}'$ *via a homomorphism* φ *if there exists a subset* A_0 *of* $A(C)$ *such that the communication structure* $\mathbf{C}_0 = (C_0, \lambda_0)$ *defined by* $B(C_0) = B(C)$, $A(C_0) = A_0$ *and for each* $x \in B(C_0) \cup A(C_0), \lambda_0(x) = \lambda(x)$ *is a covering of* $\mathbf{C}'$ *via the restriction of* φ *to* C_0*.*

A communication structure $\mathbf{C}$ will be called *pseudo-covering-minimal* if $\mathbf{C}$ is not a pseudo-covering of any $\mathbf{C}'$ such that $|B(C')| < |B(C)|$.

Obviously, if a communication structure $\mathbf{C}$ is a covering of $\mathbf{C}'$, then $\mathbf{C}$ is a pseudo-covering of $\mathbf{C}'$ and if $\mathbf{C}$ is a pseudo-covering of $\mathbf{C}'$, then $\mathbf{C}$ is a submersion of $\mathbf{C}'$.

2.3 Negotiations and Relabelling Rules

In our models, in one computation step the states of an association and its members are modified according only to their previous states. An algorithm can then be described by a set $\mathcal{R}$ of relabelling rules $r = (\lambda_r, \lambda'_r)$ where λ_r and λ'_r are two labellings of an association. A computation step is then an application of a rule to some association of the communication structure. We will note $\mathbf{C}\mathcal{R}\mathbf{C}'$ if $\mathbf{C}'$ can be obtained from $\mathbf{C}$ by applying a rule of $\mathcal{R}$ to some association of $\mathbf{C}$. Obviously, $\mathbf{C}$ and $\mathbf{C}'$ have the same underlying communication structure C, only the labelling of the active association is modified. Thus, slightly abusing the notation, $\mathcal{R}$ will stand both for a set of rules and the induced relabelling relation over labelled communication structure. The transitive closure of such a relabelling relation is noted $\mathcal{R}^*$. Computations using uniquely this type of relabelling rules are called in our paper *negotiations.*

The relation $\mathcal{R}$ is called *noetherian* on a communication structure $\mathbf{C}$ if there is no infinite relabelling sequence $\mathbf{C}_0\mathcal{R}\mathbf{C}_1\mathcal{R}\dots$ with $\mathbf{C}_0 = \mathbf{C}$. The relation $\mathcal{R}$ is noetherian if it is noetherian on each communication structure. Clearly, noetherian relations code always terminating algorithms.

An algorithm encoded with such computation rules is a distributed algorithm in the sense that two computation steps can be applied simultaneously to two distinct associations, provided that no individual belongs to both of associations.

In the following, we will consider four different models of negotiations. The most general model described above is called *labelled negotiations.* We will also deal with communication structures where the associations cannot be labelled, this model will be called *unlabelled negotiations.* We will also consider models where in one computation step, the label of at most one member can be modified, i.e., in one computation step, one member modifies its label and the label of an association it belongs to (if associations can be labelled) according to the labels of all the members of this association and to the label of the association (if the associations can be labelled). When the associations can be labelled, the model corresponding to this kind of computation steps will be called *cellular labelled negotiations* and when the associations cannot be labelled, it will be called *cellular unlabelled negotiations.*

Given a terminating algorithm $\mathcal{A}$ using labels in a set L, one will say that an algorithm $\mathcal{A}'$ using labels in a set L' simulates $\mathcal{A}$ if there exists a mapping $\pi : L' \to L$ such that for any communication structure $\mathbf{C}$, each execution of $\mathcal{A}'$ on $\mathbf{C}$ terminates and for each execution of $\mathcal{A}'$ on $\mathbf{C}$ with a final labelling λ', there exists an execution of $\mathcal{A}$ on $\mathbf{C}$ with a final labelling λ on $\mathbf{C}$, such that for each $x \in B(C) \cup A(C)$, $\pi(\lambda'(x)) = \lambda(x)$. It is obvious that any algorithm using cellular labelled negotiations is an algorithm that uses labelled negotiations; in fact, cellular labelled negotiations have the same computational power as labelled negotiations:

Proposition 1. *Any algorithm $\mathcal{A}$ using labelled negotiations can be simulated by an algorithm $\mathcal{A}'$ that uses cellular labelled negotiations.*

In the following, we say that an algorithm $\mathcal{A}$ has the termination detection property if for any execution of $\mathcal{A}$ on $\mathbf{C}$, there exists an individual $b \in B(C)$ that can detect locally (according only to its state) that the computation is over, i.e., that each individual has computed its final value.

An algorithm will be described by a recursive set of rules of the form (λ_r, λ'_r). We can see each rule r as a couple of two multisets:

$$(\{\{\lambda_{r,0}, \lambda_{r,1}, \ldots, \lambda_{r,k}\}\}, \{\{\lambda'_{r,0}, \lambda'_{r,1}, \ldots, \lambda'_{r,k}\}\}).$$

We can apply r to an association a if a has k members, if the label of a (if available) is $\lambda_{r,0}$ and if the multisets of labels $\{\{\lambda(b) \mid b \in a\}\}$ is equal to $\{\{\lambda_{r,1}, \ldots, \lambda_{r,k}\}\}$. In this case, the label of a becomes $\lambda'_{r,0}$ and the label of each member of a labelled by $\lambda_{r,i}$ becomes $\lambda'_{r,i}$. In the case of cellular negotiations, for each $i > 1$, we should have $\lambda'_{r,i} = \lambda_{r,i}$.

When we want to describe a set of rules that do not depend on the size of the association a, we will write the precondition as a logical formula that the labels of a and its members must satisfy to apply the rule. Then we describe the new labels of a and of each member of a. This description enables to encode an infinite number of relabelling rules (one for each size of association) in a finite way.

2.4 Impossibility Result

The following lemma exhibits a strong link between homomorphisms and negotiations. This is a counterpart of the lifting lemma of Angluin [1] adapted to communication structure homomorphisms.

Lemma 1 (Lifting Lemma). *Let $\mathcal{R}$ be a relabelling relation corresponding to an algorithm using labelled negotiations (resp. unlabelled negotiations, cellular unlabelled negotiations) and let $\mathbf{C}_1$ be a covering (resp. pseudo-covering, submersion) of $\mathbf{C}_2$. If $\mathbf{C}_2\mathcal{R}^*\mathbf{C}'_2$, then there exists $\mathbf{C}'_1$ such that $\mathbf{C}_1\mathcal{R}^*\mathbf{C}'_1$ and $\mathbf{C}'_1$ is a covering (resp. pseudo-covering, submersion) of $\mathbf{C}'_2$.*

Consequently, there cannot exist a naming algorithm $\mathcal{A}$ using labelled negotiations (resp. unlabelled negotiations, cellular unlabelled negotiations) on a communication structure $\mathbf{C}_1$ that is not covering-minimal (resp. pseudo-covering-minimal, submersion-minimal). Indeed, if $\mathbf{C}_1$ is a covering (resp. pseudo-covering, submersion) of $\mathbf{C}_2$ with $|B(C_2)| < |B(C_1)|$, consider a terminating execution ρ of $\mathcal{A}$ on $\mathbf{C}_2$ that leads to a final configuration $\mathbf{C}'_2$. From Lemma 1, one can construct a terminating execution on $\mathbf{C}_1$ from ρ that leads to a final configuration $\mathbf{C}'_1$ that is a covering (resp. pseudo-covering, submersion) of $\mathbf{C}'_2$. Consequently, there exists an individual in $\mathbf{C}'_2$ whose label appears at least twice in $\mathbf{C}'_1$: individuals do not have unique identities in $\mathbf{C}'_1$.

3 (Cellular) Labelled Negotiations

In this section, we give a characterization of communication structures where we can solve the naming problem using cellular labelled negotiations or labelled negotiations. We give a naming algorithm using labelled negotiations that solves the enumeration problem on any communication structure $\mathbf{C}$ that is covering-minimal.

Each individual b (resp. association a) attempts to get a number between 1 and $|B(C)|$ (resp. $|A(C)|$). Each individual (resp. association) chooses a number and collects the numbers of the associations it belongs to (resp. the numbers of its members) to construct its *local view*. Then, each individual and each association broadcasts its number with its label and its local view. If some individual b (resp. an association a) detects that there exists another individual b' (resp. another association a') with the same number, then it compares its label and its local view with the label and the local view of its opponent. If the label or the local view of b is “weaker”, then b chooses a new number and broadcasts it again. At the end of the computation, each individual and each association will have a unique number if the communication structure is covering-minimal.

Labels. Consider a communication structure $\mathbf{C} = (C, \lambda)$ with an initial labelling $\lambda\colon B(C) \cup A(C) \to L$. During the computation each individual $b \in B(C)$ will acquire new labels of the form $(\lambda(b), n(b), N(b), M(b), S(b))$ and each association $a \in A(C)$ will get labels of the form $(\lambda(a), m(a), P(a))$ where:

- the first component $\lambda(b)$ (resp. $\lambda(a)$) is just the initial label (and thus remains fixed during the computation),
- $n(b) \in \mathbb{N}$ (resp. $m(a) \in \mathbb{N}$) is the current *identity number* of b (resp. a) computed by the algorithm,
- $N(b) \in \mathcal{P}_{\mathrm{fin}}(\mathbb{N})$ (resp. $P(a) \in \mathcal{P}_{\mathrm{fin}}(\mathbb{N})$) is the *local view* of b (resp. a). Intuitively, the algorithm will try to update the current view in such a way that $N(b)$ (resp. $P(a)$) will consist of the current identities of the associations that contains b (resp. of the members of a). Therefore $N(b)$ (resp. $P(a)$) will be always a finite (possibly empty) set of integers,
- $M(b) \subseteq \mathbb{N} \times L \times \mathcal{P}_{\mathrm{fin}}(\mathbb{N})$ is the current *individual-mailbox* of b. It contains the whole information about individuals received by b during the computation.

- $S(b) \subseteq \mathbb{N} \times L \times \mathcal{P}_{\text{fin}}(\mathbb{N})$ is the current *association-mailbox* of b. It contains the whole information about associations received by b during the computation.

The fundamental property of the algorithm is based on a total order on the set $\mathcal{P}_{\text{fin}}(\mathbb{N})$ of local views, as defined by Mazurkiewicz [16]. Consider two sets N_1, N_2 of integers. Suppose that $N_1 \neq N_2$. Then $N_1 \prec_1 N_2$ if the maximal element of the symmetric difference $N_1 \triangle N_2 = (N_1 \setminus N_2) \cup (N_2 \setminus N_1)$ belongs to N_2. Note that in particular the empty set is minimal for $\prec_1$. If $N(b) \prec_1 N(b')$ then we say that the local view $N(b')$ of b' is stronger than the one of b (and $N(b)$ is weaker than $N(b')$). We extend $\prec_1$ to a total order on $L \times \mathcal{P}_{\text{fin}}(\mathbb{N})$: $(\ell, N) \prec_1 (\ell', N')$ if either $\ell <_L \ell'$ or ($\ell = \ell'$ and $N \prec_1 N'$). We will also use the reflexive closure $\preceq_1$ of $\prec_1$.

Labelled Negotiations Rules. We describe here the relabelling rules that define the enumeration algorithm. First of all, to launch the algorithm there is a special initial rule $\mathcal{R}_0$ that just extends the initial label $\lambda(b)$ (resp. $\lambda(a)$) of each individual b (resp. association a) to $(\lambda(b), 0, \emptyset, \emptyset, \emptyset)$ (resp. $(\lambda(a), 0, \emptyset)$). The label of an association a (resp. a member b of a) obtained by the application of a rule to a is denoted $(\lambda(a), m'(a), P'(a))$ (resp. $(\lambda(b), n'(b), N'(b), M'(b), S'(b))$).

The first rule $\mathcal{R}_1$ enables to update the mailboxes of all the individuals that belongs to a same association.

```
R_1:
if $\exists b, b' \in a, M(b) \neq M(b')$ or $S(b) \neq S(b')$ then
    $\forall b \in a, M'(b) := \bigcup_{b \in a} M(b)$ and $S'(b) := \bigcup_{b \in a} S(b)$ ;
```

The second rule $\mathcal{R}_2$ does not involve any synchronisation. It enables an individual b to change its identity if its current identity number $n(b)$ is 0 or if it knows that there exists another individual with the same number and a stronger label or a stronger local view.

```
R_2:
if $n(b) = 0$ or $\exists (n(b), \ell, N) \in M(b)$ such that $(\lambda(b), N(b)) \prec_1 (\ell, N)$ then
    $n'(b) := 1 + \max\{n' \mid (n', \ell', N') \in M(b)\}$;
    $M'(b) := M(b) \cup \{(n'(b), \lambda(b), N(b))\}$;
```

The rules $\mathcal{R}_3, \mathcal{R}_4, \mathcal{R}_5$ are designed such that one can apply one of these rules to some association a only if one cannot apply the preceding ones to a. The third rule enables an individual b_0 to modify its identity if it belongs to some association a such that there exists another individual $b_1 \in a$ with the same number, the same label and the same local view.

```
R_3:
if $\forall b, b' \in a, M(b) = M(b')$ and $S(b) = S(b')$ and $\forall b \in a, n(b) \neq 0$ and $\forall (n(b), \ell, N) \in M(b), (\ell, N) \preceq_1 (\lambda(b), N(b))$ and $\exists b_0, b_1 \in a$ such that $n(b_0) = n(b_1)$ then
    $n'(b_0) := 1 + \max\{n' \mid (n', \ell', N') \in M(b_0)\}$;
    $M'(b_0) := M(b_0) \cup \{(n'(b_0), \lambda(b_0), N(b_0))\}$;
```

The fourth rule is the counterpart for associations of the second rule. It enables to modify the identity of an association a if the current identity number $m(a)$ is 0 or if there exists another association with the same number and a stronger label or a stronger local view. When this rule is applied to a, the local view of the members of a is updated.

```
$\mathcal{R}_4$:
**if** $\forall b, b' \in a, M(b) = M(b'),\ S(b) = S(b')$ *and* $n(b) \neq n(b')$,
$\forall b \in a, n(b) \neq 0$ *and* $\forall (n(b), \ell, N) \in M(b), (\ell, N) \preceq_1 (\lambda(b), N(b))$
*and* $m(a) = 0$ *or* $\exists (m(a), \ell, P) \in S(b)$ *such that* $(\lambda(a), P(a)) \prec_1 (\ell, P)$ **then**
    $m'(a) := 1 + \max\{m' \mid (m', \ell', P') \in M(b)\}$;
    $P'(a) := \{n(b) \mid b \in a\}$;
    $\forall b \in a, N'(b) := N(b) \setminus \{m(a)\} \cup \{m'(a)\}$;
    $\forall b \in a, M'(b) := M(b) \cup \{(n(b'), \lambda(b'), N'(b')) \mid b' \in a\}$;
    $\forall b \in a, S'(b) := S(b) \cup \{(m'(a), \lambda(a), P'(a))\}\}$;
```

The last rule enables to update the local view of an association.

```
$\mathcal{R}_5$:
**if** $\forall b, b' \in a, M(b) = M(b'),\ S(b) = S(b')$ *and* $n(b) \neq n(b')$
*and* $\forall b \in a, n(b) \neq 0$ *and* $\forall (n(b), \ell, N) \in M(b), (\ell, N) \preceq_1 (\lambda(b), N(b))$
*and* $p(a) \neq 0$ *and* $\forall (m(a), \ell, P) \in S(b), (\ell, P) \preceq_1 (\lambda(a), P(a))$
*and* $\exists b_0 \in a$ *such that* $n(b_0) \notin P(a)$ **then**
    $P'(a) := \{n(b) \mid b \in a\}$;
    $\forall b \in a, S'(b) := S(b) \cup \{(m(a), \lambda(a), P(a))\}\}$;
```

Any execution of the algorithm satisfies monotonicity properties. Any run ρ of the algorithm on a labelled communication structure $\mathbf{C} = (C, \lambda)$ terminates and yields a final labelling $(\lambda, n_\rho, N_\rho, M_\rho, S_\rho)$ of individuals and a final labelling $(\lambda, m_\rho, P_\rho)$ of associations.

The mapping defined by n_ρ and m_ρ is a locally bijective homomorphism from $\mathbf{C}$ to $\mathbf{C}'_\rho$. Consequently, if $\mathbf{C}$ is covering-minimal, it implies that in the final configuration, $\{n_\rho(b) \mid b \in B(C)\} = [1, |B(C)|]$. From Lemma 1 and Proposition 1, we get the following theorem.

Theorem 1. *For every communication structure* $\mathbf{C}$*, there exists a naming algorithm for* $\mathbf{C}$ *using (cellular) labelled negotiations if and only if* $\mathbf{C}$ *is covering-minimal.*

Suppose that all the individuals initially know $|B(C)|$, then the termination detection of the algorithm is possible on a covering-minimal communication structure $\mathbf{C}$. Indeed, once an individual gets the identity number $|B(C)|$, it knows that all the individuals have different identity numbers that will not change any more.

4 (Cellular) Unlabelled Negotiations

We now consider unlabelled negotiations and cellular unlabelled negotiations. We give characterizations of communication structures where we can solve the naming problem using these two kinds of negotiations. A corollary of these characterizations is that unlabelled negotiations have a strictly greater computational power than cellular unlabelled negotiations.

4.1 Cellular Unlabelled Negotiations

The algorithm uses the same ideas as the algorithm of the previous section. The main difficulty is to achieve to update correctly the local view of the individuals.

Consider a communication structure $\mathbf{C} = (C, \lambda)$ with an initial labelling $\lambda \colon B(C) \to L$. During the computation each individual $b \in B(C)$ will acquire new labels of the form $(\lambda(b), n(b), N(b), M(b))$ where:

- $n(b) \in \mathbb{N}$ is the identity number of b.
- $N(b) \in \mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\mathbb{N}))$ is the local view of b. Intuitively, the algorithm will try to update the local view of b such that $N(b)$ contains a set $\{n(b') \mid b' \in a\}$ for each association a that contains b.
- $M(b)$ is the mailbox of b and it contains the whole information the individual b has about the network.

We also need to define a total order on local views. We will just generalize the order defined above. Consider two sets N_1, N_2 whose elements are some sets of integers ordered by $\prec_1$. Suppose that $N_1 \neq N_2$. Then $N_1 \prec_2 N_2$ if the maximal element for $\prec_1$ of the symmetric difference $N_1 \triangle N_2 = (N_1 \backslash N_2) \cup (N_2 \backslash N_1)$ belongs to N_2. Again, we extend $\prec_2$ to a total order on $L \times \mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\mathbb{N}))$: $(\ell, N) \prec_2 (\ell', N')$ if either $\ell <_L \ell'$ or ($\ell = \ell'$ and $N \prec_2 N'$).

Cellular Unlabelled Negotiations Rules. The label of an individual b_0 after the application of a relabelling rule to an association a that modifies the state of b_0 is denoted by $(\lambda(b_0), n'(b_0), N'(b_0), M'(b_0))$. The three first rules have the same meaning as the three first rules of the algorithm described in the previous section.

```
$\mathcal{R}_1$:
if $\exists b \in a, M(b) \setminus M(b_0) \neq \emptyset$ then
    $M'(b_0) := \bigcup_{b \in a} M(b)$;

$\mathcal{R}_2$:
if $n(b_0) = 0$ or $\exists (n(b_0), \ell, N) \in M(b_0)$ such that $(\lambda(b_0), N(b_0)) \prec_2 (\ell, N)$ then
    $n'(b_0) := 1 + \max\{n' \mid (n', \ell', N') \in M(b_0)\}$;
    $M'(b_0) := M(b_0) \cup \{(n'(b_0), \lambda(b_0), N(b_0))\}$;

$\mathcal{R}_3$:
if $\forall b, b' \in a, M(b) = M(b')$,
$\forall b \in a, n(b) \neq 0$ and $\forall (n(b), \ell, N) \in M(b), (\ell, N) \preceq_2 (\lambda(b), N(b))$
and $\exists b \in a, b \neq b_0$ such that $n(b_0) = n(b)$ then
    $n'(b_0) := 1 + \max\{n' \mid (n', \ell', N') \in M(b_0)\}$;
    $M'(b_0) := M(b_0) \cup \{(n'(b_0), \lambda(b_0), N(b_0))\}$;
```

The fourth rule enables an individual b to add the set $S' = \{n(b') \mid b' \in a\}$ of the current identity numbers of the members of an association a it belongs to. In this case, all the sets S belonging to $N(b)$ such that $S \prec_1 S'$ are removed.

The intuitive justification for the deletion of all such S is the following. Suppose that there exists an association a that contains b such that the set $S' = \{n(b') \mid b' \in a\}$ does not belong to $N(b)$. Suppose that there is a computation step that enables to modify the state of b according to the states of the members of a. Then, since the very purpose of the view $N(b)$ is to stock the identity numbers of all the members of all the associations it belongs to, we should add S' to the view $N(b)$ of b.

If the state of b is modified according to a for the first time, then adding S' to $N(b)$ is sufficient. But, it can also be the case that b modified its state according to a in the past and in the meantime another member b' of a has modified its identity. Then b should not only add S' to $N(b)$ but it should remove the old set of identity numbers corresponding to a from its view. The problem is that b cannot know which set it should remove from its view. However, since our algorithm ensures that the identity numbers of individuals can only increase, we know that the eventual old set of numbers S is weaker for $\prec_1$ than S' and consequently, by removing all the $S \prec_1 S'$, we are sure to delete all invalid information. Of course, we can also delete legitimate informations from the local view of b. But in this case, b can recover this information by some new applications of $\mathcal{R}_4$ to the other associations that contain b.

```
$\mathcal{R}_4$:
if $\forall b, b' \in a, M(b) = M(b')$ and $n(b) \neq n(b')$,
$\forall b \in a, n(b) \neq 0$ and $\forall (n(b), \ell, N) \in M(b), (\ell, N) \preceq_2 (\lambda(b), N(b))$
and $\{n(b') \mid b' \in a\} \notin N(b_0)$ then
    $S' := \{n(b') \mid b' \in a\}$;
    $N'(b_0) := N(b_0) \setminus \{S \mid S \prec_1 S'\} \cup \{S'\}$;
    $M'(b_0) := M(b_0) \cup \{(n(b_0), \lambda(b_0), N'(b_0))\}$;
```

Properties. The algorithm we described has the same interesting properties as the one described in Section 3. And from Lemma 1, we get:

Theorem 2. *For every communication structure* $\mathbf{C}$*, there exists a naming algorithm for* $\mathbf{C}$ *using cellular unlabelled negotiations if and only if* $\mathbf{C}$ *is submersion-minimal.*

Again, if the individuals initially know $|B(C)|$, then the termination detection of the algorithm is possible in a submersion-minimal communication structure: once an individual gets the number $|B(C)|$, it knows that each individual has a unique number that will not change any more.

4.2 Unlabelled Negotiations

We add time-stamps to local views in order to obtain a pseudo-covering with the final labelling. Consider a communication structure $\mathbf{C} = (C, \lambda)$ with an initial labelling $\lambda \colon B(C) \to L$. Here again, each individual $b \in B(C)$ will acquire new labels of the form $(\lambda(b), n(b), N(b), M(b))$ where:

- $n(b) \in \mathbb{N}$ is the identity number of b.
- $N(b) \in \mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\mathbb{N}) \times \mathbb{N})$ is the local view of b. Intuitively, the algorithm will try to update the local view of b such that $N(b)$ contains a set $\{n(b') \mid b' \in a\}$ for each association a that contains b. Moreover, a time-stamp o will be associated to each of these sets in order to enable an individual, when it is possible, to detect if it belongs to different associations whose members have the same numbers.
- $M(b)$ is the mailbox of b and it contains the whole information the individual b has about the network.

Again, we need a total order on local views. Consider two sets N_1, N_2 whose elements are some pairs (S, o) where $o \in \mathbb{N}$ and $S \in \mathcal{P}_{\text{fin}}(\mathbb{N})$. Given two elements (S, o) and (S', o'), one will generalize $\prec_1$ to say that $(S, o) \prec_1 (S', o')$ if $S \prec_1 S'$ or if $S =_1 S'$ and $o < o'$. We now define a new order $\prec_3$ for elements of $\mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\mathbb{N}) \times \mathbb{N})$. We say that $N_1 \prec_3 N_2$ if the maximal element for $\prec_1$ of the symmetric difference $N_1 \triangle N_2 = (N_1 \setminus N_2) \cup (N_2 \setminus N_1)$ belongs to N_2. Again, we extend $\prec_3$ to a total order on $L \times \mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\mathbb{N}))$: $(\ell, N) \prec_3 (\ell', N')$ if either $\ell <_L \ell'$ or ($\ell = \ell'$ and $N \prec_3 N'$).

Unlabelled negotiations rules. The label of an individual b after the application of a relabelling rule to an association a that contains b is denoted by $(\lambda(b), n'(b), N'(b), M'(b))$. The three first rules have the same meaning as the three first rules of the algorithms described above.

```
$\mathcal{R}_1$:
if $\exists b, b' \in a, M(b) \neq M(b')$ then
    $\forall b \in a, M'(b) := \bigcup_{b \in a} M(b)$;

$\mathcal{R}_2$:
if $n(b) = 0$ or $\exists (n(b), \ell, N) \in M(b)$ such that $(\lambda(b), N(b)) \prec_3 (\ell, N)$ then
    $n'(b) := 1 + \max\{n' \mid (n', \ell', N') \in M(b)\}$;
    $M'(b) := M(b) \cup \{(n'(b), \lambda(b), N(b))\}$;

$\mathcal{R}_3$:
if $\forall b, b' \in a, M(b) = M(b')$
and $\forall b \in a, n(b) \neq 0$ and $\forall (n(b), \ell, N) \in M(b), (\ell, N) \preceq_3 (\lambda(b), N(b))$
and $\exists b_0, b_1 \in a$ such that $n(b_0) = n(b_1)$ then
    $n'(b_0) := 1 + \max\{n' \mid (n', \ell', N') \in M(b_0)\}$;
    $M'(b_0) := M(b_0) \cup \{(n'(b_0), \lambda(b_0), N(b_0))\}$;
```

The fourth rule enables to update the local views of all the members of an association in one computation step. This rule can be applied to some association a only if the preceding ones cannot be applied to a. One can apply this rule to a if there does not exists any time-stamp o such that for each $b \in a$, $(o, \{n(b') \mid b' \in a\})$ belongs to $N(b)$. When the rule is applied, a new time-stamp o' is generated and $(o', \{n(b') \mid b' \in a\})$ is added to $N(b)$ for each $b \in a$. For the same reasons as in Section 4.1, each time we add an element (S', o') in $N(b)$, we remove all the elements of $N(b)$ that are smaller than (S', o') for $\prec_1$.

```
$\mathcal{R}_4$:
if $\forall b, b' \in a, M(b) = M(b')$ and $n(b) \neq n(b')$
and $\forall b \in a, n(b) \neq 0$ and $\forall (n(b), \ell, N) \in M(b), (\ell, N) \preceq_3 (\lambda(b), N(b))$
and $\nexists o$ such that $\forall b \in a, (\{n(b') \mid b' \in a\}, o) \in N(b)$ then
    $o' := 1 + \max\{o \mid \exists b \in a, \exists (S, o) \in N(b)\}$;
    $S' := \{n(b) \mid b \in a\}$;
    $\forall b \in a, N'(b) := N(b) \setminus \{(S, o) \mid (S, o) \prec_1 (S', o')\} \cup \{(S', o')\}$;
    $\forall b \in a, M'(b) := M(b) \cup \{(n(b'), \lambda(b'), N'(b')) \mid b' \in a\}$;
```

Properties. The algorithm we described has the same interesting properties as the ones described in Sections 3 and 4.1. Finally, we have:

Theorem 3. *For every communication structure* $\mathbf{C}$*, there exists a naming algorithm for* $\mathbf{C}$ *using unlabelled negotiations if and only if* $\mathbf{C}$ *is pseudo-covering-minimal.*

Again, if the individuals initially know $|B(C)|$, then the termination detection of the algorithm is possible in a pseudo-covering-minimal communication structure: once an individual gets the number $|B(C)|$, it knows that each individual has a unique number that will not change any more.

5 Final Remarks

The homomorphisms we introduced generalize locally constrained graph homomorphisms. These graph homomorphisms have already been studied in the literature [6,19] and one can wonder how the combinatorial properties satisfied by graph homomorphisms can be generalized to homomorphisms of communication structures. Locally constrained graph homomorphisms have also been studied from the complexity point of view [13,15]. In particular, it has been shown in [12] that it is co-NP-complete to decide whether a graph admits a naming algorithm in the models studied in [8,9,10,17]. An interesting corollary of this result is the following : The problems that ask whether a given communication structure $\mathbf{C}$ admits a naming algorithm using labelled negotiations, cellular labelled negotiations, unlabelled negotiations, cellular unlabelled negotiations respectively, are co-NP-complete.

References

1. Angluin, D.: Local and global properties in networks of processors. In: Proceedings of the 12th Symposium on Theory of Computing, STOC 1980, pp. 82–93 (1980)
2. Angluin, D., Aspnes, J., Diamadi, Z., Fisher, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Proc. of the 23th Symposium on Principles of Distributed Computing, pp. 290–299 (2004)
3. Angluin, D., Aspnes, J., Eisenstat, D.: Stably computable predicates are semilinear. In: Proc. of the 25th Symposium on Principles of Distributed Computing (2006)
4. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: On the power of anonymous one-way communication. In: Proc. 9th conf. on Principles of Distributed Computing, pp. 307–318 (2005)

5. Boldi, P., Codenotti, B., Gemmell, P., Shammah, S., Simon, J., Vigna, S.: Symmetry breaking in anonymous networks: Characterizations. In: Proc. 4th Israeli Symposium on Theory of Computing and Systems, pp. 16–26. IEEE Press, Los Alamitos (1996)
6. Boldi, P., Vigna, S.: Fibrations of graphs. Discrete Math. 243, 21–66 (2002)
7. Boldi, P., Vigna, S.: An effective characterization of computability in anonymous networks. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 33–47. Springer, Heidelberg (2001)
8. Chalopin, J.: Election and local computations on closed unlabelled edges. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 81–90. Springer, Heidelberg (2005)
9. Chalopin, J., Métivier, Y.: Election and local computations on edges. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 90–104. Springer, Heidelberg (2004)
10. Chalopin, J., Métivier, Y., Zielonka, W.: Election, naming and cellular edge local computations. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 242–256. Springer, Heidelberg (2004)
11. Chalopin, J., Métivier, Y., Zielonka, W.: Local computations in graphs: the case of cellular edge local computations. Fundamenta Informaticae 74(1), 85–114 (2006)
12. Chalopin, J., Paulusma, D.: Graphs labelings derived from models in distributed computing (submitted)
13. Fiala, J., Paulusma, D.: A complete complexity classification of the role assignement problem. Theoretical computer science (to appear)
14. Godard, E., Métivier, Y., Muscholl, A.: Characterization of Classes of Graphs Recognizable by Local Computations. Theory of Computing Systems 37(2), 249–293 (2004)
15. Kratochvil, J., Proskurowski, A., Telle, J.A.: Complexity of graph covering problems. Nordic Journal of Computing 5, 173–195 (1998)
16. Mazurkiewicz, A.: Distributed enumeration. Inf. Processing Letters 61(5), 233–239 (1997)
17. Mazurkiewicz, A.: Bilateral ranking negotiations. Fundamenta Informaticae 60, 1–16 (2004)
18. Mazurkiewicz, A.: Multilateral ranking negotiations. Fundamenta Informaticae 63, 241–258 (2004)
19. Reidemeister, K.: Einführung in die Kombinatorische Topologie. Vieweg, Brunswick (1932)
20. Yamashita, M., Kameda, T.: Computing on anonymous networks: Part i - characterizing the solvable cases. IEEE Transactions on parallel and distributed systems 7(1), 69–89 (1996)
21. Yamashita, M., Kameda, T.: Computing on anonymous networks: Part ii - decision and membership problems. IEEE Transactions on parallel and distributed systems 7(1), 90–96 (1996)

Abstracting Complex Data Structures by Hyperedge Replacement

Stefan Rieger and Thomas Noll

RWTH Aachen University
Software Modeling and Verification Group
52056 Aachen, Germany
{rieger,noll}@cs.rwth-aachen.de

Abstract. We present a novel application of hyperedge replacement grammars, showing that they can serve as an intuitive formalism for abstractly modeling dynamic data structures. The aim of our framework is to extend finite-state verification techniques to handle pointer-manipulating programs operating on complex dynamic data structures that are potentially unbounded in their size. The idea is to represent both abstraction mappings on user-defined dynamic data structures and the (abstract) semantics of pointer-manipulating operations using graph grammars, supporting a smooth integration of the two aspects. We demonstrate how our framework can be employed for analysis and verification purposes, e.g., to prove that a procedure preserves structural invariants of the heap.

1 Introduction

Techniques for analyzing pointer programs are highly desirable. Programming with pointers is error-prone with potential pitfalls such as dereferencing null pointers and the emergence of memory leaks. When considering pointer programs we face the problem of infinite state spaces arising due to the unboundedness of the heap. Thus for employing verification methods like model checking in this scenario, abstraction techniques are indispensable.

We present an approach to abstracting state spaces of pointer programs operating on linked data structures of arbitrary size and shape. In our framework, states of the heap are modeled by hypergraphs, and both pointer-manipulating operations and abstraction mappings are represented by hypergraph transformations. More concretely we employ hyperedge replacement grammars to specify data structures and their abstractions. The essential idea is to use the replacement operations which are induced by the grammar rules in two directions. By a *backward* application of some rule, a subgraph of the heap can be condensed into a single nonterminal edge, thus obtaining an *abstraction* of the heap. By applying rules in *forward* direction, certain parts of the heap which have been abstracted before can be *concretized* again. Later we will see that this operation will be required in order to avoid the necessity for defining the effect of pointer-manipulating operations on abstracted parts of the heap.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 69–83, 2008.

Due to the generality of this framework, the use of hyperedge replacement grammars does not always ensure the boundedness of the resulting abstract heaps and, thus, the finiteness of the corresponding transition systems. The formalism can therefore be extended by introducing an additional parameter which allows to limit the size of the heaps. We sketch this aspect in Sct. 4.3; it is not required for understanding the actual abstraction framework.

Altogether we obtain an expressive and highly parametrized framework which allows to specify complex dynamic data structures and their abstractions in an intuitive way. Our approach is illustrated by considering a simple programming language and an example program operating on a cyclic, doubly-linked list. Using our formalism we will be able to show that the program preserves the structure of the list, independent of its size.

2 Related Work

Related work on the topic of analyzing pointer-manipulating programs can be classified into the following (often overlapping) categories: *Shape analysis* is a static analysis technique that represents recursive data structures of unbounded size by finite structures, called "shape graphs", which are usually formalized by three-valued logical structures [6,23]. *Predicate abstraction* abstracts the state space of the program by evaluating it under a number of given predicates, obtaining a Boolean program which conservatively simulates all potential executions [3,9,19]. *Regular model checking* is a framework for unified verification of infinite-state systems based on automata theory. It represents states using words (trees) over a finite alphabet and sets of states using finite (tree) automata [7]. *Dataflow analysis* is a technique for gathering information about certain aspects of a program using its control flow graph. This approach is generally efficient but restricted to rather shallow properties of programs such as aliasing relations [17], points-to information [25], or pointer range analysis [24]. *Hoare-style approaches* extend first-order logic by reachability predicates over heap nodes [8,15]. *Separation logic* has been proposed as an extension to Hoare logic that permits local reasoning about linked structures, supporting features to support modular correctness proofs for pointer-manipulating programs [18,22].

Research in the field of *graph transformations* often concentrates on verifying and abstracting graph transformation systems, e.g. by employing so-called "Petri graphs" [4,5] or model checking state spaces generated by graph grammars [13]. We, however, *make use* of graph grammars *for* abstraction. Existing approaches with similar ideas essentially try to represent the shape of heap data structures by (abstract) graphs, and to implement statements of a programming language by graph transformation rules [20,21]. The framework presented in [1,2,10] is quite close to ours; the authors use graph reduction grammars for abstractly representing pointer structures. Their approach – which so far only handles shape safety – requires to specify an abstract transformation for each operation modifying a data structure. In contrast, we only require an abstraction specification; pointer operations do not need to be redefined in dependence of this abstraction since they are handled automatically. Another grammar-based approach to heap abstraction is presented in [14], however, it only supports tree data structures and cannot handle DAGs and general graphs as we do.

Thus our approach is unique in that it offers a new, descriptive way for specifying abstractions on arbitrary data structures. It supports dynamic memory allocation (leading to unbounded heap sizes) and destructive updates. In addition it is easily extendable to concurrent programs with dynamic thread creation along the lines of [16].

3 Hyperedge Replacement

For the realization of our framework we concentrate on hyperedge replacement grammars [11] as they provide sufficient expressive strength for our application but still share some of the nice properties of context-free string grammars. In the following we introduce some notations that will be useful in the specification of our framework.

Given a set S, $S^\star$ denotes the set of all finite sequences (strings) over S. For $s \in S^\star$ the length of s is denoted by $|s|$, the set of all elements of the sequence s is written $[s]$, and by $s(i)$ we denote the ith component of s. Given a tuple $t = (A, B, C, ...)$ we sometimes write t_A, t_B etc. for the components if their names are clear from the context.

The domain of a function f is denoted by $dom(f)$. For two functions f and g with $dom(f) \cap dom(g) = \emptyset$ we define $f \cup g$ by $(f \cup g)(x) = f(x)$ if $x \in dom(f)$ and $(f \cup g)(x) = g(x)$ if $x \in dom(g)$. For a set $S \subseteq dom(f)$ the function $f \restriction S$ is the restriction of f to S. Every $f : A \to B$ is implicitly defined on sets $f : 2^A \to 2^B$ and on sequences $f : A^\star \to B^\star$ by point-wise application. By $f[a/b]$ we denote the function update defined by $f[a/b](a) = b$ and $\forall c \neq a : f[a/b](c) = f(c)$. The identity function on a set S is $\mathbf{id}_S$.

3.1 Hypergraphs

Hyperedge replacement grammars operate on hypergraphs, which allow hyperedges connecting an arbitrary number of vertices. Let Σ be a finite ranked alphabet where $rk : \Sigma \to \mathbb{N}$ assigns to each symbol $a \in \Sigma$ its rank $rk(a)$. We partition Σ into a set of *nonterminals* $N_\Sigma \subseteq \Sigma$ and a set of *terminals* $T_\Sigma = \Sigma \setminus N_\Sigma$. We will use capital letters for nonterminals and lower case letters for terminal symbols. We assume that both the rk function and the partitioning are implicitly given with Σ.

Definition 3.1. *A* (labeled) hypergraph *over Σ is a tuple $H = (V, E, att, \ell, ext)$ where V is a set of vertices and E a set of edges, $att : E \to V^\star$ maps each edge to a sequence of attached vertices, $\ell : E \to \Sigma$ is an edge-labeling function, and $ext \in V^\star$ a sequence of pairwise distinct external vertices.*

We require that for all $e \in E$: $|att(e)| = rk(\ell(e))$. The set of all hypergraphs over Σ is denoted by HGraph_Σ*. Furthermore we use the notations $E(v) := \{e \in E \mid v \in [att(e)]\}$ for the edges attached to a vertex and $|H| := |V| + |E|$ for the size of a hypergraph.*

Thus edges are separate objects in the graph and are mapped to sequences of attached vertices. The external vertices play an important role in graph transformation steps. We will usually not distinguish between isomorphic copies of a hypergraph. Two hypergraphs H_1 and H_2 are isomorphic, written $H_1 \cong H_2$, if they are identical modulo renaming of vertices and edges.

To facilitate notation later on we introduce the notion of a *handle* which is a hypergraph consisting of only one hyperedge attached to its external nodes.

Definition 3.2. *Given $X \in \Sigma$ with $rk(X) = n$, the X-*handle *is the hypergraph* $X^\bullet = (\{v_1, ..., v_n\}, \{e\}, \{e \mapsto v_1...v_n\}, \{e \mapsto X\}, v_1...v_n) \in \mathrm{HGraph}_\Sigma$.

3.2 Hyperedge Replacement Grammars

Now we are ready to define hyperedge replacement grammars. They share some pleasant properties with context-free string grammars such as confluence and associativity [11], which is not the case for most other types of graph grammars.

Definition 3.3. *A* hyperedge replacement grammar *(HRG) over Σ is a set G of* (production) rules*, each of the form $X \to H$ with $X \in N_\Sigma$ and $H \in \mathrm{HGraph}_\Sigma$ where $|ext_H| = rk(X)$.*

We denote the set of hyperedge replacement grammars over Σ by HRG_Σ *and assume that there are no isomorphic production rules, i.e., rules with identical left-hand and isomorphic right-hand sides.*

Fig. 1 depicts a grammar generating doubly-linked lists. The only nonterminal is the symbol D, and the letters n and p are respectively used to model the next- and previous-pointers. In the (rule-)graphs the rank of all symbols is two. The small numbers close to the connecting edges represent the order of the connected vertices and the vertices shaded in gray are the external nodes. Rules p_1 and p_2 are "redundant"; this is necessary for concretization to work (see Sct. 4.1).

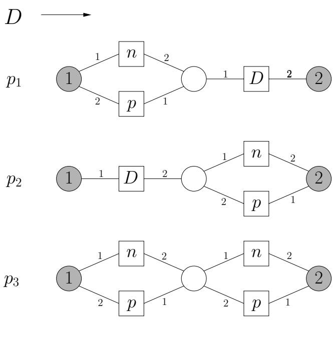


Fig. 1. HRG for Doubly-Linked Lists

The rules specify for each nonterminal X a replacement hypergraph H that will replace (the hyperedge labeled by) X when the rule $X \to H$ is applied. When a hyperedge e labeled by a nonterminal is replaced, the external vertices of the replacement graph are matched with the attached vertices of e. Thus a hyperedge replacement represents a *local* change in the graph structure.

Definition 3.4. *Let $G \in \mathrm{HRG}_\Sigma$, $H \in \mathrm{HGraph}_\Sigma$, $p = X \to K \in G$ and $e \in E_H$ such that $\ell(e) = X$. Let $E_{H-e} := E_H \setminus \{e\}$. We assume w.l.o.g. that $V_H \cap V_K = E_H \cap E_K = \emptyset$ (otherwise the components in K are renamed). The* substitution of e by K*, $J \in \mathrm{HGraph}_\Sigma$, is defined by*

$$V_J = V_H \cup (V_K \setminus [ext_K]) \qquad E_J = E_{H-e} \cup E_K$$
$$\ell_J = (\ell_H \restriction E_{H-e}) \cup \ell_K \qquad ext_J = ext_H$$
$$att_J = mod \circ ((att_H \restriction E_{H-e}) \cup att_K)$$
$$\text{with } mod = \mathbf{id}_{V_J}[ext_K(1)/att_H(e)(1), ..., ext_K(rk(e))/att_H(e)(rk(e))]$$

We write $H \Longrightarrow_G J$ if there exist e and $X \to K$ as above. The reflexive-transitive closure and the inverse of $\Longrightarrow_G$ are denoted by $\Longrightarrow_G^\star$ and $\Longrightarrow_G^{-1}$, respectively.

The *language* of a grammar $G \in \mathrm{HRG}_\Sigma$ consists of all terminal graphs (that is, graphs that have only edges with terminal labels) that can be derived from a given starting graph $H \in \mathrm{HGraph}_\Sigma$, i.e., $L(G, H) = \{K \in \mathrm{HGraph}_{T_\Sigma} \mid H \Longrightarrow_G^\star K\}$.

For actual applications it is important to not have nonterminals in the grammar from which no terminal graph is derivable ($\forall X \in N_\Sigma : L(G, X^\bullet) \neq \emptyset$). We call such grammars *productive*. Any HRG can be transformed into an equivalent productive grammar if its language is non-empty.

We are interested in (heap) graph abstractions for analysis and verification, which need to be effectively computable. Since, as we will see later, abstractions are obtained by backward applications of rules, the termination of the abstraction procedure can be ensured by requiring all rules in a HRGs to be *increasing*, meaning that the replacement graph (if it contains nonterminals) is "larger" than the handle of the respective nonterminal.

Definition 3.5. *A grammar $G \in \mathrm{HRG}_\Sigma$ is* increasing *iff for all $X \to H \in G$ such that $\ell_H(E_H) \cap N_\Sigma \neq \emptyset$ we have $|X^\bullet| < |H|$.*

Theorem 3.6. *Let $G \in \mathrm{HRG}_\Sigma$ be increasing and $H \in \mathrm{HGraph}_\Sigma$. Then the set $\{K \in \mathrm{HGraph}_\Sigma \mid K \Longrightarrow_G^+ H\}$ is finite.*

Proof. The increasingness of G implies that for any two hypergraphs H_1 and H_2 with $H_1 \Longrightarrow_G^+ H_2$ we have $|H_1| < |H_2|$ or $H_2 \in \mathrm{HGraph}_{T_\Sigma}$. Thus for every finite hypergraph H there is a bound $n \in \mathbb{N}$ such that all derivations yielding H are of length $\leq n$ (no "loops" are possible), which proves our claim. □

As we will see in Sct. 4 the result of Thm. 3.6 is essential for our abstraction technique since it allows us to compute a minimal abstract heap representation. Note that the HRG in Fig. 1 is increasing.

4 Abstraction of Heap States

For using HRGs as an abstraction mechanism for pointer-manipulating programs we have to represent heaps as hypergraphs. This is done by introducing two types of terminal edges: edges labeled with program variables (which we include in the terminal alphabet) are of rank one, edges of rank two – labeled with record selectors – are representing pointers in the heap. Formally, we let $T_\Sigma = \mathit{Var}_\Sigma \uplus \mathit{Sel}_\Sigma$ where $\mathit{rk}(\mathit{Var}_\Sigma) = \{1\}$ and $\mathit{rk}(\mathit{Sel}_\Sigma) = \{2\}$. Finally there are nonterminal edges of arbitrary rank that are used in the abstraction and that stand for (a set of) entire subgraphs.

Definition 4.1. *A* heap configuration *over an alphabet Σ is a hypergraph $H \in \mathrm{HGraph}_\Sigma$ such that $\forall x \in \mathit{Var}_\Sigma : |\{e \in E_H \mid \ell_H(e) = x\}| \leq 1$ and $\mathit{ext}_H = \varepsilon$ where Var_Σ and Sel_Σ satisfy the constraints mentioned above. We denote the set of all heap configurations over Σ – including a special configuration H_{err}*

which is reached if pointer errors occur (i.e., dereferencing of a null pointer) – by $\boldsymbol{HC}_\Sigma$. A heap configuration H is called concrete *if $H \in \boldsymbol{HC}_{T_\Sigma}$. We* identify *two heap configurations H and H' if $H \cong H'$.*

Additional notation. For $H = (V, E, att, \ell, \varepsilon) \in \mathbf{HC}_\Sigma$, $x \in Var_\Sigma$ and $v \in V$ we write $x \hookrightarrow_H v$ to denote that $\exists e \in E : \ell(e) = x \wedge att(e) = v$. Writing $x \hookrightarrow_H nil$ is equivalent to $\nexists v \in V : x \hookrightarrow_H v$. (That is, variables pointing to *nil* are represented by omitting the corresponding edge.) For $v, w \in V$ and $s \in Sel_\Sigma$ we write $v \overset{s}{\hookrightarrow}_H w$ to indicate that $\exists e \in E : \ell(e) = s \wedge att(e) = vw$.

4.1 Abstraction and Concretization

When modeling the semantics of assignments it is convenient to assume that those edges which are connected to vertices that are referenced by variables, are all labeled by terminal symbols. If there is an edge e violating this property it is called a *violation point*. For all those edges we record the indices of the attached vertices that are the targets of program variables.

Definition 4.2. *Let $H \in \boldsymbol{HC}_\Sigma$. The set of* violation points $\mathcal{VP}(H) \subseteq E_H \times \mathbb{N}$ *is given by:*

$$\begin{aligned} &(e, i) \in \mathcal{VP}(H) \\ \Leftrightarrow\ &\ell(e) \in N_\Sigma \wedge (\exists x \in Var_\Sigma, v \in V_H : x \hookrightarrow_H v \wedge v = att_H(e)(i)) \end{aligned}$$

If no violation points exist a configuration is called *admissible*. As mentioned in the introduction, this will avoid the necessity for defining the effect of pointer-manipulating operations on abstracted parts of the heap.

Definition 4.3. *The set of all admissible heap configurations is given by $\boldsymbol{aHC}_\Sigma = \{H \in \boldsymbol{HC}_\Sigma \mid \mathcal{VP}(H) = \emptyset\}$.*

We use forward derivations to restore admissibility of a configuration. This *partial concretization* (see Def. 4.7), however, raises additional requirements for the production rules. To see this, let us again consider the example from Fig. 1. Here we could omit the rule p_2 and would still obtain a grammar that suffices to generate the language of all doubly-linked lists, thus p_2 is redundant. Omitting it, though, would lead to problems when concretizing since there might be an unbounded derivation sequence starting from a nonterminal until finally one terminal symbol is generated at its place and thus we have infinitely many concretizations. To circumvent this problem we considered to use *Greibach Normal Form* for hyperedge replacement grammars [12] (a generalization of Double Greibach Normal Form of context-free string grammars). For a HRG in Greibach Normal Form a single rule application suffices for concretization. Unfortunately this idea proved to be impractical since already for simple grammars the Greibach Normal Form is often huge [12]. Thus we decided to introduce a class of HRGs that we call *heap abstraction grammars* whose definition is admittedly more complicated.

Definition 4.4. *An increasing and productive graph grammar $G \in \mathrm{HRG}_\Sigma$ is a* heap abstraction grammar *if*

1. *$\ell_H(E_H) \cap Var_\Sigma = \emptyset$ for all $X \to H \in G$ and,*
2. *for every $X \in N$ with $rk(X) = k$ there exist $G_1^X, ..., G_k^X \subseteq G$ such that*
   - $\bigcup_{i=1}^k G_i^X = G$,
   - $L(G_i^X, X^\bullet) = L(G, X^\bullet)$ *for all* $1 \leq i \leq k$, *and*
   - $\ell_H(E_H(ext_H(i))) \subseteq T_\Sigma$ *for all* $X \to H \in G_i^X$.

The first condition disallows variables (from which we do not abstract) as edge labels. The second condition enforces a kind of symmetry for rules that have nonterminal edges connected to external vertices. The idea is to use only rules from G_i^X when concretizing a nonterminal edge from the ith attached vertex (i.e. to this vertex a variable is attached). Since we have subgrammars for all i we can concretize from any direction while avoiding "loops". Note that the G_i^X are usually *not* disjoint.

In Fig. 1 rules p_1 and p_2 fulfill the conditions of Def. 4.4; the two rules together enable concretization from either "side" of a nonterminal edge while the generated graph language is retained. Thus, when concretizing a D-edge it suffices to apply *either* p_1 *or* p_3 (if p_1 concretizes "from the right-hand side"). For this example we have $G_1^D = \{p_1, p_3\}$ and $G_2^D = \{p_2, p_3\}$.

Based on the concepts presented so far we can formalize the notion of an abstraction function $\mathfrak{A}_G$, called *heap abstractor*. According to the principle that abstraction is performed by backward application of rules, $\mathfrak{A}_G$ returns some irreducible, admissible successor of the current heap configuration with respect to the inverse derivation relation $\Longrightarrow_G^{-1}$.

Definition 4.5. *Let $G \in \mathrm{HRG}_\Sigma$ be a heap abstraction grammar. A* heap abstractor *over G is a function $\mathfrak{A}_G : \boldsymbol{aHC}_\Sigma \to \boldsymbol{aHC}_\Sigma$ such that*

$$\mathfrak{A}_G(H) \in \{K \in \boldsymbol{aHC}_\Sigma \mid K \Longrightarrow_G^\star H \text{ s.t. } \nexists J \in \boldsymbol{aHC}_\Sigma \text{ with } J \Longrightarrow_G K\}.$$

Note that heap abstraction mappings are not uniquely defined. This is only the case if $\Longrightarrow_G^{-1}$ is confluent which, together with its well-foundedness that is implied by the increasingness of the HRG according to Thm. 3.6, yields unique normal forms. In general the abstractor should minimize the size of a heap configuration. Also note that this definition immediately implies the *correctness* of our abstraction in the sense that every concrete heap configuration can be re-generated from its abstraction:

Corollary 4.6. *Under the above assumptions, $H \in L(G, \mathfrak{A}_G(H))$ for every $H \in \boldsymbol{aHC}_{T_\Sigma}$.*

Since heap objects that are not reachable from program variables play no role in program semantics we delete them using a *garbage collector*. When computing the reachability of vertices we handle hyperedges of rank greater than two, i.e. nonterminal edges, conservatively as undirected edges connecting all attached vertices. We here omit the exact definition of the garbage collector and denote the mapping by $\mathrm{GC} : \mathbf{HC}_\Sigma \to \mathbf{HC}_\Sigma$.

As already mentioned before, in addition to abstraction also concretization is necessary to restore admissibility. The essential point is that we employ *partial* concretization by applying grammar rules in forward direction. Here derivation stops as soon as the resulting heap configuration is admissible, in order to minimize the degree of concretization. Thus the properties of heap abstraction grammars as required in Def. 4.4 guarantee that only a finite number of configurations can be obtained.

Definition 4.7. *Let* $G \in \mathrm{HRG}_\Sigma$ *be a heap abstraction grammar and let the* $G_i^X \subseteq G$ *be given as in Def. 4.4. The* heap concretizer, $\mathfrak{C}_G : \boldsymbol{HC}_\Sigma \to 2^{\boldsymbol{aHC}_\Sigma}$, *is then defined as follows:*

$$\mathfrak{C}_G(H) = \begin{cases} \mathfrak{C}_G(\{K \in \boldsymbol{HC}_\Sigma \mid H \Longrightarrow_{G_i^X} K\}) & \textit{if } \exists (e,i) \in \mathcal{V}P(H) \wedge \ell_H(e) = X \\ \{H\} & \textit{if } H \in \boldsymbol{aHC}_\Sigma \end{cases}$$

Note that, in contrast to a heap abstractor (Def. 4.5), the heap concretizer is uniquely defined as it yields *all* reachable (first) admissible configurations.

4.2 Pointer Programs and Their Semantics

Previously we already introduced the memory model, abstraction and concretization techniques but we still did not consider any programming language. In the following we will do this; the language itself is kept minimal to reduce the formal effort in the specification of the semantics, though it is sufficient to model most standard concepts in pointer programs.

Definition 4.8. *A* pointer program π *is a sequence of statements* $s_1; \ldots; s_r$ *with* $s_i \in \mathrm{CMD}$ *where* CMD *is the set of the following commands:*

PExp := PExp	*(pointer assignment)*	**if** BExp **goto** n	*(conditional jump)*
new(PExp)	*(object creation)*	**goto** n	*(unconditional jump)*

Furthermore we have:

$$\begin{aligned} \mathrm{PExp} &::= \; nil \mid x \;(x \in Var_\Sigma) \mid x.s \;(s \in Sel_\Sigma) \\ \mathrm{BExp} &::= \; \mathrm{PExp} = \mathrm{PExp} \mid \mathrm{BExp} \wedge \mathrm{BExp} \mid \neg\mathrm{BExp} \end{aligned}$$

Please note that for simplicity the programming language does not support arbitrary dereferencing depths. This is no restriction since this feature can be emulated by multiple assignments. An object deletion command is omitted since a *nil*-assignment with a subsequent garbage collection has the same effect. The programming language can be extended with unbounded threads and atomic regions using the concepts we introduce in [16].

```
delete() {
 1 if x = nil goto 10;
 2 if x = x.n goto 9;
 3 y := x.n;
 4 x := x.p;
 5 x.n := y;
 6 y.p := x;
 7 y := nil;
 8 goto 10;
 9 x := nil;
10 }
```

Fig. 2. Delete from an arbitrary Cyclic Doubly-Linked List

In Fig. 2 an example program is shown that deletes an element from a cyclic doubly-linked list. The selectors n and p respectively model the next- and previous-pointers. The variable x is assumed to point to some object in the structure while y is used as an auxiliary variable.

In the pointer semantics we use the special value err to denote that a pointer error (e.g. *nil* dereference) occurred.

Definition 4.9. *For* $H = (V, E, att, \ell, ext) \in \boldsymbol{aHC}_\Sigma$ *the* semantics of pointer expressions $\mathcal{P}_H[\![\cdot\,]\!] : \mathrm{PExp} \to V \cup \{nil, \mathrm{err}\}$ *is defined as follows:*

$$\begin{aligned}
\mathcal{P}_H[\![nil\,]\!] &= nil \\
\mathcal{P}_H[\![x\,]\!] &= v && \textit{if}\ \ x \hookrightarrow_H v \\
\mathcal{P}_H[\![x\,]\!] &= nil && \textit{if}\ \ x \hookrightarrow_H nil \\
\mathcal{P}_H[\![x.s\,]\!] &= v && \textit{if}\ \ \mathcal{P}_H[\![x\,]\!] \neq nil\ \wedge\ \mathcal{P}_H[\![x\,]\!] \overset{s}{\hookrightarrow}_H v \\
\mathcal{P}_H[\![x.s\,]\!] &= nil && \textit{if}\ \ \mathcal{P}_H[\![x\,]\!] \neq nil\ \wedge\ \nexists v \in V : \mathcal{P}_H[\![x\,]\!] \overset{s}{\hookrightarrow}_H v \\
\mathcal{P}_H[\![x.s\,]\!] &= \mathrm{err} && \textit{if}\ \ \mathcal{P}_H[\![x\,]\!] = nil
\end{aligned}$$

The semantics of Boolean expressions $\mathcal{B}_H[\![\cdot\,]\!] : \mathrm{BExp} \to \mathbb{B} \cup \{\mathrm{err}\}$ is as usual but strict, i.e. if one of the arguments (pointer or Boolean expression) yields err the result is also err. Next we can formulate the semantics of assignments and **new**-statements, still without considering the additional steps, e.g. concretization, garbage collection and abstraction.

Definition 4.10. *Let* $H \in \boldsymbol{aHC}_\Sigma$ *and* $\alpha, \alpha' \in \mathrm{PExp}$*. Then we define* $H[\alpha/\alpha'] \in \boldsymbol{HC}_\Sigma$ *as follows:*

- $H[x.s/\alpha'] = H_{\mathrm{err}}$ *if* $x \hookrightarrow_H nil$ *or* $\mathcal{P}_H[\![\alpha'\,]\!] = \mathrm{err}$
- *Otherwise we distinguish the cases given in Fig. 3 where the modifications are represented by graph transformations. Here the triangle vertex is assumed to be* $\mathcal{P}_H[\![\alpha'\,]\!]$*. Thus there is more than one possible result. Graph objects not shown in the source or target graphs remain unchanged.*

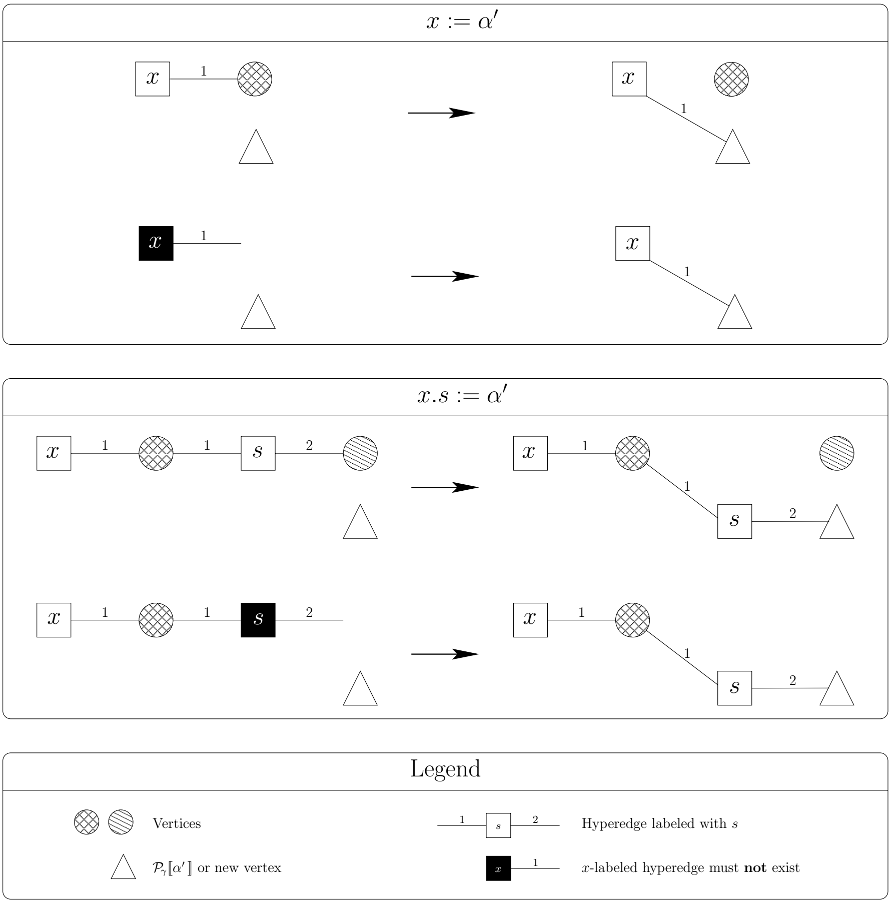
$H[\alpha/\mathrm{new}] \subseteq \boldsymbol{aHC}_\Sigma$ *is given similarly by:*

- $H[x.s/\mathrm{new}] = H_{\mathrm{err}}$ *if* $x \hookrightarrow_H nil$
- *Otherwise the cases given in Fig. 3 apply where the triangle vertex is a new vertex inserted into* V_H *before applying the transformations.*

Combining all the concepts introduced before we obtain the abstract heap semantics that captures the effect of the commands on the heap. For the more involved commands (assignment, **new**) the following steps are necessary:

1. execution of the actual assignment (nondeterministic)
2. garbage collection
3. partial concretization (nondeterministic)
4. re-abstraction

A fifth step may become necessary if the abstraction grammar is not suitable for the data structures occurring in the program. We then need to “artificially” bound the heap configuration by collapsing vertices. This is done by a so-called



$x := \alpha'$
x
1
$x.s := \alpha'$
s
2
Legend
Vertices
Hyperedge labeled with s
$\mathcal{P}_\gamma[\![\alpha']\!]$ or new vertex
x-labeled hyperedge must **not** exist

Fig. 3. Assignment / Creating Objects

heap compactor which we only briefly sketch in Sct. 4.3 in favor of concentrating on the actual abstraction.

Finally we can introduce an abstract "transition relation" that captures the effect of the statements in our programming language.

Definition 4.11. *Let $G \in \mathrm{HRG}_\Sigma$ be a heap abstraction grammar, and $\mathfrak{A}_G : \boldsymbol{aHC}_\Sigma \to \boldsymbol{aHC}_\Sigma$ a heap abstractor. The* abstract heap transformation relation *$\overset{h}{\Rightarrow} \subseteq (\boldsymbol{aHC}_\Sigma \times \mathrm{CMD} \times \boldsymbol{aHC}_\Sigma)$ is given as follows for $H \in \boldsymbol{aHC}_\Sigma$, $H \neq H_{\mathrm{err}}$ (we omit the* **if** *and* **goto** *statements since their semantics is straightforward and has no effect on the heap):*

$$\frac{K \in \mathfrak{A}_G(\mathfrak{C}_G(\mathrm{GC}(H[\alpha/\alpha'])))}{H, \boldsymbol{\alpha := \alpha'} \overset{h}{\Rightarrow} K} \qquad \frac{K \in \mathfrak{A}_G(\mathfrak{C}_G(\mathrm{GC}(H[\alpha/\mathrm{new}])))}{H, \mathbf{new}(\boldsymbol{\alpha}) \overset{h}{\Rightarrow} K}$$

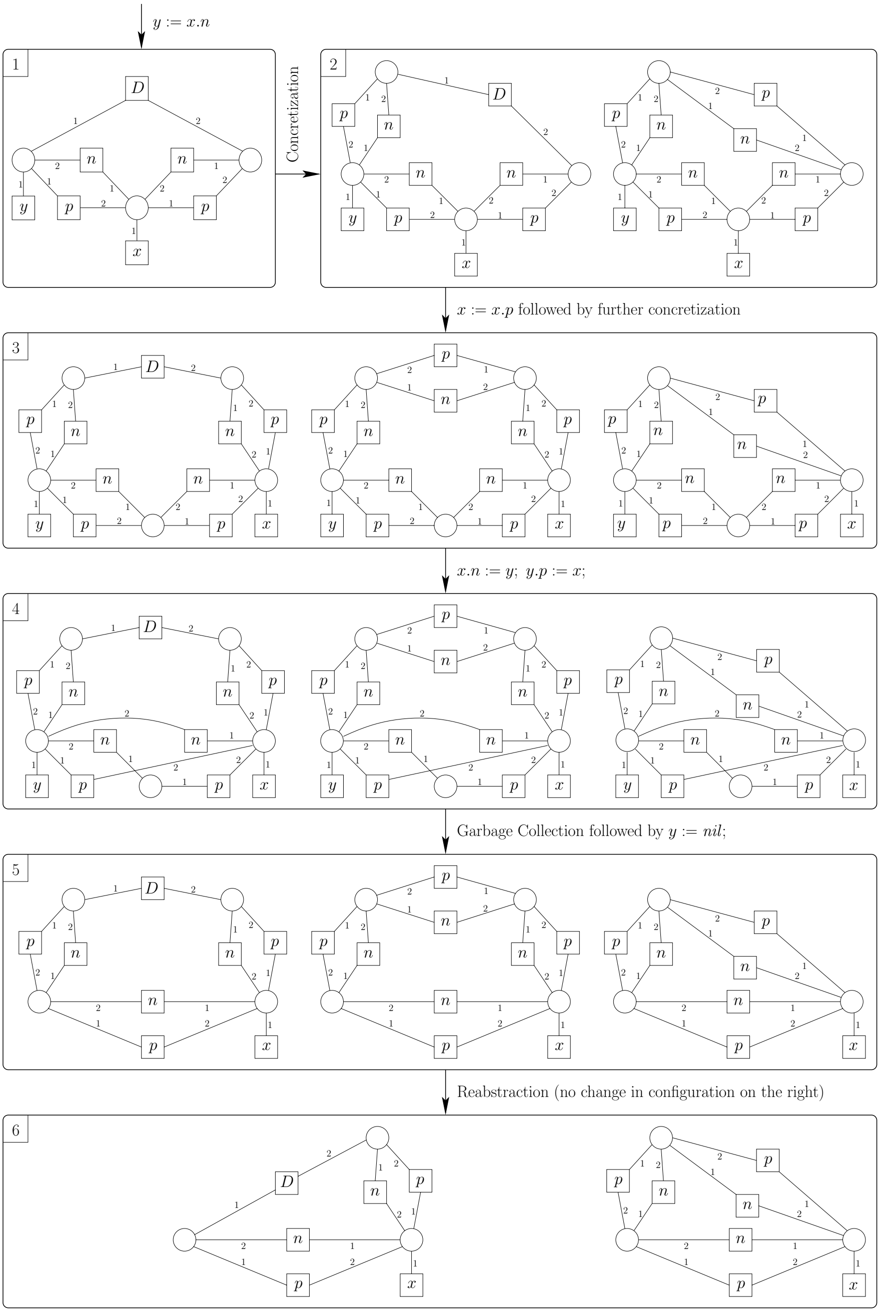

$y := x.n$
1
Concretization
2
$x := x.p$ followed by further concretization
3
$x.n := y;\ y.p := x;$
4
Garbage Collection followed by $y := nil$;
5
Reabstraction (no change in configuration on the right)
6

Fig. 4. Delete on Abstract Graph

Figure 4 shows the semantics of the `delete()` operation from Fig. 2 based on the HRG for doubly-linked lists (Fig. 1). We start with a configuration with three nodes and one D-edge, that is, it represents arbitrary large cycles with at least four nodes (see the left heap in subfigure 6). The **if**-statements have no effect in this case. After the first assignment $y := x.n$, we obtain the configuration depicted in subfigure 1. Here the variable y is too close to the D-edge and thus the configuration is not admissible. We have to concretize it using the rules p_1 and p_3 (from G_1^D), obtaining two resulting configurations where one only contains terminal edges (application of p_3). Rule p_2 is not applicable since it would not produce a terminal edge on the left-hand side. Note that this does not violate correctness, since the HRG from Fig. 1 is a heap abstraction grammar.

The next assignment $x := x.p$ makes a further concretization necessary since the variable x is now too close to the D-edge. Now rules p_2 and p_3 (from G_2^D) are applied to the left-hand graph from subfigure 2, and we obtain two results one of which is concrete. The third graph is resulting from the right-hand graph in subfigure 2.

The two assignments $x.n := y$ and $y.p := x$ exchange the next- and previous-pointers in the subgraph between x and y; the result is shown in subfigure 4. The following garbage collection (the lower vertex is unreachable) and the *nil*-assignment to y yield the states visualized in subfigure 5. A re-abstraction applying rules p_1 to the left-hand and p_3 to the middle graph leads to the same result. The left-hand heap in subfigure 6 is again the initial graph, and the right-hand is the concrete one that results from deleting one node in the (minimal) cyclic list with four elements. (It is the same as in subfigure 5 since no rule is applicable).

Hence we just proved that `delete()` preserves the structure of cyclic doubly-linked lists of arbitrary size. The heap compactor is not required for our example since all abstract configurations have less than five nodes. For an insert-operation one could easily give a similar proof and would obtain even less configurations (due to the lower degree of nondeterminism).

The correctness proof for our abstraction technique requires to first define the transformation relation on concrete heaps, which is straightforward, and then to relate concrete and abstract computations in the following way. Whenever a concrete heap $H \in \mathbf{aHC}_{T_\Sigma}$ is transformed into $H' \in \mathbf{aHC}_{T_\Sigma}$ and its abstraction $\mathfrak{A}_G(H) \in \mathbf{aHC}_\Sigma$ is (abstractly) transformed into $H'' \in \mathbf{aHC}_\Sigma$, then $H' \in L(G, H'')$. That is, every concrete computation has its abstract counterpart, and thus our abstraction constitutes a safe approximation of the system.

4.3 Enforcing Finiteness

This section is optional reading and gives a short overview of the *heap compactor* which may be necessary to enforce a finite state space in certain situations where unsuitable abstraction grammars are used. The cost is an inherent loss in precision. The compactor works by merging vertices to form a special *sink* vertex if the configuration exceeds a size bound given a priori. This vertex that can represent *arbitrary* subgraphs has to be considered in the semantics and yields additional nondeterminism.

Definition 4.12. *A* heap compactor *is a function* $\kappa : \boldsymbol{aHC}_\Sigma \times \mathbb{N} \to \boldsymbol{aHC}_\Sigma$*. For* $H, K \in \boldsymbol{aHC}_\Sigma$ *and* $k \in \mathbb{N}$*,* $\kappa(H,k) = H$ *if* $|V_H| \leq k$ *and otherwise* $\kappa(H,k) = K$ *such that:*

- $|V_K| = k$
- $sink \in V_K \subset (V_H \cup \{sink\})$
- $E_K = \{e \in E_H \mid [att_H(e)] \setminus \{sink\} \neq \emptyset \ \vee \ \ell(e) \in Var_\Sigma\}$
- $att_K = mod \circ att_H$ *where*
  $$mod : V_H \to V_K,\ mod(v) = \begin{cases} v & \text{if } v \in V_K \\ sink & \text{otherwise} \end{cases}$$
- $\ell_K = \ell_H \restriction E_K$

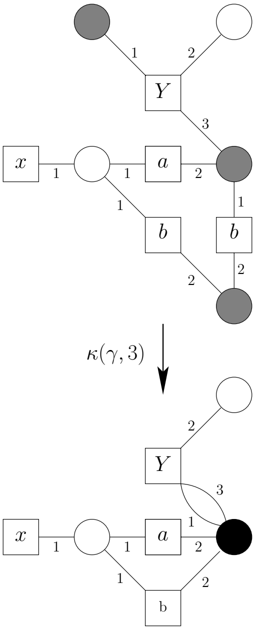


Fig. 5. Heap Compactor (Example)

Thus the heap compactor only modifies a configuration if the abstractor (which is to be executed beforehand) does not "compress" it enough. Its purpose is to guarantee finiteness of the semantics. If the constant k is large enough, small inconsistencies as they occur often temporarily when manipulating data structures do not result in a loss of precision since the compactor does not modify configurations with at most k nodes.

In Fig. 5 the compactor is visualized by an example. The vertices shaded in gray are merged to form the *sink* vertex visualized in black. For the actual implementation of the heap compactor a heuristics that merges connected vertices (otherwise potential dependencies between independent parts of the graph are created) and those that are distant from the program variables seems promising. The latter will reduce the probability that the *sink* vertex plays a role in the program semantics.

Modifying the expression and assignment semantics is mostly straightforward and is therefore omitted here. One essentially needs to consider additional nondeterministic cases which are introduced by the sink vertex. This leads for example to a multi-valued Boolean semantics: if an expression refers to the sink vertex we cannot decide anymore whether it is true or not and thus have to consider both cases. For assignments we get a nondeterministic step if a variable references the sink vertex.

5 Conclusions and Future Work

We have presented a framework for the analysis of pointer-manipulating programs operating on arbitrary dynamic data structures. The abstraction mechanism is parametrized via a hyperedge replacement graph grammar that models the data structure(s) used in the program. We showed how the abstract states can be transformed and how abstract state spaces can be generated. When employing a compactor our method ensures that these state spaces are always finite, even if the underlying data structure is outside of the specification. Smaller inconsistencies that naturally occur when manipulating data structures can be handled without loss of precision.

The programming language can be extended with concurrency, e.g. unbounded threads and atomic regions, without major changes [16]. This works essentially by modelling the control-flow semantics separately from the heap semantics by a Petri net and then combining both for state-space exploration. Hereby an orthogonal abstraction is applied on the control-flow part.

Currently we are working on an implementation of our framework. We are planning to introduce a logic to formulate verification properties and a model checking algorithm to verify those on a given program. Furthermore we will analyze how data structure definitions – as they occur in many programming languages – can be used for automatically generating an appropriate abstraction grammar.

References

1. Bakewell, A., Plump, D., Runciman, C.: Checking the shape safety of pointer manipulations. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 48–61. Springer, Heidelberg (2004)
2. Bakewell, A., Plump, D., Runciman, C.: Specifying pointer structures by graph reduction. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 30–44. Springer, Heidelberg (2004)
3. Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis by predicate abstraction. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 164–180. Springer, Heidelberg (2005)
4. Baldan, P., Corradini, A., König, B.: Verifying Finite-State Graph Grammars: An Unfolding-Based Approach. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 83–98. Springer, Heidelberg (2004)
5. Baldan, P., König, B.: Approximating the behaviour of graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 14–29. Springer, Heidelberg (2002)
6. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
8. Bozga, M., Iosif, R., Lakhnech, Y.: On logics of aliasing. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 344–360. Springer, Heidelberg (2004)
9. Dams, D., Namjoshi, K.S.: Shape analysis through predicate abstraction and model checking. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 310–323. Springer, Heidelberg (2002)
10. Dodds, M., Plump, D.: Extending C for checking shape safety. In: Graph Transformation for Verification and Concurrency 2005. ENTCS, vol. 154(2), pp. 95–112. Elsevier, Amsterdam (2006)

11. Drewes, F., Kreowski, H.-J., Habel, A.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Foundations, vol. I, pp. 95–162. World Scientific, Singapore (1997)
12. Engelfriet, J.: A Greibach Normal Form for Context-Free Graph Grammars. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 138–149. Springer, Heidelberg (1992)
13. 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)
14. Lee, O., Yang, H., Yi, K.: Automatic verification of pointer programs using grammar-based shape analysis. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 124–140. Springer, Heidelberg (2005)
15. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, S., Srivastava, S., Yorsh, G.: Simulating reachability using first–order logic with applications to verification of linked data structures. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 99–115. Springer, Heidelberg (2005)
16. Noll, T., Rieger, S.: Verifying dynamic pointer-manipulating threads. In: Cuellar, J., Maibaum, T.S.E. (eds.) FM 2008. LNCS, vol. 5014. Springer, Heidelberg (2008)
17. Nystrom, E.M., Kim, H.-S., Hwu, W.W.: Bottom–up and top–down context–sensitive summary–based pointer analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 165–180. Springer, Heidelberg (2004)
18. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL 2004, pp. 268–280. ACM Press, New York (2004)
19. Podelski, A., Wies, T.: Boolean heaps. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 268–283. Springer, Heidelberg (2005)
20. Rensink, A.: Canonical graph shapes. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 401–415. Springer, Heidelberg (2004)
21. Rensink, A., Distefano, D.: Abstract graph transformation. In: Proc. of Int. Workshop on Software Verification and Validation (SVV 2005). Electr. Notes Theor. Comput. Sci, vol. 157(1) (2006)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
23. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3–valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
24. Yong, S.H., Horwitz, S.: Pointer-range analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 133–148. Springer, Heidelberg (2004)
25. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: PLDI 2004, pp. 145–157. ACM Press, New York (2004)

Inductively Sequential Term-Graph Rewrite Systems*

Rachid Echahed

CNRS, LIG
46, avenue Félix Viallet, F-38031 Grenoble, France
Rachid.Echahed@imag.fr

Abstract. Definitional trees have been introduced by Sergio Antoy in order to design an efficient term rewrite strategy which computes needed outermost redexes. In this paper, we consider the use of definitional trees in the context of term-graph rewriting. We show that, unlike the case of term rewrite systems, the strategies induced by definitional trees do not always compute needed redexes, in presence of term-graph rewrite systems. We then define a new class called inductively sequential term-graph rewrite systems (istGRS) for which needed redexes are still provided by definitional trees. Systems in this class are not confluent in general. We give additional syntactic criteria over istGRS's which ensure the confluence property with respect to the set of admissible term-graphs.

1 Introduction

Many declarative languages are based on term rewrite systems (TRS). There are good reasons for that, they actually benefit from a solid logical foundations (equational logic, model-theory, proof methods) as well as very efficient implementation techniques. Term rewrite systems have been used also as a unifying computational model for declarative languages sharing both functional and logic features with very efficient operational semantics [5].

However, real-life programs, very often, deal with complex data-structures built by means of pointers (e.g., circular lists, doubly-linked lists, etc.). Such data-structures can be modeled as term-graphs [6,19] and are sometimes mandatory for efficiency reasons, namely time and space complexity of algorithms. Term rewriting constitutes a computational model which is Turing-complete and thus can encode theoretically any transformation over term-graphs, but such encodings are in general cumbersome and too costly. Thus term-graphs appear as a good trade-off to use rewrite systems to compute with general data-structures without using all the machinery specific to graph transformations [20,13,14]. In recent works, e.g. [4,3,18] term-graph rewriting has also been considered as a means to implement naturally, in declarative languages, the call-time choice semantics introduced in [16].

* This work has been partly funded by the project ARROWS of the French *Agence Nationale de la Recherche.*

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 84–98, 2008.

A new class of term-graph rewrite systems (hereafter, noted tGRS) has been introduced recently in [8]. This class is a conservative extension of those introduced in [9,10]. It provides some features dedicated especially to pointer rewriting such as *redirection of pointers* or *node constraints* (see section 2). These features allow one to write, in a rule-based language, algorithms with efficient space complexity such as *in-situ list reversal* or those manipulating node constraints such as the *length of circular lists.* Such algorithms were not possible to encode directly in previous works regarding term-graph rewriting such as [19,9,10].

The new features of tGRSs are very appealing. That is why, we intend to pursue our efforts in investigating the class of tGRSs. In [8], a categorical approach has been proposed, [12] present a discussion about the use of term-graphs with priorities as a means to overcome the non-confluence issues and [11] presents the first general and complete narrowing procedure which is able to synthesize solutions with circular data-structures.

The present paper is a first step towards the conception of efficient rewrite strategies in presence of subclasses of tGRSs. We particularly consider the use of *Definitional Trees* introduced by Antoy in his seminal paper [1]. Definitional trees have been successfully used in defining efficient strategies either in term rewriting and narrowing [1,2,5], graph rewriting and graph narrowing [9,10,4]. We show that, the strategies induced by Definitional trees do not compute needed redexes in general. Then, we define a particular class of term-graph rewrite systems for which the induced strategies are efficient and compute needed redexes.

The paper is organized as follows. The next section defines the class of term-graph rewrite systems that we consider. In section 3 we show some negative results regarding the use of definitional trees and introduce the class of inductively sequential tGRSs, for which Definitional trees help to compute needed redexes. In section 4 we show the confluence property for a subclass of inductively sequential term-graph rewrite systems. Section 5 concludes the paper.

2 Preliminary Definitions

In this section we define a class of term-graph rewrite systems, denoted tGRS. We define the shape of its rules and the process of rewriting. The right-hand sides of the rules consist of sequences of actions. These actions are intended to decompose the transformation of graphs into consecutive *atomic* actions.

Definition 1 (Signature). *A* many-sorted signature $\Sigma = \langle S, \Omega \rangle$ *consists of a set* S *of sorts and an* S*-indexed family of sets of operation symbols* $\Omega = \uplus_{s \in S} \Omega_s$ *with* $\Omega_s = \uplus_{w \in S^*} \Omega_{w \to s}$*. We shall write* $f : s_1 \dots s_n \to s$ *whenever* $f \in \Omega_{s_1 \dots s_n \to s}$ *and say that* f *is of* sort s *and* rank $s_1 \dots s_n$*. A* constructor-based signature Σ *is a triple* $\Sigma = \langle S, \mathcal{C}, \mathcal{D} \rangle$ *such that* S *is a set of sorts,* $\mathcal{C}$ *is an* S*-indexed family of sets of* constructor symbols*,* $\mathcal{D}$ *is an* S*-indexed family of sets of* defined operations*,* $\mathcal{C} \cap \mathcal{D} = \emptyset$ *and* $\langle S, \mathcal{C} \uplus \mathcal{D} \rangle$ *is a signature.*

A term-graph is defined in this paper as a set of nodes and edges between the nodes [6]. Each node may be labeled with an operation symbol or not. A node which is not labeled will act as a variable. Let $\mathcal{N} = \uplus_{s\in S}\mathcal{N}_s$, be an S-indexed family of countable sets of *nodes*. $\mathcal{N}$ is supposed to be fixed throughout the rest of the paper.

Definition 2 (Term-Graph)
A term-graph *g over $\langle \Sigma, \mathcal{N}\rangle$ is a tuple $g = \langle \mathcal{N}_g, \mathcal{N}_g^{\Omega}, \mathcal{L}_g, \mathcal{S}_g\rangle$ such that :*

1. *$\mathcal{N}_g$ is the set of nodes of g, i.e., $\mathcal{N}_g = \uplus_{s\in S}(\mathcal{N}_g)_s$ with $(\mathcal{N}_g)_s \subseteq \mathcal{N}_s$.*
2. *$\mathcal{N}_g^{\Omega}$ is the subset of labeled nodes of g, $\mathcal{N}_g^{\Omega} \subseteq \mathcal{N}_g$*
3. *$\mathcal{L}_g$, the* labeling function *of g, is an S-indexed family of functions associating an operation symbol to each labeled node of g, i.e., $\mathcal{L}_g = \uplus_{s\in S}(\mathcal{L}_g)_s$ with $(\mathcal{L}_g)_s : (\mathcal{N}_g^{\Omega})_s \to \Omega_s$.*
4. *$\mathcal{S}_g$, the* successor function *of g, is an S-indexed family of functions associating a (possibly empty) string of nodes to each labeled node of g, i.e., $\mathcal{S}_g = \uplus_{s\in S}(\mathcal{S}_g)_s$ with $(\mathcal{S}_g)_s : (\mathcal{N}_g^{\Omega})_s \to \mathcal{N}_g^*$ such that for every node $n \in (\mathcal{N}_g)_s$:*
   - *if $(\mathcal{L}_g)_s(n) = f$ with $f : s_1 \dots s_k \to s$, then there exist $n_1, \dots, n_k \in \mathcal{N}_g$ such that $(\mathcal{S}_g)_s(n) = n_1 \dots n_k$ and $n_i \in (\mathcal{N}_g)_{s_i}$ for all $i \in 1..k$.*
   - *if $(\mathcal{L}_g)_s(n) = c$ with $c \in \Omega_{\varepsilon,s}$ (c is a constant), then $(\mathcal{S}_g)_s(n) = \varepsilon$ (i.e., n has no successor).*

   We write $n \in \mathcal{S}_g(m)$ if n is a successor of m.

We write ar(n) *for the arity of node n which is equal to the length of $\mathcal{S}_g(n)$. A rooted term-graph, denoted by g^n, is a term-graph g with a distinguished node n ($n \in \mathcal{N}_g$) called the root of g. n will be denoted by $\mathcal{R}oot_g$. Let g be a term-graph and n and m two nodes of g ($n, m \in \mathcal{N}_g$), we write $n \rightsquigarrow_g m$ iff $m \in \mathcal{S}_g(n)$. We will say that node m is reachable in g from node n iff $n \overset{*}{\rightsquigarrow}_g m$. A rooted term-graph g^n is a* constructor-rooted term-graph *if and only if the root n is labeled by a constructor (i.e. $\mathcal{L}_g(n) \in \mathcal{C}$). A rooted term-graph g^n is a* constructor term-graph *if and only if every reachable node m from the root n ($n \overset{*}{\rightsquigarrow}_g m$), m is either labeled by a constructor symbol ($\mathcal{L}_g(m) \in \mathcal{C}$) or m is not labeled ($m \notin \mathcal{N}_g^{\Omega}$).*

In the sequel, we will assume that all formulae we are considering are well-sorted, and thus drop subscripts related to the many-sorted framework.

As the formal definition of term-graphs is not very convenient to write examples, we recall below the linear notation [6] of term-graphs. In the following grammar, the variable A (resp. n) ranges over the set Ω (resp. $\mathcal{N}$):

$$\begin{array}{lcl} \textsc{TermGraph} & ::= & \textsc{Node} \mid \textsc{Node} + \textsc{TermGraph} \\ \textsc{Node} & ::= & n{:}A(\textsc{Node},\dots,\textsc{Node}) \mid n{:}\bullet \mid n \end{array}$$

The root of a rooted term-graph defined by means of a linear expression is the first node of the expression. $n{:}\bullet$ means that node n is not labeled. $+$ stands for the union of graph definitions.

Example 1. Let G_1^a be the graph (see Fig. 1) defined by $G_1^a = \langle \mathcal{N}_{G_1^a}, \mathcal{N}_{G_1^a}^{\Omega}, \mathcal{L}_{G_1^a}, \mathcal{S}_{G_1^a}\rangle$ such that:

- $\mathcal{N}_{G_1^a} = \{a, b, c, d, e\}$
- $\mathcal{N}_{G_1^a}^{\Omega} = \{a, b, c, e\}$

- $\mathcal{L}_{G_1^a}(a) = succ; \mathcal{L}_{G_1^a}(b) = f; \mathcal{L}_{G_1^a}(c) = g; \mathcal{L}_{G_1^a}(e) = h$
- $\mathcal{S}_{G_1^a}(a) = b; \mathcal{S}_{G_1^a}(b) = ce; \mathcal{S}_{G_1^a}(c) = de; \mathcal{S}_{G_1^a}(e) = b$

G_1^a could also be written using the linear notation as follows:

$$G_1^a = a : succ(b : f(c : g(d : \bullet, e : h(b)), e))$$

Definition 3 (Homomorphism). *Let g_1^n and g_2^m be two rooted term-graphs. A* homomorphism *h from g_1^n to g_2^m is a mapping $h : \mathcal{N}_{g_1^n} \to \mathcal{N}_{g_2^m}$ which preserves the root, the labeled nodes and the labeling and successor functions, i.e., $h(n) = m$, $h(\mathcal{N}_{g_1^n}^{\Omega}) \subseteq \mathcal{N}_{g_2^m}^{\Omega}$, and for each labeled node, p, in g_1^n, $\mathcal{L}_{g_2^m}(h(p)) = \mathcal{L}_{g_1^n}(p)$ and $\mathcal{S}_{g_2^m}(h(p)) = h^*(\mathcal{S}_{g_1^n}(p))$ where h^* denotes the extension of h to strings (of nodes) defined by $h^*(p_1 \ldots p_k) = h(p_1) \ldots h(p_k)$.*

Notice that homomorphisms, as defined above, can map unlabeled nodes to labeled ones.

Definition 4 (Actions). *An action has one of the following forms. We omit to give sort constraints which are quite straightforward and thus we assume that all constructions are well-sorted.*

- *a* ***node definition*** *or* ***node labeling*** *$\alpha : f(\alpha_1, \ldots, \alpha_n)$ where $\alpha, \alpha_1, \ldots, \alpha_n$ are nodes and f is a label of rank $s_1, \ldots, s_n$. This means that α is labeled by f and $\alpha_1 \ldots \alpha_n$ are the successor nodes of α ($\mathcal{S}(\alpha) = \alpha_1 \ldots \alpha_n$).*
- *an* ***edge redirection*** *or* ***local redirection*** *$\alpha \gg_i \beta$ where α, β are nodes and $i \in \{1, \ldots, \mathrm{ar}(\mathcal{L}(\alpha))\}$. This is an edge redirection and means that the target of the i^{th} edge outgoing α is redirected to point to the node β.*
- *a* ***global redirection*** *$\alpha \gg \beta$ where α and β are nodes. This means that all edges pointing to α are redirected to point to the node β.*

The result of applying an action a to a term-graph g is denoted by $a[g]$ and is defined as the following term-graph g':

- *If $a = \alpha : f(\alpha_1, \ldots, \alpha_n)$ then $\mathcal{N}_{g'} = \mathcal{N}_g \cup \{\alpha, \alpha_1, \ldots, \alpha_n\}$, $\mathcal{L}_{g'}(\alpha) = f$, $\mathcal{L}_{g'}(\beta) = \mathcal{L}_g(\beta)$ if $\beta \neq \alpha$, and $\mathcal{S}_{g'}(\alpha) = \alpha_1, \ldots, \alpha_n$, $\mathcal{S}_{g'}(\beta) = \mathcal{S}_g(\beta)$ if $\beta \neq \alpha$. $\cup$ denotes classical union.*
- *If $a = \alpha \gg_i \beta$ then $\mathcal{N}_{g'} = \mathcal{N}_g$, $\mathcal{L}_{g'} = \mathcal{L}_g$, and if $\mathcal{S}_g(\alpha) = \alpha_1, \ldots, \alpha_i, \ldots, \alpha_n$ then $\mathcal{S}_{g'}(\alpha) = \alpha_1, \ldots, \alpha_{i-1}, \beta, \alpha_{i+1}, \ldots, \alpha_n$ and for any node γ we have $\mathcal{S}_{g'}(\gamma) = \mathcal{S}_g(\gamma)$ iff $\gamma \neq \alpha$. If α does not occur in $\mathcal{N}_g$, then $g' = g$.*
- *If $a = \alpha \gg \beta$ then $\mathcal{N}_{g'} = \mathcal{N}_g$, $\mathcal{L}_{g'} = \mathcal{L}_g$ and for all nodes δ such that $\mathcal{S}_g(\delta) = \alpha_1, \ldots, \alpha_n$ then $\mathcal{S}_{g'}(\delta) = \alpha'_1, \ldots, \alpha'_n$ such that for i in $1..n$, $\alpha'_i = \beta$ if $\alpha_i = \alpha$, and $\alpha'_i = \alpha_i$ if $\alpha_i \neq \alpha$. If α does not occur in $\mathcal{N}_g$, then $g' = g$.*

The application of an action a to a rooted term-graph g^n is a rooted term-graph g'^m such that $g' = a[g]$ and root m is defined as follows:

- *$m = n$ if a is not of the form $n \gg p$.*
- *$m = p$ if a is of the form $n \gg p$.*

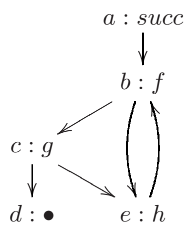


Fig. 1. Term-graph G_1^a

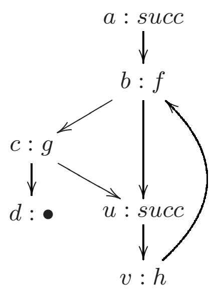


Fig. 2. Term-graph G_2^a

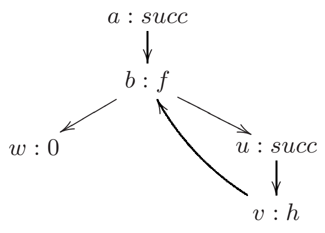


Fig. 3. Term-graph G_3^a

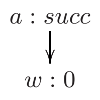


Fig. 4. Term-graph G_4^a

The application of a sequence of actions u to a (rooted) term-graph g is defined inductively as follows : $u[g] = g$ if u is the empty sequence and $u[g] = u'[a[g]]$ if $u = a; u'$ where ; is the concatenation operation.

Example 2
Let G_1^a be the graph defined in *Example 1* (see Fig. 1).
Let G_2^a be the graph (see Fig. 2) $G_2^a = a : succ(b : f(c : g(d : \bullet, u : succ(v : h(b))), u))$
Let G_3^a be the graph (see Fig. 3) $G_3^a = a : succ(b : f(w : 0, u : succ(v : h(b))))$
Let G_4^a be the graph (see Fig. 4) $G_4^a = a : succ(w : 0)$

Below we give some examples of the application of actions on the graphs above. The first line shows the application of the actions $v : h(b)\ ;\ u : succ(v)\ ;\ e \gg u$ on the term-graph G_1^a. The second line shows the application of the actions $w : 0\ ;\ c \gg w$ on the term-graph G_2^a. The last line shows the application of the action $b \gg w$ on the term-graph G_3^a.

$v : h(b)\ ;\ u : succ(\ v)\ ;\ e \gg u\ \ [G_1^a] = u : succ(v)\ ;\ e \gg u[G_1^a + v : h(b)] = e \gg u\ \ [G_1^a + u : succ(v : h(b))] = G_2^a + e : h(b)$

$w : 0\ \ ;\ \ c \gg w\ \ [G_2^a] = c \gg w\ \ [G_2^a + w : 0] = G_3^a + c : g(d : \bullet, u)$

$b \gg w\ \ [G_3^a] = G_4^a + b : f(w, u : succ(v : h(w)))$

Definition 5 (Node Constraint). *A* node constraint *is a (possibly empty) conjunction of disequations between nodes:* $\bigwedge_{i=1}^{n}(\alpha_i \not\doteq \beta_i)$*. A substitution* $\sigma : \mathcal{N} \to \mathcal{N}$ *is a* solution *of a constraint* $\phi = \bigwedge_{i=1}^{n}(\alpha_i \not\doteq \beta_i)$ *iff for any* $i \in [1..n]$*, we have* $\sigma(\alpha_i) \neq \sigma(\beta_i)$*. We denote by* $sol(\phi)$ *the set of solutions of* ϕ*.*

Notice that we do not use equality constraints. Such equalities may be encoded directly into term-graphs.

Definition 6 (Rule, system)
A term-graph rewrite rule *is an expression of the form* $[l \mid c] \to r$ *where* r *is a sequence of actions,* c *is a constraint and* l *is a rooted term-graph s.t. for any node* α *occurring in* l*, we have* $\mathcal{R}oot_l \overset{*}{\rightsquigarrow}_l \alpha$ *(i.e. any node occurring in the left-hand side must be reachable from the root* $\mathcal{R}oot_l$*). A rule* ρ_2 *is said to be a variant of a rule* ρ_1 *iff* ρ_2 *is obtained from* ρ_1 *by (one-one) renaming all the nodes in* ρ_1*. A* term-graph rewrite system *is a set of rewrite rules.*

Example 3. We first define an operation, *sameloc*, which tests whether two arguments are located at the same place or not. Such operation is sometimes used to enhance the implementation of equality in declarative languages.

$r : sameloc(n : \bullet, n) \to q : true; r \gg q$
$[r : sameloc(n : \bullet, m : \bullet) \mid n \not\doteq m] \to q : false; r \gg q$

As a second example, we define below the operation *length* which deals with cyclic data-structures. $length(p : \bullet)$ computes the number of elements of any, possibly circular, list matched by node p.

$r : length(p : \bullet) \to r' : length'(p, p); r \gg r'$
$r : length'(p_1 : nil, p_2 : \bullet) \to r' : 0; r \gg r'$
$r : length'(p_1 : cons(n : \bullet, p_2 : \bullet), p_2) \to r' : s(0); r \gg r'$
$[r : length'(p_1 : cons(n : \bullet, p_2 : \bullet), p_3 : \bullet) \mid p_2 \not\doteq p_3] \to r' : s(q : \bullet); q : length'(p_2, p_3); r \gg r'$

Pointers help very often to enhance the efficiency of algorithms. In the following, we define the operation *reverse* which performs the so-called "in-situ list reversal".

2 $o : reverse(p : \bullet) \to o' : reverse'(p, q : nil); o \gg o'$
$o : reverse'(p_1 : cons(n : \bullet, q : nil), p_2 : \bullet) \to p_1 \gg_2 p_2; o \gg p_1$
$o : reverse'(p_1 : cons(n : \bullet, p_2 : cons(m : \bullet, p_3 : \bullet), p_4 : \bullet) \to p_1 \gg_2 p_4; o \gg_1 p_2; o \gg_2 p_1$

The last example illustrates the encoding of classical term rewrite systems. We define the addition on naturals as well as the function *double* with their usual meanings.

$r : +(n : 0, m : \bullet) \to r \gg m$
$r : +(n : succ(p : \bullet), m : \bullet) \to q : succ(k : +(p, m)); r \gg q$
$r : double(n : \bullet) \to q : +(n, n); r \gg q$

Definition 7 (Matching). *Let $[l \mid c] \to r$ be a rewrite rule and g^n a rooted term-graph. We say that the left-hand side $[l \mid c]$ matches the term-graph g^n at node p, and denoted by $[l \mid c] \leq g^p$ iff p is reachable from n (i.e. $n \overset{*}{\rightsquigarrow}_g p$) and there exists a homomorphism, also called* matcher, *h from l to g^p, i.e. $h : \mathcal{N}_l \to \mathcal{N}_g$ such that $h(\mathcal{R}oot_l) = p$ and h is a solution of constraint c, i.e., $h \in sol(c)$.*

Definition 8 (Rewrite Step). *Let ρ be the rewrite rule $[l \mid c] \to r$ and g^n be a rooted term-graph. We say that g^n rewrites to g_1^m at node p by using the rule ρ iff there exists a matcher $h : l \to g^p$ which is a solution of constraint c and $g_1^m = h(r)[g^n]$. We write $g^n \to_{[p,\,[l|c]\to r]} g_1^m$, $g^n \to_p g^m$ or simply $g^n \to g^m$.*

Example 4. Let f, g and h be three defined operations specified by the following rewrite rules:

$n : f(p : 0, q : \bullet) \to n \gg p$
$n : g(p : \bullet, q : succ(m : \bullet)) \to w : 0; n \gg w$
$n : h(p : \bullet) \to u : succ(v : h(p)); n \gg u$

Let G_1^a, G_2^a, G_3^a and G_4^a be the graphs defined in Example 2. We recall their definitions below.

$G_1^a = a : succ(b : f(c : g(d : \bullet, e : h(b)), e))$
$G_2^a = a : succ(b : f(c : g(d : \bullet, u : succ(v : h(b))), u))$
$G_3^a = a : succ(b : f(w : 0, u : succ(v : h(b))))$
$G_4^a = a : succ(w : 0)$

From the rules given in this example, we can get the following derivation. Notice that we did not report the nodes which are not reachable from the roots of the considered term-graphs.

$$G_1^a \to_e G_2^a \to_c G_3^a \to_b G_4^a$$

3 Inductively Sequential Term-Graph Rewrite Systems

Inductively sequential term rewrite systems have been introduced by Antoy in [1]. Such systems are defined over constructor-based signatures. The left-hand sides of the rules are patterns of the form $f(k_1, \cdots, k_n)$ where f is a defined symbol and the sub-terms (i.e., the k_i's) are constructor terms. By definition, the rules of an inductively sequential term rewrite system are stored in data-structures called definitional trees. Thanks to these data-structures, several efficient rewriting and narrowing strategies have been devised (e.g. [1,5]).

In this section we consider a subclass of tGRSs which consists of systems that can be stored within definitional trees.

Definition 9 (Definitional tree). *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be a tGRS such that Σ is a constructor-based signature. A tree $\mathcal{T}$ is a* partial definitional tree, *or* pdt, *with pattern $[\pi \mid C]$ iff one of the following cases holds:*

- $\mathcal{T} = rule([\pi \mid C] \to r)$, *where* $[\pi \mid C] \to r$ *is a variant of a rule of* $\mathcal{R}$.
- $\mathcal{T} = position.branch([\pi \mid C], o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, *where* o *is a non-labeled node of* π, o *is of sort* s, $c_1, \ldots, c_k$ $(k > 0)$ *are different constructors of the sort* s *and for all* $j \in 1..k$, $\mathcal{T}_j$ *is a pdt with pattern* $[\pi_j | C]$, *such that* π_j *is obtained from* π *by applying an action which labels the node* o *with constructor* c_j, *i.e.,* $\pi_j = o : c_j(o_1 : \bullet, \ldots, o_n : \bullet)[\pi]$, *where* n *is the number of arguments of* c_j *and* $o_1, \ldots, o_n$ *are new nodes.*
- $\mathcal{T} = share.branch([\pi \mid C], \mathcal{T}_1, \mathcal{T}_2)$, *where* $\mathcal{T}_1$ *is a pdt with pattern* $[\pi \mid C \wedge n \dot{\neq} m]$ *such that* n *and* m *are nodes occurring in* π *and the constraint* $n \dot{\neq} m$ *does not occur in* C *and* $\mathcal{T}_2$ *is a pdt with pattern* $[\pi' \mid C]$ *such that* π' *is obtained from* π *by collapsing the two nodes* n *and* m *(and their successors). I.e.* π' *is obtained by encoding the constraint* $n \doteq m$ *into* π.

We write $pattern(\mathcal{T})$ *to denote the pattern argument of a pdt.*
A definitional tree $\mathcal{T}$ *of a defined operation* f *is a* finite *pdt with a pattern of the form* $[p : f(o_1 : \bullet, \ldots, o_n : \bullet) \mid true]$, *also denoted by* $p : f(o_1 : \bullet, \ldots, o_n : \bullet)$, *where* n *is the number of arguments of* f, $p, o_1, \ldots, o_n$ *are new nodes, and for every rule* $[l \mid C] \to r$ *of* $\mathcal{R}$, *with* l *of the form* $f(g_1, \ldots, g_n)$, *there exists a leaf* $rule([l' \mid C'] \to r')$ *of* $\mathcal{T}$ *such that* $[l' \mid C'] \to r'$ *is a variant of* $[l \mid C] \to r$.

Example 5. We consider the auxiliary operation *length′* defined in Example 3. We recall first its rules and provide a definitional tree for it. We give only the pattern or the rule for every node of the tree.

(Rule1) $r : length'(p_1 : nil, p_2 : \bullet) \to r' : 0; r \gg r'$

(Rule2) $r : length'(p_1 : cons(n : \bullet, p_2 : \bullet), p_2) \to r' : s(0); r \gg r'$

(Rule3) $[r : length'(p_1 : cons(n : \bullet, p_2 : \bullet), p_3 : \bullet) \mid p_2 \dot{\neq} p_3] \to r' : s(q : \bullet); q : length'(p_2, p_3); r \gg r'$

Readers familiar with classical definitional trees [1] should notice the introduction of a new kind of nodes called *share.branch*. In the context of term-graph rewriting, sharing of data-structures plays an important role which cannot be handled easily in the framework of term (tree) rewriting. The addition of the nodes *share.branch* still ensures the property of non overlapping of the patterns situated at the leaves of a definitional tree. We can easily prove the following statement.

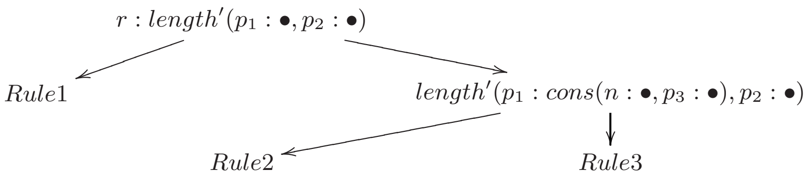

$r : length'(p_1 : \bullet, p_2 : \bullet)$
$Rule1$
$length'(p_1 : cons(n : \bullet, p_3 : \bullet), p_2 : \bullet)$
$Rule2$
$Rule3$

Fig. 5. A definitional tree of operation *length′*

Proposition 1. *Let $\mathcal{T}$ be a definitional tree of a defined operation f. Let $[l_1 \mid c_1] \rightarrow r_1$ and $[l_2 \mid c_2] \rightarrow r_2$ be two different rules of $\mathcal{T}$. Then, the left-hand sides $[l_2 \mid c_2]$ and $[l_1 \mid c_1]$ do not overlap. I.e., there exist no term-graph g, and matchers $h_1 : l_1 \rightarrow g$ and $h_2 : l_2 \rightarrow g$ which fulfil respectively constraints c_1 and c_2.*

Hereafter, we define the rewrite strategy Φ induced by definitional trees. We start by the following technical definition of constructor paths.

Definition 10 (Constructor Path). *We will say that a node p is reachable from a node n_0 in a term-graph g through a* constructor path *iff there exists a path in g, say $n_0 \rightsquigarrow_g n_1 \rightsquigarrow_g \ldots \rightsquigarrow_g n_k \rightsquigarrow_g p$ such that, for all $i \in 0..k$, $\mathcal{L}_g(n_j)$ is a constructor symbol ($\in \mathcal{C}$).*

Definition 11 (A term-graph rewrite strategy). *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be a tGRS such that Σ is constructor-based and the rules of every defined operation are stored in a definitional tree. Let g^n be a rooted term-graph. Let p be a reachable node from the root n through a constructor path in g^n such that p is labeled by a defined operation f and let $\mathcal{T}_f$ be a definitional tree of f. Φ is the partial function defined by $\Phi(g^n) = \varphi(g^p, \mathcal{T}_f)$.*

Below, we define the partial function φ. Let g^n be a rooted term-graph such that $\mathcal{L}_{g^n}(n) \in \mathcal{D}$ (i.e. the root n is labeled with a defined operation) and $\mathcal{T}$ a pdt such that $pattern(\mathcal{T}) \leq g^n$. When it is defined, the value $\varphi(g^n, \mathcal{T})$ is a pair (p, R) such that the term-graph g^n can be reduced at node p using the rule R. More precisely, $\varphi(g^n, \mathcal{T})$ is defined as follows:

$$\varphi(g^n,\mathcal{T}) = \begin{cases} (n, [\pi' \mid C'] \rightarrow r') & \textit{if } \mathcal{T} = rule([\pi \mid C] \rightarrow r) \textit{ and} \\ & \quad [\pi' \mid C'] \rightarrow r' \textit{ is a variant of } [\pi \mid C] \rightarrow r \ ; \\ \varphi(g^n, \mathcal{T}_i) & \textit{if } \mathcal{T} = share.branch([\pi \mid C], \mathcal{T}_1, \mathcal{T}_2) \textit{ for} \\ & \quad \textit{the unique } i \textit{ such that } pattern(\mathcal{T}_i) \leq g^n \textit{ and } i \in 1..2; \\ \varphi(g^n, \mathcal{T}_i) & \textit{if } \mathcal{T} = position.branch([\pi \mid C], o, \mathcal{T}_1, \ldots, \mathcal{T}_k) \textit{ for} \\ & \quad \textit{the unique } i \textit{ such that } pattern(\mathcal{T}_i) \leq g^n \textit{ and } i \in 1..k; \\ (p, R) & \textit{if } \mathcal{T} = position.branch([\pi \mid C], o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad [\pi \mid C] \textit{ matches } g^n \textit{ at the root } n \textit{ by} \\ & \quad \textit{homomorphism } h : \pi \rightarrow g, \\ & \quad h(o) \textit{ is labeled with a defined operation } f \textit{ (in } g\textit{)}, \\ & \quad \mathcal{T}' \textit{ is a definitional tree of } f \textit{ and} \\ & \quad \varphi(g^{h(o)}, \mathcal{T}') = (p, R). \end{cases}$$

Example 6. We illustrate the use of the strategy Φ. We consider again the operations and the rules given in Example 4.

(R1) $n : f(p : 0, q : \bullet) \rightarrow n \gg p$
(R2) $n : g(p : \bullet, q : succ(m : \bullet)) \rightarrow w : 0; n \gg w$
(R3) $n : h(p : \bullet) \rightarrow q : succ(m : h(p)); n \gg q$

First, we provide a definitional tree for each operation.

$\mathcal{T}_f = position.branch(n : f(p : \bullet, q : \bullet), p, rule(R1))$
$\mathcal{T}_g = position.branch(n : g(p : \bullet, q : \bullet), q, rule(R2))$
$\mathcal{T}_h = rule(R3)$

The following derivation given in Example 4 is developed by the strategy Φ.

$$G_1^a \rightarrow_e G_2^a \rightarrow_c G_3^a \rightarrow_b G_4^a$$

One can easily verify the following equalities:

$\Phi(G_1^a) = (e,\text{R3})$
$\Phi(G_2^a) = (c,\text{R2})$
$\Phi(G_3^a) = (b,\text{R1})$

The aim of the definition of the strategy Φ is to compute needed nodes to be contracted during the transformation of a term-graph. We define below the notions of needed nodes and outermost nodes in the framework of term-graph rewriting.

Definition 12 (needed node, outermost redex). *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be a tGRS such that Σ is constructor-based. Let g_1^n and g_2^m be two term-graphs and $B = g_1^n \xrightarrow{*} g_2^m$ a rewrite derivation. A node q labeled with a defined operation in g_1^n and reachable from the root n* is a residual node by B *if q remains reachable from the root m in g_2^m. Then, we call* descendant of g_1^q *the rooted term-graph g_2^q. A node q in g is* needed *iff in every rewrite derivation from g to a constructor normal form, a descendant of g^q is rewritten at its root q. A node q labeled with a defined operation in g^n is an* outermost node *of g^n iff $q = n$ or q is reachable from n through a constructor path. A redex u rooted by q in g^n is an* outermost redex *iff $q = n$ or q is reachable from n through a path $p_0 \rightsquigarrow_{g^n} p_1 \rightsquigarrow_{g^n} \ldots \rightsquigarrow_{g^n} p_k$ such that $p_0 = n$, $p_k = q$ and g^{p_i} is not a redex for all $i \in 0..(k-1)$.*

Unlike the case of terms, we show in the following proposition that, in general, the strategy Φ does not compute needed nodes when it is applied on term-graphs. We will give later in Definition 13 sufficient conditions which ensure the neededness of the nodes computed by the strategy Φ.

Proposition 2. *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be a tGRS such that Σ is constructor-based and the rules of every defined operation are stored in a definitional tree. Let g^n be a rooted term-graph. Then,*

1. *the computation of $\phi(g^n)$ may be infinite.*
2. *if $\phi(g^n) = (p, R)$, the node p is not needed in general.*
3. *if $\phi(g^n)$ is not defined, g^n can still have a constructor normal form.*

Proof. The proof is given by counter examples. Let us consider the following tGRS which satisfies the conditions of the proposition.

$$r : f_1(p : 0) \rightarrow r \gg p$$

$$r : f_1(p : succ(p' : \bullet)) \rightarrow r \gg p$$

$$r : h_1(p : 0, q : succ(n : \bullet)) \rightarrow q \gg p$$

$$r : g_1(p : 0) \rightarrow r \gg p$$

1. Let E^n be the term-graph $n : f_1(m : f_1(n))$. Then, by definition of the strategy ϕ, $\phi(E^n) = \varphi(E^n, \mathcal{T}_{f_1})$, for some definitional tree, $\mathcal{T}_{f_1}$, of f_1. By definition of φ and from the patterns of the rules defining f_1, one can easily verify that $\phi(E^n)$ does not halt.
2. Let $G^n = n : succ(r : succ(p : f_1(q : succ(s : h_1(u : 0, r)))))$. We can easily verify that $\phi(G^n) = (p, r_1 : f_1(p_1 : succ(p'_1 : \bullet)) \to r_1 \gg p_1)$. However, the node p is not needed in G^n since one may obtain the desired normal form $n : succ(u : 0)$ after one rewrite step performed at node s.
3. Let us consider the graph $H^n = n : succ(r : succ(p : g_1(q : succ(s : h_1(u : 0, r)))))$. Then, $\phi(H^n) = \varphi(H^p, \mathcal{T}_{g_1})$ is not defined for any definitional tree $\mathcal{T}_{g_1}$. However, if we rewrite H^n at node s we get a constructor normal form $H_1^n = n : succ(u : 0)$.

To overcome the issues pointed by Proposition 2, we propose below sufficient syntactic conditions over rewrite rules.

Definition 13 (inductively sequential tGRS). *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be a tGRS. SP is called* inductively sequential *tGRS iff (i) the signature Σ is constructor-based, (ii) the rules which define every defined operation are stored in a definitional tree and (iii) the nodes which can be subject to local or global redirections are the roots of the left-hand sides of the rules. That is to say, for all rules $[\pi \mid C] \to r$ in $\mathcal{R}$, for all global (respectively, local) redirections of the form $p \gg q$ (respectively, $p \gg_i q$ for some i), occurring in the right-hand side r, we have $p = Root_\pi$.*

Example 7. The rewrite systems given in Example 3 and Example 4 are all inductively sequential but the one which defines the operation *reverse*.

The following proposition summarizes the main properties of φ in presence of inductively sequential term-graph rewrite systems.

Proposition 3. *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be an inductively sequential tGRS, f a defined operation, $\mathcal{T}_f$ a definitional tree of f, and g^n a rooted term-graph whose root is labeled with f (i.e. $\mathcal{L}_{g^n}(n) = f$). If $\varphi(g^n, \mathcal{T}_f) = (p, R)$, then (i) in every rewrite derivation from g^n to a constructor-rooted term-graph, a descendant of g^p is rewritten at the root p, in one or more steps, into a constructor-rooted term-graph ; (ii) g^p is a redex of g matched by the left-hand side of R ; (iii) g^p is an outermost redex of g^n. (iv) If $\varphi(g^n, \mathcal{T})$ is not defined, then g^n cannot be rewritten into a constructor-rooted term-graph.*

Theorem 1. *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be an inductively sequential tGRS, and g^n a rooted term-graph. If $\Phi(g) = (p, R)$, then g^p is an outermost needed redex of g^n and g^n can be rewritten at node p with rule R. If $\Phi(g)$ is not defined, then g^n cannot be rewritten into a constructor term-graph.*

4 Confluence

In this section, we consider the property of confluence which could be of great interest for deterministic computations. Ensuring confluence in presence of term-graph rewrite systems is not an easy task (see e.g., [17]). For example, a rewrite system as simple as the two following rules $f(x) \to x$ and $g(x) \to x$ is not confluent. Indeed, the term-graph $n : f(m : g(n))$ can be reduced to two different term-graphs $n : f(n)$ and $m : g(m)$. The two last term-graphs cannot be reduced to a common term-graph. In [9,10], a subclass of circular term-graphs, called admissible term-graphs, has been introduced. It has been shown that, for a large class of term-graph rewrite systems, the rewrite relation induced over admissible term-graphs is confluent. In this section, we generalise that result to the admissible inductively sequential tGRSs.

Definition 14 (admissible rooted term-graph). *[9,10] A rooted term-graph g^n is admissible iff for all nodes m, labeled by a defined operation (i.e., $\mathcal{L}_{g^n}(m) \in \mathcal{D}$), m is not reachable from itself (i.e., m does not belong to a cycle $m \not\overset{*}{\rightsquigarrow} m$).*

Definition 15 (admissible inductively sequential tGRS). *Let $SP{=}\langle \Sigma, \mathcal{R}\rangle$ be an inductively sequential tGRS. SP is called admissible iff for all rules $[\pi \mid C] \to r$ in $\mathcal{R}$ the following conditions are satisfied*

- *for all global (respectively, local) redirections of the form $p \gg q$ (respectively, $p \gg_i q$ for some i), occurring in the right-hand side r, we have $p = Root_\pi$ and $q \neq Root_\pi$.*
- *for all actions of the form $\alpha : f(\beta_1, \ldots, \beta_n)$, for all $i \in 1..n$, $\beta_i \neq Root_\pi$*
- *the set of actions of the form $\alpha : f(\beta_1, \ldots, \beta_n)$, appearing in r, do not construct a cycle consisting only of newly introduced nodes in r and including a node labeled with a defined operation. If we denote by $\rightsquigarrow_r$ the reachability over the new nodes introduced in r, this condition could be specified as : for all nodes, α, introduced in r and labeled by a defined operation, $\alpha \not\overset{*}{\rightsquigarrow}_r \alpha$.*
- *Constraint C includes disequations of the form $p \neq q$ where p and q are labeled by constructor symbols.*

Example 8. All the previous inductively sequential systems are admissible or can be modified to fulfil the required conditions. Below we provide an admissible inductively sequential tGRS which defines equality over naturals.

$$p : eq(n : \bullet, n) \to q : true; p \gg q$$

$$[p : eq(n : 0, m : 0) \mid n \neq m] \to q : true; p \gg q$$

$$[p : eq(n : succ(n' : \bullet), m : succ(m' : \bullet)) \mid n \dot{\neq} m] \to p \gg_1 n'; p \gg_2 m'$$

$$p : eq(n : succ(n' : \bullet), m : 0) \to q : false; p \gg q$$

$$p : eq(n : 0, m : succ(m' : \bullet)) \to q : false; p \gg q$$

The following proposition states that the class of admissible term-graphs is closed under the rewrite relation induced by an admissible inductively sequential tGRS.

Proposition 4. *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be an admissible inductively sequential tGRS and g^n an admissible rooted term-graph. If g^n rewrites to g^m via a rewrite rule in $\mathcal{R}$, then g^m is also an admissible rooted term-graph.*

Definition 16 (Confluence). *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be an admissible inductively sequential tGRS. We say that* the rewriting relation $\rightarrow$ is confluent w.r.t the class of admissible term-graphs *iff for all rooted admissible term-graphs g_1^n, $g_2^{n'}$, g_3^m and $g_4^{m'}$ such that g_1^n and $g_2^{n'}$ are identical up to renaming of nodes ($g_1^n \sim g_2^{n'}$), $g_1^n \xrightarrow{*} g_3^m$ and $g_2^{n'} \xrightarrow{*} g_4^{m'}$, there exist two admissible graphs g_5^o and $g_6^{o'}$ such that $g_3^m \xrightarrow{*} g_5^o$, $g_4^{m'} \xrightarrow{*} g_6^{o'}$ and $g_5^o \sim g_6^{o'}$.*

We state below a new confluence result regarding the class of admissible inductively sequential tGRS. The reader familiar with the confluence property may notice that systems in this class are not always confluent modulo bisimilarity (two term-graphs are said bisimilar iff they represent the same rational term). For instance the application of the operation *length*, as defined in Example 3, to two bisimilar and non isomorphic lists, should yield different values.

Theorem 1. *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be an admissible inductively sequential tGRS. Then the rewriting relation $\rightarrow$ is confluent w.r.t the class of admissible term-graphs.*

The proof of Theorem 1 is obtained by classical induction on the length of the considered rewrite derivations and leans basically on the following key result.

Lemma 1. *Let $SP = \langle \Sigma, \mathcal{R} \rangle$ be an admissible inductively sequential tGRS and g^n, g_1^m and g_2^o be three admissible term-graphs. If $g^n \rightarrow g_1^m$ and $g^n \rightarrow g_2^o$, then there exist two graphs g_3^p and g_4^q such that $g_1^m \xrightarrow{\varepsilon} g_3^p$, $g_2^o \xrightarrow{\varepsilon} g_4^q$ and g_3^p and g_4^q are equal up to renaming of nodes ($g_3^p \sim g_4^q$). The notation $g \xrightarrow{\varepsilon} g'$ means that g' is either g (zero rewrite step) or it is obtained from g after one rewrite step.*

5 Conclusion

Definitional trees [1] give rise to efficient rewrite and narrowing strategies. In this paper we investigated ways to use Definitional trees with the aim to propose new efficient strategies for term-graph rewriting. We succeeded to show the computation of needed redexes in the particular class of inductively sequential tGRSs. We gave also counter-examples illustrating some negative results. These results give an idea about the limits of the use of Definitional trees in the context of term-graph rewriting. On the other hand, we proposed a new class of admissible term-graph rewrite systems for which the rewrite relation is confluent with respect to admissible term-graphs and for which Definitional trees still behave nicely. The presented results open some directions of work. In

[11], a general narrowing procedure has been proposed. The class of inductively sequential tGRSs seem to be a good candidate to develop an efficient narrowing strategy for term-graphs. Abstraction techniques has been successfully used in the context of term rewrite systems (see, e.g., [7,15]). Extensions of abstraction methods to term-graph rewrite systems worth also to be investigated.

References

1. Antoy, S.: Definitional trees. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 143–157. Springer, Heidelberg (1992)
2. Antoy, S.: Evaluation strategies for functional logic programming. Journal of Symbolic Computation 40(1), 875–903 (2005)
3. Antoy, S., Braßel, B.: Computing with subspaces. In: Proc. of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2007), pp. 121–130. ACM Press, New York (2007)
4. Antoy, S., Brown, D.W., Chiang, S.-H.: Lazy context cloning for non-deterministic graph rewriting. Electr. Notes Theor. Comput. Sci. 176(1), 3–23 (2007)
5. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM 47(4), 776–822 (2000)
6. Barendregt, H., van Eekelen, M., Glauert, J., Kenneway, R., Plasmeijer, M.J., Sleep, M.: Term graph rewriting. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) PARLE 1987. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
7. Bert, D., Echahed, R.: Abstraction of Conditional Term Rewriting Systems. In: Lloyd, J. (ed.) Proc. of International Logic Programming Symposiumon (ILPS), pp. 162–176. MIT Press, Cambridge (1995)
8. Duval, D., Echahed, R., Prost, F.: Modeling pointer redirection as cyclic term-graph rewriting. Electr. Notes Theor. Comput. Sci. 176(1), 65–84 (2007)
9. Echahed, R., Janodet, J.C.: Admissible graph rewriting and narrowing. In: Proc. of Joint International Conference and Symposium on Logic Programming (JICSLP 1998), pp. 325–340. MIT Press, Cambridge (1998)
10. Echahed, R., Janodet, J.-C.: Parallel admissible graph rewriting. In: Fiadeiro, J.L. (ed.) WADT 1998. LNCS, vol. 1589, pp. 122–137. Springer, Heidelberg (1999)
11. Echahed, R., Peltier, N.: Narrowing data-structures with pointers. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 92–106. Springer, Heidelberg (2006)
12. Echahed, R., Peltier, N.: Non strict confluent rewrite systems for data-structures with pointers. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 137–152. Springer, Heidelberg (2007)
13. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformations, Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
14. Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformations, Concurrency, Parallelism and Distribution, vol. 3. World Scientific, Singapore (1999)
15. Hanus, M.: Call pattern analysis for functional logic programs. In: Proc. of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2008) (July 2008) (to appear)
16. Hußmann, H.: Nondeterministic algebraic specifications and nonconfluent term rewriting. J. Log. Program 12(3&4), 237–255 (1992)

17. Kennaway, J.R., Klop, J.K., Sleep, M.R., Vries, F.J.D.: On the adequacy of graph rewriting for simulating term rewriting. ACM Transactions on Programming Languages and Systems 16(3), 493–523 (1994); previous version: Technical Report CS-R9204, CWI, Amsterdam (1992)
18. López-Fraguas, F.J., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: A simple rewrite notion for call-time choice semantics. In: PPDP, the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Wroclaw, Poland, July 14-16, 2007, pp. 197–208. ACM, New York (2007)
19. Plump, D.: Term graph rewriting. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 3–61. World Scientific, Singapore (1999)
20. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Foundations, vol. 1. World Scientific, Singapore (1997)

Mobile Agents Implementing Local Computations in Graphs

Bilel Derbel[1,⋆], Mohamed Mosbah[2,⋆⋆], and Stefan Gruner[3,⋆⋆⋆]

[1] Laboratoire d'Informatique Fondamentale de Lille (LIFL),
Université des Sciences et Technologies de Lille 1, France
bilel.derbel@lifl.fr

[2] Laboratoire Bordelais de Recherche en Informatique (LaBRI),
Université Bordeaux 1/ENSEIRB, France
mosbah@labri.fr

[3] Department of Computer Science,
University of Pretoria, South Africa
sgruner@cs.up.ac.za

Abstract. Mobile agents are a well-known paradigm for the design and implementation of distributed systems. However, whilst their popularity continues to grow, a uniform theory of mobile agent systems is not yet sufficiently elaborated, in comparison with classical models of distributed computation. In this paper we show how to use mobile agents as an alternative model for implementing distributed local computation rules. In doing so, we approach a general and unified framework for local computations which is consistent with the classical theory of distributed computations based on graph relabeling systems.

Keywords: Distributed algorithms, Mobile agents, Relabeling systems.

1 Introduction

Models of *local computations*, described by *graph relabeling systems* provide a useful theoretical framework to specify and reason about various aspects of distributed computation with distributed algorithms [9,10,2]. Assuming that the reader is already familiar with this theoretical background, we will only briefly recapitulate the basic characteristics and features of modeling distributed systems by local computations and graph relabeling systems. This well-known paradigm will be our starting point from where we shall proceed towards a more recent paradigm of distributed computation by *mobile agents*.

Our aim is to demonstrate that all basic building blocks of the graph relabeling paradigm can be implemented by the activities of *mobile agents*, leading to

[⋆] Supported by the DOLPHIN INRIA project team.

[⋆⋆] This work has been supported by the french project "Agence Nationale de la Recherche ANR-06-SETI-015".

[⋆⋆⋆] Supported by the National Research Foundation (NRF) of the Republic of South Africa.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 99–114, 2008.

the hypothesis that mobile agents are as powerful as classical distributed systems, i.e., message passing systems [4]. In practice, the use of mobile agents for the implementation of distributed algorithms can have advantages over classical implementations, because roaming agents can better cope with temporary network failures and also consume less computational resources, in comparison with the global network activities induced by classical implementations of distributed algorithms. In addition, mobile agents allow to bring a new level of abstraction in distributed computing. For instance, in the message passing model, the nodes represent both the topology of the network and the autonomous computation entities. In opposite, in the mobile agent model, the nodes define only the topology of the network, while the agents define the computation entities of the network.

The consideration (description, reconstruction) of agent systems in terms of graph transformation systems is not a new idea; take for example [8] as an early contribution to this field of study. In [8], however, graph transformation techniques are used to model *internal* properties and/or actions of agents, whereas the focus of our paper is on their *external* properties, mainly *motion* between network places, motivated by our intention to demonstrate the possibility of expressing (respectively implementing) classical distributed algorithms in terms of mobile agent systems. To this end, graph transformation systems can be regarded as the bridge formalism between the domain of classical distributed algorithms and the domain of mobile agent systems.

Graph Relabeling Systems: Processor networks, which are the substrate of distributed computation, are represented by labeled graphs $G = (V, E, L, \lambda)$ with a set of labels L and a (possibly partial) labeling function $\lambda : (V \uplus E) \longrightarrow L$ that attaches labels to vertices (nodes) and/or edges (arcs) of the network graph. The labels, which may lexically appear arbitrarily complex, are used to model the internal states of the network components during the run of a distributed algorithm on the network. A final label configuration represents the result of a terminated algorithm. Thereby, the models must be designed in such a way that three *locality conditions* are always fulfilled:

c1: Relabeling does not modify the underlying graph structure (from a topological point of view);

c2: Each step can only relabel a limited, connected sub graph (fixed in size);

c3: The applicability of a relabeling step in a "neighborhood" is constrained only by the local conditions within such a neighborhood, not by the global state of the entire network.

Distributed algorithms described in such a framework are usually composed of *basic units* which correspond to certain types of relabeling rules. These various rule types, which are classified and explained in [5], comprise constructs such as: single node relabeling depending on only one neighbor, two neighbor relabeling, single node relabeling depending on labels of all neighbors (star relabeling), single node relabeling in the center of a *ball* of radius 2, relabeling of an entire ball of radius $k > 1$.

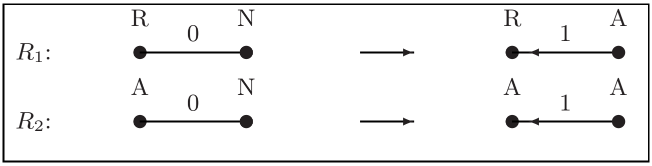


Fig. 1. Rules for a distributed construction of a rooted spanning tree

However, not every rule type is suitable for composing (describing) a particular distributed algorithm. For example, whereas the distributed computation of a *spanning tree* across the underlying network can well be described in terms of the most simple rule type (relabeling one node depending on only one neighbor), others, more complicated distributed algorithms can only be described in terms of more complicated types of relabeling rules [11]. Anyway, a relabeling system gives us a uniform and unified methodology for describing and proving distributed algorithms. For example, Figure 1 shows a simple relabeling rule system, consisting of two rules only, by means of which a spanning tree of a graph can be computed in a distributed and self-organizing fashion[1].

Mobile Agent Systems: One can ask how to turn a set of relabeling rules into an executable distributed algorithm — in other words, how to implement a distributed algorithm described with a relabeling system into a practical distributed setting. Because we can find many types of distributed systems relying on the type of communication (e.g., messages, shared memory), the type of synchrony (e.g., synchronous, asynchronous), and the type of computation entities (e.g., processors, mobile agents), many solutions are possible. For instance, some algorithms are known for the case of classical message passing systems [13,14,12,7]. In this paper we are interested in a uniform mobile agent solution.

When a distributed algorithm is to be implemented by means of mobile agents, a variety of issues must be considered. Amongst those, there are some especially important considerations concerning the nature of *synchronization*, the notion of *agent*, as well as the organization of *agent processing* whenever an agent arrived at a particular node in the network.

In order to perform local relabeling classically, some type of synchronization is needed for a short period of time between the involved nodes. In the usual implementation of a local relabeling step [3], *messages* are sent between the involved nodes such that, depending on the information contents of those messages, synchronization can be achieved. In a pure mobile agent system, however, there are no messages; there are only agents moving from node to node.[2] Consequently, the notion of "synchronization" looses its traditional meaning: In a classical distributed system, all nodes are active during the same time. They might not have

[1] In the example of Fig. 1, it is implicitly assumed that at the beginning there exists a unique node with label R (root) in the network graph, and all the other nodes have label N.

[2] Thus the agent itself "is" a message.

a common clock and might follow their own local speed of pace, but no node is supposed to fall asleep until the termination of the algorithm. In a mobile agent system, on the contrary, a network node is asleep as long as no agent is locally present: consequently, the notion of "synchrony" in a mobile agent system can only be circumscribed in terms of particular *patterns of agent moves* between two quasi-synchronized neighbor nodes.

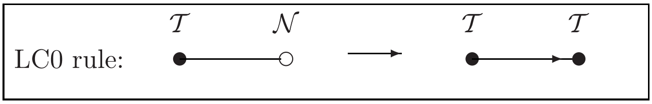
Remainder of the Paper: In the remainder of the paper we present some mobile agent implementations of relabeling systems. In Section 2, we describe two implementations of two basic classes of local relabeling rules called type LC0 and LC1. In Section 3 we describe a *general* methodology (or framework) for implementing *any* class of relabeling rules.

2 Basic Agent Operations

As mentioned in the Introduction, there are several classes of local relabeling rules, according to the various types of neighborhoods in which those rules can be anchored (e.g. edge-shaped neighborhood, star-shaped neighborhood, ball-shaped neighborhood, and the like). In the following sub-sections we present novel solutions for rules of type LC0 and LC1 in terms of mobile agents.

2.1 Blocking-Free LC0

The LC0 rule (which is well-known to be suitable for the distributed computation of *spanning trees*) looks like this:



It replaces Nonterminal nodes by Terminal nodes, and increases thus its own applicability by every actual application. One should note that an initial labeling of the graph that will allow the application of such a rule must have at least one node with a type $\mathcal{T}$ label.

In the following we present a simple implementation which does not use any blocking in the agent code at all. Instead, mutual exclusion will be provided by the operating systems of the nodes themselves which provides internal *waiting FIFO queues* to cope with the arrival of more than one agent at the same node at the same time. Thus, the here presented approach makes only minimalist requirements as far as the internal structure (code) of agents are concerned.

Preliminaries, Part A: Agents

- All LC0 agents are assumed to be *identical* which means that (a) they carry the *same program code* and (b) they do *not* carry any static unique ID that could distinguish them from each other once and for all.

- In this model an agent can *not* continue the execution of the program code after migrating from one node v to another node v'. Thus they must always start with their very first line of program code again, whenever they arrive at another processing place.
- However, an agent needs *one bit* of *persistent memory* C in order to remember the context from where he came. Because of the specification of the LC0 Rule, namely $\{\mathcal{T}—\mathcal{N}\} \Longrightarrow \{\mathcal{T} \rightarrow \mathcal{T}\}$, an agent must have visited a $\mathcal{T}$ node before he may change an $\mathcal{N}$ node to $\mathcal{T}$. Coming from another $\mathcal{N}$ node the agent may *not* update an $\mathcal{N}$ node, because otherwise the agent would implement a *wrong* rule, namely $\{\mathcal{N}—\mathcal{N}\} \Longrightarrow \{\mathcal{N} \rightarrow \mathcal{T}\}$, for which there is no specification. Without any kind of persistent memory, which the agent can carry along in his 'rucksack' while traveling from node to node, the agent could not remember the type of node from where he came and could thus not correctly implement LC0[3]. At agent creation time we set $C :=$ false and update the value to true as soon as the agent has found its very first LC0 context node of type $\mathcal{T}$.

These types of agents are deliberately specified minimalistically to be the most primitive and 'non-intelligent' agents we can think of; yet these primitive entities will be sufficient to implement the above-mentioned LC0 rule, if only the underlying network environment provides the following features:

Preliminaries, Part B: Environment

- According to the LC0 rule of above, a node possesses one out of two distinguishable types T: These are $\mathcal{T}$, respectively $\mathcal{N}$.
- A node shall also be equipped with a waiting queue for incoming agents. Because of the system being fully asynchronous, the *waiting time* of an agent in a node's waiting queue is completely arbitrary; an agent could vanish in a queue for several hours as well as for just a few micro seconds.
- We assume that an agent gets *exclusive access* to the processor of a node from the beginning to the end of its agent code, while other agents are waiting in the queue until that agent has finished its task. There is *no* round robin (or any other pseudo-simultaneous) processor sharing amongst a multitude of agents sitting in the same node at the same time, which means that two agents can never disturb each other while sitting in the same network node at the same time.
- We assume that a node maintains *locally unique* channel names (port names) to each of its adjacent edges e_i.
- We assume that a node will be able to inform an incoming agent a_0 about local identity of the channel e_i through which that agent entered that node. This information will be stored in the operating system of the node even while the according agent is waiting in the node's internal queue.

[3] Technically speaking, the agent must modify its own program code —like in the 'core war' game— when modifying its own persistent memory, for the agent consists of nothing else but program code.

– After an LCO rule $\{\mathcal{T}—\mathcal{N}\} \Longrightarrow \{\mathcal{T} \rightarrow \mathcal{T}\}$ has been successfully applied to a node v, i.e. another branch (→) of a spanning tree has been constructed and v has become part of it by changing its type from $\mathcal{N}$ to $\mathcal{T}$, v will internally mark the according channel port. A link memory L shall store this information. Note that a node can have at most *one incoming* edge in a spanning tree, such that L is either empty or it carries the name of one of the ports of its node.

Algorithm (Pseudo-Code)

Based on the assumptions of above, the algorithm of Fig. 2 (the code of which is carried by the mobile agents), implements relabeling rules of type LC0. *Deadlock-Freeness* of the procedure is guaranteed because no kinds of blocking techniques (semaphores, etc.) are used at all. *Mutual Exclusion* of agents in one node is guaranteed by the underlying operating system of that node which is assumed to provide a FIFO queue for incoming agents. *Correctness* of the spanning tree construction is guaranteed by the fact that any node can have at most one incoming link, and any agent can create at most one such links at the same time, and at most one agent can be active in the same node at the same time.

Also note that the agent's memory C is actually a *monotonous* function: As soon as the agent has found his first context node of type $\mathcal{T}$, C will switch

```
PROCEDURE AGENT[C] ............ //C is persistent during migration!
BEGIN
ARRIVE @ node;
BEGIN ATOMIC SECTION

if( T(node) == ‘$\mathcal{T}$’ ) ................. // found potential LC0 rule context
C := true;
p := getAnyPort(node); ...................... // try to find node type ‘$\mathcal{N}$’
LEAVE(p);

if( T(node) == ‘$\mathcal{N}$’ AND C == true ) ............ // application possible
i := getMyIncomingPort(node);
L(node) := i;
T(node) := ‘ $\mathcal{T}$’; ................................ // update accomplished
p := getAnyPort(node); ...................... // try to find further work
LEAVE(p);

if( T(node) == ‘$\mathcal{N}$’ AND C == false ) ............ // not seen $\mathcal{T}$-context
p := getAnyPort(node); .................... // try to find node type ‘ $\mathcal{T}$’
LEAVE(p);

END ATOMIC SECTION
END
```

Fig. 2. LC0 implementation: code for an agent

to true and will never be switched back to false again, for any node visited immediately afterward is either a $\mathcal{T}$ too, or will switch from $\mathcal{N}$ to $\mathcal{T}$ in the course of the operation. This means that our agent implementation of above is *almost state-less*, and can indeed be made completely state-less if it can be externally guaranteed that the starting place of an agent (at creation time) is a node which has type $\mathcal{T}$.

2.2 LC1 with Two Different Types of Agents

LC1 is the star rule type that updates a single node in relation to all its neighbors. In other words, LC1 works like a *generalized cellular automaton* rule in the sense of [6]. We can thus sketch the update type of LC1 as:

$$\boxed{\text{LC1 rule: } \overset{\mathcal{N}}{\circ}\,\{\,\overset{\mathcal{X}}{\longrightarrow\!\!\!\!\otimes}\,\}^{*} \quad \longrightarrow \quad \overset{\mathcal{T}}{\bullet}\,\{\,\overset{\mathcal{X}}{\longrightarrow\!\!\!\!\otimes}\,\}^{*}}$$

whereby $\mathcal{X}$ stands for any node label in the neighborhood which will remain the same; only the center of the star is updated when the neighborhood condition is fulfilled. In the following implementation of this rule we will use a *blocking* technique, such that two agents who wish to update neighbor nodes cannot interfere with each other. The according agents of type Star will be used to implement the core of LC1.

However, whenever blocking is allowed, the resulting system is deadlock-prone. To break the symmetry of a mutual-block situation, an agent of type “Lamport” will crawl through the web and assign *priority labels* wherever a mutual-block situation is detected. Consequently an area with a higher priority can be served first by the agents of type Star. In the following we first present the code of the Star agents, thereafter the code of the “Lamport” agent.[4]

Preliminaries. Without loss of generality (only for the sake of intuitive description) we assume that a star center is connected by with its neighbor nodes by means of one *hyper edge*. Given a node set $\mathcal{V}$, a hyper edge is a structure $h = (v, V)$, whereby $v \in \mathcal{V}$ and $V \subseteq \mathcal{V}$. Thus an agent shall be able to use the information $h(v) \in V$ (and for any $v' \in V$, $h^{-}(v') = v$) for the purpose of traveling between a center of a star and its fringes (orientation). A node shall be endowed with a rich internal state, made up of the following components:

- $p \in \mathbb{N}_0 \uplus \{-1\}$ is a priority flag which will be used to solve conflicts between competing neighbor activities. (The value -1 means that this node has not yet been ranked in any priority order.)
- m is the node’s main label, which can be updated as a result of any LC1 rule application.
- h is the node’s hyper edge information which is used by a Star agent to navigate within a star shaped neighborhood.

[4] The idea is inspired by Lamports well-known “Bakery” protocol.

- $M = [m_1, \ldots, m_n]$ is a field with a buffer m_i for every neighbor node $v_i = h(v) \in V$. According to rule LC1 the center node can be updated as soon as information from all its neighbors are collected, and M will be used exactly for this purpose.

Similar to the previous example (LC0), a Star agent shall possess a small, persistent runtime environment which the agent can carry from node to node during migration. The main components of this runtime environment are

- A number memory (‘my-prio’, init.:*nil*), such that priority considerations can be made;

```
AG DEF PERSISTENT VAR:
    my-label(init:nil), my-prio(init:nil),
    my-memo(init:nil), my-counter(init:0) ;
BEGIN
if IF (my-count = 0) AND (host-prio = −1) then
    host-prio := 0 ; ........................................ // mark center active
    my-prio := host-prio ;
    my-count := 1 ; ......................................... // prepare for work
if (my-count = 1) then
    my-label := host-hyp ; .................................. // remember hyper edge
    my-prio := host-prio ;
    my-count := 2 ; ......................................... // prepare for fringe
    DO select neighbor N with host-M[N] = nil ;
    DO move and enqueue into [my-label → N] ;
if (my-count = 2) AND ((host-prio = −1) OR (my-prio < host-prio)) then
    my-memo := host-m ; ..................................... // collect info
    my-count := 3 ; ......................................... // prepare for center
    DO move and enqueue into [my-label⁻] ; .................. // go back
if (my-count = 2) AND (my-prio ≥ host-prio) then
    my-memo := nil ; ........................................ // fringe is blocked
    my-count := 3 ; ......................................... // prepare for center
    DO move and enqueue into [my-label⁻] ; .................. // go back
if (my-count = 3) then
    DO update host-M ← my-memo ; ............................ // bring info
    if IF (host-M contains nil) then  my-count := 1 else my-count := 4
if (my-count = 4) then
    // all neighbors checked
    DO update host-m = • ; .................................. // rule application in center
    host-prio := −1 ; ....................................... // job done
    my-count := 0 ;
    DO move away to another job ;
if (otherwise) then
    // nothing to do here my-count := 0 ;
    DO move away to another job ;
```

Fig. 3. LC1 implementation: code for a Star agent

```
if (host-prio = 0) then
    // found node in critical section
    host-prio := my-number ; ............................ // allocate priority
    my-number := my-number +1 ;
    DO move away to another job ;
```

Fig. 4. LC1 implementation: code for the Lamport agent

- A hyper edge memory, ('my-label', init.:*nil*) such that the agent has orientation within a star-shaped neighborhood of nodes;
- A work-mode flag ('my-counter', init.:0) such that the agent can determine whether he is in the center of a star, or at the fringe of a star, or in search for another job;
- A memory ('my-memo', init.:*nil*) for reading a node's label and transporting this information back into the center of a star.

Agent Star (Pseudo-Code). Based on the preliminaries of above, the algorithm of Fig. 3 for the Star agent should be more or less self-explanatory — note, however, that the agent code is started from the very first line of the program whenever the agent arrives at a new node, (thus: no persistent program pointer and consequently no code-continuation in the process of migration):

Basically the algorithm says: When you have collected information from all the neighbors then you must apply the LC1 rule. However, if a neighbor is prior, then you cannot collect its information and you must return to the center undone; and try again later. The priority labels are allocated by the supportive Lamport agent which is described in the following.

Agent "Lamport" (Pseudo-Code). This agent is very simple, see Fig. 4 for the detailed description. However, to ensure *uniqueness* of the priority numbers, we stipulate that there be only *one* instance of "Lamport" in the network. Because the code of this agent is only short, we can assume that it will work sufficiently fast to do his job across the network. Whenever the Lamport agent finds a critical node (with number 0), it will allocate a unique number $n > 0$ to it. This is also the reason why the Star agent has to update his own priority memory whenever he comes back into the center — because the Lamport agent could have visited the center in the meantime while the Star agent was in the fringe.

Because of the uniqueness of the priority numbers allocated by the Lamport agent, the Star agents can never deadlock, though they can temporarily protect their current neighborhoods against other Star agents roaming in the network.

3 A General Mobile Agent Framework for Relabeling Systems

After having presented two particular examples (LC0, LC1) in the previous section, we are now aiming for a constructive and general method of implementing *any* local graph relabeling system by means of mobile agents. In doing so, we approach a general and unified framework for local computations which is consistent with the classical theory of local computation based on graph relabeling systems.

In the rest of the paper, we consider a k*-locally generated* relabeling system R. We recall that R is called k*-locally generated* if any relabeling rule of R is entirely defined by the precondition and the relabeling of a generic ball of radius at most k. Intuitively speaking, only the labels of nodes and edges in a ball of radius k are changed. One application of this type of relabeling systems is for studying graph reduction rules and graph recognizers in a distributed and static environment. For instance, in [15] it is shows how to encode *handy reduction rules* envolving vertex (edge) deletion (addition) in a distributed environement by mean of k-locally generated relabeling systems.

Before going into the details, let us define the model we will consider. We will assume that each node is equipped with a *whiteboard* where agents can read and write information under mutual exclusion. The label of a node is stored in its whiteboard. The whiteboard $WB(v)$ of a node v contains also other variables allowing agents to exchange information and to communicate together (e.g., to decide whether a node may be relabeled). More precisely, for every node v we will denote by $WB(v).c$ the couple (X, i) with X a label from set $\{\mathbf{M}, \mathbf{Locked}\}$, and i is an integer value. In our general approach, we will assume that every agent has a unique identifier. In fact, if the agents (and the network) are anonymous and if $k > 2$, there exists *no deterministic* distributed algorithm in the asynchronous mobile agent model allowing to execute a k-locally generated relabeling system for *any* graph. This claim can be proved using the equivalence result of [4]. Roughly speaking, the equivalence result there says that mobile agents and message passing systems have the same power from a computability point of view[5]. Since it is well known that it is impossible to implement a k-locally generated relabeling system for any graph using messages (see e.g., [1,13]), our claim is straightforward. For simplicity and clarity, we assume that the identifier of agent A_i (with $i \in \{1, \cdots, n\}$) is i.

Assume that we have n agents which have been scattered over the entire network. Our goal is to make the agents apply the relabeling rules given by R in a distributed way. The examples of the previous section have shown that the major challenge consists in making the agents execute the rules in an independent and concurrent way, that is, if an agent is being executing some rule in some region, then no other agent should execute a rule simultaneously on the same region — otherwise the relabeling may be wrong or ill-defined. We first present an algorithm for the case there is exactly one agent in the network, thereafter we extend the solution for the more general case of many agents ($n > 1$).

3.1 Single Agent Implementation

For now we assume that we have only one agent in the network to *implement* a distributed algorithm *specified* by a local graph relabeling system. Two problems must be solved in this scenario:

[5] In other words, what can be computed by message passing can also be computed by mobile agents and vice versa.

- How shall the agent traverse the entire network without omitting any node?
- How does the agent recognize the neighborhood of a node in order to apply a relabeling rule on this node in that neighborhood?

The traveling problem can be solved by means of a *spanning tree.* Thus, first we make the agent construct a rooted spanning tree T of the entire network. Many spanning tree algorithms are described in the literature, and any kind of spanning tree will do (see also the next section for a Depth First Search (DFS) tree algorithm). Now, the agent can use T as a *map* for traveling across the network. For instance, from the root of T, the agent could perform a DFS-traversal of T. Whenever the agents visit a new node, he temporarily interrupts his DFS-traversal in order to apply a local relabeling rule. Thereafter the agent continues the DFS-traversal to visit another node in T. Once the entire network is traversed the agent will start a new DFS-traversal, and so on, until no further relabeling rules are applicable. This method ensures that all the nodes of the graph will be visited at some time by the agent, such that node starvation is impossible.

Now we need to describe how the agent can execute a graph relabeling rule after arrival at some node v. The idea is to make the agent "learn" the k-neighborhood of v in order to be able to check if a relabeling rule can be applied. In order to learn the node's k-neighborhood, the agent first constructs a Breadth First Spanning (BFS) tree $T_{B(v,k)}$ of the ball $B(v,k)$ rooted at v (for instance this can be done in a layered fashion). Then the agent collects the entire topology of $B(v,k)$ by traversing the neighborhood tree $T_{B(v,k)}$. In case the network nodes have *unique identifiers* the learning of a node's k-neighborhood is quite straightforward. In case that no such unique node identifiers are available it is also not too difficult to let the agent himself create such identifiers for the visited nodes (e.g., when constructing the initial spanning tree T). Having "learned" the topology of $B(v,k)$, and having noticed that some relabeling rule r is applicable in the context of $B(v,k)$, the agent visits $B(v,k)$ again (using the neighborhood tree $T_{B(v,k)}$) and attaches new labels according to rule r.

3.2 Multiple Agent Implementation

In the remainder of this section, we extend the previous single agent approach and describe our generic framework for implementing a k-locally generated relabeling system for any integers $k, n \geq 1$.

Initializing and Traveling the Network. The key idea of our approach is to partition the graph G into a set of n regions $(G_i)_{i\in\{1,\ldots,n\}}$ and to assign a region G_i to every agent A_i. Each agent then applies the applicable relabeling rules in its own region, independent of other agents. Thereby we have to consider how the regions are assigned to the agents, and how the application of rules is managed at the borderline between two regions.

Without loss of generality, we can assume that a node contains no more than one agent at the beginning. In fact, if this assumption is not satisfied then the agents

```
mark the initial departure node as root of $T_{G_i}$;
find a new un-explored node neighboring the current node;
if a new un-explored node $v$ is found then
    mark the new explored node $v$ as part of sub-graph $G_i$;
    update the rooted tree $T_{G_i}$ ;
    continue the exploration (DFS-traversal) from node $v$ (go to line 2);
else
    move back to the previous parent node $u$ using the rooted tree $T_{G_i}$ ;
    if node $u$ is the root and all outgoing edges of $u$ were explored then
        stop the exploration;
        $T_{G_i}$ is ready;
    else
        continue the exploration from node $u$ (go to line 2);
```

Fig. 5. Algorithm InitNetwork for constructing a region G_i: high level code for agent number i

with the lowest identifiers travel the network searching for a new departure node. If no free node is found (which can be detected by performing a DFS-traversal of the network), the agent searching for a departure node vanishes (it dies).

At the beginning, each agent executes algorithm of Fig. 5. This algorithm is an adaptation of the classical DFS-tree algorithm for a mobile agent system. For simplicity, we have omitted the details showing how an agent marks a node or an edge (which is straightforward using the above-mentioned whiteboards of the nodes). After termination, every agent has computed a spanning tree denoted by T_{G_i}. In other words, the region G_i is defined to be the subgraph of G induced by the tree constructed by agent A_i.

Note that it might possibly happen that an agent *fails* to compute a tree. In this case, the agent should vanish and the actual number of agents in the network is decreased. Moreover, the case of a unique agent corresponds to the case where there is only one region (the whole graph). However, the algorithm InitNetwork of Figure 5 allows to construct a spanning forest of G even when the agents do not have unique identifiers which could be of independent interest. Note that algorithm InitAgent can be easily encoded in a high level way using rules type LC0 or LC1.

Executing the local relabeling. Now that the regions $(G_i)_{i\in\{1,\dots,n\}}$ are constructed, every agent is responsible for executing relabeling rules in its own region. In the interior of a region, the rules could be executed like specified by our single agent implementation. However, some conflicts may occur at the borderline between two adjacent regions. The main purpose of the following paragraphs is to show how to deal with these conflicts. First, each agent A_i constructs a BFS-spanning tree $T_{B_i(v,k)}$ of $B(v,k)$ for each node $v \in G_i$ (note that $T_{B_i(v,k)}$ may contain nodes in another region $G_j \neq G_i$). Then, each agent A_i traverses G_i in a DFS fashion using T_{G_i}. When agent A_i is at a node $v \in G_i$, it tries to apply a rule using the following four phase strategy:

1. In the first phase, agent A_i traverses $T_{B_i(v,k)}$ and collects the labels of $B(v,k)$ in order to check if a rule can be applied. If no rule can be applied, then A_i continues the traversal of T_{G_i}. Otherwise, A_i goes to the second step.
2. In the second phase, agent A_i traverses $T_{B_i(v,k)}$ and tries to mark the $WB(w).c$ field of nodes $w \in B(v,k)$ using an extra label **(M,i)** as following:
   - If a node $w \in B(v,k)$ is marked with label **(locked,j)** for any $j \neq i$, then agent A_i waits until node w is unlocked by agent A_j (see next phase).
   - If a node $w \in B(v,k)$ is already marked **(M,j)** by another agent $A_j \neq A_i$, then there are two cases:
     - If $i < j$ then A_i unmarks all the nodes he has already marked and continues the traversal of T_{G_i} (go to step 1).
     - Otherwise, A_i marks w with label **(M,i)** and continues the traversal of $T_{B_i(v,k)}$ (exploration of $B(v,k)$).
3. In the third phase, if A_i succeeds in marking all the nodes of $B(v,k)$ with **(M,i)**, then it traverses $T_{B_i(v,k)}$ once again in order to lock all the nodes in $B(v,k)$ by marking them with the extra label **(locked,i)**, i.e., the neighborhood ball is ready to be relabelled according to a rule. If the label of at least one node $w \in B(v,k)$ is not **(M,i)** then A_i unmarks all nodes marked with label **(M,i)** or those locked with label **(locked,i)** and continues the DFS-traversal of T_{G_i} (in other words, it reinitializes the $WB(w).c$ field of nodes $w \in B(v,k)$ marked by himself and goes to phase 1). When an agent A_i traverses $T_{B_i(v,k)}$ in order to lock the nodes, it also collects the topology of $B(v,k)$ at the same time in order to prepare executing a rule which avoids to make another traversal.
4. The fourth phase is executed if and only if the agent A_i has succeeded locking all nodes in $B(v,k)$. Hence, the agent traverses $B(v,k)$ for the fourth time in order to apply a rule. At the same time, it unlocks the nodes in $B(v,k)$. Finally, the agent continues the DFS-traversal of T_{G_i} and starts another cycle in the first phase again.

Note that an agent executes the second phase if and only if it finds a rule to execute after the first traversal in the first phase. Nevertheless, it may happen that in the fourth phase, no rule can be applied since the label of some nodes in $B(v,k)$ may change. In addition, a node w marked **(locked,i)** by an agent A_i can be updated *only* by agent A_i himself. In other words, if an agent j wants to mark node w, then he must wait until agent A_i unmarks it.

Correctness Analysis. First we argue that the relabeling done by an agent A_i locally on a ball $B(v,k)$ is correct. In fact, the relabeling of a ball is always done according to a valid relabeling rule described by the relabeling system given in input. Furthermore, whenever an agent is being relabeling a node w (or an edge) of a ball $B(v,k)$ in phase 4, *no* other agent could be relabeling a node w' in $B(v,k)$ at the same time. The latter property is quite easily proven, too, since an agent A_i begins relabeling a ball $B(v,k)$ in phase 4 if and only if the entire ball $B(v,k)$ has been marked with label **(locked,i)**, and nodes marked with label **(locked,i)** cannot be unlocked by other agents. Now, it remains the prove that the relabeling is globally correct.

Lemma 1. *Our framework is deadlock free, i.e., an agent cannot be blocked infinitely often in any node.*

Proof. **(Sketch)** The only case where an agent A_i may wait at a node w is when w is marked **(locked,j)** with $i \neq j$ (phase 2). In other words, the agent A_i may wait if the node w was locked by another agent A_j. From the description of phase 3 and 4, we are sure that node w will be unlocked by agent A_j. Since node w was locked by agent A_j this means that agent A_j has succeeded into applying phase 2, i.e., it has marked all nodes in the corresponding $B(v, k)$ ball with label **(M,j)**. Thus, agent A_j is applying either phase 3 or phase 4, while agent A_i is waiting in node w. From the description of phase 3 and 4, agent A_j is never blocked and it always unlocks the nodes in $B(v, k)$. □

The deadlock freeness property stated in the previous lemma is not sufficient to prove the correctness of our framework. In fact, it only ensures that the agents will not be blocked waiting for each others, but it does not ensure that the relabeling rules will be effectively applied. In the following, we argue that if a rule r has to be executed in any node v in order to continue the relabeling of the graph, then there exists an agent A_i that succeeds in relabeling $B(v, k)$ within a finite time according to r.

Note that in the first stage of our framework, an agent at node v always verifies whether a rule can be applied. Thus if an agent starts marking the nodes of some ball $B(v, k)$, then this means that some rule can be applied in $B(v, k)$. Now, observe that if an agent A_i fails preparing a ball $B(v, k)$ in phase 3 i.e., it fails locking the nodes of $B(v, k)$, then there must exist another agent A_j applying a rule in a ball $B(w, k)$ such that $j > i$ and $B(v, k) \cup B(w, k) \neq \varnothing$. The agent A_j may also fail preparing ball $B(w, k)$ because of a neighboring agent A_ℓ with a higher identifier. Using the deadlock freeness property we are sure that among all agents who passe the first phase, at least the agent having the highest identifier will succeed applying a rule. Now, suppose that some rule r has to be executed in some ball $B(v, k)$ in order to continue the relabeling of the graph, that is no other rule can be applied in any other node before rule r is applied in $B(v, k)$. Then, the agent A_i in the region G_i containing v will be the only agent who passes the first stage of our framework and will not be disturbed by other neighboring agents when preparing the ball $B(v, k)$ in phase 3.

Therefore we can conclude that our generic framework is correct; there is no deadlock, there is no rule starvation, and the relabeling performed by agents works as intended.

4 Conclusion

In this paper, we have argued that mobile agent paradigm is suitable for implementing distributed algorithms based on relabeling systems. By doing so, we are approaching a more comprehensive theory of distributed algorithms in which *(i)* relabeling systems are considered as a formal tool-box for designing algorithms and *(ii)* our mobile agent algorithms are considered as a practical tool-box for implementing them. Consequently, the mobile agent algorithms given in this paper can be considered as the key to a complete solution for designing, proving and implementing distributed algorithms using relabeling systems.

We believe that mobile agents will play an important role into bringing about a new theoretical and a practical approach to some classical distributed problems. Indeed, the abstraction provided by mobile agents allows both an encapsulation and a modularization of distributed computations over a network, which should lead to feasible solutions.

References

1. Angluin, D.: Local and global properties in networks of processors. In: 12th Symposium on theory of computing, pp. 82–93 (1980)
2. Bauderon, M., Métivier, Y., Mosbah, M., Sellami, A.: From local computations to asynchronous message passing systems. Technical Report, LaBRI, RR-1271-02 (2002)
3. Bauderon, M., Mosbah, M.: A unified framework for designing, implementing and visualizing distributed algorithms. In: Bottoni, P., Minas, M. (eds.) Graph Transformation and Visual Modeling Techniques (GT-VMT 2002), Barcelona, Spain. Electronic Notes in Theoretical Computer Science, vol. 72 (2003)
4. Chalopin, J., Godard, E., Métivier, Y., Ossamy, R.: Mobile agent algorithms versus message passing algorithms. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 187–201. Springer, Heidelberg (2006)
5. Chalopin, J., Paulusma, D.: Graph labelings derived from models in distributed computing. In: Fomin, F.V. (ed.) WG 2006. LNCS, vol. 4271, pp. 301–312. Springer, Heidelberg (2006)
6. Gruner, S.: Mobile agent systems and cellular automata. Technical Report 1400-06, LaBRI - University of Bordeaux 1 (2006)
7. Hibaoui, A.E., Métivier, Y., Robson, J., Saheb-Djahromi, N., Zemmari, A.: Analysis of a randomized dynamic timetable handshake algorithm. Technical Report 1402-06, LaBRI (2006)
8. Knirsch, P., Kreowski, H.-J.: A note on modeling agent systems by graph transformation. In: Münch, M., Nagl, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 79–86. Springer, Heidelberg (2000)
9. Litovsky, I., Métivier, Y., Sopena, E.: Different local controls for graph relabelling systems. Math. Syst. Theory 28, 41–65 (1995)
10. Litovsky, I., Métivier, Y., Sopena, E.: Graph relabelling systems and distributed algorithms. In: Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G. (eds.) Handbook of graph grammars and computing by graph transformation, vol. 3, pp. 1–56. World Scientific, Singapore (1999)
11. Litovsky, I., Métivier, Y., Zielonka, W.: On the Recognition of Families of Graphs with Local Computations. Inf. and Comp. 118(1), 110–119 (1995)
12. Métivier, Y., Saheb, N., Zemmari, A.: Randomized rendez vous. In: Mathematics and computer science: Algorithms, trees, combinatorics and probabilities, Trends in mathematics, pp. 183–194. Birkhäuser, Basel (2000)

13. Métivier, Y., Saheb, N., Zemmari, A.: Randomized local elections. Information Processing Letters 82, 313–320 (2002)
14. Métivier, Y., Saheb, N., Zemmari, A.: Analysis of a randomized rendez vous algorithm. Information and Computation 184, 109–128 (2003)
15. Ossamy, R.: An algorithmic and computational approach to local computations. In: Ph. D Thesis. LaBRI, University of Boredaux 1, France (December 2005)

A Decentralized Implementation of Mobile Ambients[*]

Fabio Gadducci and Giacoma Valentina Monreale

Dipartimento di Informatica, Università di Pisa
largo Pontecorvo 3c, I-56127 Pisa, Italy
gadducci@di.unipi.it, vale@di.unipi.it

Abstract. We present a graphical implementation for finite processes of the mobile ambients calculus. Our encoding uses unstructured (i.e., non hierarchical) graphs and it is sound and complete with respect to the structural congruence of the calculus (that is, two processes are equivalent iff they are mapped into isomorphic graphs). With respect to alternative proposals for the graphical implementation of mobile ambients, our encoding distinguishes the syntactic structure of a process from the activation order of a process components. Our solution faithfully captures a basic feature of the calculus (ambients can be nested and reductions are propagated across ambient nesting) and it allows to model the reduction semantics via a graph transformation system containing just three rules.

1 Introduction

Among recently introduced nominal calculi, mobile ambients [1] proved to be a popular specification formalism. Besides the standard operators for parallel composition and name restriction, it introduces the notion of *ambient*, i.e., a named environment where system evolutions may take place. The application domains of the calculus proved quite large, as witnessed by its use in system biology [2]. Moreover, the calculus inspired novel verification tools such as spatial logic [3], where the logical operators reflect the topological structure of a system.

As it is nowadays standard for nominal calculi, the operational semantics of mobile ambients is expressed by a set of (structural) axioms, plus a set of inference rules, inducing a reduction relation on processes. With respect to similar foundational calculi, though, those rules are rather complex, reflecting the rich structure of processes. Such a complexity is confirmed by the current (distributed) implementations for the calculus, as surveyed in [4]. Besides the usual problems of nominal calculi, linked with the use of message passing for addressing the so-called *magic matching* issue (the implicitly global choice for the subprocess where the reduction has to take place), the abstract machines have to "separate the logical distribution of ambients (the tree structure given by the syntax) from their physical distribution (the actual sites they are running on)" [4, p.117]: the states of the machine thus have to explicitly record the nesting of ambients.

[*] Research partially supported by the EU FP6-IST IP 16004 SEnSOria.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 115–130, 2008.

This paper presents a graphical implementation for the mobile ambients calculus that exploits the dichotomy between the tree structure of a process and the topology associated to its activation points, i.e., to those ambients that actually allow for the evolution of the subprocesses they contain. The encoding is then exploited to recast the complex operational semantics of the calculus by an easy and natural presentation via DPO rules, thus inheriting the wealth of tools and techniques for system analysis that are available for graph transformation.

It has been indeed since its origins in the late 1960's that the theory of graph transformation has been successfully applied in those areas where both static and dynamic modeling of systems by graphical structures play an important role. Graph rewriting has been used, for example, as a computational model for functional programming languages and for specifying distributed systems and visual languages. Only in recent years, though, graph rewriting has been used for process algebras specification: graphs model processes and graph transformation techniques simulate the reduction semantics of the calculus at hand.

The widespread acceptance of π-calculus made it the formalism usually considered when proposing a graphical framework for the description of concurrent and distributed systems (see e.g. [5] and the references therein), and even more so after the introduction of Milner's *bigraphs* [6]. However, the richness of mobile ambients may prove it a more suitable testbed for the use of graph-based formalisms in the description of process calculi.

The earliest proposal we are aware of is [7], from where our solution lifts the use of unstructured graphs in the encoding of processes. Besides introducing a slender graph syntax (according to [5]), the difference with the previous solution lies in the chosen representation of the states: the lack of records for the activation points in [7] forced the introduction of suitable rules for forwarding the information about "being enabled" to subprocesses. The presence of such spurious rules, possibly inhibiting the execution of some reductions, made the correspondence between graph transformations and process reductions only weakly sound and complete (see e.g. [7, Theorems 5.3 and 5.4]). Thus, it made less meaningful the application of standard tools from graph transformation (such as the different parallelism theorems) for discussing about properties of process evolution. Our chosen state representation allows instead for the reuse of such techniques, as surveyed in [5] for the π-calculus.

As far as other proposals for graphical implementation are concerned, we are aware of [8,9], using the so-called Synchronized Hyperedge Replacement framework, as well as of [10], in the mold of the standard DPO approach. Moreover, in [11] an encoding of mobile ambients by bigraphs is just outlined. As future work they plan to pursue this in detail, but still no encoding has been proposed.

In general, those SHR solutions are eminently hierarchical, meaning that each edge/label is itself a structured entity, and possibly a graph. More precisely, "sequential processes become edge labels: when an action is performed, an edge labelled by $M.P$ is rewritten as the graph corresponding to P" [8, p. 11]. This is unfortunate for calculi such as mobile ambients, where the topology of the systems plays a major role in discussing e.g. about distributed implementation and parallel execution of reductions [12]. Moreover, to the expressive power of the SHR framework corresponds a rather complex mechanism for rule application, which compares unfavourably with the basic DPO matching of our solution.

As far as [10] is concerned, the main difference with respect to our proposal is in the use of a process representation where the nesting of ambients is made explicit by the presence of suitable edges, instead of being implicit in the representation of each process, as in our proposal. The resulting encoding of processes is thus centralized, and this condition results in a complex set of graph transformation rules. Moreover, the encoding of process reduction is sound, yet not complete, thus not allowing the reuse of tools for system analysis that we mentioned earlier.

This paper is organized as follows. Section 2 briefly recalls the mobile ambients calculus. In Section 3 we introduce (typed hyper-)graphs and their extension with interfaces, while Section 4 presents the DPO approach to their rewriting. Then, in Section 5 we introduce a graphical encoding for processes of the mobile ambients calculus, and we present our first result, namely, that our encoding is sound and complete with respect to the structural congruence of mobile ambients. The main results of our paper are presented in Section 6, which introduces a graph transformation system for modelling the reduction semantics of mobile ambients. Finally, Section 7 concludes the paper.

2 Mobile Ambients

In this section we briefly recall the mobile ambients calculus [1]. In particular, we introduce the syntax and the reduction semantics for the finite, communication free fragment of the mobile ambients calculus.

Table 1 shows the syntax of the calculus. We assume a set $\mathcal{N}$ of *names* ranging over by $m, n, o, \ldots$ Also, we let $P, Q, R, \ldots$ range over the set $\mathcal{P}$ of processes.

Table 1. Syntax of mobile ambients

$P ::= \mathtt{0}, n[P], M.P, (\nu n)P, P_1 \vert P_2$	$M ::= \mathit{in}\ n, \mathit{out}\ n, \mathit{open}\ n$

The restriction operator $(\nu n)P$ binds n in P. A name n occurring in the scope of the operator (νn) is called *bound*, otherwise it is called *free*. We denote the set of free names of a process P by $fn(P)$. We adopt the standard notion of α-conversion of bound names and the standard definition for name substitution. We write $P\{^m/_n\}$ for the process obtained by replacing each free occurrence of n in P with m, and by α-converting the bound names to avoid conflicts with m.

The semantics of the mobile ambients calculus is given by a structural congruence between processes and a reduction relation. The *structural congruence*, denoted by $\equiv$, is the least relation on processes that satisfies the equations and

Table 2. Structural congruence

$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	$P\|\mathsf{0} \equiv P$
$P \equiv Q \Rightarrow M.P \equiv M.Q$	$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	$(\nu n)(P\|Q) \equiv P\|(\nu n)Q \quad$ if $n \notin fn(P)$
$P \equiv Q \Rightarrow P\|R \equiv Q\|R$	$(\nu n)m[P] \equiv m[(\nu n)P] \quad$ if $n \neq m$
$P\|Q \equiv Q\|P$	$(\nu n)\mathsf{0} \equiv \mathsf{0}$
$(P\|Q)\|R \equiv P\|(Q\|R)$	$(\nu n)P \equiv (\nu m)(P\{^m/_n\})$ if $m \notin fn(P)$

the rules shown in Table 2. The congruence relates processes which intuitively specify the same system, up-to a syntactical rearrangement of its components, and it is then used to define the operational semantics.

The *reduction relation*, denoted by →, describes the evolution of processes over time: $P \rightarrow Q$ means that P reduces to Q, that is, P can execute a computational step and it is transformed into Q. Table 3 shows the reduction rules. The first three rules are the only three axioms for the reduction relation. In particular, the *Red-In* rule enables an ambient n to enter a sibling ambient m. The *Red-Out* rule enables an ambient n to get out of its parent ambient m. Finally, the last axiom allows to dissolve the boundary of an ambient n. The *Red-Res*, *Red-Amb* and *Red-Par* rules say that reduction can occur underneath restriction, ambient and parallel composition, respectively. Finally, the last rule says that the reduction relation is closed under the structural congruence ≡.

As we said above, the structural congruence is used to define the reduction relation. It is possible to take into account different structural congruence relations. As in [7], we consider the structural congruence, denoted by $\equiv'$, defined as the least relation that satisfies the axiom in Table 4 and all the equations and the rules in Table 2, except the axiom $(\nu n)\mathsf{0} \equiv \mathsf{0}$. We denote by $\rightarrow'$ the reduction relation defined by the rules shown in Table 3, but closed under the structural congruence $\equiv'$. Note that considering the structural congruence $\equiv'$ does not change substantially the reduction semantics. Indeed, the equality introduced by the axiom in Table 4 holds in the only observational equivalence for mobile ambients that we are aware of, proposed in [13]. In particular, two

Table 3. Reduction relation

$n[in\ m.P\|Q]\|m[R] \rightarrow m[n[P\|Q]\|R]$	(Red-In)
$m[n[out\ m.P\|Q]\|R] \rightarrow n[P\|Q]\|m[R]$	(Red-Out)
$open\ n.P\|n[Q] \rightarrow P\|Q$	(Red-Open)
$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$	(Red-Res)
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$	(Red-Amb)
$P \rightarrow Q \Rightarrow P\|R \rightarrow Q\|R$	(Red-Par)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red-Cong)

Table 4. The additional axiom of the structural congruence

$(\nu n)M.P \equiv' M.(\nu n)P \quad \text{if } n \notin fn(M)$

processes that are congruent according to the axiom in Table 4 are also reduction barbed congruent.

3 Graphs and Graphs with Interfaces

This section presents some definitions concerning (hyper-)graphs, typed graphs and graphs with interfaces. It also introduces two operators on graphs with interfaces. We refer to [14] and [15] for a detailed introduction.

Definition 1 (Graphs). *A (hyper-)graph is a four-tuple $\langle V, E, s, t\rangle$ where V is the set of nodes, E is the set of edges and $s, t : E \to V^*$ are the source and target functions.*

From now on we denote the components of a graph G by V_G, E_G, s_G and t_G.

Definition 2 (Graph morphisms). *Let G, G' be graphs. A (hyper-)graph morphism $f : G \to G'$ is a pair of functions $\langle f_V, f_E\rangle$, such that $f_V : V_G \to V_{G'}$ and $f_E : E_G \to E_{G'}$ and they preserve the source and target functions, i.e. $f_V \circ s_G = s_{G'} \circ f_E$ and $f_V \circ t_G = t_{G'} \circ f_E$.*

The category of graphs is denoted by **Graph**. We now give the definition of typed graph [16], i.e., a graph labelled over a structure that is itself a graph.

Definition 3 (Typed graphs). *Let T be a graph. A* typed graph *G over T is a graph $|G|$ with a graph morphism $\tau_G : |G| \to T$.*

Definition 4 (Typed graph morphisms). *Let G, G' be typed graphs over T. A* typed graph morphism *$f : G \to G'$ is a graph morphism $f : |G| \to |G'|$ consistent with the typing, i.e., such that $\tau_G = \tau_{G'} \circ f$.*

The category of graphs typed over T is denoted by T-**Graph**. In the following, we assume a chosen type graph T.

To define the encoding for processes inductively, we need operations to compose graphs. So, we equip typed graphs with suitable "handles" for interacting with an environment. The following definition introduces graphs with interfaces.

Definition 5 (Graphs with interfaces). *Let J, K be typed graphs. A graph with input interface J and output interface K is a triple $\mathbb{G} = \langle j, G, k\rangle$, where G is a typed graph, $j : J \to G$ and $k : K \to G$ are injective typed graph morphisms, and they are called input and output morphisms, respectively.*

Definition 6 (Interface graph morphisms). *Let $\mathbb{G}, \mathbb{G}'$ be graphs with the same interfaces. An interface graph morphism $f : \mathbb{G} \Rightarrow \mathbb{G}'$ is a typed graph morphism $f : G \to G'$ between the underlying graphs that preserves the input and output morphisms.*

We denote a graph with input interface J and output interface K by $J \xrightarrow{j} G \xleftarrow{k} K$. If the interfaces J and K are *discrete*, i.e., they contain only nodes, we represent them by sets. With an abuse of notation, in the following we refer to the nodes belonging to the image of the input morphism as inputs. Similarly, we refer to the nodes belonging to the image of the output morphism as outputs. We often refer implicitly to a graph with interfaces as the representative of its isomorphism class. Moreover, we sometimes denote the class of isomorphic graphs and its components by the same symbol.

Now, we define two binary operators on graphs with discrete interfaces.

Definition 7 (Sequential and parallel composition). *Let $\mathbb{G} = J \xrightarrow{j} G \xleftarrow{k} K$ and $\mathbb{G}' = K \xrightarrow{j'} G' \xleftarrow{k'} I$ be graphs with discrete interfaces. Their sequential composition is the graph with discrete interfaces $\mathbb{G} \circ \mathbb{G}' = J \xrightarrow{j''} G'' \xleftarrow{k''} I$, where G'' is the disjoint union $G \uplus G'$, modulo the equivalence on nodes induced by $k(x) = j'(x)$ for all $x \in V_K$, and j'' and k'' are the uniquely induced arrows.*

Let $\mathbb{G} = J \xrightarrow{j} G \xleftarrow{k} K$ and $\mathbb{G}' = J' \xrightarrow{j'} G' \xleftarrow{k'} K'$ be graphs with discrete interfaces, such that $\tau_J(x) = \tau_{J'}(x)$ for all $x \in V_J \cap V_{J'}$ and $\tau_K(y) = \tau_{K'}(y)$ for all $y \in V_K \cap V_{K'}$. Their parallel composition is the graph with discrete interfaces $\mathbb{G} \otimes \mathbb{G}' = (J \cup J') \xrightarrow{j''} G'' \xleftarrow{k''} (K \cup K')$, where G'' is the disjoint union $G \uplus G'$, modulo the equivalence on nodes induced by $j(x) = j'(x)$ for all $x \in V_J \cap V_{J'}$ and $k(y) = k'(y)$ for all $y \in V_K \cap V_{K'}$, and j'', k'' are the uniquely induced arrows.

Intuitively, the sequential composition $\mathbb{G} \circ \mathbb{G}'$ is obtained by taking the disjoint union of the graphs underlying $\mathbb{G}$ and $\mathbb{G}'$, and gluing the outputs of $\mathbb{G}$ with the corresponding inputs of $\mathbb{G}'$. Similarly, the parallel composition $\mathbb{G} \otimes \mathbb{G}'$ is obtained by taking the disjoint union of the graphs underlying $\mathbb{G}$ and $\mathbb{G}'$, and gluing the inputs (outputs) of $\mathbb{G}$ with the corresponding inputs (outputs) of $\mathbb{G}'$. Note that both operations are defined on "concrete" graphs. However, their results do not depend on the choice of the representatives of their isomorphism classes.

A *graph expression* is a term over the syntax containing all graphs with discrete interfaces as constants, and parallel and sequential composition as binary operators. An expression is *well-formed* if all the occurrences of both sequential and parallel composition are defined for the interfaces of their arguments, according to Definition 7. The interfaces of a well-formed graph expression are computed inductively from the interfaces of the graphs occurring in it; the value of the expression is the graph obtained by evaluating all the operators in it.

4 Graph Rewriting

In this section we introduce the basic definitions for the DPO approach to the rewriting of (typed hyper-)graphs [17,18] and graphs with interfaces.

Definition 8 (Graph production). *A T-typed graph production $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of a production name p and of a span of graph morphisms $(L \xleftarrow{l} K \xrightarrow{r} R)$ with l mono in T-*__Graph__.

Definition 9 (Graph transformation system). *A T-typed graph transformation system (*GTS*) $\mathcal{G}$ is a pair $\langle T, P \rangle$, where T is a type graph and P is a set of productions, all with different names.*

Definition 10 (Graph derivation). *Let $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ be a T-typed graph production and let G be a T-typed graph. A match of p in G is a morphism $m_L : L \to G$. A direct derivation from G to H via production p and match m_L is a diagram as depicted in Figure 1, where (1) and (2) are pushouts in T-*__Graph__. *We denote this derivation by $p/m : G \Longrightarrow H$, for $m = \langle m_L, m_K, m_R \rangle$, or simply by $G \Longrightarrow H$.*

Before giving the definition of derivation between graphs with interfaces, we introduce the notion of track function.

Definition 11 (Track function). *Let p be a graph production and let $p/m : G \Longrightarrow H$ be a direct derivation, as in Figure 1. The track function $tr(p/m)$ associated with the derivation is the partial graph morphism $r^* \circ (l^*)^{-1} : G \to H$.*

$$\begin{tikzcd}
p: & L \arrow[d, "m_L"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "m_K"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m_R"] \\
 & G & D \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

Fig. 1. A direct derivation

The track function identifies the items before and after a derivation. It is used to give the definition of derivation between graphs with interfaces.

Definition 12 (Graph with interfaces derivation). *Let $\mathbb{G} = J \xrightarrow{j} G \xleftarrow{k} K$ and $\mathbb{H} = J \xrightarrow{j'} H \xleftarrow{k'} K$ be graphs with interfaces, and let $p/m : G \Longrightarrow H$ be a direct derivation such that the track function $tr(p/m)$ is total on $j(J)$ and $k(K)$. We say that $p/m : \mathbb{G} \Longrightarrow \mathbb{H}$ is a direct derivation of graphs with interfaces if $j' = tr(p/m) \circ j$ and $k' = tr(p/m) \circ k$.*

Intuitively, a derivation between graphs with interfaces is a direct derivation between the underlying graphs, such that inputs and outputs are preserved.

5 Graphical Encoding for Processes of Mobile Ambients

This section introduces a graphical encoding for processes of the mobile ambients calculus. First of all, we present a suitable type graph, depicted in Figure 2, and then we define an inductive encoding by exploiting the composition operators introduced in Definition 7. This corresponds to a variant of the usual construction of the tree for a term of an algebra: names are interpreted as variables, so they are mapped to leaves of the graph and can be safely shared.

As we can see, in the type graph there are three types of node. Intuitively, a node of type $\circ$ represents an ambient name, while a graph that has as roots a pair of nodes $\langle \diamond, \bullet \rangle$ represents a process. More precisely, the node of type $\diamond$ represents the activating point for reductions of the process represented by the graph. We need two different types of node to model processes by graphs because each graph has to model both syntactical dependences between the operators of the process and their activation dependences. Indeed, in mobile ambients the nesting of operators does not reflect the activation dependences between them, since reductions can occur inside ambients. So, in order to model a process, we use $\bullet$ nodes to model the syntactical dependences between the operators of the process, and $\diamond$ nodes to model their activation dependences.

Each edge of the type graph, except the *go* edge, simulates an operator of mobile ambients. Note that the *act* edge actually represents three edges, namely *in*, *out* and *open*. These three edges simulate the capabilities of the calculus, while the *amb* edge simulates the ambient operator. Notice that there are no edges to simulate the restriction operator and the parallel composition. Finally, the *go* edge is a syntactical device for detecting the "entry" point for the computation. We need it later to simulate the reduction semantics of mobile ambients. It allows to avoid that a reduction can occur underneath a capability operator.

All edges, except *go* edge, have the same type of source, that is the node list $\langle \diamond, \bullet \rangle$, while they have different types of target. In particular, the *amb* edge has the node list $\langle \bullet, \circ \rangle$ as target, while the *in*, *out* and *open* edges have the same type of target, i.e. the node list $\langle \diamond, \bullet, \circ \rangle$. Note that the three latter edges have a node $\diamond$ in the target. This node represents the activating point for the reductions of the continuation of the capability. It is different from the activating point of the

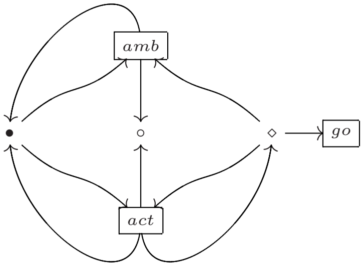


Fig. 2. The type graph (for $act \in \{in, out, open\}$)

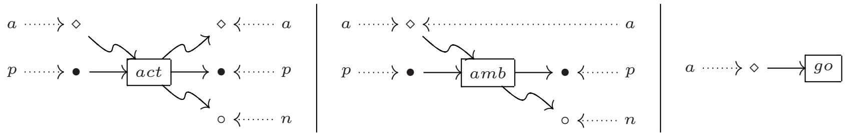


Fig. 3. Graphs act_n (with $act \in \{in, out, open\}$); amb_n; and go (left to right)

outermost capability operator, because the reductions of the continuation can occur only after the action regulated by the capability is executed. The *amb* edge instead has no node of type ◇ in its target. In fact, the activating point for the reductions of the process inside an ambient is the same activating point of the outermost ambient. This occurs because process reductions permeate ambients.

Now we define a class of graphs such that all processes can be encoded into an expression containing only those graphs as constants, and parallel and sequential composition as binary operators. Figures 3 and 4 depict these constant graphs. In particular, Figure 3 presents the graphs that correspond to the edges of the type graph. Figure 4 presents additional constant graphs needed for the formal presentation of our encoding. Note that in the graphs of the two figures we denote the input interface on the left and the output interface on the right. For example, the graph amb_n in the middle of Figure 3 has as input interface $\{a, p\}$ and as output interface $\{a, p, n\}$. Since a and p are constants used by our encoding, we assume that $p, a \notin \mathcal{N}$, while $n \in \mathcal{N}$ (where $\mathcal{N}$ is the set of names of mobile ambients).

In the following, we use $\mathtt{0}_{a,p}$ as shorthand for $\mathtt{0}_a \otimes \mathtt{0}_p$. Moreover, for a set of names Γ, we use id_Γ and new_Γ as shorthands for $\bigotimes_{n \in \Gamma} id_n$ and $\bigotimes_{n \in \Gamma} new_n$, respectively. Note that both last expressions are well defined, because the $\otimes$ operator is associative. The definition below introduces the encoding of processes into graphs with interfaces. It maps each finite process into a graph expression.

Definition 13 (Encoding for processes). *Let P be a finite process and let Γ be a set of names such that $fn(P) \subseteq \Gamma$. The encoding of P, denoted by $[\![P]\!]_\Gamma$, is defined by structural induction according to the rules in Table 5.*

Note that the encoding $[\![M.P]\!]_\Gamma$ represents the encoding of *in n.P*, *out n.P* and *open n.P*, while act_n represents the in_n, out_n and $open_n$ graphs, respectively.

Our encoding solves the α-conversion of restricted names by denoting them with ○ nodes that are not in the image of the variable morphism. The mapping

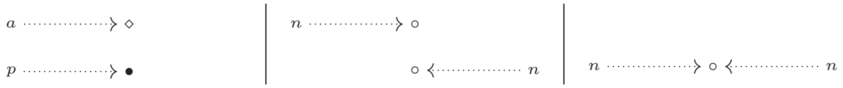


Fig. 4. Graphs $\mathtt{0}_a$ and $\mathtt{0}_p$; $\mathtt{0}_n$ and new_n; and id_n (top to bottom and left to right)

Table 5. Encoding for processes

$$
\begin{array}{lcll}
[\![\mathsf{0}]\!]_{\Gamma} & = & \mathsf{0}_{a,p} \otimes new_{\Gamma} & \\
[\![n[P]]\!]_{\Gamma} & = & amb_n \circ (id_n \otimes [\![P]\!]_{\Gamma}) & \\
[\![M.P]\!]_{\Gamma} & = & act_n \circ (id_n \otimes [\![P]\!]_{\Gamma}) & \\
[\![(\nu n)P]\!]_{\Gamma} & = & [\![P\{^m/_n\}]\!]_{\Gamma \cup \{m\}} \circ (\mathsf{0}_m \otimes id_{\Gamma}) & \text{for } m \notin \Gamma \\
[\![P|Q]\!]_{\Gamma} & = & [\![P]\!]_{\Gamma} \otimes [\![Q]\!]_{\Gamma} &
\end{array}
$$

is well defined in the sense that the result is independent of the choice of the name m in the rule for restriction.

The encoding $[\![P]\!]_{\Gamma}$, where Γ is a set of names such that $fn(P) \subseteq \Gamma$, is a graph with interfaces $(\{a, p\}, \Gamma)$. We note that the mapping is not surjective. In fact, there are graphs with interfaces $(\{a, p\}, \Gamma)$ that are not in the image of the encoding. The encoding of a process P is the graph $[\![P]\!]_{fn(P)}$. Below we give an example of encoding of a process.

Example 1. Let us consider the example below, originally proposed in [1], which illustrates a form of planned dissolution of an ambient n:

$$R = n[acid[out\ n.open\ n.P]|Q]|open\ acid.\mathsf{0}.$$

Figures 5 depicts the graph encoding $[\![R]\!]_{fn(R)}$. We represent the graph encodings for the processes P and Q by $\mathbb{G}_P$ and $\mathbb{G}_Q$, respectively. Moreover, for the sake of simplicity, we assume that the ambient names n and *acid* do not belong to the set of free names of P and Q. For the moment, the reader can ignore the edge labelled *go* and the labels of the nodes.

The leftmost edges, labelled *amb* and *open*, have the same roots, into which the names a and p are mapped. Those two edges represent the topmost operators of the two parallel components of the process. The edges in the middle, representing from left to right the operators *acid*[_] and *out n*._, respectively, are linked to the same $\diamond$ root. Intuitively, this means that they have the same activating point of the outermost ambient, and hence the reductions can permeate the two ambients n and *acid*. Instead, the rightmost edge, labelled *open*, has a different $\diamond$ source

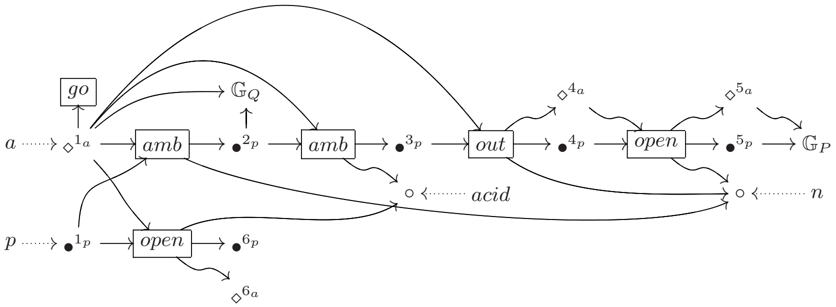


Fig. 5. Graph encoding for the process $n[acid[out\ n.open\ n.P]|Q]|open\ acid.\mathsf{0}$

that is the target of the edge *out*. Intuitively, this means that this capability *open* can be executed only after the action *out*.

The following theorem states that our encoding is sound and complete with respect to the structural congruence $\equiv'$.

Theorem 1. *Let P, Q be processes and let Γ be a set of names, such that $fn(P) \cup fn(Q) \subseteq \Gamma$. Then, $P \equiv' Q$ if and only if $[\![P]\!]_\Gamma = [\![Q]\!]_\Gamma$.*

6 A Graph Transformation System for Mobile Ambients

This section presents a graph rewriting system that models the reduction semantics of the mobile ambients calculus.

First of all, we enrich the encoding introduced in Definition 13 in order to avoid performing reductions underneath capability operators. To do this we attach a *go* edge to the $\diamond$ root node of each graph representing a process. The *go* edge is a syntactical device needed for detecting the "entry" point for the computation of the process. Given a process P and a set of names Γ such that $fn(P) \subseteq \Gamma$, its enriched encoding is the graph $[\![P]\!]_\Gamma \otimes go$. We denote it by $[\![P]\!]_\Gamma^{go}$.

Figure 6 presents the rules of the GTS $\mathcal{R}_{amb}$, which simulates the reduction semantics $\rightarrow'$ introduced in Section 2. The GTS $\mathcal{R}_{amb}$ contains just three rules, namely p_{in}, p_{out} and p_{open}. They simulate the *Red-In*, *Red-Out* and *Red-Open* reductions, respectively. The action of the three rules is described by the node identifiers. These identifiers are of course arbitrary. They correspond to the actual elements of the set of nodes and are used to characterize the track function.

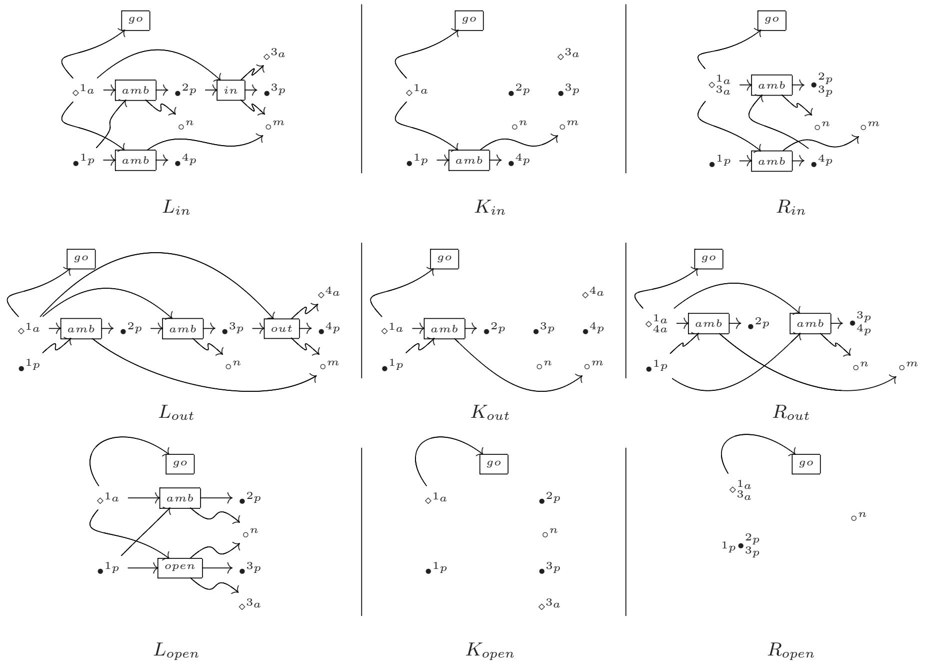
Now we discuss the rules of the GTS $\mathcal{R}_{amb}$. In order to give a clear explanation of the rule actions, we denote by amb_n an *amb* edge having in its target a $\circ$ node identified by n. Let us consider the p_{in} production. The p_{in} rule preserves the amb_m edge, removes the amb_n edge and re-creates this last one under amb_m. Note that, after the reduction, the *in* edge disappears and the nodes identified by 2_p and 3_p are coalesced. Moreover, the $\diamond$ node under the *in* prefix is activated.

The p_{out} rule preserves the amb_m edge and removes the amb_n edge, too. It also re-creates this last one with the same source nodes of amb_m. Analogously to p_{in}, after the reduction the *out* edge disappears and the nodes identified by 3_p and 4_p are coalesced. Moreover, the $\diamond$ node under the *out* prefix is activated.

Finally, the p_{open} production removes both *amb* and *open* edges. After the reduction, all the $\diamond$ nodes and all the $\bullet$ nodes are coalesced, respectively. Furthermore, the $\diamond$ node under the *open* prefix is activated.

It seems noteworthy that three rules suffice for recasting the reduction semantics of mobile ambients. That is possible for two reasons. First, the closure of reduction with respect to contexts is obtained by the fact that graph morphisms allow the embedding of a graph within a larger one. Second, no distinct instance of the rules is needed, since graph isomorphism takes care of the closure with respect to structural congruence, and interfaces of the renaming of free names.

We now introduce the main theorems of the paper. They state that our encoding is sound and complete with respect to the reduction relation $\rightarrow'$.



L_{in} K_{in} R_{in}

L_{out} K_{out} R_{out}

L_{open} K_{open} R_{open}

Fig. 6. The rewriting rules p_{in}, p_{out} and p_{open} (top to bottom)

Theorem 2 (Soundness). *Let P, Q be processes and Γ a set of names, with $fn(P) \subseteq \Gamma$. If $P \rightarrow' Q$, then $\mathcal{R}_{amb}$ entails a direct derivation $[\![P]\!]_{\Gamma}^{go} \Longrightarrow [\![Q]\!]_{\Gamma}^{go}$.*

Intuitively, a process reduction is simulated by applying a rule on an enabled event, that is, by a match covering a subgraph with the *go* operator on top.

Theorem 3 (Completeness). *Let P be a process and Γ a set of names, with $fn(P) \subseteq \Gamma$. If $\mathcal{R}_{amb}$ entails a direct derivation $[\![P]\!]_{\Gamma}^{go} \Longrightarrow \mathbb{G}$, then there exists a process Q, such that $P \rightarrow' Q$ and $\mathbb{G} = [\![Q]\!]_{\Gamma}^{go}$.*

The correspondence holds since a rule is applied only if there is a match that covers a subgraph with the *go* operator on the top. This allows the occurrence of reductions inside activated ambients, but not inside capabilities. In fact, if an *amb* operator is activated, that is, its $\diamond$ source node has an outgoing *go* edge, then all operators inside it are activated too, because they have the same source node $\diamond$ as the *amb* operator. Differently, a reduction can not occur inside the outermost capability, because the activating point for the reductions of the continuation of a capability is different from the activating point of the outermost capability.

The following example shows the application of some rules of the GTS $\mathcal{R}_{amb}$ to the graph encoding for the process considered in Example 1.

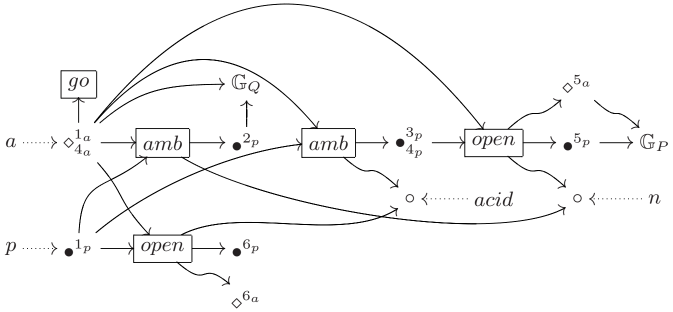


Fig. 7. Graph encoding $[\![acid[open\ n.P]|n[Q]|open\ acid.0]\!]^{go}_{fn(R)}$

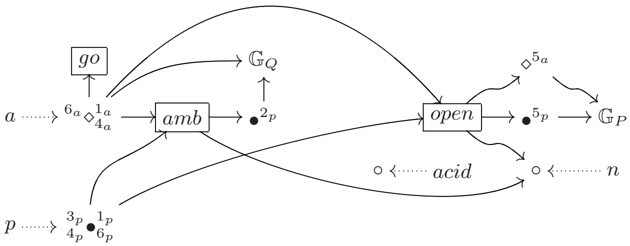


Fig. 8. Graph encoding $[\![open\ n.P|n[Q]]\!]^{go}_{fn(R)}$

Example 2. Let us consider again the process shown in Example 1:

$$R = n[acid[out\ n.open\ n.P]|Q]|open\ acid.0.$$

The graphical encoding for the process above is depicted in Figure 5. The nodes are labelled in order to denote the track function of the derivation. The edge labelled *go* denote the entry point for the computation of the process. Note that the two edges *amb*, the edge *out* and the outermost edge *open* can be involved in a reduction step because they have the same activation node with an outgoing *go* edge. Instead, the rightmost edge, labelled *open*, is not activated, since its $\diamond$ source is the target of another edge.

The application of the p_{out} rule to the graph in Figure 5 results in the graph in Figure 7, which is the encoding for the process $acid[open\ n.P]|n[Q]|open\ acid.0$. In fact, this rewriting step simulates the transition $n[acid[out\ n.open\ n.P]|Q]|open\ acid.0 \rightarrow' acid[open\ n.P]|n[Q]|open\ acid.0$.

Now, we can apply the p_{open} rule to the graph in Figure 7, and we obtain the graph in Figure 8. Note that this rewriting step simulates the transition $acid[open\ n.P]|n[Q]|open\ acid.0 \rightarrow' open\ n.P|n[Q]$.

Finally, by applying the p_{open} rule to the graph in Figure 8, we get the graph in Figure 9. The derivation mimics the reduction $open\ n.P|n[Q] \rightarrow' P|Q$.

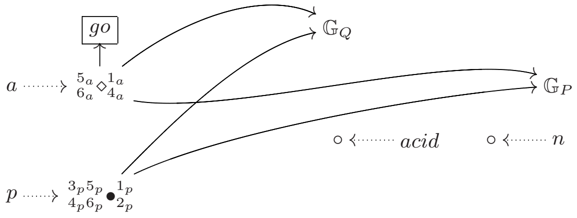

go
$\mathbb{G}_Q$
a
$^{5_a}_{6_a}\diamond^{1_a}_{4_a}$
$\mathbb{G}_P$
$acid$
n
p
$^{3_p 5_p}_{4_p 6_p}\bullet^{1_p}_{2_p}$

Fig. 9. Graph encoding $[\![P|Q]\!]^{go}_{fn(R)}$

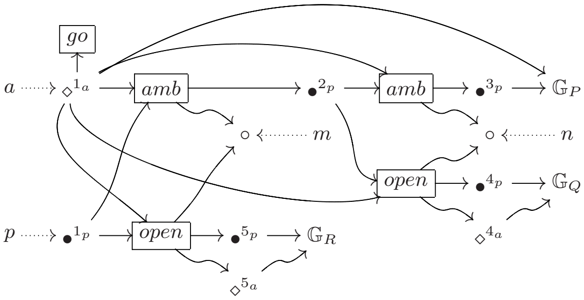


Fig. 10. Graph encoding for the process $m[n[P]|open\ n.Q]|open\ m.R$

The rewriting steps shown in the example above simulate a sequence of process reductions all occurring on the top. The next example shows instead how our encoding is able to simulate process reductions nested inside ambients.

Example 3. Let S be the process $m[n[P]|open\ n.Q]|open\ m.R$, previously proposed in [7]. The encoding $[\![S]\!]^{go}_{fn(S)}$ is depicted in Figure 10. For the sake of simplicity, we assume that the names m and n do not belong to the free names of P, Q and R.

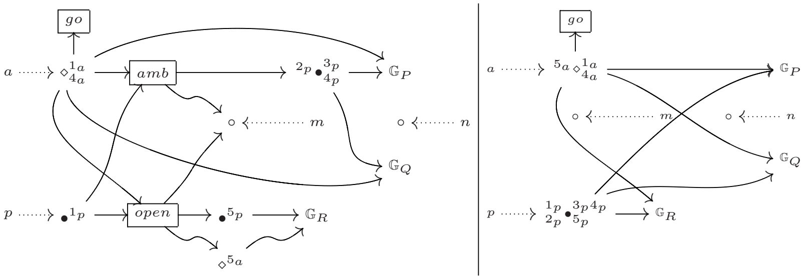


Fig. 11. Graph encodings $[\![m[P|Q]|open\ m.R]\!]^{go}_{fn(S)}$ and $[\![P|Q|R]\!]^{go}_{fn(S)}$ (left to right)

The rewriting step, resulting in the graph on the left of Figure 11, is obtained by applying the p_{open} rule and it simulates the process reduction nested inside the ambient m, namely, $m[n[P] | open\ n.Q] | open\ m.R \rightarrow' m[P|Q] | open\ m.R$. The application of the p_{open} rule to the graph on the left results instead in the graph on the right of Figure 11. This rewriting step mimics the transition $m[P|Q] | open\ m.R \rightarrow' P|Q|R$. Note that, as discussed in [7, Section 5.3], these two reductions are parallel independent, hence they can be executed in any order, obtaining two derivations that differ only in the scheduling of the two steps. The two derivations thus correspond to the same graph process [16].

7 Conclusions and Further Work

We presented an encoding for finite, communication-free processes of mobile ambients into graphs, proving its soundness and completeness with respect to the operational semantics of the calculus. Differently from alternative proposals, it is based on unstructured graphs and standard DPO approach tools, thus allowing for the reuse of analysis techniques from the graph transformation mold, along the lines of graphical encodings presented in [5,7]. Most importantly, our encoding has the ability both to model the syntactic structure of a process and to keep track of its activation points, that is, of those ambients where reductions may actually take place. Therefore, it allows a simply and faithful modeling of the reduction semantics of mobile ambients.

For the sake of space, we discarded from our presentation the communication primitives of the calculus, as well as recursive expressions: both could be tackled along the lines of the solution in [5]. The article also offers a list of applications for the graphical encoding of π-calculus [5, Section 8], which could be immediately lifted to our encoding of mobile ambients. They range from the use of graphs for verifying system properties expressed by spatial logic, to the use of the *borrowed contexts* [19] approach for deriving a labelled transition system for mobile ambients. It should be remarked that this array of applications is possible thanks to our graphical implementation, where the tree structure of a process is decoupled from its activation points. Moreover, the lack of activation rules (needed instead in [7]) guarantees a direct correspondence between process reductions and graph derivations, thus allows for the simultaneous execution of reductions, possibly nested inside ambients as well as sharing some resources.

Our next step is the study of the labelled transition system for mobile ambients, which can be obtained by exploiting the borrowed context technique discussed above. The borrowed approach proved to be able to characterize strong bisimulation for a simple process calculi, namely, Milner's CCS [20]. Since we are aware only of the process equivalence proposed by [13], the analysis of the graph-based equivalence could prove pivotal in validating that proposal.

References

1. Cardelli, L., Gordon, A.: Mobile ambients. Theor.Comp.Sci. 240(1), 177–213 (2000)
2. Regev, A., Panina, E., Silverman, W., Cardelli, L., Shapiro, E.: Bioambients: an abstraction for biological compartments. Theor.Comp.Sci. 325(1), 141–167 (2004)
3. Caires, L., Cardelli, L.: A spatial logic for concurrency (part I). Information and Computation 186(2), 194–235 (2003)
4. Hirschkoff, D., Pous, D., Sangiorgi, D.: An efficient abstract machine for safe ambients. Journal of Logic and Algebraic Programming 71(2), 114–149 (2007)
5. Gadducci, F.: Graph rewriting for the π-calculus. Mathematical Structures in Computer Science 17(3), 407–437 (2007)
6. Milner, R.: Pure bigraphs: Structure and dynamics. Information and Computation 204(1), 60–122 (2006)
7. Gadducci, F., Montanari, U.: A concurrent graph semantics for mobile ambients. In: Brookes, S., Mislove, M. (eds.) Mathematical Foundations of Programming Semantics. Electr.Notes in Theor.Comp.Sci., vol. 45. Elsevier Science, Amsterdam (2001)
8. Ferrari, G., Montanari, U., Tuosto, E.: A LTS semantics of ambients via graph synchronization with mobility. In: Restivo, A., Ronchi Della Rocca, S., Roversi, L. (eds.) ICTCS 2001. LNCS, vol. 2202, pp. 1–16. Springer, Heidelberg (2001)
9. Cenciarelli, P., Talamo, I., Tiberi, A.: Ambient graph rewriting. In: Martì-Oliet, N. (ed.) Rewriting Logic and its Applications. ENTCS, vol. 117, pp. 335–351. Elsevier, Amsterdam (2005)
10. Mylonakis, N., Orejas, F.: Another fully abstract graph semantics for the ambient calculus. Graph Transformation for Verification and Concurrency (2007)
11. Jensen, O., Milner, R.: Bigraphs and mobile processes. Technical Report 580, Computer Laboratory, University of Cambridge (2003)
12. Levi, F., Sangiorgi, D.: Mobile safe ambients. ACM Trans. Program. Lang. Syst. 25(1), 1–69 (2003)
13. Merro, M., Zappa Nardelli, F.: Behavioral theory for mobile ambients. Journal of the ACM 52(6), 961–1023 (2005)
14. Bruni, R., Gadducci, F., Montanari, U.: Normal forms for algebras of connections. Theor.Comp.Sci. 286(2), 247–292 (2002)
15. Corradini, A., Gadducci, F.: An algebraic presentation of term graphs, via gs-monoidal categories. Applied Categorical Structures 7, 299–331 (1999)
16. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae 26(3/4), 241–265 (1996)
17. 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, vol. 1, pp. 163–245. World Scientific, Singapore (1997)
18. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1, pp. 95–162. World Scientific, Singapore (1997)
19. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Mathematical Structures in Computer Science 16(6), 1133–1163 (2006)
20. Bonchi, F., Gadducci, F., König, B.: Process bisimulation via a graphical encoding. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 168–183. Springer, Heidelberg (2006)

Network Applications of Graph Bisimulation

Pietro Cenciarelli[1], Daniele Gorla[1], and Emilio Tuosto[2]

[1] Dip. di Informatica, "Sapienza" Università di Roma, Italy
[2] Dept. Computer Science, Univerity of Leicester, UK

Abstract. *Synchronising Graphs* is a system of parallel graph transformation designed for modeling process interaction in a network environment. We propose a theory of *context-free* synchronising graphs and a novel notion of bisimulation equivalence which is shown to be a congruence with respect to graph composition and node restriction. We use this notion of equivalence to study some sample network applications, and show that our bisimulation equivalence captures notions like functional equivalence of logical switches, equivalence of channel implementations and level of fault tolerance of a network.

1 Introduction

Synchronising Graphs (SG) is a system of parallel graph transformation designed for modeling process interaction in a network environment. The system is inspired by [8], and it stems from the *Synchronized Hyperedge Replacement* (SHR) of [10], with which it has been compared in [4]. In the SG model, as in SHR, hyperedges represent agents, or software components, while nodes are thought of as communication channels, synchronisation points or, more generally, network communication infrastructure. The idea that hypergraphs may interact by synchronising action and co-action pairs at specific synchronisation points (the nodes) is quite intuitive, while the flexibility of the model in representing diverse network topologies and communication protocols makes SG fit as a common semantic framework for interpreting different calculi. We followed this idea in [3], where *Mobile Ambients* [2] and the *distributed CCS* of [22] (without restriction) were both modeled in SG by using a common recursive architecture.

Here, we explore an orthogonal issue, namely the *behavioural equivalence* of SG. Indeed, such equivalences are often sought in the theory of concurrency for proving the conformance of an implementation with respect to a specification or for achieving a sort of compositionality in the semantics. If we identify the meaning of a process in its abstract behaviour (which is traditionally considered its bisimulation equivalence class), compositionality requires that, when equivalent processes (e.g. a specification and an implementation) are plugged into the same context, they behave in the same way. This amounts to proving that bisimulation equivalence is a congruence. However, although such results are abundant in the literature for process calculi, not so for graph rewriting, where system behaviour is typically context dependent.

To our knowledge, the most strictly related notions of behavioural equivalence proposed for related systems of graph transformation are [13] and [14]. The first paper proposes a behavioural equivalence for a model called *synchronised graph rewriting*; as

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 131–146, 2008.

pointed out by the authors, this equivalence is rather coarse, in that it is not able to distinguish graphs with different degrees of parallelism. In this paper, we meet their challenge for a finer notion and propose one capable of detecting parallelism (it is indeed possible to implement in SG Plotkin's parallel or). The behavioural equivalence of [14] refers to a system of graph rewriting, the SHR of [10], which differs from ours in the mathematical presentation of graphs, in their LTS and, more importantly, in the proof theory. Syntax is pervasive in SHR, which is more deeply rooted in the field of process calculi, of which it shares notions such as structural congruence and name binding. Nodes are treated as names are in process calculi. Unlike in SG, no semantic difference can be made between two nodes beside them being distinct. Not always so in graph rewriting, where transformations may depend on attachment to specific nodes. As shown in Section 3 (example 5), such dependency may cause non-compositional behaviour. Hence, while compositionality is to be expected in hyperedge replacement [14], not so for SG, which allows, as many graph rewrinting systems [7], context-dependent specifications. Thus, we characterise the theories of synchronising graphs, called *context-free* where compositionality holds. A natural notion of bisimulation equivalence is introduced to capture their abstract behaviour, and proven a congruence in any context-free theory. A similar result is presented in [14] for hyperedge replacement by exploiting the sytactic presentation of graphs and referring to results obtained in [23] in the context of structural operational semantics. Here we provide a *direct* proof, which relies on no syntax and sheds light on the meta-theoretical properties of our system (Lemmas 1 and 2). While imposing on axiom formats built-in features of SHR, our result is no special case of that in [14], as discussed in the conclusions.

Then, we use our framework for modeling four network applications where the proposed notion of bisimulation equivalence is shown to capture interesting properties. In the first application we consider network implementations of *logical switches*. Here bisimulation equivalence corresponds to functional equivalence, in that equivalent networks have an identical input-output behaviour. Then, we consider network implementations of *communication channels*, where information items can travel in parallel. Bisimulation equivalence is shown to capture the notion of (static) channel capacity. In the third scenario, we refine the previous model by introducing *node charges*, that are consumed upon passage of information. This feature cruises in several wireless applications and becomes a crucial issue in "extreme" applications like the *Smart Dust* [24]. Here bisimulation equivalence implies identity of dynamic capacity but provides a finer notion of observational equivalence which can be employed for net optimisation. Finally, we study the impact of *failure* in a communication net and show that bisimulation equivalence characterises exactly the degree of *fault tolerance*, or *robustness*, of a net.

The paper is structured as follows. In Section 2, we present the general model of SG. In Section 3 we investigate the possible sources of context dependency in SG and focus on *context-free* SG; a novel notion of bisimulation equivalence is then introduced and shown to be a congruence. Section 4 presents the applications. Section 5 concludes the paper by discussing related work and by hinting at current and future research. For space reasons, all proofs have been omitted; the interested reader can find them in the on-line version at `http://www.dsi.uniroma1.it/~gorla/publications.htm`.

2 Synchronising Graphs

Let $\mathcal{N}$ be a set of *nodes*, which we consider fixed throughout. A *graph* (E, G, R) consists of a set E of *hyperedges*, an attachment function $G : E \to \mathcal{N}^*$ and a set $R \subseteq |G|$ of nodes, called *restricted*, where $|G| = \{x \in \mathcal{N} \mid \exists\, e \in E \text{ s.t. } x \in G\,e\}$ is the set of nodes of the graph. When clear from the context or when not important, we shall write a graph by simply specifying its attachment function. When $G\,e = x_1x_2 \ldots x_n$ (we shall often abbreviate $x_1x_2 \ldots x_n$ as $\boldsymbol{x}$), we call n the *arity* of e and say that the i-th *tentacle* of e is attached to x_i. We denote by $\mathit{res}\,(G)$ the set of restricted nodes of G, and by $\mathit{fn}\,(G)$ the set $|G| - \mathit{res}\,(G)$ of *free* nodes. We write $e(\boldsymbol{x})$ for an hyperedge such that $G\,e = \boldsymbol{x}$.

Let $\mathit{Act} = \{a, b, \ldots\} \cup \{\overline{a}, \overline{b}, \ldots\}$ be a set of *actions*; we call $\overline{a}$ the *co-action* of a, and intend a by $\overline{\overline{a}}$. A *pre-transition* is a triple (G, Λ, H), written $G \xrightarrow{\Lambda} H$ (or just Λ for short), where $\Lambda \subseteq \mathcal{N} \times \mathit{Act} \times \mathcal{N}^*$ is a relation, while G and H are graphs, called respectively the *source* and the *destination* of Λ. Intuitively, $(x, a, \boldsymbol{y}) \in \Lambda$ expresses the occurrence of action a at node x, which can be thought as a communication channel, while the elements of $\boldsymbol{y}$, called *objects*, are thought of as arguments. When $\boldsymbol{y}$ is the empty sequence ϵ, (x, a, ϵ) is written (x, a).

In SG the occurrence of both $(x, a, \boldsymbol{y})$ and $(x, \overline{a}, \boldsymbol{z})$ in Λ is called a *synchronisation*, and it corresponds to the *silent* action τ of most process calculi. Synchronising hyperedges may exchange information. This is implemented in SG by unifying the lists $\boldsymbol{y}$ and $\boldsymbol{z}$ of objects, which are required to be of the same length. Only two agents at a time may synchronise at one node. Moreover, if an action occurs at a restricted node, then it *must* synchronise with a corresponding co-action, as we consider *observable* the unsynchronised actions. A restricted node may be "opened" by unifying it with an argument of an observable action, or with a node which is not restricted.

Notation. If $\varphi \subseteq A \times B$ is a relation and $a \in A$, we write $\varphi\, a$ the set $\{b \in B : (a, b) \in \varphi\}$. The *domain* of φ is the set $\mathit{dom}\,(\varphi) = \{a \in A : \exists\, b \in B\,.\,(a, b) \in \varphi\}$. A function $f : A \to B$ is said to *agree* with φ when $f\,x \in \varphi\,x$, for all $x \in A$. If φ is an equivalence relation, $[x]_\varphi$ is the equivalence class of an element x, which we write $[x]$ when φ is understood. A *unifier* of φ is a function f which agrees with φ as above and such that $f[x]$ is a singleton, for all x.

If $f : \mathcal{N} \to \mathcal{N}$ is a function on nodes and (E, G, R) is a graph, we write fG the graph (E, fG, fR) obtained by substituting all nodes x in G with fx. More precisely, for all $e \in E$, if $G\,e = x_1 \ldots x_n$ then $(fG)\,e = fx_1 \ldots fx_n$.

Given a pre-transition $G \xrightarrow{\Lambda} H$, we denote by $|\Lambda|$ the set $|G| \cup |H|$ and by $\mathit{res}\,(\Lambda)$ the set $\mathit{res}\,(G) \cup \mathit{res}\,(H)$. By $\mathit{obj}\,(\Lambda)$ we denote the set $\{y \in \mathcal{N} : \exists\,(x, a, \boldsymbol{y}) \in \Lambda \text{ such that } y \in \boldsymbol{y}\}$. We omit parentheses and braces when listing the elements of Λ above a transition arrow.

An *action set* is a relation $\Lambda \subseteq \mathcal{N} \times \mathit{Act} \times \mathcal{N}^*$ such that, for all nodes x, $\Lambda\,x$ has *at most* two elements and, when so, it is of the form $\{(a, \boldsymbol{y}), (\overline{a}, \boldsymbol{z})\}$, where $\boldsymbol{y}$ and $\boldsymbol{z}$ are vectors of identical length. Given an action set Λ, we denote by $\overset{\Lambda}{=}$ the smallest equivalence relation on nodes such that, if $(x, a, y_1y_2 \ldots y_n)$ and $(x, \overline{a}, z_1z_2 \ldots z_n)$ are in Λ, then $y_i \overset{\Lambda}{=} z_i$, for $i = 1 \ldots n$. By a slight abuse, we say that a function agrees with (or unifies) an action set Λ to mean that it agrees with (unifies) the relation $\overset{\Lambda}{=}$. Arguments of unsynchronised actions are called *dangling*. More precisely, we call dangling in Λ the elements of the set $dng(\Lambda) = \{z \in obj(\Lambda) \;:\; \Lambda\, x = \{(a, \boldsymbol{y})\}$ and $z \overset{\Lambda}{=} y,$ for some x and $y \in \boldsymbol{y}\}$.

Definition 1. *A* transition *is a pre-transition* $G \xrightarrow{\Lambda} H$ *such that:*

1. Λ *is an action set such that* $\mathrm{dom}(\Lambda) \cup obj(\Lambda) \subseteq |G|$*;*
2. *if a node* x *is restricted in* G *then* $\Lambda\, x$ *is not a singleton;*
3. *if* $x \in |H|$*, then* $x \in fn(H)$ *if and only if* $x \in fn(G) \cup dng(\Lambda)$*.*
4. $H = \rho H$ *for some unifier* ρ *of* Λ *such that* $\rho\, x \in fn(G)$ *for all* $x \in fn(G)$*.*

Condition 1 expresses the *locality* of action: graphs can only act upon their own nodes. By this condition, for example, the pre-transition $e(x) \xrightarrow{x,a,y} d(y)$, legal in SHR, is not a transition, because $y \notin |e(x)|$. A consequence of 1 and 3 is that all free nodes in the destination of a transition must occur in the source. Hence, while $e(x) \xrightarrow{\emptyset} \nu y\, d(y)$ is a legal transition, $e(x) \xrightarrow{\emptyset} d(y)$ is not. This rules *ownership* of nodes: the access to a new channel is only acquired via synchronisation. Condition 4 enforces fusions. It also grants a privilege to the free nodes when they are fused with the bound, which allows $\nu y\, e(x\, y) \xrightarrow{\substack{x,\bar{a},x\\ x,a,y}} d(x)$ and forbids $\nu y\, e(x\, y) \xrightarrow{\substack{x,\bar{a},x\\ x,a,y}} d(y)$. This restriction is not essential for the theory of synchronising graphs while it simplifies the meta-theory without loss of generality.

In SG, synchronisation is subject to a non-interference condition: two transitions can be synchronised provided they are disjoint and they share no restricted nodes. Formally, $G \xrightarrow{\Lambda} H$ and $F \xrightarrow{\Theta} K$ are said to be *non-interfering*, written $\Lambda \,\#\, \Theta$, whenever $\Lambda \cap \Theta = \emptyset$ and $res(\Lambda) \cap |\Theta| = res(\Theta) \cap |\Lambda| = \emptyset$. It is an easy check that the only nodes two non-interfering transitions may have in common are the free nodes in their sources.

The rules of the system of synchronising graphs are given below. The *composite* of two graphs (E, G, R) and (D, F, S), written $G|F$, is defined when E and D are disjoint and moreover $res(G) \cap |F| = res(F) \cap |G| = \emptyset$; when so, $G|F$ is the graph $(E \cup D, G + F, R \cup S)$, where $G + F$ is the attachment function mapping $e \in E$ to $G\, e$ and $d \in D$ to $F d$. We let $\nu x\, G$ denote the graph $(E, G, R \cup \{x\})$ when $x \in |G|$, while $\nu x\, G = G$ otherwise.

$$[\,\mathrm{sync}\,]\ \frac{G \xrightarrow{\Lambda} H \qquad F \xrightarrow{\Theta} K}{G|F \xrightarrow{\Lambda \cup \Theta} \rho(H|K)}\ \Lambda \,\#\, \Theta \text{ and } \rho \text{ unifies } \Lambda \cup \Theta$$

$$[\,\mathrm{open}\,]\ \frac{G \xrightarrow{\Lambda} H}{\nu x\, G \xrightarrow{\Lambda} H}\ x \in dng(\Lambda) \qquad\qquad [\,\mathrm{res}\,]\ \frac{G \xrightarrow{\Lambda} H}{\nu x\, G \xrightarrow{\Lambda} \nu x\, H}\ x \notin dng(\Lambda)$$

A *theory* of synchronising graphs is a set of transitions which is closed under the inference rules. The smallest theory including a given set $\mathcal{A}$ of transitions is said to be *generated* by the *axioms* in $\mathcal{A}$.

Note that inference rules assume, as implicit side condition, that the conclusion be a transition. Hence, for example, the rule [sync] does not apply to $\nu y\, e(x\, y) \xrightarrow{x,a,y} f(y)$ and $\nu z\, d(x\, z) \xrightarrow{x,\bar{a},z} g(z)$ because the conclusion would violate condition 3 of definition 1.

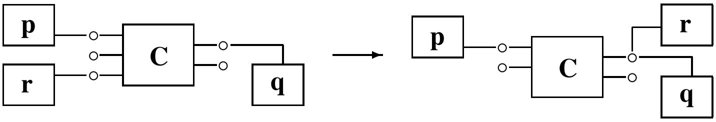


Fig. 1. A non-deterministic commuter

Also note that, differently from the π-calculus [19], we do not have a "close" rule to close the scope of a restricted name after having opened it via an "open" rule. This is related to the fact that every inference in SG can be rewritten in a sort of 'normal form' where all the applications of [res] and [open] needed to infer the judgement come after all the applications of [sync] (see Lemmata 1 and 2 later on). This is similar to the presentations of the LTS for the π-calculus that include structural equivalence: in those cases, a "close" rule is omitted because redundant. Thus, for example, we can build an inference for the graph $\nu xG \mid \nu yH$ where the two parallel components synchronise, assuming that $G \xrightarrow{z,a,x} G'$ and that $H \xrightarrow{z,\bar{a},y} H'$: indeed, $\nu xG \mid \nu yH$ is just another (but exactly identical) way of writing the graph $\nu x(\nu y(G \mid H))$, that reduces to, e.g., $\nu x(\nu y(G' \mid H'\{x/y\}))$ after a transition $\{(z, a, x), (z, \bar{a}, y)\}$.

Example 1 (A non-deterministic commuter). Consider a system consisting of several input and output *sockets*. The system, which we shall call *non-deterministic commuter*, acts by non-deterministically connecting client processes (possibly attached to an input socket) with one of the output sockets (where server processes may be attached). Connections are established one at a time. Figure 1 depicts a commuter **C** with three input and two output sockets. A client process **r** is being connected with a server **q**.

Non-deterministic commuters can be engineered in SG by assembling simple components (edges) of the form $in\,(x\,u)$ and $out\,(u\,y)$, representing input and output sockets respectively. Clients are meant to be attached to the x node of a socket, while servers are attached to y. The node u represents an internal communication channel of the system. As elsewhere in the paper, we may use the same name to denote distinct edges representing components of the same kind. For example, $in\,(x\,u) \mid in\,(z\,u)$ will denote a graph with *two* edges, each representing an input socket. Then, ignoring the two unused sockets of **C** (viz., the second input socket and the second output socket), the initial state of the commuter is represented by the graph $\nu u\,(in\,(x\,u) \mid in\,(z\,u) \mid out\,(u\,y))$. The system's behaviour is specified by the following two transitions, where a and $\overline{a}$ represent the input and output actions respectively:

$$in\,(x\,u) \xrightarrow{u,a,x} \emptyset \qquad\qquad out\,(u\,y) \xrightarrow{u,\overline{a},y} out\,(u\,y)$$

To be precise, these are to be considered as axiom *schemes*, and we assume one axiom of the first kind for each input socket and one of the second for each output. In the present example, we further assume that any hyperedge can perform a passive (empty) transition to itself. Then, ignoring **p** and its socket, the transition of figure 1 is obtained

as $\quad r(z) \mid q(y) \mid \nu u\,(in\,(z\,u) \mid out\,(u\,y)) \xrightarrow{\substack{u,\overline{a},y \\ u,a,z}} r(y) \mid q(y) \mid \nu u\,out\,(u\,y).$ □

To conclude the presentation of SG, we give two meta-theoretical lemmata showing that any transition in a given theory can be inferred by a canonical derivation where all applications of [sync] precede [res] and [open].

Notation. Let $G \xrightarrow{\Lambda} H$ and $F \xrightarrow{\Theta} K$ be transitions; we denote by $\Lambda * \Theta$ the set of transitions of the form $G|F \xrightarrow{\Lambda \cup \Theta} \rho(H|K)$ obtained by synchronising Λ and Θ with [sync]. Clearly, $\Lambda * \Theta$ is empty when Λ and Θ interfere. The expression $(\Lambda * \Theta) * \Phi$ stands for $\bigcup_{\Xi \in \Lambda * \Theta} (\Xi * \Phi)$. Similarly, we let $\nu x\, \Lambda$ be the transition which results from restricting Λ on x by an application of [res] or [open]. The expression $\nu x\, (\Lambda * \Theta)$ denotes the set of transitions of the form $\nu x\, \Xi$ with $\Xi \in \Lambda * \Theta$.

Lemma 1. *Let Λ and Θ be transitions and let x occur unrestricted in the source of Λ. Then, $(\nu x\, \Lambda) * \Theta \subseteq \nu x\, (\Lambda * \Theta)$.*

The opposite inclusion does not hold: $\nu y\, e(x\,y) \mid \nu z\, d(x\,z) \xrightarrow{\substack{x,\overline{a},z \\ x,a,y}} \nu y\, (h(y) \mid k(y))$ is included in $\nu y\, (\Lambda * \Theta)$ where Λ is $e(x\,y) \xrightarrow{x,a,y} h(y)$ and Θ is $\nu z\, d(x\,z) \xrightarrow{x,\overline{a},z} k(z)$, while $(\nu y\, \Lambda) * \Theta$ is empty because the result of applying [sync] to $\nu y\, \Lambda$ and Θ violates condition 3 of definition 1.

Lemma 2. *Synchronisation is associative:* $(\Lambda * \Theta) * \Xi = \Lambda * (\Theta * \Xi)$.

3 Context-Free Theories and Behavioural Equivalence

One of the aims of the present paper is to characterise the theories of synchronising graphs in which the behaviour of a graph is not affected by the context. The following examples will clarify this concept.

Example 2. In the theory generated by a unique axiom $e|d \xrightarrow{\emptyset} \emptyset$, the two processes e and d, considered in isolation, have the same behaviour: none of them can move. However, if set in the context $[\,\_\,]|d$, the two processes exhibit quite different behaviour, as $e|d$ can move while $d|d$ cannot.

Example 3. In the theory generated by a unique axiom $\nu x\, e(x) \xrightarrow{\emptyset} \emptyset$, the process $e(x)$ cannot move, thus exhibiting the same catatonic behaviour as the empty process $\emptyset$. However, when set in a context $\nu x\, [\,\_\,]$ where x is restricted, $\nu x\, e(x)$ can move while $\nu x\, \emptyset = \emptyset$ cannot.

Example 4. In the theory generated by the four axioms $h(x\,y) \xrightarrow{\substack{x,\overline{a},x \\ x,a,y}} \emptyset$, $d \xrightarrow{\emptyset} d$, $e(x\,y) \xrightarrow{\emptyset} e(x\,y)$ and $e(x\,x) \xrightarrow{x,a} \emptyset$, $e(x\,y)$ behaves just like the process d, cycling forever over itself. However, when put in parallel with $h(x\,y)$, $e(x\,y)$ yields a trace which $h(x\,y) \mid d$ does not have: $h(x\,y)|e(x\,y) \xrightarrow{\substack{x,\overline{a},x \\ x,a,y}} e(x\,x) \xrightarrow{x,a} \emptyset$.

Example 5. In the theory generated by the three axioms $h(x\,y) \xrightarrow{\substack{x,\overline{a},x \\ x,a,y}} \emptyset$, $d \xrightarrow{\emptyset} d$ and $e(x) \xrightarrow{\emptyset} e(x)$, the processes $e(x)$ and d have the same behaviour. However, when put in parallel with $h(x\,y)$, $e(x)$ yields a transition to a catatonic state, namely $e(y)$, which $h(x\,y)\,|\,d$ cannot reach.

These are in fact the *only* possible sources of context dependency in a theory of synchronising graphs. This is shown in the present section by providing a notion of bisimulation equivalence on graphs and then proving that, in any theory generated by axioms including no transitions of the kind described in the examples, the proposed equivalence is a congruence with respect to restriction and parallel composition.

In this section we abandon the brute force notion of node substitution in a graph G adopted in the previous section and denote by hG the graph obtained by applying a substitution h to the *free* nodes of G, while restricted nodes are suitably renamed so as to avoid capture. This simplifies the treatment while remaining consistent with the theory developed so far. In particular, note that the new interpretation of $\rho(H|K)$ in [sync] does not alter the set of derivable transitions.

An *instance* of a transition $G \xrightarrow{\Lambda} H$ is a transition of the form $hG \xrightarrow{h\Lambda} \rho\, hH$ where h is a node substitution $\mathcal{N} \to \mathcal{N}$ and ρ is a unifier of $h\Lambda$. A *production* is a transition whose source consists of a single hyperedge $e(\boldsymbol{x})$, where all components of $\boldsymbol{x}$ are distinct and none of them is restricted. A theory of synchronising graphs is called *context-free* when it is generated by all the instances of a given set of productions. Note that the constraints that productions are asked to satisfy prevent the first three examples of context dependency to occur, while the use of *all* their instances for generating the theory accounts for the fourth example.

We now move to the definition of our behavioural equivalence; to this aim, we call *parameters* the elements of the set $\mathcal{P} = \mathcal{N} \times Act \times \mathbb{N}$. Intuitively, a parameter (x, a, i) is an abstraction over the i-th argument y_i of an action $(x, a, \boldsymbol{y})$. We call *observations* the elements of the set $\mathcal{O} = \mathcal{N} \cup \mathcal{P}$. Given an action set Λ, the relation $\overset{\Lambda}{=}$ extends to a relation $\overset{\Lambda}{=}_o$ on observations that is the smallest equivalence relation containing $\overset{\Lambda}{=}$ such that $(x, a, i) \overset{\Lambda}{=}_o (x, \overline{a}, i)$ and moreover $(y, b, j) \overset{\Lambda}{=}_o z$ if $(y, b, z_1 \ldots z_j \ldots z_n) \in \Lambda$ and $z = z_j$.

Not all pairs of $\overset{\Lambda}{=}_o$ are observable. The set $obs(\Lambda)$ of *observables* of a transition $G \xrightarrow{\Lambda} H$ consists of its observable nodes, the set of which we denote by $|\Lambda|_o$, together with the parameters of unsynchronised actions: $obs(\Lambda) = |\Lambda|_o \cup \{(x, a, i) \in \mathcal{P} \,:\, \Lambda\, x = \{(a, \boldsymbol{y})\}$ and $0 \le i \le |\boldsymbol{y}|\}$, where $|\Lambda|_o = \{x \in fn(G) \,:\, x \in dom(\Lambda)$ or x is dangling or $x \overset{\Lambda}{=} y \neq x$ for some $y \in fn(G)\}$. Note that, by the definition of $|\Lambda|_o$, while x is observable in $e(x\,y) \xrightarrow{\substack{x,\overline{a},y \\ x,a,y}} H$, y is not because, although it is free in $e(x\,y)$, "self-fusion" has no bearing on the interacting environment. The *observable part* of the relation $\overset{\Lambda}{=}_o$, written $\overset{\Lambda}{\simeq}$, is the equivalence relation obtained by restricting $\overset{\Lambda}{=}_o$ to $obs(\Lambda)$; thus, we let $p \overset{\Lambda}{\simeq} q$ if and only if $p \overset{\Lambda}{=}_o q$ and $p, q \in obs(\Lambda)$.

Definition 2. *Two transitions $G \xrightarrow{\Lambda} H$ and $F \xrightarrow{\Theta} K$ are called* equivalent *when $\overset{\Lambda}{\simeq} = \overset{\Theta}{\simeq}$ and* $\{x \in fn(G) \,:\, |\Lambda\, x| = 2\} = \{y \in fn(F) \,:\, |\Theta\, y| = 2\}$.

In our study of behavioural equivalence we follow a standard practice in process algebra where alpha-equivalent terms are considered as identical. In our context this amounts to defining behaviour on *classes* of alpha-equivalent graphs, that is graphs which are identical up to renaming of restricted nodes. We shall call such classes *abstract graphs*, and write them in bold, $\mathbf{G}$. Any theory of synchronising graphs yields a transition system of abstract graphs which includes $\mathbf{G} \xrightarrow{\Lambda} \mathbf{H}$ if and only if $G \xrightarrow{\Lambda} H$ is in the theory, for some $G \in \mathbf{G}$ and $H \in \mathbf{H}$.

Notice that the notion of free names can be extended to abstract graphs since, for every G and G' in $\mathbf{G}$, it holds that $fn(G) = fn(G')$; thus, the notion of equivalent transitions scales to abstract graphs as well. A transition $\mathbf{G} \xrightarrow{\Lambda} \mathbf{H}$ is said to be *fair* with an abstract graph $\mathbf{F}$ when none of the nodes in $obj(\Lambda) \setminus fn(\mathbf{G})$ is free in $\mathbf{F}$.

Definition 3. *A* simulation *is a binary relation $\mathcal{S}$ on abstract graphs such that* $\mathbf{G}\,\mathcal{S}\,\mathbf{F}$ *implies that, for all transitions* $\mathbf{G} \xrightarrow{\Lambda} \mathbf{H}$ *fair with* $\mathbf{F}$*, there exists a transition* $\mathbf{F} \xrightarrow{\Theta} \mathbf{K}$ *such that Λ and Θ are equivalent, and* $\mathbf{H}\,\mathcal{S}\,\mathbf{K}$*. An abstract graph* $\mathbf{G}$ *is* simulated *by a graph* $\mathbf{F}$*, written* $\mathbf{G} \prec \mathbf{F}$*, if there exists a simulation $\mathcal{S}$ such that* $\mathbf{G}\,\mathcal{S}\,\mathbf{F}$*. A* bisimulation *is a symmetric simulation. Two abstract graphs* $\mathbf{G}$ *and* $\mathbf{F}$ *are called* bisimulation equivalent*, written* $\mathbf{G} \sim \mathbf{F}$*, when they are related by a bisimulation.*

Notice that the fairness condition asked for $\mathbf{G} \xrightarrow{\Lambda} \mathbf{H}$ in the previous definition is standard in name-passing calculi, e.g. the π-calculus [19].

Composition and restriction extend to abstract graphs. In particular, $\nu x\,\mathbf{G}$ is $[\nu x\,G]_\alpha$, for some $G \in \mathbf{G}$ such that $x \notin res(G)$, while $\mathbf{G}|\mathbf{F}$ is $[G|F]_\alpha$, for some $G \in \mathbf{G}$ and $F \in \mathbf{F}$ such that $G|F$ is defined. Note that the above definitions do not depend on specific choices of G and F. A relation $\mathcal{R}$ on abstract graphs is called a *congruence* when $\mathbf{G}\,\mathcal{R}\,\mathbf{F}$ implies $\nu x\,\mathbf{G}\,\mathcal{R}\,\nu x\,\mathbf{F}$ and $\mathbf{G}|\mathbf{H}\,\mathcal{R}\,\mathbf{F}|\mathbf{H}$, for all x and $\mathbf{H}$.

Theorem 1. *Bisimulation equivalence is a congruence.*

Proof (Sketch). The result is proven by showing that the symmetric relation

$$\mathcal{R} = \{(\nu \boldsymbol{x}\,(\mathbf{G}|\mathbf{U}), \nu \boldsymbol{x}\,(\mathbf{F}|\mathbf{U}))\ :\ \mathbf{G} \prec \mathbf{F}\}$$

is a simulation. Then, closure under parallel composition is obtained by letting $\boldsymbol{x}$ be the empty vector; closure under name restriction is obtained by letting $\mathbf{U}$ be $[(\emptyset;\emptyset;\emptyset)]_\alpha$. Lemmas 1 and 2 are used. See appendix for detail. □

4 Network Applications

4.1 A Non-deterministic Commuter (Example 1 Continued)

The internal communication channel of the non-deterministic commuter can be implemented by a local network without affecting the observable behaviour of the system. We build such internal infrastructure by means of simple components, called *connectors*, of the form $c(u_1u_2v)$. Connectors echo the information received from u_1 (call it the *input* node) over u_2 (the *output* node) using a *service* node v for the matching. Once v has served its purpose, a new service node is created. In symbols:

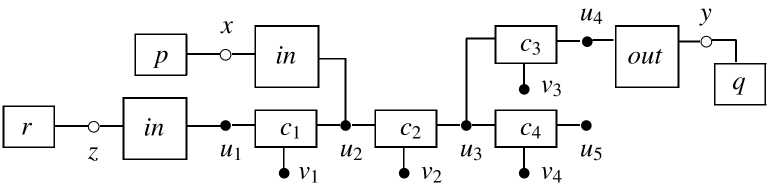


Fig. 2. An implementation of the commuter in Figure 1 (we draw labeled boxes for hyperedges and bullets for nodes; the latter are solid when restricted and clear otherwise; tentacles are represented by lines connecting hyperedges with nodes)

$c(u_1u_2v) \xrightarrow{\substack{u_1,\overline{a},v\\ u_2,a,v}} vw\, c(u_1u_2w)$. The internal channel of the commuter in Figure 1 can be implemented by the net G of four connectors in Figure 2. In symbols, grouping all indexed names into vectors:

$$\begin{aligned} G = \nu\, \boldsymbol{u}\, \boldsymbol{v}\; & p(x) \mid q(y) \mid r(z) \mid in\,(z\,u_1) \mid in\,(x\,u_2) \\ & \mid c_1(u_1u_2v_1) \mid c_2(u_2u_3v_2) \mid c_3(u_3u_4v_3) \mid c_4(u_3u_5v_4) \mid out\,(u_4y). \end{aligned}$$

With this implementation, the transition in Example 1 is simulated by an equivalent transition $G \xrightarrow{\Lambda} H$, where (by ignoring all the unused sockets and connectors) $H = r(y) \mid q(y) \mid \ \nu\, \boldsymbol{u}\, \boldsymbol{w}\, (c_1(u_1u_2w_1) \mid c_2(u_2u_3w_2) \mid c_3(u_3u_4w_3) \mid out\,(u_4y))$ and Λ is $\{(u_1, a, z), (u_1, \overline{a}, v_1), (u_2, a, v_1), (u_2, \overline{a}, v_2), (u_3, a, v_2), (u_3, \overline{a}, v_3), (u_4, a, v_3), (u_4, \overline{a}, y)\}$.

In general, a graph made of sockets and connectors behaves like a non-deterministic commuter when it is a tree (that is, connected and acyclic) in which output sockets are attached by their first tentacle, input sockets by their second, no connector is attached by its service node, and moreover there exists a node, called *pivot*, that may split the graph into two (possibly disconnected) subgraphs, one including all the input and the other all the output sockets. In our implementation, nodes u_2, u_3 and u_4 are all pivotal. Of course, in the absence of a pivot, the internal infrastructure may allow for parallel connections, which are not contemplated in the specification of Example 1.

Proposition 1. *Any abstract graph* **G** *satisfying the conditions above is bisimulation equivalent to the abstract graph corresponding to the non-deterministic commuter obtained by deleting all the connectors from* **G** *and attaching all sockets to the pivot node.*

4.2 Functional Equivalence

We now consider a more general kind of non-deterministic commuters, allowing multiple connections to occur at once. Hence, the internal structure of a commuter can now be any acyclic graph of connectors where all nodes are restricted. Input sockets are attached by their second tentacle, while their first is attached to a free node called *input node*; and dually for output sockets and *output nodes*.

A *connection* in a commuter C is a path from an input to an output node of C. A set of *disjoint* such connections (i.e. no node is shared by two connections in the set) is called a *service* of C. If s is a service, we write $\hat{s}$ the *partial* function from the input to the output nodes of C such that $\hat{s}(x) = y$ if and only if there exists a connection from x

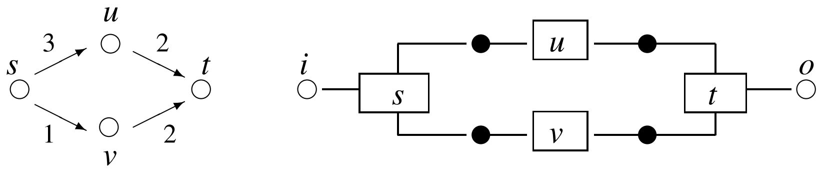


Fig. 3. A channel with maximum flow 3 and its SG representation

to y in s. We say that two commuters are *functionally equivalent* when, for each service s of one, there exists a service r of the other such that $\hat{s} = \hat{r}$, and vice-versa.

Proposition 2. *Two non-deterministic commuters are functionally equivalent if and only if their alpha-equivalence classes are bisimulation equivalent.*

4.3 The *Maximum Flow* in a Net

Consider an application where a *sender* sends discrete pieces of information, called *items*, to a *receiver*. The communication infrastructure is represented by a directed acyclic graph $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges. An edge (u, v) is an *input* for v and an *output* for u. We assume that the graph features a *unique* vertex with no input edges, called *source*, representing the sender. Similarly, the receiver is represented by a unique vertex with no input edges, called *target*.

We further endow each edge with a *capacity*, that is an upper bound to the number of items it can transmit *at a time*: n items can travel simultaneously through an edge provided its capacity is not less than n. No items are lost during transmission, and all items in input to a node are immediately presented in output in equal number. Hence, if the sender feeds the net with n items simultaneously, n items are received at once by the receiver, provided the edge capacities are not exceeded. This gives rise to the notion of *network flow* and of *maximum network flow* [6], i.e. the maximum number of items which can be simultaneously fed to the net. We call *channels*, and use metavariables A, B... to denote them, graphs $(\mathcal{V}, \mathcal{E})$ as above, endowed with a function $c : \mathcal{E} \to \mathbb{N}$ assigning to each edge an integer capacity, which we assume *strictly* greater than 0.

A *flow* in a channel A is a function $f : \mathcal{E} \to \mathbb{N}$ such that $f(u, v) \leq c(u, v)$ and $\sum_u f(u, v) = \sum_w f(v, w)$, for all $v \in \mathcal{V}$ except for source and target. The value of a flow f at the source s is $f(s) = \sum_u f(s, u)$, while $f(v) = \sum_u f(u, v)$ for all other vertices v. Clearly, $f(s) = f(t)$, and we call this number the *value* of f in A. A *positive* flow is one with value strictly greater than 0. The *maximum flow* of A, written $\phi\,(A)$, is the greatest value of a flow in A.

A channel $A = (\mathcal{V}, \mathcal{E}, c)$ is modelled by a synchronising graph $\hat{A}$ as follows. The nodes of $\hat{A}$ are the elements of $\mathcal{E} \cup \{i, o\}$, where i and o are called respectively the *input* and *output* nodes. All nodes in $\hat{A}$ are restricted, except i and o. Hyperedges are the vertices of $\mathcal{V}$. They are attached to nodes as follows: $v\,(\boldsymbol{x}\boldsymbol{y})$ represents a vertex $v \notin \{s, t\}$ where $\boldsymbol{x}$ is a vector including all input edges of v and $\boldsymbol{y}$ all outputs. Source and target are respectively $s\,(i\,\boldsymbol{y})$ and $t\,(\boldsymbol{x}\,o)$. Figure 3 represents a channel and its representation as a synchronising graph.

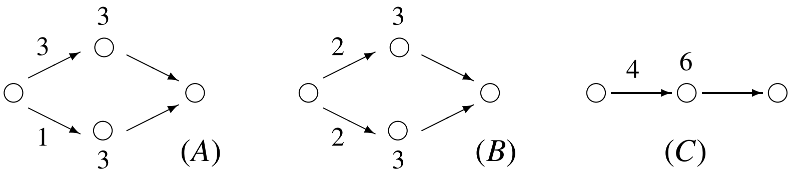


Fig. 4. Three consumable channels. Omitted capacities and energy charges are assumed high enough as to not influence the flow dynamics

The theory of channels features actions of the form n and $\overline{n}$, where $n \in \mathbb{N}$, and no parameters. It is generated by all axioms of the form:

$$v\,(x_1 \dots x_n\, y_1 \dots y_m) \xrightarrow{\substack{x_1\, h_1, \dots, x_n\, h_n \\ y_1\, \bar{k}_1, \dots, y_m\, \bar{k}_m}} v\,(x_1 \dots x_n\, y_1 \dots y_m),$$

where $\sum_{i=1}^{n} h_i = \sum_{j=1}^{m} k_j$ and, assuming $c(i) = c(o) = \infty$, $h_i \leq c(x_i)$, $k_j \leq c(y_j)$ for all nodes x_i and y_j. It is easy to see that A supports a flow of value k if and only if $\hat{A}$ has a transition Λ whose only observable actions are $\Lambda(i) = \{k\}$ and $\Lambda(o) = \{\overline{k}\}$. The following result shows that, in this simple model, bisimulation equivalence captures precisely the notion of maximum flow.

Proposition 3. *Let $\hat{A} \in \mathbf{A}$ and $\hat{B} \in \mathbf{B}$; then, $\mathbf{A} \sim \mathbf{B}$ if and only if $\phi\,(A) = \phi\,(B)$.*

4.4 The *Dynamic Flow* of a Net

In real applications the nodes of a wireless network are often supplied with a *finite* amount of energy which is consumed in routing information. The *Smart Dust* [24], where nodes are *motes* of 1mm diameter, is an extreme example of energy-sensitive application. We give a simple account of such scenarios by charging the channels of Section 4.3 with consumable energy and studying their behaviour.

A *consumable channel* $A = (\mathcal{V}, \mathcal{E}, c, \eta)$ is a channel as above, endowed with an energy function $\eta : \mathcal{V} \to \mathbb{N}$. A flow in A is just as in Section 4.3, with the additional requirement that $f(v) \leq \eta\,(v)$ for all $v \in \mathcal{V}$. The energy inside a channel decreases at each flow. For simplicity, we shall assume that the passage of one information item through a vertex consumes one energy unit. Then, the energy dynamics is described by a transition system over consumable channels with transitions $(\mathcal{V}, \mathcal{E}, c, \eta) \xrightarrow{k} (\mathcal{V}, \mathcal{E}, c, \eta'\,)$, whenever the channel to the left admits a flow f of value k, and $\eta'(v) = \eta\,(v) - f(v)$ for all $v \in \mathcal{V}$. Clearly, $A \xrightarrow{k} A'$ implies $k \leq \phi\,(A)$.

A *computation* of a channel A is a sequence of transitions $A \xrightarrow{k_1} A_1 \dots \xrightarrow{k_n} A_n$, which we shorten as $\langle k_1, \dots, k_n \rangle$. The *dynamics* $\Phi\,(A)$ of a channel A is the set of all its computations. The channels depicted in Figure 4 all have a maximum flow of 4. However, while (B) and (C) have same dynamics, not so for (A) as it does not admit a computation $\langle 4, 2 \rangle$ while the others do. In Section 4.5 we distinguish channels such as (B) and (C) by introducing the notion of *robustness*.

As before, we model a consumable channel A by a synchronising graph $\hat{A}$, and relate the dynamics of the former with the observable behaviour of the latter. $\hat{A}$ is defined just as in Section 4.3 except that the hyperedge $v_n(\boldsymbol{x}\,\boldsymbol{y})$ corresponding to a vertex v is now labelled by the energy $n = \eta(v)$. The axioms $v_n(\boldsymbol{x}\,y) \xrightarrow{\cdots} v_{n'}(\boldsymbol{x}\,y)$ are as in Section 4.3 with the additional requirement that, writing p the value $\sum_{j=1}^{m} k_j$ of the transition, $p \leq n$ and $n' = n - p$.

Channel dynamics do not provide a good notion of behavioural equivalence for consumable channels. For example, consider the channels (B) and (C) in Figure 4: by always taking the upper path, after $\langle 2, 1 \rangle$ (B) becomes a net that cannot transmit three information items simultaneously (because of the capacity bound on its lower edge), whereas (C) can always perform $\langle 2, 1, 3 \rangle$. Bisimulation equivalence captures such differences in channel behaviours; moreover, it also yields a technique for proving that two channels have identical dynamics, i.e. the same set of traces:

Proposition 4. *Let* $\hat{A} \in \mathbf{A}$ *and* $\hat{B} \in \mathbf{B}$*; then,* $\mathbf{A} \sim \mathbf{B}$ *implies* $\Phi(A) = \Phi(B)$.

4.5 Network Robustness

The channels (B) and (C) of Figure 4 have the same dynamics in the world described in Section 4.4 but not in a more realistic setting where vertices may *fail*. In such a case B is to be considered more *robust* than C. Robustness is ususally defined as the minimum number of faults that would block a net. Here we show a model where bisimulation equivalence captures precisely this notion of robustness.

Since the interplay between robustness and dynamic flow is subtle, we shall make the simplifying assumption that every node has infinite energy and every edge has capacity 1. Since flow values are not of interest here, we further assume that channels may pass at most one information item at a time. We let $r(A)$ denote the minimum number of nodes that must be removed to disconnect a channel A (i.e. source from target).

We represent the behaviour of a faulty channel A by augmenting the theory of synchronising graphs of Section 4.4 with new axioms for failure. As anticipated, all hyperedges of $\hat{A}$ are now labelled by ∞, and failure is represented by a sudden drop of $v_\infty(\boldsymbol{x}\,\boldsymbol{y})$ to $v_0(\boldsymbol{x}\,\boldsymbol{y})$. The axioms of flow are just as in Section 4.4 with the only difference that $c(i) = c(o) = 1$. For modeling failure, we introduce a new action † and the following axiom schemes, where we write v when the energy of the vertex (∞ or 0) is irrelevant:

$$v_\infty(\boldsymbol{x}\,\boldsymbol{y}) \xrightarrow{\substack{x_i,\dagger \\ y_j,\dagger}} v_0(\boldsymbol{x}\,\boldsymbol{y}) \qquad s(\boldsymbol{i}\,\boldsymbol{y}) \xrightarrow{\substack{i,0 \\ y_j,\bar{\dagger}}} s(\boldsymbol{i}\,\boldsymbol{y}) \qquad t(\boldsymbol{x}\,o) \xrightarrow{\substack{x_i,\bar{\dagger} \\ o,\bar{0}}} t(\boldsymbol{x}\,o)$$

$$v(\boldsymbol{x}\,\boldsymbol{y}) \xrightarrow{\substack{x_i,\bar{\dagger} \\ y_j,\dagger}} v(\boldsymbol{x}\,\boldsymbol{y}) \text{ for } v \notin \{s,t\} \qquad v(\boldsymbol{x}\,\boldsymbol{y}) \xrightarrow{\substack{x_i,\dagger \\ y_j,\bar{\dagger}}} v(\boldsymbol{x}\,\boldsymbol{y}) \text{ for } v \notin \{s,t\}$$

The first axiom accounts for the failure of v while the two axioms to the bottom are to transmit such an information respectively towards the source and the target. Notice that, in the first axiom, we can freely pick any $x_i \in \boldsymbol{x}$ and $y_j \in \boldsymbol{y}$ since every x_i and y_j lie in a path from s to t (because we work with connected graphs). By the remaining two axioms, source and target *hide* occurrences of † by issuing 0 on the free input and output nodes. Hence, failures are not *explicitly* observable. Note that we engineered our model as to admit one failure at a time. This allows us to test robustness by *counting* the number of steps that a channel requires in order to die. Simultaneous failures could of course have been modeled, at the cost of exposing the failure action † over the free nodes i and o.

The following result shows that, in this model, the robustness of a faulty channel is captured precisely by the notion of bisimulation equivalence.

Proposition 5. *Let $\hat{A} \in \mathbf{A}$ and $\hat{B} \in \mathbf{B}$; then, $\mathbf{A} \sim \mathbf{B}$ if and only if $r(A) = r(B)$.*

5 Conclusions

Synchronised graph rewriting has been proposed as a unifying semantic framework for process calculi [12,10,15,3]; to fulfill this project, graphs must be endowed with an abstract notion of behaviour. In this paper we do so by introducing a notion of bisimulation equivalence for a system of context-free synchronising graphs and by proving it a congruence with respect to parallel composition and node restriction. Bisimulation equivalence can be used to prove the correctness of system implementations, or (dually) of optimisation steps. For example, we have developed an application where the specification of a simple component, called non-deterministic commuter, is shown to be equivalent to an implementation in which the internal communication channel is replaced by a local net.

Bisimulation techniques could have been used, of course, *directly* in each one of the applications we have considered, without passing through an encoding into SG. However, the gain from our effort is twofold. On the one hand, matching the proposed notion of graph bisimulation with well known properties in the theory of networks is a good test for naturality and flexibility. On the other hand, SG may provide "mechanical" support for reasoning about such properties: systems such as the *Concurrency workbench* [5] support bisimulation proofs in the framework of process algebra. It is a challenging project to endow SG with a similar capability.

Finally, our rule of synchronisation is reminiscent of the communication law of the *Fusion Calculus* [21]. Linking to Fusion is therefore a natural gateway for us to the universe of process algebra. We are working through this direction and have developed a context-free theory of synchronising graphs which can be viewed, in a precise sense, as a parallel and syntax-free version of the Fusion calculus. We believe that our translation is fully abstract w.r.t. proper notions of bisimulation equivalences, but we still have not been able to prove such a result.

Related work. SG is closely related to the synchronised hyperedge rewriting (SHR) approach [10] from which it takes inspiration. SHR rewriting acts on *syntactic judgements*, that is *term-graphs* equipped with an *interface* consisting of their set of free nodes. The syntax-driven presentation of SHR enables several properties to be proven at a rather high abstraction level. For example, mimicking the approach in [20] for the π-calculus, it is proven in [14] that a given notion of bisimulation is a congruence for SHR parametrically in a synchronisation algebra with mobility, thus accounting for several styles of interaction. While renouncing the generality of SHR in abstracting over synchronisation algebras, SG exhibits a much simpler system of inference rules.

Productions are built-in SHR so as to make rewriting context-independent in a much similar way our productions do in context-free theories. Indeed, in SHR the dependencies described in Section 3 (and, in particular, those in Examples 4 and 5) are avoided by defining rewriting rules on productions rather than on graphs. Albeit being resolutive, this approach introduces some complexity in the definition of the operational semantics of SHR. For example, all possible instances must be considered when synchronising productions. We prefer to maintain the simple presentation of Section 2 and apply the (simpler) rewriting rules [sync], [open] and [res] to contex-free theories.

However, even in the setting of contex-free theories, SHR still differs from SG both in the notion of transition and in the proof theory. For example, while $x \vdash e(x) \xrightarrow{(x,a,y),\, id} x, y \vdash d(x, y)$ is legal in SHR, where nodes are treated as variables, $e(x) \xrightarrow{x,a,y} d(x, y)$ violates the principle of locality of Definition 1 in SG, where nodes are "constants". As for the proof theory, consider an application in which agent $d(x)$ dies. This is done in SG by the production $d(x) \xrightarrow{\emptyset} \emptyset$, which is mimicked in SHR by the production $x \vdash d(x) \xrightarrow{\emptyset,\, id} x \vdash nil$. Say this transition occurs in a larger context including an idle agent $e(y)$. In SG: $d(x) \,|\, e(y) \xrightarrow{\emptyset} e(y)$. In SHR: $x, y \vdash d(x) \,|\, e(y) \xrightarrow{\emptyset,\, id} x, y \vdash e(y)$. Node x remains in the context, even if no edge is attached to it. After that, and for the rest of its life, $e(y)$ can procede computation in SHR only if synchronising with the identical transition $x \vdash nil \xrightarrow{\emptyset,\, id} x \vdash nil$ of the graph consisting of a unique node x and no edges (no such a graph exists in SG). Identities are therefore fundamental in SHR, and all syntactic judgements are granted one. On the other hand, identities can be provided in a context-free theory *if desired*. Interleaving could be inhibited, for example, by not providing identities. This also impacts on behavioural equivalence as no distinction can be made in SHR between edges whose only transition is the identity and edges with no transitions at all.

The above examples show that no sensible matching can be made between SHR (with synchronisation à la Milner) and context-free theories of synchronising graphs, and none can be viewed as generalising the other.

Other interesting approaches have been applied to give congruential observational semantics to graph rewriting. Notably, in [9] *borrowed contexts* enable the derivation *minimal contexts* in a DPO (double-push out) approach. The idea, inspired by [17,16], consists in computing the minimal context within which a system can react. The resulting observational semantics, where observations are given by such minimal contexts, provides a bisimulation which is a congruence "by construction". A similar approach is taken in [18], where bigraphs are equipped with rules to form a *bigraphical reactive system* providing a bisimilarity which is a congruence. An interesting research direction is applying the mentioned approaches to SG and then compare the resulting observational semantics with the one defined here. Indeed, it is not clear what are the relationships between "natural" equivalences and those obtained via borrowed-context or reactive approaches. Initial studies for process algebras show that such equivalences may not coincide: for example, [11] shows a congruential bisimilarity obtained with a borrowed-context approach that is finer than open bisimilarity in the π-calculus. Recently, the notion of *saturated semantics* [1] has been shown to provide suitable congruential bisimilarities (e.g. the open bisimilarity for π-calculus can be obtained). This approach is quite promising but, at the best of our knowledge, it has not been applied to observational semantics of graphs rewriting.

References

1. Bonchi, F., König, B., Montanari, U.: Saturated semantics for reactive systems. In: Proc. of LICS, pp. 69–80. IEEE, Los Alamitos (2006)
2. Cardelli, L., Gordon, A.: Mobile Ambients. Theor. Comp. Science 1(240), 177–213 (2000)
3. Cenciarelli, P., Talamo, I., Tiberi, A.: Ambient Graph Rewriting. Electronic Notes in Theoretical Computer Science 117, 335–351 (2005)
4. Cenciarelli, P., Tiberi, A.: Rational Unification in 28 Characters. Electronic Notes in Theoretical Computer Science 127(5), 3–20 (2005)
5. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: A semantics-based tool for the verification of concurrent systems. ACM ToPLaS 15(1), 36–72 (1993)
6. Cormen, T., Leiserson, C., Rivest, R.: Introduction to algorithms. MIT Press, Cambridge (1990)
7. 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: Handbook of Graph Grammars and Computing by Graph Transformation, ch. 3, vol. 1
8. Degano, P., Montanari, U.: A model for distributed systems based on graph rewriting. Journal of the ACM 34, 411–449 (1987)
9. Ehrig, H., König, B.: Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 151–166. Springer, Heidelberg (2004)
10. Ferrari, G., Montanari, U., Tuosto, E.: A LTS semantics of ambients via graph synchronization with mobility. In: Restivo, A., Ronchi Della Rocca, S., Roversi, L. (eds.) ICTCS 2001. LNCS, vol. 2202. Springer, Heidelberg (2001)
11. Ferrari, G., Montanari, U., Tuosto, E.: Model Checking for Nominal Calculi. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 1–24. Springer, Heidelberg (2005)
12. Hirsch, D., Montanari, U.: Synchronized hyperedge replacement with name mobility: A graphical calculus for name mobility. In: Agha, G.A., De Cindio, F., Rozenberg, G. (eds.) APN 2001. LNCS, vol. 2001. Springer, Heidelberg (2001)
13. König, B., Montanari, U.: Observational equivalence for synchronized graph rewriting with mobility. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 145–164. Springer, Heidelberg (2001)
14. Lanese, I.: Synchronization Strategies for Global Computing Models. PhD thesis, Univ. of Pisa (2006)
15. Lanese, I., Montanari, U.: A graphical fusion calculus. In: Proc. of COMETA 2003 (2003)
16. Leifer, J.: Operational Congruences for Reactive Systems. PhD thesis, Univ. of Cambridge (UK) (2001)
17. Leifer, J., Milner, R.: Deriving Bisimulation Congruences for Reactive Systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)
18. Milner, R.: Bigraphical Reactive Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154. Springer, Heidelberg (2001)
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Information and Computation 100, 1–77 (1992)

20. Montanari, U., Buscemi, M.: A First Order Coalgebraic Model of π-Calculus Early Observational Equivalence. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 449–465. Springer, Heidelberg (2002)
21. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proc.of LICS. IEEE Computer Society Press, Los Alamitos (1998)
22. Riely, J., Hennessy, M.: Distributed Processes and Location Failures. Theoretical Computer Science 266, 693–735 (2001)
23. Turi, D., Plotkin, G.D.: Towards a mathematical operational semantics. In: Proc. of LICS. IEEE Computer Society Press, Los Alamitos (1997)
24. Warneke, B., Last, M., Liebowitz, B., Pister, K.S.J.: Smart dust: Communicating with a cubic-millimeter computer. IEEE Computer 34(1), 44–51 (2001)

Graph Transformation for Topology Modelling

Mathieu Poudret[1,2], Agnès Arnould[2], Jean-Paul Comet[3],
and Pascale Le Gall[1,4]

[1] Programme d'Épigénomique, Génopole, F-91000 Évry
pascale.legall@epigenomique.genopole.fr
[2] XLIM-SIC UMR 6172 CNRS, Univ. de Poitiers, F-86962 Futuroscope
{poudret,arnould}@sic.univ-poitiers.fr
[3] I3S, UMR 6070 CNRS, Univ. de Nice-Sophia-Antipolis, F-06903 Sophia-Antipolis
comet@unice.fr
[4] MAS, Ecole Centrale Paris, Grande Voie des Vignes, F-92195 Châtenay-Malabry

Abstract. In this paper we present meta-rules to express an infinite class of semantically related graph transformation rules in the context of pure topological modelling with G-maps. Our proposal is motivated by the need of describing specific operations to be done on topological representations of objects in computer graphics, especially for simulation of complex structured systems where rearrangements of compartments are subject to change. We also define application of such meta-rules and prove that it preserves some necessary conditions for G-maps.

Keywords: topology-based geometric modelling, graph transformation, generalized map.

1 Introduction

Simulation of complex structured systems is a specialised area of topology-based modelling (or topological modelling for short). Topological models deal with the representation of the structure of objects (their decomposition into topological units: vertices, edges, faces and volumes) and with the neighbourhood relations between topological units. Thus topological structures are specific graphs. Among numerous topological models, generalized maps [Lie89, Lie94] (or G-maps) constitute a mathematically-defined model. Intuitively, edges between nodes indicate which nodes are neighbours and edge labels indicate which kind of neighbouring is concerned (i.e. connection of volumes, faces or edges). G-maps are thus a particular class of graphs with labelled edges defined by constraints ensuring that neighbouring relations are consistently organised. Topology-based modellers and simulators aggregate a large number of operations to edit objects. Most operations are designed to be dedicated to some application scopes. Moreover, they are usually implemented by a dedicated algorithm finely tuned in order to optimise its efficiency.

Using the framework of graph transformations [Roz97, EEPT06], we propose in this paper to model topological operations with transformation rules. Thus,

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 147–161, 2008.

we will be able to develop a simulator as a simple engine of rules applications. In a previous work, we defined transformation rules adapted to G-maps [PCG+07] using the algebraic approach of graph transformation based on labelled graphs and the double-pushout approach. Our first framework contains classical rules defined on an explicit pattern and a first class of meta-rules defined on patterns that carry isomorph topological units (volume, face, edge, vertex). This first proposal was satisfactory in the sense that we defined the four basic operations of G-maps (those from which all others can be defined) in terms of graph transformation rules. Even if we have already used our framework for the simulation of complex biological structured systems [PCLG+08], this first framework was not powerful enough to directly define complex topological operations. However, to facilitate the derivation of efficient simulation algorithms from high-level transformation rules, it becomes essential to be able to describe a large class of complex topological operations directly in term of transformation rules, instead of the composition of elementary topological operations. Indeed, we take advantage of such an approach both by ensuring for free some constraints of G-maps and by directly defining efficient algorithms by means of dedicated coverages of G-maps driven by the form of the considered high-level transformation rules.

In this paper, we present a more general class of meta-rules for G-maps which allows one to directly define a large class of topological operations. Intuitively, our meta-rules are built over graphs whose edges are labelled by new symbols playing the role of variables. The name of the symbols will indicate by which kind of topological units they can be substituted. So, our variables may be perceived as typed variables, each type representing a class of topological units of similar nature as volumes, faces or edges within the framework of G-maps. Thus, our meta-rules are more abstract and expressive than simple transformation rules over G-maps and take advantage of variables to generate a large family of basic transformation rules sharing the same effect according to a topological point of view. The use of variables to abstract graph transformation rules has been previously addressed [Hof05], in particular to model software transformations for refactoring purpose [HJE06]. In [Hof05], variables can be graph variables, attribute variables and cloning variables. In particular, cloning variables are mechanisms for duplicating some scheme extracted from the variable-based transformation, according to a given cardinality. The application of a transformation rule with cloning variables can then be expressed in term of application of simple transformation rules. In a similar approach, the application of our meta-rules will imply a mechanism of scheme cloning. However, our meta-rules will be specialised with respect to the underlying class of graphs on which they are applied, that is, the class of G-maps. Cloning mechanism will allow us to capture the class of all topological units of same nature (as volume, face, ...) which are of different size according to the considered 3D-object. Moreover, as G-maps are strongly constrained graphs, we will give some simple conditions on our meta-rules, ensuring both the dangling condition on all underlying basic transformation rules issued from the meta-rules and some of the constraints characterising G-maps among all labelled graphs.

The paper is organised as follows. Section 2 briefly presents graph transformation rules. Section 3 presents the G-map topological model. In Section 4, we introduce graph transformation meta-rules for modelling high-level topological operations and their application is defined by means of some intermediate cloning steps. In Section 5, we prove that some constraints of G-maps are preserved through the application of graph transformation meta-rules. Section 6 provides some concluding remarks.

2 Preliminaries

Let us first recall some notions and notations concerning graph transformations extracted from [EEPT06].

A graph G with labels in Σ_E is a couple (V, E) such that V is a set of vertices and $E \subset V \times \Sigma_E \times V$ is a set of non-oriented labelled edges. A path in G is a sequence $(v_0, l_1, v_1), (v_1, l_2, v_2), ..., (v_{k-1}, l_k, v_k)$ of E edges. We say that this path links v_0 to v_k and is labelled by the word $l_1 l_2 ... l_k \in \Sigma_E^*$. If $v_0 = v_k$, the path is called a cycle.

We introduce orbit graphs as particular sub-graphs, those which are generated by a vertex and an identified subset of labels. Indeed, these orbit graphs are useful to easily represent and manipulate topological cells (like faces or volumes) in the context of topological modelling.

Definition 1 (orbit). *Let us consider $G = (V, E)$ a graph with labels in Σ_E, $\{l_1, ..., l_k\} \subset \Sigma_E (k \geq 0)$ a set of labels and a vertex v of G.*

We call orbit $< l_1, ..., l_k > (v)$, the subset of V vertices reachable from v with paths labelled by words of $\{l_1, ..., l_k\}^*$. The orbit $< l_1, ..., l_k > (v)$ is said to be adjacent to v.

We call orbit graph $<< l_1, ..., l_k >> (v)$, the subgraph of G with vertices in $< l_1, ..., l_k > (v)$ and with edges in $\{(v', l, v") \in E \ / \ v', v" \in < l_1, ..., l_k > (v)$ and $l \in \{l_1, ..., l_k\}\}$.

A graph morphism $f : G \to H$ between two graphs G and H with labels in Σ_E, consists of two functions f_V from G vertices to H vertices and f_E from G edges to H edges, such that labelled edges are preserved[1]. Such a morphism is injective (resp. bijective) if both f_V and f_E are injective (resp. bijective). A bijective morphism is named isomorphism. G and H are said isomorphic if there exists an isomorphism $f : G \to H$.

In the sequel, for our purposes, we only consider injective graph morphisms, which formalise the classical inclusion relation. Thus, we present the algebraic graph transformation approach and use the category **Graph** of graphs and graph morphisms (see chapter 2 of [EEPT06]).

A *production rule* $p : L \leftarrow K \to R$ is a pair of graph morphisms $l : K \to L$ and $r : K \to R$. L is the left-hand side, R is the right-hand side and K is the common interface of L and R. The left-hand side L represents the pattern of the

[1] For each edge (v, l, v') of G, $f_E((v, l, v')) = (f_V(v), l, f_V(v'))$.

rule, while the right-hand side R describes the production. K describes a graph part which has to exist to apply the rule, but which is not modified. Intuitively, $L\backslash K$ is the removed part[2] while $R\backslash K$ is the added part.

The rule p *transforms* G into a graph H, denoted by $G \Rightarrow_{p,m} H$, if there are a match graph morphism $m : L \rightarrow G$ and two square diagrams which are graph pushouts as in the following figure.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

A direct graph transformation can be applied from a production rule p on a graph G if one can find a match m of the left-hand side L in G such that m is an (injective) morphism.

When a graph transformation with a production rule p and a match m is performed, all the vertices and edges which are matched by $L\backslash K$ are removed from G. The removed part is not a graph, in general, but the remaining structure $D := (G\backslash m(L)) \cup m(K)$ still has to be a legal graph (see following dangling condition), *i.e.* no edges should dangle (source and target vertices of all remaining edges should also remain). This means that the match m has to satisfy a suitable gluing condition, which makes sure that the gluing of $L\backslash K$ and D is equal to G (see (1) in the figure). In the second step of a direct graph transformation, D and $R\backslash K$ are glued together to obtain the derived graph H (see (2)).

More formally, we use graph morphisms $K \rightarrow L$, $K \rightarrow R$, and $K \rightarrow D$ to express how K is included in L, R, and D, respectively. This allows us to define the gluing constructions $G = L +_K D$ and $H = R +_K D$ as the pushout[3] constructions (1) and (2) in the figure, leading to a double pushout.

A graph morphism $m : L \rightarrow G$ from the left-hand side of a production rule $p : L \leftarrow K \rightarrow R$ to a graph G satisfies the *dangling condition* if no edge of $G\backslash m(L)$ is adjacent to a vertex of $m(L\backslash K)$. This dangling condition makes sure that the gluing of $L\backslash K$ and D is equal to G. Intuitively, all edges of G incident to a removed vertex are also removed.

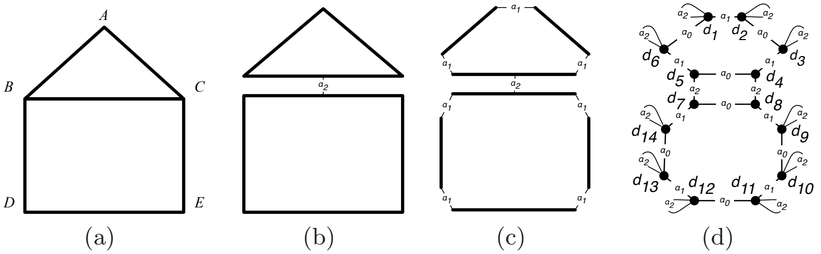
Finally, a *graph transformation*, or, more precisely, a graph transformation sequence, consists of zero or more direct graph transformations.

3 Generalized Maps

The generalized maps (or G-maps) introduced by P. Lienhardt [Lie89, Lie94] define the topology of an n-dimensional subdivision space. G-maps allow the representation of the quasi-varieties, orientable or not. To represent cellular space

[2] The substraction $L\backslash K$ between two graphs $L = (V_L, E_L)$ and $K = (V_K, E_K)$ is defined from the set substraction on vertices and edges $L\backslash K = (V_L\backslash V_K, E_L\backslash E_K)$. Thus $L\backslash K$ may not be a graph.

[3] Let $f : A \rightarrow B$ and $g : A \rightarrow C$ be two graph morphisms, $D = B +_A C$ is the pushout object of B and C *via* A, or more precisely, *via* (A, f, g).



(a) (b) (c) (d)

Fig. 1. Decomposition of a 2D object

subdivisions, we can choose other topological representations like combinatorial maps [Tut84]. Nevertheless, G-maps have the advantage of providing a homogeneous definition for all dimensions. Thus, operation specifications are simpler.

Intuitively, the main idea of G-maps is to decompose an object into basic elements, also called darts (graph vertices), which are connected together (with graph edges). The decomposition of a 2D object is shown in Fig. 1. The 2D object is displayed on Fig. 1(a). In Fig. 1(b), the object is split in order to focus on the two faces (topological 2-cells) which compose it. In an n-G-map, $n + 1$ kinds of labelled-edges (from α_0 to α_n) allow one to recover the knowledge about neighbourhood relations between the topological cells. Thus, in Fig. 1(b), an α_2 edge makes explicit the adjacency relation which previously exists between faces ABC and $BCDE$. On Fig. 1(c), the faces are decomposed into lower dimension elements: the 1-cells. In the same manner, α_1 edges makes explicit the adjacency relations between the 1-cells. Finally, in the 2-G-map of Fig. 1(d), edges are split into α_0-connected 0-dimensional darts (represented with black dots). We notice that the index i of α_i labelled edges gives the dimension of the considered adjacency relation.

Definition 2 (G-map). *Let $n \geq 0$. An* n-dimensional generalised map *(or n-G-map) is a graph G with labels in $\Sigma_E = \{\alpha_0, \ldots, \alpha_n\}$, such that:*

- *The following $\mathcal{C}_G(\Sigma_E)$ condition is satisfied:*
  each vertex of G has exactly one adjacent l-edge for each label $l \in \Sigma_E$.
- *The following consistency constraint is satisfied:*
  for each pair of labels $\alpha_i, \alpha_j \in \Sigma_E$ such that $i + 2 \leq j$, there exist a cycle labelled $\alpha_i\alpha_j\alpha_i\alpha_j$ from each vertex v of G.

The first condition of this definition (which is denoted by $\mathcal{C}$ for short, in the sequel) ensures that each vertex of an n-G-map has exactly $n+1$ adjacent edges labelled by $\alpha_0, ..., \alpha_n$. For example, in Fig. 1(d), the vertex d_4 is α_0-linked with d_5, α_1-linked with d_3, and α_2-linked with d_8. On the border of the objects, some darts do not have all of its neighbours. For instance, on Fig. 1(d) the vertex d_1 is α_0-linked to d_6 and α_1-linked to d_2, but d_1 is not linked to another vertex by an α_2-edge, because d_1 denotes the top corner of the object of Fig. 1(a) and thus is on the border of the 2D object. However, according to the $\mathcal{C}$ condition all vertices must have exactly one adjacent label for each dimension. Thus, there is an α_2-loop adjacent to vertex d_1.

The second point of the n-G-map definition expresses some consistency constraints on the adjacency relations denoted by the labelled edges. Intuitively, in an G-map, if two i-dimensional topological cells are stuck together then they are stuck along a $(i-1)$-dimensional cell. For instance, on Fig. 1(d), the 2-cell defined by $\{d_1, ..., d_6\}$ is stuck with the 2-cell defined by $\{d_7, ..., d_{14}\}$ along the 1-cell defined by the four vertices $\{d_5, d_4, d_8, d_7\}$. The consistency constraint requires that there is a cycle $\alpha_0\alpha_2\alpha_0\alpha_2$ starting from each vertex of $\{d_5, d_4, d_8, d_7\}$. Thanks to loops, this property is also satisfied on object borders. For example, on the bottom of the object of Fig. 1(d), we have the cycle $\alpha_0\alpha_2\alpha_0\alpha_2$ from d_{11} and d_{12}.

The following definition explains the notion of i-cell in terms of G-map orbits.

Definition 3 (i-cell). *Let us consider G an n-G-map, v a vertex of G and $i \in [0, n]$. The i-cell adjacent to v is the orbit graph (see definition 1) of G $<< \alpha_0, ..., \alpha_{i-1}, \alpha_{i+1}, ..., \alpha_n >> (v)$. The i-cell adjacent to v is noted i-cell(v).*

Let us illustrate this definition on the Fig. 1. The 2D geometric object Fig. 1(a) is composed of 0-cells (the geometric points A, B, C, D and E), 1-cells (the geometric segments AB, BC, AC, BD, DE and CE), and 2-cells (the two geometric faces ABC and $BCDE$). The corresponding 2-G-map Fig. 1(d) contains the same cells denoted by the following sub-G-maps. The geometric triangle ABC is denoted by 2-cell(d_1), *i.e.* the orbit graph $<< \alpha_0, \alpha_1 >> (d_1)$ which contains all vertices reachable from d_1 using α_0 and α_1 labelled edges. The geometric segment BC is denoted by the 1-cell(d_5), *i.e.* the orbit graph $<< \alpha_0, \alpha_2 >> (d_5)$ which contains the four vertices d_4, d_5, d_7 and d_8. The geometric points are denoted by 0-cells and their numbers of vertices depend on their numbers of adjacent segments. For example A (denoted by 0-cell(d_1), *i.e.* the orbit graph $<< \alpha_1, \alpha_2 >> (d_1)$), contains the two vertices d_1 and d_2.

We have already seen that applying a production rule on a graph requires to find a matching morphism satisfying the dangling condition. The following proposition shows that in case of graphs verifying the $\mathcal{C}$ condition (see definition 2), the dangling condition only depends on the form of the production rule, and that the derivation then preserves the $\mathcal{C}$ property.

Proposition 1. *Let p: $L \leftarrow K \rightarrow R$ be a production rule, and m: $L \rightarrow G$ be a match morphism on a graph G with labels in Σ_E which satisfies the $\mathcal{C}_G(\Sigma_E)$ condition.*

1. *m satisfies the dangling condition iff $L \backslash K$ satisfies the $\mathcal{C}_{L\backslash K}(\Sigma_E)$ condition*[4].
2. *Moreover, if the rule p satisfies the following $\mathcal{C}_p(\Sigma_E)$ condition, then the derived graph H produced by the direct graph transformation $G \Rightarrow_{p,m} H$ satisfies the $\mathcal{C}_H(\Sigma_E)$ condition.*

[4] The condition $\mathcal{C}_G$ is defined for a graph G but can be extended for a structure which is not a graph. In this case, adjacent edges can dangle.

$\mathcal{C}_p(\Sigma_E)$: $\mathcal{C}_{L\backslash K}(\Sigma_E)$ and $\mathcal{C}_{R\backslash K}(\Sigma_E)$ are satisfied and each preserved vertex of K has the same adjacent labelled edges in L and R (i.e. $\forall v \in K$, $\forall l \in \Sigma_E$, v has an l-edge adjacent in L iff v has an l-edge adjacent in R and if they exist they are unique[5]).

Proof. Let us prove the first point. Let us suppose that m satisfies the dangling condition. By hypothesis, G satisfies the $\mathcal{C}_G(\Sigma_E)$ condition, thus for each deleted vertex v of $m(L\backslash K)$ and for each label $l \in \Sigma_E$ there exists a unique l-edge adjacent to v in G noted (v, l, v'). Thus, thanks to the dangling condition, (v, l, v') is an edge of $m(L)$. Because m is injective, $L\backslash K$ satisfies the $\mathcal{C}_{L\backslash K}(\Sigma_E)$ condition.

Reciprocally, let us suppose that $L\backslash K$ satisfies the $\mathcal{C}_{L\backslash K}(\Sigma_E)$ condition. Since G satisfies the $\mathcal{C}_G(\Sigma_E)$ condition, each edge of G adjacent to a vertex of $m(L\backslash K)$ is an edge of $m(L\backslash K)$. So, the dangling condition is satisfied.

Let us now prove the second point. Let us suppose that G, the removed structure $L\backslash K$ and the created structure $R\backslash K$ satisfy, respectively, $\mathcal{C}_G(\Sigma_E)$, $\mathcal{C}_{L\backslash K}(\Sigma_E)$ and $\mathcal{C}_{R\backslash K}(\Sigma_E)$ conditions and that each preserved vertex of K has the same labelled edges in left-hand side L and in right-hand side R. Thanks to the first point, the dangling condition is satisfied and thus the direct graph transformation $G \Rightarrow_{p,m} H$ exists. Let v be a vertex of H and $l \in \Sigma_E$ a label:

- If v is not a matched vertex, *i.e.* v is a vertex of $G\backslash m(L)$. Thanks to the $\mathcal{C}_G(\Sigma_E)$ condition, there exists a unique l-edge adjacent to v in G, noted (v, l, v'). v' may be a vertex of $m(L)$ or not, but (v, l, v') is not a matched edge, *i.e.* (v, l, v') is not an edge of $m(L)$, because L is a graph. Thus thanks to the direct graph transformation, (v, l, v') is the unique l-edge adjacent to v in H;
- If v is an added vertex, *i.e.* v is not a vertex of G. Thanks to direct graph transformation, there exist a vertex u of R such that the double pushout produces v in H from u. However, thanks to hypothesis, R have exactly one l-edge adjacent to u. Thus H have exactly one l-edge adjacent to v;
- If v is a matched vertex, *i.e.* v is a vertex of $m(K)$. Thanks to the $\mathcal{C}_G(\Sigma_E)$, there exists an unique l-edge adjacent to v in G, noted (v, l, v'). And thanks to the m injectivity, there exists a unique vertex u in L such that $m_V(u) = v$.
  - If there does not exist any l-edge adjacent to u in L, thanks to hypothesis, there does not exist any l-edge adjacent to u in R. Thus, (v, l, v') is an edge of $G\backslash m(L)$ and thanks to direct graph transformation, (v, l, v') is an edge of H. Moreover, thanks to the $\mathcal{C}_G(\Sigma_E)$ condition, (v, l, v') is the unique l-edge adjacent to v in H;
  - If there exists an edge (u, l, u') in L, thanks to hypothesis, it is the unique l-edge adjacent to u in L and there exists an unique l-edge adjacent to u in R noted (u, l, u''). Moreover, thanks to the $\mathcal{C}_G(\Sigma_E)$ condition, the unique l-edge adjacent to v in G is $m_E((u, l, u'))$. Thus the double pushout of the direct graph transformation produces an unique l-edge adjacent to v from the (u, l, u'') edge.

[5] We suppose, without loss of generality, that the morphisms l and r of the double-pushout figure (see section 2) are the identity.

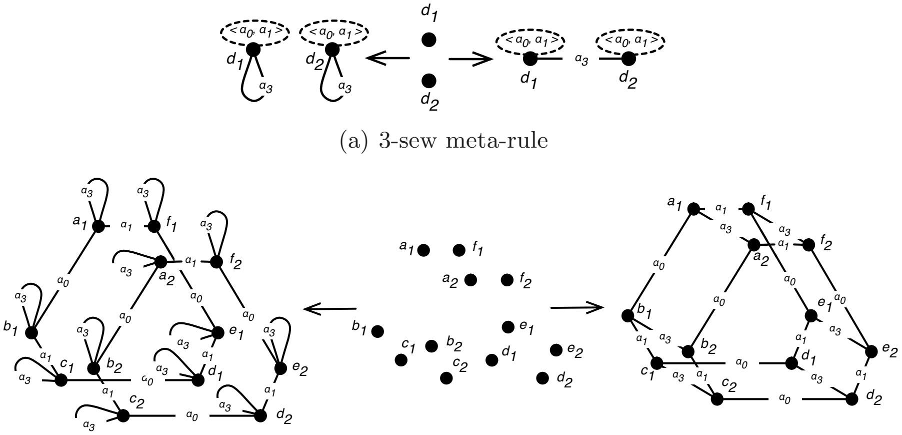
Consequently, there exists a unique l-labelled edge adjacent to v in H. In other words, H satisfies the $\mathcal{C}_H(\Sigma_E)$ condition. □

The proposition 1 ensures that all derivations with an adequate production rule preserve the $\mathcal{C}$ condition of G-maps (see definition 2). Let us notice that like in classical operation definitions (mathematical definitions, algorithms or formal specifications), the G-map consistency constraint (second point of definition 2) has to be verified individually for each production rule.

4 Topological Operations in Terms of Graph Transformation

The set of basic topological operations for G-maps has been defined [Lie89] and includes different operations, namely vertex addition, vertex suppression, sew and unsew. In previous works [PCG+07], we have shown that first and second operations can be directly translated into transformation rules satisfying the $\mathcal{C}$ condition and moreover the consistency constraint of G-map (see definition 2).

Nevertheless, both sew and unsew operations are generic and cannot be defined directly in terms of graph transformation rules because they depend on the orbits. To overcome this limitation, we introduced in [PCG+07] a concept of graph transformation meta-rules which abstracts a set of graph transformation rules along an orbit. The idea is to propagate a local transformation pattern (expressed on a few vertices) along an orbit of the graph, independently of the form of this orbit. To specify which part of the local pattern is associated to the elements of the orbit, we introduce an additional label, which denotes the orbit. Graphically, these meta-labelled edges are noted with dotted lines. Thus the 3-sew meta-rule (which aims at sticking two volumes along one face) of Fig. 2(a)



(a) 3-sew meta-rule

(b) 3-sew of triangular faces

Fig. 2. 3-sew rules

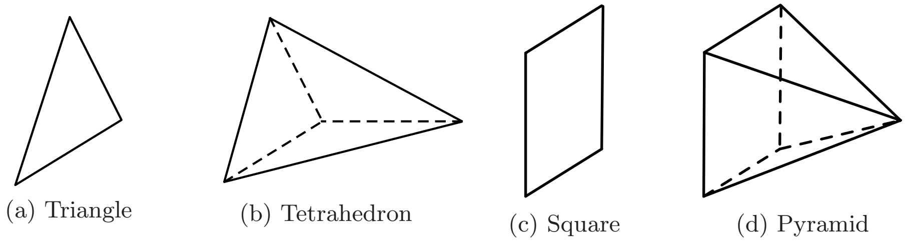
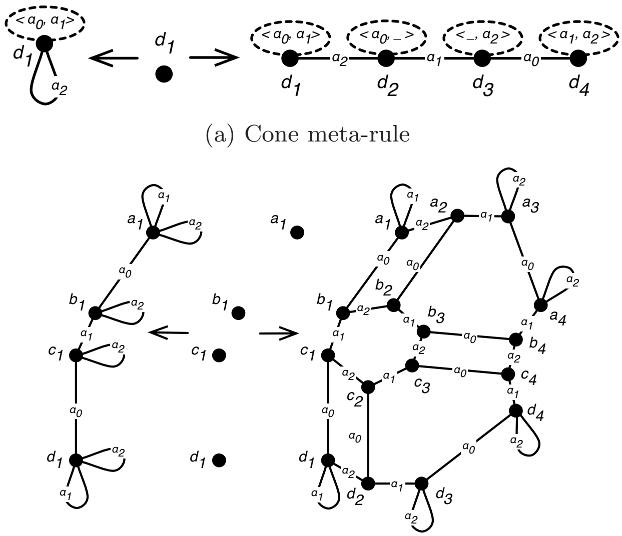

(a) Triangle (b) Tetrahedron (c) Square (d) Pyramid

Fig. 3. Cone operation

may be applied along a triangular face to define the classical rule of Fig. 2(b), or along any other face orbit. More precisely, a meta-edge $(d_1, < \alpha_0, \alpha_1 >, d_1)$ of a meta-rule specifies a sub-graph labelled on $\{\alpha_0, \alpha_1\}$ and thus matches an orbit graph $<< \alpha_0, \alpha_1 >> (d_1)$ in each classical rule (see Fig. 2(b)). The pattern connected to d_1, compounded of a α_3 classical loop, must be repeated along this orbit. Thus, Fig. 2(b) vertices a_1, ..., f_1, a_2, ..., f_2 have an α_3 loop. Finally, $(d_1, < \alpha_0, \alpha_1 >, d_1)$ and $(d_2, < \alpha_0, \alpha_1 >, d_2)$ must be expended in two isomorphic orbit graphs. Thus, $<< \alpha_0, \alpha_1 >> (d_1)$ and $<< \alpha_0, \alpha_1 >> (d_2)$ are isomorphic faces in Fig. 2(b).

This previous framework is enough to specify basic operations, and thus is complete because all 3-G-map operations may be specified from the basic ones. But, from a user point of view, to specify an operation as a large composition of basic operations is less easy and efficient that specifying it directly. Unfortunately, the previous framework is not general enough to directly specify most of complex operations. Indeed, previous meta-rules are defined along a unique orbit, thus every meta-edges are expended as isomorphic orbit graphs. For example, the four meta-edges of sew rule Fig. 2(a) are expended to four isomorphic triangular faces (see Fig. 2(b)). But, for most operations we need to match (and/or to produce) different kinds of orbit graphs. In the cone operation (which aims at producing a cone-shaped volume from one base face), different kinds of orbit graphs are necessary to produce, for instance, a tetrahedron from a triangular face or a pyramid from a square face (see Fig. 3). This operation cannot be defined from several copies of the base 2-cell. But, it may be defined from copies of base vertices linked together in the right manner. Especially, the top 0-cell of a cone is dual[6] of the base 2-cell. Intuitively, the 2-cells adjacent to the base are also adjacent to the top. The classical rule of Fig. 4(b) defines the cone operation on a face corner. Here, the top orbit graph $<< \alpha_1, \alpha_2 >> (d_4)$ is a copy of the base orbit graph $<< \alpha_0, \alpha_1 >> (d_1)$ with a renaming of links. Thus, when a_1 and b_1 are α_0 linked, a_4 and b_4 are α_1 linked and when b_1 and c_1 are α_1 linked, b_4 and c_4 are α_2 linked. Moreover, when a_1 and b_1 are α_0 linked, a_2 and b_2 are also α_0 linked and when b_1 and c_1 are α_1 linked, b_2 and c_2 are not linked. The α_2 loop of the left-hand side of the rule figure 4(a), means that only isolated faces (which are not linked to another one) can be matched in order to produce cones.

[6] Two topological cells are dual if they are isomorphic up to a renaming of their labels.



(a) Cone meta-rule

(b) Rule of a corner cone

Fig. 4. Cone rules

The following definition allows us to generalise graph and production rules notions by adding meta-edges that denote isomorphic orbit graphs up to a renaming of their edges labels.

Definition 4 (meta-graph and production rule). *Let $\beta = \{\alpha'_1, ..., \alpha'_k\} \subset \Sigma_E$ a subset of labels and Γ_β be the set of all renaming functions $\gamma : \beta \to \Sigma_E \cup \{\_\}$. A renaming function γ is named meta-label and is said full if $\gamma(\beta) \subset \Sigma_E$ (without "_")*[7]*.*

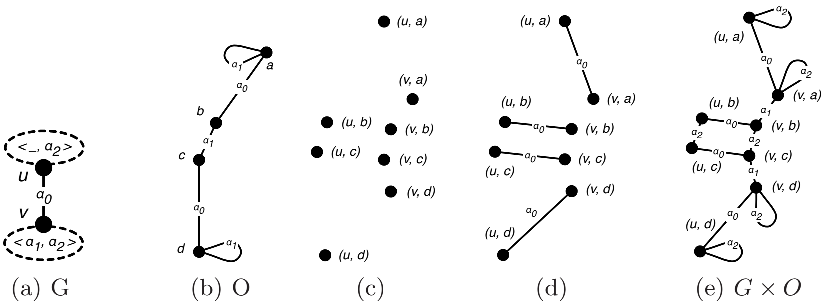
A meta-graph *on β, or meta-graph, is a graph with label in $\Sigma_E \cup \Gamma_\beta$ such that each meta-labelled edge is a loop. A meta-graph is said full if all its meta-labels are full. Graphically, a meta-loop γ is labelled by the renamed orbit $< \gamma(\alpha'_1), ..., \gamma(\alpha'_k) >$.*

A production meta-rule *on β, or meta-rule, is a production rule $p : L \leftarrow K \to R$ on the full sub-category of generalised meta-graph on β, such that the meta-graph L is full.*

The meta-rule Fig. 4(a) specifies the cone operation. In this example, we can see four different kinds of orbits: a full orbit graph for the base (2-cell $<< \alpha_0, \alpha_1 >> (d_1)$), two partial ones for the side faces[8] (2-cells $<< \alpha_0, \_ >> (d_2)$ and $<< \_, \alpha_2 >> (d_3)$) and another full one for the top (0-cell $<< \alpha_1, \alpha_2 >> (d_4)$). All of them are translated copies of matched 2-cell $<< \alpha_0, \alpha_1 >> (d_1)$, using respectively renaming functions $\gamma_1 : \alpha_0 \mapsto \alpha_0, \alpha_1 \mapsto \alpha_1$, $\gamma_2 : \alpha_0 \mapsto \alpha_0, \alpha_1 \mapsto \_$,

[7] Where $\gamma(\beta)$ names the set $\{\gamma(l) \mid l \in \beta\}$.

[8] Formally, this two subgraphs are not orbits in the sense of definition 1.



(a) G (b) O (c) (d) (e) $G \times O$

Fig. 5. Expansion of a meta-graph G on $\{\alpha_0, \alpha_1\}$ along a graph O

$\gamma_3 : \alpha_0 \mapsto \_, \alpha_1 \mapsto \alpha_2$ and $\gamma_4 : \alpha_0 \mapsto \alpha_1, \alpha_1 \mapsto \alpha_2$. By lack of space, we do not explain how by means of similar production rules, we can express other topological operations like the extrusion operation (to create box from a face) or the rounding operation (to round angular edges or vertices).

As seen on examples, the semantic of meta-graph patterns is given by expanding the meta-patterns along an orbit on β.

Definition 5 (expansion). *Let $\beta = \{\alpha'_1, ..., \alpha'_k\} \subset \Sigma_E$ be subset of labels and O be a graph with labels in β. The* expansion *of a meta-graph G on β along O is the Cartesian-like product $G \times O$ such that:*

- *The set of vertices is the Cartesian product of vertex sets* $\{(u, a) \mid u$ *is a vertex of* G *and* a *is a vertex of* $O\}$*;*
- *The set of edges is* $\{((u, a), l, (v, a)) \mid (u, l, v)$ *is an edge of* G *and* a *is a vertex of* $O\} \cup$ $\{((u, a), \gamma(l), (u, b)) \mid (u, \gamma, u)$ *is a meta-edge of* G*,* $\gamma(l) \in \Sigma_E$ *and* (a, l, b) *is an edge of* $O\}$.

The expansion of a morphism $f : G \to H$ along O, is the morphism $f \times O : G \times O \to H \times O$ which associates the vertex $(f_V(u), a)$ of $H \times O$ to each vertex (u, a) of $G \times O$.

The expansion of a production meta-rule $p : L \xleftarrow{l} K \xrightarrow{r} R$ along O is the production rule $p \times O : L \times O \xleftarrow{l \times O} K \times O \xrightarrow{r \times O} R \times O$.

In Fig. 5, we expand a graph G on $\{\alpha_0, \alpha_1\}$ (see Fig. 5(a)) along a graph O labelled in $\{\alpha_0, \alpha_1\}$ (see Fig. 5(b)). Actually, G is extracted from the right-hand side of the cone meta-rule (as shown in Fig. 4(a)) and O represents corner 2-cell. The first step of the expansion process (see Fig. 5(c)) consists in copying the vertices of G along O (computing $V_G \times V_O$). The next steps consist in, respectively, copying classical edges of G along O (see Fig. 5(d)) and copying renamed edges of O along the G meta-edges (see Fig. 5(e)). Then, Fig. 4(b) is obtained by expansion of cone meta-rule Fig. 4(a) along a face corner pattern.

Proposition 2. *Let $f : G \to H$ be a morphism between the two meta-graphs G and H on β, and O a graph with label in β. The expansion $f \times O$ always exists.*

Proof. For each edge (u, l, v) of G and each vertex a of O, $((u, a), l, (v, a))$ is an edge of $G \times O$ and $((f_V(u), a), l, (f_V(v), a))$ is an edge of $H \times O$.

For each meta-edge (u, γ, u) of G and each edge (a, l, b) of O, if $\gamma(l) \in \Sigma_E$ then $((u, a), \gamma(l), (u, b))$ is an edge of $G \times O$ and $((f_V(u), a), \gamma(l), (f_V(u), b))$ is an edge of $H \times O$.

$G \times O$ has no other edge. □

The following proposition shows that the expansion does not depend on β labels. Its proof is left to the reader.

Proposition 3. *Let β and δ be two subset of labels, $\iota : \delta \to \beta$ a bijective function, G be a meta-graph on β and O a graph labelled in β.*

Let H be the meta-graph on δ obtained from G by renaming each meta γ-loop to a $\gamma \circ \iota$-loop and P be the graph labelled on δ obtained from O by renaming each label of O along ι^{-1}. Then we have $G \times O = H \times P$.

The previous proposition founds the graphical notation of meta-loops with implicit renaming functions.

By definition, if there exist several meta-edges on the left-hand side and on the right-hand side of a production meta-rule, the expansion replaces all these meta-edges with distinct sub-graphs (each of them is isomorphic, up to a renaming of their edges labels, to the β-labelled graph O).

Definition 6 (direct graph meta-transformation). *Let G be a graph labelled on Σ_E and $p : L \leftarrow K \to R$ a production meta-rule on β.*

The meta-rule p direct meta-transforms *G into a graph H labelled on Σ_E, denoted $G \Rightarrow_{p,O,m} H$, if there are a graph O with labels in β and a match morphism $m : L \times O \to G$ such that $G \Rightarrow_{p \times O,m} H$ is a direct graph transformation.*

Classically, a *graph meta-transformation*, or more precisely, a graph meta-transformation sequence, consists in zero or more direct graph transformations. We should notice that, a production rule without any meta-edge can be seen as a meta-rule on the empty set. Indeed, such production meta-rules and the corresponding classical production rule allow one to produce the same direct transformed graphs.

5 Consistency of G-Maps and Transformation Rules

We have already seen that applying a production meta-rule on a graph requires to find a matching morphism which satisfies the dangling condition. The following proposition shows that, as in proposition 1, in case of graphs in which each vertex has exactly one adjacent l-edge for each label l (i.e. the condition $\mathcal{C}$), the dangling condition uniquely depends on the form of the production meta-rule, and that the derivation then preserves the $\mathcal{C}$ property. Let us first define the extension of condition $\mathcal{C}$ (see proposition 1) to meta-graphs and meta-rules:

$\mathcal{C}_G(\Sigma_E)$ Let G be a graph on β. For each label $l \in \Sigma_E$, each vertex has exactly one adjacent edge s. t. either it is l-labelled or it is γ-labelled with $l \in \gamma(\beta)$;

$\mathcal{C}_p(\Sigma_E)$ Let p be the rule $L \leftarrow K \rightarrow R$ on β, $\mathcal{C}_{L\backslash K}(\Sigma_E)$ and $\mathcal{C}_{R\backslash K}(\Sigma_E)$ are satisfied and each preserved vertex of K has the same adjacent labelled edges in L and R (in the extended way).

Proposition 4. *Let $p : L \leftarrow K \rightarrow R$ be a production rule on β, O be a graph with labels in β and $m : L \times O \rightarrow G$ be a match morphism on a graph G with labels in Σ_E which satisfies the $\mathcal{C}_G(\Sigma_E)$ condition.*

1. *m satisfies the dangling condition iff O satisfies the $\mathcal{C}_O(\beta)$ condition and $L\backslash K$ satisfies the $\mathcal{C}_{L\backslash K}(\Sigma_E)$ condition.*
2. *Moreover, if the rule p satisfies the $\mathcal{C}_p(\Sigma_E)$ condition, then the derived graph H produced by the direct graph transformation $G \Rightarrow_{p,0,m} H$ satisfies the $\mathcal{C}_H(\Sigma_E)$ condition.*

Proof. Let us first prove the following lemma: for each vertex u of a meta-graph G and each vertex a of O, vertices u in G and (u, a) in $G \times O$ have the same labelled edges (in the extended way).

- If u has an adjacent l-labelled edge (u, l, v) in G, then (u, a) has an adjacent l-labelled edge $((u, a), l, (v, a))$ in $G \times O$;
- If u has an adjacent meta γ-edge in G, and a has an adjacent l-labelled edge (a, l, b) in O, such that $\gamma(l) \in \Sigma_E$, then (u, a) has an adjacent labelled edge $((u, a), \gamma(l), (u, b))$ in $G \times O$; And by definition, $G \times O$ has no other edges.

The proof of the proposition lies directly in this lemma. □

The condition of the proposition 4 ensures that a full γ-edge in the left-hand side of the meta-rule matches a complete $< \gamma(\beta) >$-orbit of the transformed graph and respectively full γ-edges of right-hand side match complete $< \gamma(\beta) >$-orbit of produced graph.

Thanks to proposition 4, a transformation of a G-map along the cone meta-rule of Fig. 4(a) preserves the $\mathcal{C}$ property of G-maps. Since each vertex of the cone meta-rule has exactly three links labelled by α_0, α_1 and α_2 (in extended way), the expanded rule (see Fig. 4(b) for example) has the same property.

Moreover, it is easy to prove that the consistency property of G-maps (see second condition of definition 2) is preserved by application of the cone meta-rule. Indeed, in the left-hand side, because d_1 has a α_2-loop, its α_0-neighbour has also an α_2-loop. Moreover, in all expanded rules along a graph O, because of $\mathcal{C}_O(\{\alpha_0, \alpha_1\})$, each expanded vertex has an α_2-loop and an α_0-neighbour. And thus, each expanded vertex has an $\alpha_0\alpha_2\alpha_0\alpha_2$ labelled cycle. For example, in the cone expansion Fig. 4(b), a_1b_1 and c_1d_1 are two $\alpha_0\alpha_2\alpha_0\alpha_2$ labelled cycles.

In the right-hand side, since d_1 and d_2 are α_2-linked together, their α_0-neighbours are also α_2-linked together. Indeed, because of $\mathcal{C}_O(\{\alpha_0, \alpha_1\})$, each vertex of O has an α_0-edge. And thus, each expanded vertex has an $\alpha_0\alpha_2\alpha_0\alpha_2$ cycle. In the cone rule example Fig. 4(b), $a_1b_1b_2a_2$ and $c_1d_1d_2c_2$ are two $\alpha_0\alpha_2\alpha_0\alpha_2$ labelled cycles. In the same way, d_3 and d_4 are α_0-linked together, thus their α_2-neighbours are also α_0-linked together. In the cone rule example Fig. 4(b), a_3a_4, $b_3b_4c_4c_3$ and d_3d_4 are three $\alpha_0\alpha_2\alpha_0\alpha_2$ labelled cycles.

6 Conclusion and Perspectives

In this paper we focus on the formalisation of complex topological operations on G-maps. Pursuing previous works, we propose a general class of meta-rules for G-maps which allows us to directly define a large class of topological operations, helpful in the context of modelling of complex structured systems, as the cone operation taken as illustration in the paper. We prove that thanks to strong G-maps constraints concerning edge labelling, the dangling condition of a meta-rule can be statically verified independently of the G-map on which it is applied. We will search for sufficient syntactical conditions on rules to ensure G-map consistency constraints.

This rule-based approach to specify topological evolution of objects will be useful for coupling transformations of objects with more classical rule-based approaches for simulating complex systems. In the context of modelling of complex biological systems, such a simulation paradigm has been broadly considered leading to an enormous amount of successfull applications [CFS06, RPS+04, Car05]. In these models, the compartmentalisation captures a static topology (focusing on exchange between compartments and molecular interactions) or simple topological modifications (resulting, for example, from endocytosis or exocytosis). Nevertheless, although biological systems are composed of molecules, the structure of the system and components both play essential roles in the biological functions of the system. Indeed the understanding of biological systems needs to take into account molecular phenomena (possibly abstracted by continuous concentrations), communication channels and space structuring of the cells at a same accuracy level. Thus it is an important challenge to understand the effects of spatial structure on the different concentrations, and reciprocally, the consequences of the evolution of concentrations on the spatial structure.

A general framework for rule-based simulations taking into account both molecular phenomena and subcellular compartment rearrangments would handle embedded G-maps. In previous work [PCG+07], we sketched out embedded G-maps by associating labels with vertices to represent geometric aspects (shapes of objects, distances between them, etc.) and by associating other labels to represent biochemical quantities (protein concentrations, protein fluxes through a subcellular wall, etc.). We have already used such topological transformation rules to simulate the evolution of biological subcellular compartments [PCLG+08]. To apply topological transformation rules, we have first to match the left-hand side of a rule. The pattern-matching problem is recognised as difficult in the general case of general graphs (without any constraint). In the particular case of G-maps, we have applied heuristics derived from usual G-map coverage involved in classical computer graphics operations. Our future works will then focus on the definition of embedded G-maps, and of associated graph transformation rules. Then it will be mandatory to study the conditions which ensure that the application of a transformation rule leads to another embedded G-maps in a coherent way.

References

[Car05] Cardelli, L., Calculi, B.: Interactions of biological membranes. In: Danos, V., Schachter, V. (eds.) CMSB 2004. LNCS (LNBI), vol. 3082, pp. 257–280. Springer, Heidelberg (2005)

[CFS06] Calzone, L., Fages, F., Soliman, S.: Biocham: an environment for modeling biological systems and formalizing experimental knowledge. Bioinformatics 22(14), 1805–1807 (2006)

[EEPT06] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. In: Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)

[HJE06] Hoffmann, B., Janssens, D., Van Eetvelde, N.: Cloning and expanding graph transformation rules for refactoring. Electr. Notes Theor. Comput. Sci. 152, 53–67 (2006)

[Hof05] Hoffmann, B.: Graph transformation with variables. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 101–115. Springer, Heidelberg (2005)

[Lie89] Lienhardt, P.: Subdivision of n-dimensional spaces and n-dimensional generalized maps. In: SCG 1989, pp. 228–236. ACM Press, New York (1989)

[Lie94] Lienhardt, P.: n-dimensional generalised combinatorial maps and cellular quasimanifolds. In: IJCGA (1994)

[PCG$^+$07] Poudret, M., Comet, J.-P., Le Gall, P., Arnould, A., Meseure, P.: Topology-based geometric modelling for biological cellular processes. In: LATA 2007, Tarragona, Spain, March 29 - April 4 (2007), http://grammars.grlmc.com/LATA2007/proc.html

[PCLG$^+$08] Poudret, M., Comet, J.-P., Le Gall, P., Képès, F., Arnould, A., Meseure, P.: Topology-based abstraction of complex biological systems: Application to the Golgi apparatus. Theory in Biosciences (2008)

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

[RPS$^+$04] Regev, A., Panina, E.M., Silverman, W., Cardelli, L., Shapiro, E.: Bioambients: an abstraction for biological compartments. Theor. Comput. Sci. 325(1), 141–167 (2004)

[Tut84] Tutte, W.: Graph Theory. Encyclopedia of Mathematics and its Applications, vol. 21. Addison-Wesley, Reading (1984)

Embedding and Confluence of Graph Transformations with Negative Application Conditions

Leen Lambers[1], Hartmut Ehrig[1], Ulrike Prange[1], and Fernando Orejas[2]

[1] Institute for Software Engineering and Theoretical Informatics
Technical University Berlin, Germany
{leen,ehrig,uprange}@cs.tu-berlin.de
[2] Department L.S.I, Technical University of Catalonia, Spain
orejas@lsi.upc.edu

Abstract. The goal of this paper is the generalization of embedding and confluence results for graph transformation systems to transformation systems with negative application conditions (NACs). These conditions restrict the application of a rule by expressing that a specific structure must not be present before or after applying the rule to a certain context. Such a condition influences each rule application and transformation and therefore changes significantly the properties of the transformation system. This behavior modification is reflected by the generalization of the Embedding Theorem and the Critical Pair Lemma or Local Confluence Theorem, formulated already for graph transformation systems without negative application conditions. The results hold for adhesive high-level replacement systems with NACs and are formulated in this paper for the instantiation to double-pushout graph transformation systems with NACs. All constructions and results are explained on a running example.

1 Introduction

In graph transformation, negative application conditions (NACs) express that certain structures at a given time are forbidden. They are a widely used feature for several applications of graph transformation e.g., [1,2]. In order to allow confluence analysis for these applications, the theory already worked out for graph transformation systems (gts) without NACs has to be generalized to gts with NACs. The notion of critical pairs is central in this theory. It was first developed in the area of term rewriting systems (e.g., [3]) and, later, introduced in the area of graph transformation for hyper-graph rewriting [4,5] and then for all kinds of transformation systems fitting into the framework of adhesive high-level replacement (HLR) categories [6]. We tailored the theory presented in this paper for gts with NACs and not for other kind of constraints or application conditions, since NACs are already widely used in practice.

For gts without NACs, embedding of a graph transformation sequence without NACs and local confluence of a gts without NACs has been investigated in detail in [6]. Recall that in order to be able to embed a transformation without

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 162–177, 2008.

NACs into some larger context by some extension morphism k, this morphism should be consistent as defined in [6]. Using the results on concurrency for graph transformation with NACs [7] we introduce in this paper the definition of NAC-consistency of an extension morphism. This is an additional condition on top of standard consistency enabling the generalization of the Embedding Theorem to transformations with NACs. Recall moreover that, for a gts without NACs, in order to be locally confluent it suffices that all critical pairs are strictly confluent. Having generalized the notion of critical pairs [8], completeness [8], and embedding to transformations with NACs in this paper, we moreover introduce a sufficient condition on the critical pairs with NACs. This condition implies local confluence of a gts with NACs as stated in the introduced Critical Pair Lemma with NACs. The proofs of these results are given in a technical report [9] on the level of adhesive HLR systems. In return, all results are illustrated in this paper by an example modeling order and payment transactions in a restaurant by typed graphs and rules with NACs.

The structure of this paper is as follows. In Section 2, we introduce preliminaries on gts with NACs and main results on concurrency for graph transformation with NACs. In Section 3, it is explained under which conditions it is possible to embed transformations with NACs. In Section 4, results on confluence of transformation systems with NACs are formulated. Section 5 concludes this paper with remarks on future work and a short summary.

2 Graph Transformation Systems with NACs

In this section, we reintroduce gts with NACs and some preliminary results that we need for the remaining paper. NACs are an important feature for the modeling of transformation systems, expressing that a certain structure is not present when performing the transformation [10] and thus enhancing the expressiveness of the transformation. In order to provide a rich theory for such transformations with NACs, they are integrated into the framework of adhesive HLR systems [6]. In [7] it is remarked that gts with NACs are a valid instantiation of adhesive HLR systems with NACs. In this paper, we concentrate on formulating the results for graph transformation with NACs and showing their significance on an example.

Definition 1 (typed graph and graph morphism)
A graph $G = (G_E, G_V, s, t)$ *consists of a set* G_E *of edges, a set* G_V *of vertices and two mappings* $s, t : G_E \to G_V$*, assigning to each edge* $e \in G_E$ *a source* $q = s(e) \in G_V$ *and target* $z = t(e) \in G_V$*. A* graph morphism $f : G_1 \to G_2$ *between two graphs* $G_i = (G_{i,E}, G_{i,V}, s_i, t_i)$*,* $(i = 1, 2)$ *is a pair* $f = (f_E : G_{E,1} \to G_{E,2}, f_V : G_{V,1} \to G_{V,2})$ *of mappings, such that* $f_V \circ s_1 = s_2 \circ f_E$ *and* $f_V \circ t_1 = t_2 \circ f_E$*. A* type graph *is a distinguished graph* TG*. A* typed graph $G^T : (G, type)$ *over* TG *is a graph* G *and a graph morphism* $type : G \to TG$*. A* typed graph morphism $f : G_1^T \to G_2^T$ *is a graph morphism* $f : G_1 \to G_2$ *with* $type_2 \circ f = type_1$*.*

From now on we only consider typed graphs and morphisms over a given type graph TG and omit the prefix typed and the index T in our notations.

Definition 2 (injective, surjective, overlapping, pair factorization). *A graph morphism $f : G_1 \to G_2$ is injective (resp. surjective) if f_V and f_E are injective (resp. surjective) mappings. Two graph morphisms $m_1 : L_1 \to G$ and $m_2 : L_2 \to G$ are* jointly surjective *if $m_{1,V}(L_{1,V}) \cup m_{2,V}(L_{2,V}) = G_V$ and $m_{1,E}(L_{1,E}) \cup m_{2,E}(L_{2,E}) = G_E$. A pair of jointly surjective morphisms (m_1, m_2) is also called an* overlapping *of L_1 and L_2. A* pair factorization *of two graph morphisms $(m_1 : G_1 \to H, m_2 : G_2 \to H)$ consists of a pair of jointly surjective morphisms $(e_1 : G_1 \to E, e_2 : G_2 \to E)$ and an injective morphism $m : E \to H$ such that $m \circ e_1 = m_1$ and $m \circ e_2 = m_2$ and is unique up to isomorphism.*

Definition 3 (rule and match). *A graph transformation* rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ *consists of a rule name p and a pair of injective graph morphisms $l : K \to L$ and $r : K \to R$. The graphs L, K and R are called the left-hand side (lhs), the interface, and the right-hand side (rhs) of p, respectively. Given a rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ and a graph G, one can try to apply p to G if there is an occurrence of L in G i.e. a graph morphism, called* match $m : L \to G$.

A negative application condition or NAC as introduced in [10] forbids a certain graph structure to be present before or after applying a rule.

Definition 4 (negative application condition)

$$\begin{tikzcd}
L \arrow[r, "n"] \arrow[d, "g"] & N \arrow[dl, dotted, "X" description, "q"'] \\
G &
\end{tikzcd}$$

- *A* negative application condition *or $NAC(n)$* on *L is a graph morphism $n : L \to N$. A graph morphism $g : L \to G$* satisfies *$NAC(n)$ on L i.e. $g \models NAC(n)$ if and only if $\nexists\ q : N \to G$ which is injective such that $q \circ n = g$.*
- *A $NAC(n)$ on L (resp. R) for a rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ is called* left *(resp.* right*) NAC on p. $NAC_{p,L}$ (resp. $NAC_{p,R}$) is a set of left (resp. right) NACs on p. $NAC_p = (NAC_{p,L}, NAC_{p,R})$, consisting of a set of left and a set of right NACs on p is called a set of NACs on p.*

Definition 5 (graph transformation with NACs)

- *A* graph transformation system with NACs *is a set of rules where each rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ has a set $NAC_p = (NAC_{p,L}, NAC_{p,R})$ of NACs on p.*
- *A* direct graph transformation *$G \overset{p,g}{\Rightarrow} H$ via a rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ with $NAC_p = (NAC_{p,L}, NAC_{p,R})$ and a match $g : L \to G$ consists of the double pushout [11] (DPO) at the right where g satisfies each NAC in $NAC_{p,L}$, written $g \models NAC_{p,L}$, and h satisfies each NAC in $NAC_{p,R}$, written $h \models NAC_{p,R}$. Since pushouts in* **Graph** *always exist, the DPO can be constructed if the pushout complement of $K \to L \to G$ exists. If so, we say that the match g satisfies the gluing condition of rule p. A* graph transformation*, denoted as $G_0 \overset{*}{\Rightarrow} G_n$, is a sequence $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ of direct graph transformations.*

$$\begin{tikzcd}
L \arrow[d, "g"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d, "h"] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Remark 1. From now on we only consider gts with rules having an empty set of right NACs. This is without loss of generality, because each right NAC can be translated into an equivalent left NAC as explained in [6], where Theorem 7.17 can be specialized to NACs as remarked in [7].

Example 1. Here we introduce our running example *Restaurant* modeling order and payment transactions in a restaurant. The type and start graph of *Restaurant* are depicted in Fig. 1. Note that all results reviewed and introduced in this paper hold in particular for typed gts with NACs, since they are a valid instantiation of adhesive HLR systems with NACs. The rule *openAccountTable* shown in Fig. 1 models the opening of an account for one of the tables in the restaurant. This rule holds two NACs: *notOpened* forbids the rule to be applied twice to the same table and *noAlert* specifies that an account can not be opened if there exists an alert for the table. An alert can be generated by rule *alertUnpaidTable* if a staff member of the restaurant notices that guests have left a table without paying. In this case an exception handling starts making sure that unpaid orders are considered when doing the daily accounting. The NAC *noAlert* for this rule avoids it to be applied if an alert for the table already exists. *noteUnpaidOrder* notes an unpaid order for a table by deleting it. *resetAlert* can reset the alert for a table if it e.g. appeared to be a false alarm or all unpaid orders are processed by *noteUnpaidOrder* in the meanwhile. Furthermore, *gatherOrder* can assign an order for a table if there is no alert, *payOrder* models the paying of an order for the case that there is no alert expressed by NAC *noAlert* and finally *closeAccountTable* can close the account of a table if all orders of a table are processed which is expressed by NAC *noOrders*. Note that the rules *payOrder* and *noteUnpaidOrder* have the same effect on the system since they both just delete an order, but the first one can only be applied if there is no alert in contrast to the second one. Of course it is possible to augment this system with information on the price of the order, keeping track of a list of paid resp. unpaid orders etc. For the purpose of this paper though we restrict ourselves to these more simple operations.

In the following sections, we repeatedly need two constructions translating NACs via morphisms and rules. More precisely, the mapping D translates NACs downwards along morphisms. Given a diagram as depicted in Fig. 2, consider a $NAC_{L'_c}$ on L'_c and a morphism m_c, then $D_{m_c}(NAC_{L'_c})$ translates $NAC_{L'_c}$ into equivalent NACs on L_c. The basic idea of the construction of D is to consider all suitable overlappings of L_c and $NAC_{L'_c}$. The mapping DL translates NACs down- and leftwards, i.e. given the diagram in Fig. 2 with NAC_{L_n} on L_n, a morphism e_n, and a rule $L_c \leftarrow C_c \rightarrow E$, then $DL_{p_c}(NAC_{L_n})$ translates the NACs on L_n to equivalent NACs on L_c. The construction of DL is based on D translating NAC_{L_n} to equivalent NACs on E, and then on the well-known construction of right to left application conditions [6]. For more details see [7,9].

Now we introduce the concurrent rule with NACs p_c induced by a transformation with NACs $t : G_0 \overset{n+1}{\Longrightarrow} G_{n+1}$ via a sequence of rules $p_0, \ldots, p_n$. Intuitively, a concurrent rule with NACs summarizes in one rule which parts of a graph G_0

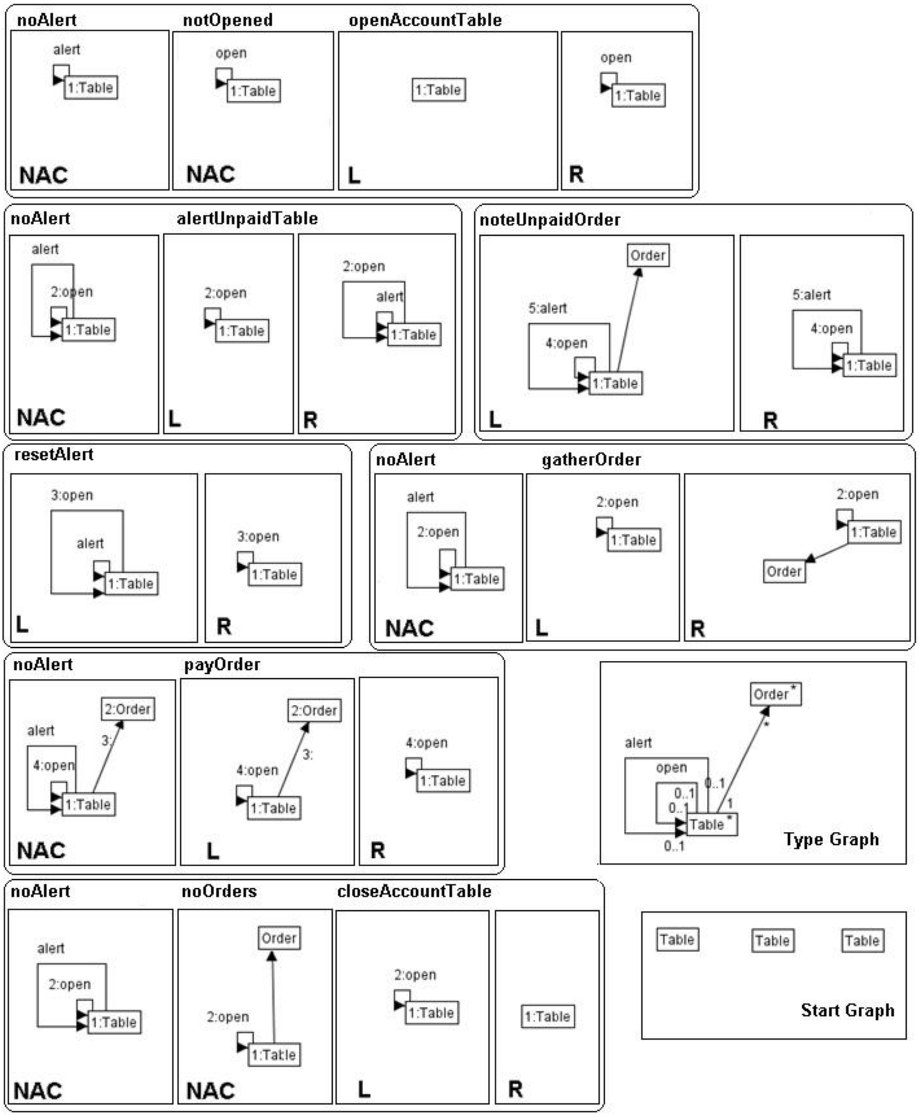


Fig. 1. Start graph, type graph and rules of *Restaurant*

should be present, preserved, deleted, and produced by t. Moreover we have a summarized set of NACs on the concurrent rule p_c expressing which graph parts are forbidden when applying $p_0, \ldots, p_n$ to G_0 leading to t. Note that in [7,9] it is proven that it is possible to repeat the transformation t in one step via the concurrent rule p_c with NACs. In addition, whenever it is possible to apply a concurrent rule with NACs p_c it is also possible to resequentialize this one-step

$$\begin{tikzcd}
L_0 \arrow[r, "m'_c"] & L'_c \arrow[d, dotted, "m_c"] \arrow[ddd, bend right, "g'_c"'] & K'_c \arrow[l] \arrow[r] \arrow[d, dotted] & R'_c \arrow[dr, dotted, "e'_c"'] \arrow[dddr, "h'_c"] & & L_n \arrow[dl, dotted, "e_n"] \arrow[dddl, "g_n"'] & K_n \arrow[l] \arrow[r] \arrow[d, dotted] & R_n \arrow[d, dotted] \arrow[ddd, bend left, "g_{n+1}"] \\
 & L_c \arrow[dd, dotted, "g_c"] & C_c \arrow[l, dotted, "l"'] \arrow[rr, dotted] \arrow[dd, dotted] & (1) & E \arrow[dd, dotted, "h"] & & C_n \arrow[ll, dotted] \arrow[r, dotted, "r"] \arrow[dd, dotted] & R_c \arrow[dd, dotted, "h_c"'] \\
 & & & K_c \arrow[ul, dotted, "k_c"] \arrow[urrr, dotted, "k_n"'] & & & & \\
 & G_0 & D_n \arrow[l] \arrow[rr] & & G_n & & D \arrow[ll] \arrow[r] & G_{n+1}
\end{tikzcd}$$

Fig. 2. Definition of concurrent rule with NACs

transformation into single steps via the sequence of original rules with NACs which led to this concurrent rule.

Definition 6 (concurrent rule with NAC, concurrent (co-, lhs-)match induced by $G_0 \overset{n+1}{\Longrightarrow} G_{n+1}$)

$n = 0$ *For a direct transformation $G_0 \Rightarrow G_1$ via match $g_0 : L_0 \to G_0$, comatch $g_1 : R_0 \to G_1$ and rule $p_0 : L_0 \leftarrow K_0 \to R_0$ with NAC_{p_0} the* concurrent rule p_c with NAC *induced by $G_0 \Rightarrow G_1$ is defined by $p_c = p_0$ with $NAC_{p_c} = NAC_{p_0}$, the* concurrent comatch h_c *is defined by $h_c = g_1$, the* concurrent lhs-match *by $id : L_0 \to L_0$ and the* concurrent match g_c *by $g_c = g_0 : L_0 \to G_0$.*

$n \geq 1$ *Consider $p'_c : L'_c \leftarrow K'_c \to R'_c$ (resp. $g'_c : L'_c \to G_0$, $h'_c : R'_c \to G_n$,$m'_c : L_0 \to L'_c$), the concurrent rule with NACs (resp. concurrent match, comatch, lhs-match) induced by $G_0 \overset{n}{\Longrightarrow} G_n$. Let $((e'_c, e_n), h)$ be the pair factorization of the comatch h'_c and match g_n of $G_n \Rightarrow G_{n+1}$. According to Fact 5.29 in [6] PO-PB decomposition, PO composition and decomposition lead to the diagram in Fig. 2 in which (1) is a pullback and all other squares are pushouts. For a transformation sequence $G_0 \overset{n+1}{\Longrightarrow} G_{n+1}$ the* concurrent rule p_c with NACs *(resp. concurrent match, comatch, lhs-match) induced by $G_0 \overset{n+1}{\Longrightarrow} G_{n+1}$ is defined by $p_c = L_c \overset{l \circ k_c}{\leftarrow} K_c \overset{r \circ k_n}{\to} R_c$ ($g_c : L_c \to G_0$, $h_c : R_c \to G_{n+1}$, $m_c \circ m'_c : L_0 \to L_c$). Thereby NAC_{p_c} is defined by $NAC_{p_c} = DL_{p_c}(NAC_{L_n}) \cup D_{m_c}(NAC_{L'_c})$.*

Example 2. Consider the graph *Middle2Orders* depicted in Fig. 3 and a transformation t : *Middle2Orders* $\overset{alertUnpaidTable}{\Rightarrow} G_1 \overset{noteUnpaidOrder}{\Rightarrow} G_2 \overset{noteUnpaidOrder}{\Rightarrow} G_3 \overset{resetAlert}{\Rightarrow} G_4 \overset{closeAccountTable}{\Rightarrow} G_5 \overset{closeAccountTable}{\Rightarrow}$ *Startgraph* in which an alert is generated for the middle table, consequently both orders on the middle table are noted as unpaid, the alert is then reset, the middle table account is closed and the right table account is closed as well. The lhs L_c and rhs R_c of the concurrent rule p_c induced by transformation t is depicted in Fig. 3 together with the concurrent transformation via p_c summarizing t into one step. It deletes two

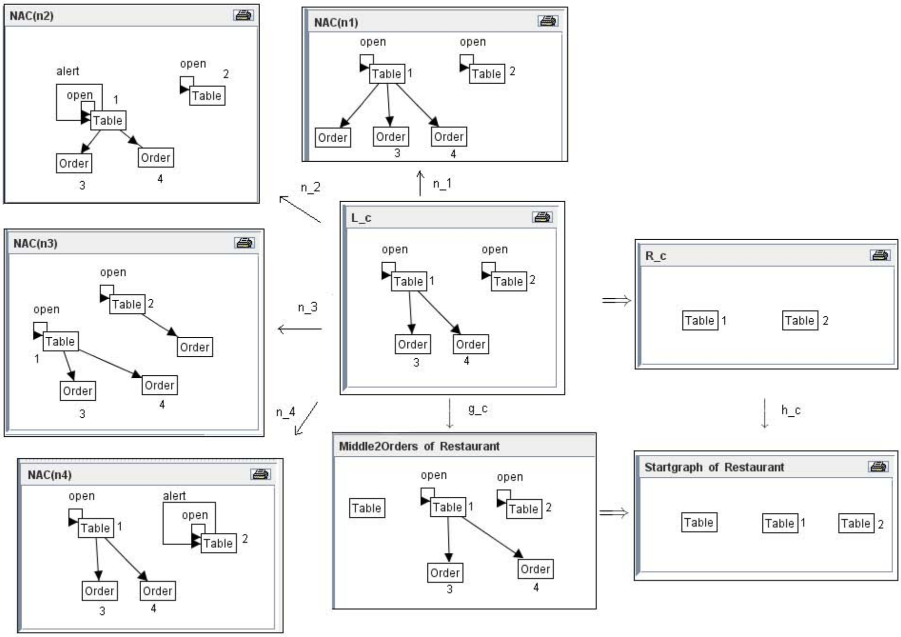


Fig. 3. Concurrent rule and transformation with NACs

different orders belonging to the middle table, closes its account and in addition closes the account of the right table. Note that the node ids in this figure define the morphisms. The concurrent NAC_{p_c} induced by t holds a $NAC(n_1)$ forbidding more than two orders, a $NAC(n_2)$ forbidding an alert for the table holding already two orders, a $NAC(n_3)$ forbidding any order, and $NAC(n_4)$ forbidding an alert for the other table. Note that the same $NAC(n_2)$ is induced by the NACs of *alertUnpaidTable* and *closeAccountTable* applied to the middle table.

Finally, for the Embedding Theorem with NACs we reintroduce the notion of extension diagram with NACs [8].

Definition 7 (extension diagram with NACs)

$$\begin{tikzcd}
G_0 \arrow[r, Rightarrow, "t"', "*"] \arrow[d, "k_0"'] \arrow[dr, phantom, "(1)"] & G_n \arrow[d, "k_n"] \\
G'_0 \arrow[r, Rightarrow, "t'"', "*"] & G'_n
\end{tikzcd}$$

An extension diagram *is a diagram (1), where,* $k_0 : G_0 \to G'_0$ *is a graph morphism, called extension morphism, and* $t : G_0 \overset{*}{\Rightarrow} G_n$ *and* $t' : G'_0 \overset{*}{\Rightarrow} G'_n$ *are graph transformations via the same rules* $(p_0, \cdots, p_{n-1})$ *with NACs, and matches* $(m_0, \cdots, m_{n-1})$ *and extended matches* $(k_0 \circ m_0, \cdots, k_{n-1} \circ m_{n-1})$*, respectively, defined by the DPO diagrams on the right for each* p_i*. Since* t *and* t' *are transformations with NACs, the matches* $(m_0, \cdots, m_{n-1})$ *and extended matches* $(k_0 \circ m_0, \cdots, k_{n-1} \circ m_{n-1})$ *have to satisfy the NACs of the rules* $(p_0, \cdots, p_{n-1})$*.*

$$\begin{tikzcd}
L_i \arrow[d, "m_i"'] & K_i \arrow[l, "l_i"] \arrow[r, "r_i"'] \arrow[d, "j_i"] & R_i \arrow[d, "n_i"] \\
G_i \arrow[d, "k_i"'] & D_i \arrow[l, "f_i"] \arrow[r, "g_i"'] \arrow[d, "d_i"] & G_{i+1} \arrow[d, "k_{i+1}"] \\
G'_i & D'_i \arrow[l, "f'_i"] \arrow[r, "g'_i"'] & G'_{i+1}
\end{tikzcd}$$

3 Embedding of Transformations with NACs

Recall [6] that in order to be able to embed a transformation without NACs into some other context by an extension morphism k, this morphism should not identify graph elements which are deleted and on the other hand preserved by the transformation. Moreover k should not map any node which is deleted by the transformation to a node which is the source or target of an additional edge in the new context. This condition on the extension morphism k can be checked by computing its boundary and context, and checking then consistency as defined in [6]. Combined with the results on concurrency for graph transformation with NACs [7] it is possible to define also NAC-consistency of an extension morphism. This is an additional condition needed on top of standard consistency to generalize the embedding of transformations to transformations with NACs. Note that in order to make the difference between consistency and NAC-consistency of an extension morphism clear, we call consistency from now on *boundary consistency.*

Now we can formulate the definition of NAC-consistency for an extension morphism k_0 w.r.t. a transformation t. It expresses that the extended concurrent match induced by t should fulfill the concurrent NAC induced by t.

Definition 8 (NAC-consistency). *A morphism $k_0 : G_0 \to G'_0$ is called NAC-consistent w.r.t. a transformation $t : G_0 \overset{*}{\Rightarrow} G_n$ if $k_0 \circ g_c \models NAC_{p_c}$ with NAC_{p_c} the* concurrent NAC *and g_c the* concurrent match induced by *t.*

The Embedding Theorem for rules with NACs needs NAC-consistency of the extension morphism k_0 on top of boundary consistency. Note that in [9] also the Extension Theorem with NACs is proven describing the fact that boundary and NAC-consistency are not only sufficient, but also necessary conditions for the construction of extension diagrams for transformations with NACs.

Theorem 1 (Embedding Theorem with NACs [9])

Given a transformation $t : G_0 \overset{n}{\Longrightarrow} G_n$ with NACs. If $k_0 : G_0 \to G'_0$ is boundary consistent and NAC-consistent w.r.t. t then there exists an extension diagram with NACs over t and k_0 as defined in Def. 7 and depicted on the right.

$$\begin{tikzcd}
G_0 \arrow[r, Rightarrow, "t"] \arrow[d, "k_0"'] \arrow[dr, phantom, "(1)"] & {}^{*}G_n \arrow[d, "k_n"] \\
G'_0 \arrow[r, Rightarrow, "t'"] & {}^{*}G'_n
\end{tikzcd}$$

Example 3. Consider the transformation $t : Middle2Orders \overset{*}{\Rightarrow} Startgraph$ with its concurrent rule p_c and NAC_{p_c} as described in Example 2 and depicted as concurrent transformation in Fig. 3. In addition, consider an inclusion morphism $k : Middle2Orders \to Middle3Orders$ in which the graph $Middle3Orders$ has an additional order for the middle table. The morphism k is not NAC-consistent, since the concurrent NAC induced by t is not satisfied by $k \circ g_c$. This is because $NAC(n_1)$ forbidding more than 2 orders on the middle table is not satisfied. Thus it is not possible to embed transformation t into *Middle3Orders* by the extension morphism k, since k is not NAC-consistent. Intuitively speaking, it is not possible to change the state of the tables in the restaurant in the same way as transformation t if the middle table holds an extra order. This is because rule *closeAccountTable* would forbid closing the account for the middle table still holding one order. Consider a different inclusion morphism $k' : \mathit{Middle2Orders} \rightarrow \mathit{4Tables}$ in which *4Tables* just holds an extra table. Now k' is NAC-consistent and it is possible to embed t into *4Tables*. Intuitively speaking, the embedded transformation t' changes the states of the tables in the restaurant in the same way as transformation t, but does this in a restaurant with an extra table.

4 Confluence of Transformations with NACs

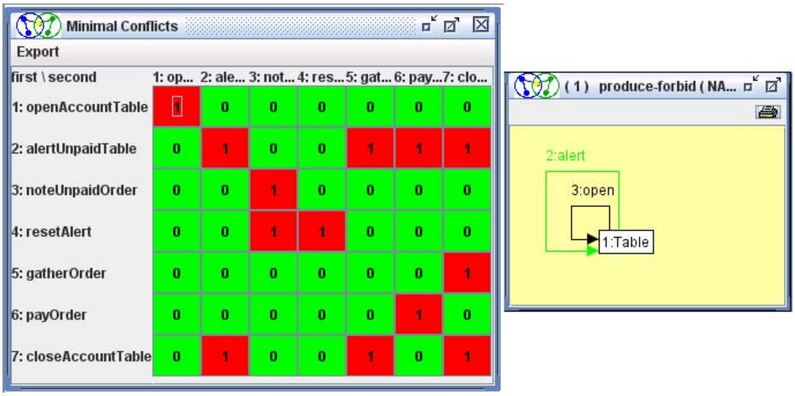
Local confluence of a transformation system without NACs can be inferred from the strict confluence of all critical pairs (see Critical Pair Lemma [6]). If a critical pair is strictly confluent via some transformations t_1 and t_2, we say that (t_1, t_2) is a *strict solution* of the critical pair. Intuitively speaking, strict confluence means that the common structure which is preserved by the critical pair should be preserved by the strict solution of this critical pair as well. For the Critical Pair Lemma with NACs we need a stronger condition though to obtain local confluence of the transformation system. In addition to strict confluence of all critical pairs we need also NAC-confluence. NAC-confluence of a critical pair ensures that for each context into which the critical pair can be embedded, such a strict solution can be embedded into this context as well without violating the NACs present in t_1 and t_2. In particular, a critical pair is NAC-confluent if the NAC-consistency (as defined in Def. 8) of each extension morphism w.r.t. a strict solution of the critical pair follows from the NAC-consistency of the extension morphism w.r.t. the critical pair itself.

First we state the definition of a critical pair with NACs. A critical pair describes a conflict between two rules in a minimal context. Therefore we consider in the following only overlaps of graphs in order to rule out superfluous context. Moreover, it is proven in [8] that the following critical pair definition satisfies completeness. This means intuitively that for each pair of conflicting transformations there exists a critical pair expressing the same conflict in a minimal context.

Definition 9 (critical pair). *A critical pair is a pair of direct transformations* $K \overset{(p_1,m_1)}{\Rightarrow} P_1$ *with* NAC_{p_1} *and* $K \overset{(p_2,m_2)}{\Rightarrow} P_2$ *with* NAC_{p_2} *such that:*

1. (a) $\nexists h_{12} : L_1 \rightarrow D_2 : d_2 \circ h_{12} = m_1$ *and* (m_1, m_2) *jointly surjective*
   (use-delete-conflict)
   or
   (b) *there exists* $h_{12} : L_1 \rightarrow D_2$ *s.t.* $d_2 \circ h_{12} = m_1$, *but for one of the NACs* $n_1 : L_1 \rightarrow N_1$ *of* p_1 *there exists an injective morphism* $q_{12} : N_1 \rightarrow P_2$ *s.t.* $q_{12} \circ n_1 = e_2 \circ h_{12}$ *and* (q_{12}, h_2) *jointly surjective (forbid-produce-conflict)*

   or
2. (a) $\nexists h_{21} : L_2 \rightarrow D_1 : d_1 \circ h_{21} = m_2$ *and* (m_1, m_2) *jointly surjective*
   (delete-use-conflict)
   or



first \ second	1: op...	2: ale...	3: not...	4: res...	5: gat...	6: pay...	7: clo...
1: openAccountTable	1	0	0	0	0	0	0
2: alertUnpaidTable	0	1	0	0	1	1	1
3: noteUnpaidOrder	0	0	1	0	0	0	0
4: resetAlert	0	0	1	1	0	0	0
5: gatherOrder	0	0	0	0	0	0	1
6: payOrder	0	0	0	0	0	1	0
7: closeAccountTable	0	1	0	0	1	0	1

Fig. 4. Conflict Matrix for *Restaurant* and minimal context for *(alertUnpaidTable,gatherOrder)*

(b) there exists $h_{21} : L_2 \to D_1$ *s.t.* $d_1 \circ h_{21} = m_2$*, but for one of the NACs* $n_2 : L_2 \to N_2$ *of* p_2 *there exists an injective morphism* $q_{21} : N_2 \to P_1$ *s.t.* $q_{21} \circ n_2 = e_1 \circ h_{21}$ *and* (q_{21}, h_1) *jointly surjective (produce-forbid-conflict).*

$$\begin{tikzcd}
 & & N_1 \arrow[rr, dotted, no head] \arrow[ddrrrr, dotted, "q_{12}"] & & N_2 \arrow[ddllll, dotted, "q_{21}"'] & & \\
R_1 \arrow[d, "h_1"'] & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d] & L_1 \arrow[u, "n_1"'] \arrow[dr, "m_1"] \arrow[drrr, dotted, "h_{12}"] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"'] \arrow[dlll, dotted, "h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] \arrow[d] & R_2 \arrow[d, "h_2"] \\
P_1 & D_1 \arrow[l, "e_1"] \arrow[rr, "d_1"'] & & K & & D_2 \arrow[ll, "d_2"] \arrow[r, "e_2"'] & P_2
\end{tikzcd}$$

Example 4. All critical pairs of a gts can be computed by the graph transformation tool AGG [12]. They are illustrated by a Conflict Matrix as for our *Restaurant* example in the left part of Fig. 4. More precisely, entry (p_j, p_i) (row, column) in this matrix denotes the number of critical pairs $K \overset{(p_j,m_j)}{\Rightarrow} P_j$ with NAC_{p_j} and $K \overset{(p_i,m_i)}{\Rightarrow} P_i$ with NAC_{p_i} describing delete-use and produce-forbid-conflicts as defined in Def. 9. Consider in particular entry $(alertUnpaidTable, gatherOrder)$. This critical pair expresses a produce-forbid conflict, since when an alert is set for a certain table it is impossible to gather an order for it afterwards. The minimal context expressing this conflict is in the right part of Fig. 4 depicting graph P_j. The critical pair itself is depicted in the left part of Fig. 5.

Definition 10 (strict NAC-confluence). *A critical pair* $P_1 \overset{p_1,g_1}{\Leftarrow} K \overset{p_2,g_2}{\Rightarrow} P_2$ *is* strictly NAC-confluent *if*

- *it is* strictly confluent *via some transformations* $t_1 : K \overset{p_1,g_1}{\Rightarrow} P_1 \Rightarrow^* X$ *and* $t_2 : K \overset{p_2,g_2}{\Rightarrow} P_2 \Rightarrow^* X$ *(see [6])*
- *and it is* NAC-confluent for t_1 and t_2*, i.e. for every injective morphism* $k_0 : K \rightarrow G$ *which is NAC-consistent w.r.t.* $K \overset{p_1,g_1}{\Rightarrow} P_1$ *and* $K \overset{p_2,g_2}{\Rightarrow} P_2$ *it follows that* k_0 *is NAC-consistent w.r.t.* t_1 *and* t_2*.*

$$\begin{tikzcd} & & P_1 \arrow[dr, Rightarrow, "*"] & \\ G & K \arrow[l, "k_0"'] \arrow[ur, Rightarrow, "{(p_1,g_1)}"] \arrow[dr, Rightarrow, "{(p_2,g_2)}"'] & & X \\ & & P_2 \arrow[ur, Rightarrow, "*"'] & \end{tikzcd}$$

Remark 2. Injectivity of k_0 is sufficient by completeness of critical pairs [8].

Example 5. Consider again the Conflict Matrix of *Restaurant* depicted in Fig. 4. Let us investigate the following critical pairs for strict NAC-confluence:

- *(resetAlert,noteUnpaidOrder).* It describes a delete-use-conflict, since *resetAlert* deletes the alert and *noteUnpaidOrder* can only be applied if an alert is set for the table. This critical pair can be resolved by a strict solution by on the one hand applying *payOrder* and on the other hand *resetAlert*. Now we investigate if this critical pair is also NAC-confluent for this solution. Rule *resetAlert* does not hold a NAC and therefore there is nothing to prove. On the other hand *payOrder* holds a NAC which forbids an alert on the table. The rules *resetAlert* and *noteUnpaidOrder* can only be applied if an alert on the table is present. After applying *resetAlert* though this alert is in any case deleted. The only problem that can occur is that in another context more than one alert is present for a table. Cardinality constraints on the type graph of *Restaurant* as depicted in Fig. 1 forbid this possibility though. Therefore we can conclude that this critical pair is strictly NAC-confluent.
- *(gatherOrder,closeAccountTable).* It describes a produce-forbid-conflict, since *gatherOrder* produces an order which is forbidden by *closeAccountTable*. This pair can be resolved on the one hand by paying the order and closing the table account and on the other hand nothing. This solution demands no alert on the table because of NAC *noAlert* on *payOrder* and no orders because of NAC *noOrders* on *closeAccountTable*. It becomes clear that this critical pair is NAC-confluent for this solution because it only occurs on a table without an alert and without any order. These are exactly the restrictions for which the above solution holds.
- *(alertUnpaidTable,gatherOrder).* We described this critical pair already in Example 4 and it is depicted in Fig. 5. There exists a strict solution for this critical pair resetting the alert on the one hand and paying the order on the other hand. This solution is depicted also in Fig. 5 (part without rounded rectangle). This critical pair is NAC-confluent for this solution since *alertUnpaidTable* can be applied only on a table without any order and therefore when paying the order there will not be any alert either.

  Consider though a somewhat larger strict solution for this critical pair, namely on the one hand resetting the alert and closing the account and on the other hand paying the order and then closing the account. In Fig. 5 this means we consider now as well the rounded rectangle. The critical pair is

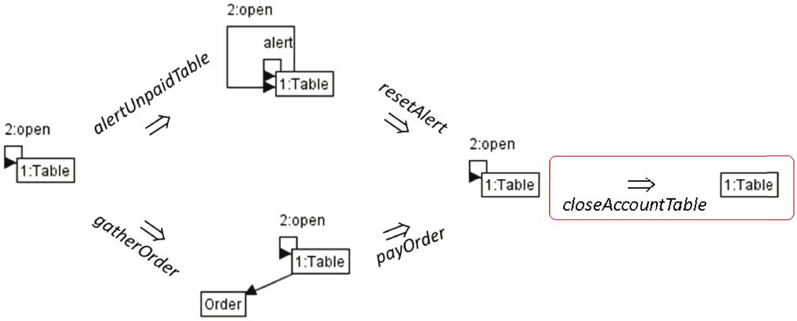


Fig. 5. Critical pair *(alertUnpaidTable,gatherOrder)* with its solution

not NAC-confluent for this solution. This is because *closeAccountTable* has a NAC *noOrders* which is not present in the NACs of the critical pair. This means that the graph consisting of a table with an open account could be embedded into a graph with a table with an open account and some order already present. In this case it is not possible to apply the same strict solution to the produce-forbid-conflict, since there are too many orders on the table. This example demonstrates very nicely why in this case the minimal context in which the conflict is resolved is not sufficient to resolve the conflict also in any other valid context in the same way.

Theorem 2 (Local Confluence Theorem - Critical Pair Lemma with NACs [9]). *Given a gts with NACs, it is locally confluent if all its critical pairs are strictly NAC-confluent.*

Example 6. Our running example transformation system *Restaurant* is locally confluent, since all critical pairs are strictly NAC-confluent. Consider again the Conflict Matrix of *Restaurant* as depicted in Fig. 4.

- All critical pairs on the diagonal of the matrix are trivially strictly NAC-confluent since they are of the form $P \overset{p,g}{\Leftarrow} K \overset{p,g}{\Rightarrow} P$.
- *(gatherOrder,closeAccountTable)* has been discussed in Example 5.
- *(closeAccountTable,gatherOrder)* denotes a critical pair in delete-use-conflict and can be resolved analogously to *(gatherOrder,closeAccountTable)*.
- *(closeAccountTable,alertUnpaidTable)* denotes a critical pair in delete-use-conflict and can be resolved, without transformation on the result of *closeAccountTable*, just applying the rules *resetAlert* and then *closeAccountTable* to the result of *alertUnpaidTable*. Thereby *closeAccountTable* can be applied since an alert can only be generated if no alert was there yet. Therefore after resetting this one alert, there will be no alert anymore on the table. Moreover there are no orderings on the table, since the table could be closed already at the start of the transformation.
- *(alertUnpaidTable,gatherOrder)* has been discussed in Example 5.
- *(alertUnpaidTable,payOrder)* will be discussed in Example 8.

- *(alertUnpaidTable,closeAccountTable)* denotes a critical pair in produce-forbid-conflict and can be resolved analogously to *(closeAccountTable, alertUnpaidTable)*.
- *(resetAlert,noteUnpaidOrder)* has been discussed in Example 5

Given a critical pair and a strict solution for it, it would be desirable to have a constructive method to check for NAC-confluence of this solution. Therefore the following theorem formulates a constructive sufficient condition for a critical pair $P_1 \overset{p_1}{\Leftarrow} K \overset{p_2}{\Rightarrow} P_2$ which is strictly confluent via some transformations $t_1 : K \overset{p_1,g_1}{\Rightarrow} P_1 \Rightarrow^* X$ and $t_2 : K \overset{p_2,g_2}{\Rightarrow} P_2 \Rightarrow^* X$ to be also NAC-confluent for t_1 and t_2. This means by definition that for every injective extension morphism $k_0 : K \to G$ which is NAC-consistent w.r.t. $K \overset{p_1,g_1}{\Rightarrow} P_1$ and $K \overset{p_2,g_2}{\Rightarrow} P_2$ it follows that k_0 is also NAC-consistent w.r.t. t_1 and t_2. In the following theorem two different conditions on each single $NAC(n_{1,j})$ (resp. $NAC(n_{2,j})$) of the concurrent NAC induced by transformation t_1 (resp. t_2) are given which lead to NAC-confluence if one of them is satisfied. The first condition expresses that there exists a suitable NAC on p_1 (resp. p_2) which evokes the satisfaction of $NAC(n_{1,j})$ (resp. $NAC(n_{2,j})$). The second condition first asks for a suitable morphism between the lhs's of the concurrent rules induced by both transformations t_1 and t_2. Moreover it expresses that there exists a suitable NAC on p_2 (resp. p_1) which evokes the satisfaction of $NAC(n_{1,j})$ (resp. $NAC(n_{2,j})$). Note that in the following theorem Def. 6 is used in order to refer to a concurrent rule, match and lhs-match induced by a transformation t, and it is referred to the downward translation of a NAC on L via a morphism $m_c : L \to L_c$ to a set of equivalent NACs on L_c denoted as $D_{m_c}(NAC_L)$ and defined explicitly in [7].

Theorem 3 (Sufficient Condition for NAC-confluence). *Given a critical pair $P_1 \overset{p_1}{\Leftarrow} K \overset{p_2}{\Rightarrow} P_2$ which is strictly confluent via the transformations $t_1 : K \overset{p_1,g_1}{\Rightarrow} P_1 \Rightarrow^* X$ and $t_2 : K \overset{p_2,g_2}{\Rightarrow} P_2 \Rightarrow^* X$. Let $L_{c,1}$ (resp. $L_{c,2}$) be the left-hand side of the concurrent rule $p_{c,1}$ (resp. $p_{c,2}$), $m_{c,1} : L_1 \to L_{c,1}$ (resp. $m_{c,2} : L_2 \to L_{c,2}$) the lhs-match and $g_{c,1} : L_{c,1} \to K$ (resp. $g_{c,2} : L_{c,2} \to K$) the concurrent match induced by t_1 (resp. t_2). Then the critical pair $P_1 \overset{p_1}{\Leftarrow} K \overset{p_2}{\Rightarrow} P_2$ is also NAC-confluent for t_1 and t_2 and thus strictly NAC-confluent if one of the following conditions holds for each $NAC(n_{1,j}) : L_{c,1} \to N_{1,j}$ (resp. $NAC(n_{2,j}) : L_{c,2} \to N_{2,j}$) of the concurrent $NAC_{p_{c,1}}$ induced by t_1 (resp. $NAC_{p_{c,2}}$ induced by t_2)*

- *there exists a $NAC(n'_{1,i}) : L_{c,1} \to N'_{1,i}$ (resp. $NAC(n'_{2,i}) : L_{c,2} \to N'_{2,i}$) in $D_{m_{c,1}}(NAC_{L_1})$ (resp. $D_{m_{c,2}}(NAC_{L_2})$) and an injective morphism $d_{ij} : N'_{1,i} \to N_{1,j}$ (resp. an injective morphism $d_{ij} : N'_{2,i} \to N_{2,j}$) such that (1) (resp. (1')) commutes.*
- *there exists a morphism $l_{21} : L_{c,2} \to L_{c,1}$ (resp. $l_{12} : L_{c,1} \to L_{c,2}$) s.t. (2) (resp. (2')) commutes and in addition a $NAC(n'_{2,i}) : L_{c,2} \to N'_{2,i}$ (resp. $n'_{1,i} : L_{c,1} \to N'_{1,i}$) in $D_{m_{c,2}}(NAC_{L_2})$ (resp. $D_{m_{c,1}}(NAC_{L_1})$) with an injective morphism $m_{ij} : N'_{2,i} \to N_{1,j}$ (resp. an injective morphism $m_{ij} : N'_{1,i} \to N_{2,j}$) s.t. $n_{1,j} \circ l_{21} = m_{ij} \circ n'_{2,i}$ (resp. $n_{2,j} \circ l_{12} = m_{ij} \circ n'_{1,i}$).*

$$\begin{tikzcd}
N_{1,j} & N'_{1,i} \arrow[l, dotted, "d_{ij}"'] & (3) & N'_{2,i} \arrow[lll, dotted, bend right=30, "m_{ij}"'] \\
(1) & L_{c,1} \arrow[ul, "n_{1,j}"] \arrow[u, "n'_{1,i}"'] \arrow[dr, "g_{c,1}"'] & (2) & L_{c,2} \arrow[ll, dotted, "l_{21}"] \arrow[u, "n'_{2,i}"'] \arrow[dl, "g_{c,2}"] \\
& & K &
\end{tikzcd}
\qquad
\begin{tikzcd}
N'_{1,i} \arrow[rrr, dotted, bend left=30, "m_{ij}"] & (3') & N'_{2,i} \arrow[r, dotted, "d_{ij}"] & N_{2,j} \\
L_{c,1} \arrow[u, "n'_{1,i}"] \arrow[rr, dotted, "l_{12}"] \arrow[dr, "g_{c,1}"'] & (2') & L_{c,2} \arrow[u, "n'_{2,i}"] \arrow[ur, "n_{2,j}"'] \arrow[dl, "g_{c,2}"] & (1') \\
& K & &
\end{tikzcd}$$

Example 7 – Consider again the critical pair *(gatherOrder,closeAccountTable)* as described in Example 5. The strict solution consists on the one hand of a transformation that pays the order and then closes the account and on the other hand nothing. Thus for the solution of this critical pair t_2 merely consists of $t_2 : K \overset{closeAccountTable}{\Rightarrow} P_2$ and therefore this case is trivial. Thus it remains to consider the diagram on the left in Theorem 3. Namely, on the other hand $t_1 : K \overset{gatherOrder}{\Rightarrow} K_1 \overset{payOrder}{\Rightarrow} K_2 \overset{closeAccount}{\Rightarrow} P_2$ is not trivial. The concurrent rule $p_{c,1}$ of this transformation closes the account of a table with a concurrent $NAC_{p_{c,1}}$ consisting of a $NAC_{n_{1,1}}$ forbidding any alert and a single NAC $NAC_{n_{1,2}}$ forbidding any order for this table. Now $NAC_{n_{1,1}}$ is induced on the one hand by the downward translation of NAC *noAlert* of rule *gatherOrder*, since they can be connected by an identity i.e. $d_{1,1} = id$. On the other hand $NAC_{n_{1,2}}$ is induced by NAC *noOrders* on rule *closeAccountTable*. This is because the lhs $L_{c,1}$ of the concurrent rule $p_{c,1}$ consists of an open table and it is thus identical to the lhs $L_{c,2}$ of *closeAccountTable*. Therefore the morphism l_{21} is in this case the identity as well as the morphism m_{12} connecting both single NACs.

– Consider also the critical pair *(resetAlert,noteUnpaidOrder)* as described in Example 5. The sufficient condition as described in Theorem 3 is not fulfilled for the strict solution described in Example 5. It was possible though to conclude NAC-confluence according to Def. 10 for this solution anyway as explained in Example 5.

The following corollary follows directly from Theorem 3. It states that NAC-confluence for a critical pair is automatically fulfilled if a strict solution can be found via rules without NACs.

Corollary 1. *A critical pair* $P_1 \overset{p_1}{\Leftarrow} K \overset{p_2}{\Rightarrow} P_2$ *is strictly NAC-confluent if it is strictly confluent via the transformations* $t_1 : K \overset{p_1,g_1}{\Rightarrow} P_1 \Rightarrow^* X$ *(resp.* $t_2 : K \overset{p_2,g_2}{\Rightarrow} P_2 \Rightarrow^* X$*) and both* $P_1 \Rightarrow^* X$ *and* $P_2 \Rightarrow^* X$ *are transformation sequences without NACs.*

Example 8. Consider now in the Conflict Matrix in Fig. 4 the critical pair corresponding to *(alertUnpaidTable,payOrder)*. A strict solution for this critical pair on the one hand notes the unpaid order and resets the alert and on the other hand does nothing. Then we have a table with an open account and without orders. The rules *noteUnpaidOrder* and *resetAlert* are rules without NACs therefore according to the former corollary this critical pair is automatically strictly NAC-confluent and does not have to be investigated further.

5 Conclusion

In this paper, the Embedding Theorem and Local Confluence Theorem formulated in [6] for graph transformations without NACs are extended to graph transformations with NACs. These results hold not only for the instantiation of double-pushout gts with NACs as shown in this paper, but also for more general adhesive HLR systems with NACs as proven in [9]. In our results including NACs extra conditions such as NAC-consistency of the extension morphism (resp. NAC-confluence of the set of critical pairs) are required in order to lead to a correct embedding with NACs (resp. confluent gts with NACs). These additional conditions are explained in our running example.

Future work consists of trimming the results towards efficient tool support and generalizing the theory for transformation systems with NACs described in [9] to transformation systems with more general application conditions as defined in [13].

References

1. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovsky, T., Prange, U., Varro, D., Varro-Gyapay, S.: Model Transformation by Graph Transformation: A Comparative Study. In: Proc. Workshop Model Transformation in Practice, Montego Bay, Jamaica (October 2005)
2. Bottoni, P., Schürr, A., Taentzer, G.: Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In: Proc.IEEE Symposium on Visual Languages, Long version available as technical report SI-2000-06, University of Rom (September 2000)
3. Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems. JACM 27(4), 797–821 (1980)
4. Plump, D.: Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In: Sleep, M., Plasmeijer, M., van Eekelen, M.C. (eds.) Term Graph Rewriting, pp. 201–214. Wiley, Chichester (1993)
5. 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)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
7. Lambers, L., Ehrig, H., Orejas, F., Prange, U.: Parallelism and Concurrency in Adhesive High-Level Replacement Systems with Negative Application Conditions. In: Ehrig, H., Pfalzgraf, J., Prange, U. (eds.) CC 2007. Elsevier, Amsterdam (to appear, 2008)

8. Lambers, L., Ehrig, H., Orejas, F.: Conflict Detection for Graph Transformation with Negative Application Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 61–76. Springer, Heidelberg (2006)
9. Lambers, L.: Adhesive high-level replacement systems with negative application conditions. Technical report, Technische Universität Berlin (2007), http://iv.tu-berlin.de/TechnBerichte/2007/2007-14.pdf
10. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions 26, 287–313 (1996)
11. 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, Foundations, vol. 1, pp. 163–245. World Scientific, Singapore (1997)
12. 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. 446–456. Springer, Heidelberg (2004)
13. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Constraints and application conditions: From graphs to high-level structures. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 287–303. Springer, Heidelberg (2004)

Formal Analysis of Model Transformations Based on Triple Graph Rules with Kernels

Hartmut Ehrig and Ulrike Prange

Technische Universität Berlin, Germany
{ehrig,uprange}@cs.tu-berlin.de

Abstract. Triple graph transformation has become an important approach for model transformations. Triple graphs consist of a source, a target and a connection graph. The corresponding rules also contain these parts and describe the simultaneous construction of both the source and the target model. From these rules, forward rules can be derived which describe the model transformation from a given source model to a target model. The forward transformation must be source consistent in order to define a valid model transformation. Source consistency implies that the source and the target model correspond to each other according to a triple transformation.

In this paper, the relationship between the source consistency of forward transformations, and NAC consistency and termination used in other model transformation approaches is analysed from a formal point of view. We define the kernel of a forward rule and construct NACs based on this kernel. Then we give sufficient conditions such that source consistency implies NAC consistency and termination. Moreover, we analyse how to achieve local confluence independent of source consistency. Both results together provide sufficient conditions for functional behaviour of model transformations. Our results are illustrated by an example describing a model transformation from activity diagrams to CSP.

1 Introduction

Model transformations are most important for model-driven software development. In recent years, triple graph grammars (TGGs) introduced by A. Schürr [1] have been shown to be a suitable basis to define model transformations in various application areas [2,3]. TGGs are based on triple graphs and triple rules which allow to consistently co-develop two related structures modeled by graphs. TGG rules are triples of non-deleting graph rules and generate the language of triple graphs, which can be projected to the first and third component, usually called source and target language, respectively.

In [1], it was shown that a triple rule tr can be decomposed into a source rule tr_S and a forward rule tr_F, and similarly for the transformations, where the forward rules can be used to define model transformations from source to target models. Dually, triple rules and transformations can be decomposed into target and backward rules and transformations leading to model transformations from

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 178–193, 2008.

target to source models, and hence to bidirectional model transformations between different domain-specific modeling languages [4, 5, 6]. The decomposition result in [1] has been extended by a corresponding composition result in [5] leading to a bijective correspondence between triple graph transformation sequences and consistent sequences of corresponding source and forward transformation sequences.

A forward transformation $G_S \xRightarrow{tr_F^*} G$ is called a forward model transformation from the source projection of G_S to the target projection of G if it is "source consistent". As defined in [5], "source consistency" means that there is a generating sequence $\varnothing \xRightarrow{tr_S^*} G_S$ for the source model via the corresponding source rules such that the matches of the source components in $G_S \xRightarrow{tr_F^*} G$ are defined by the items generated by $\varnothing \xRightarrow{tr_S^*} G_S$. Actually, source consistency can be considered as a control condition for forward transformation sequences in order to obtain a model transformation from the source to the target model.

In most practical approaches using TGGs for model transformations there is no formal control condition for how to apply the forward rules. But the intuitive idea is to apply each forward rule to a corresponding item of the source graph [7]. In other graph transformation approaches to model transformations (see [5]), the control condition is given by different layers of rules with negative application conditions (NACs), where the rules in each layer are applied as long as possible. Termination is checked by suitable termination criteria [8].

In this paper, the relationship between source consistency of forward transformation sequences on the one hand and NAC consistency and termination on the other hand is analysed from a formal point of view. For this purpose, we consider triple graph rules $tr : L \to R$ with kernels, where the kernel is a distinguished triple $k(tr) = (x, r, y) \in R$ of connected nodes in the source, connection, and target graphs created by the triple rule tr. This allows to define the corresponding kernels $k(tr_S) = x$ and $k(tr_F) = k(tr)$ for the source and forward rules, respectively. Moreover, for a forward rule $tr_F = L \to R$ we define so-called kernel NACs $NAC(tr_F) = L \cup k(tr_F)$. This means that a NAC consistent forward transformation via tr_F cannot be applied twice at the same match.

Our first main result shows that the source consistency of $G_S \xRightarrow{tr_F^*} G$ implies NAC consistency and termination. In fact, an even slightly weaker notion called "kernel source consistency" is sufficient for this result, where source consistency is restricted to the kernel elements. In our second main result we give sufficient conditions for local confluence of forward rules with kernel NACs. Both results together lead to confluence and termination of forward rules with kernel NACs and hence to functional behaviour of the corresponding model transformation.

This paper is organized as follows: In Section 2, we review the basic definitions for triple graph grammars. An example model transformation from activity diagrams to CSP is introduced in Section 3. In Section 4, kernels of triple rules and some basic properties are defined. In Section 5, the main results are stated and proven, and applied to our example model transformation in Section 6. Finally, the conclusion is given in Section 7.

2 Triple Graph Transformation

In this section, we give a short introduction of the basic notions of triple graphs and triple graph grammars. which have been introduced in [1] as an approach to consistently develop related structures.

Triple graphs consist of a source, a target and a connection graph which is embedded into both source and target graphs. For the underlying graphs, we use for simplicity the category **Graphs** of graphs and graph morphisms, but the main results in [5] have been formulated already in the framework of adhesive HLR systems [8].

Definition 1 (Triple Graph). *A* triple graph $G = (S_G \overset{s_G}{\leftarrow} C_G \overset{t_G}{\rightarrow} T_G)$ *consists of graphs* S_G*,* C_G*, and* T_G *called source graph, connection graph, and target graph, respectively, and graph morphisms* $s_G : C_G \rightarrow S_G$ *and* $t_G : C_G \rightarrow T_G$*.*

For triple graphs G *and* H*, a* triple graph morphism $f = (f_S, f_C, f_T) : G \rightarrow H$ *consists of graph morphisms* $f_S : S_G \rightarrow S_H$*,* $f_C : C_G \rightarrow C_H$ *and* $f_T : T_G \rightarrow T_H$ *such that* $s_H \circ f_C = f_S \circ s_G$ *and* $t_H \circ f_C = f_T \circ t_G$*.*

Triple graphs and triple graph morphisms form the category **TripleGraphs***.*

Remark 1. The category **TripleGraphs** is isomorphic to the comma category $\mathbf{ComCat}(\mathbf{S}, \mathbf{C}, \mathbf{T}, \{\mathbf{s} : \mathbf{C} \rightarrow \mathbf{S}, \mathbf{t} : \mathbf{C} \rightarrow \mathbf{T}\})$ where $\mathbf{S}, \mathbf{C}, \mathbf{T} = \mathbf{Graphs}$ and all functors are the identities on **Graphs**.

For simplicity, for a morphism $f : G \rightarrow H$ and $x \in S_G$ we write $f(x)$ instead of $f_S(x)$, and similarly for $x \in C_G, T_G$.

The concept of typing plays an important role in modeling. Thus we introduce typed triple graphs.

Definition 2 (Typed Triple Graph). *Given a triple graph* TG *as type graph, a* typed triple graph $(G, type^G)$ *consists of a triple graph* G *and a triple graph morphism* $type^G : G \rightarrow TG$*.* G *is said to by typed over* TG *with typing* $type^G$*.*

For typed triple graphs $(G, type^G)$ *and* $(H, type^H)$*, a* typed triple graph morphism f *is a triple graph morphism* $f : G \rightarrow H$ *such that* $type^H \circ f = type^G$*.*

Triple graphs typed over TG *and typed triple graph morphisms form the category* $\mathbf{TripleGraphs_{TG}}$*.*

Remark 2. The category $\mathbf{TripleGraphs_{TG}}$ is isomorphic to the slice category $\mathbf{TripleGraphs}\backslash TG$.

For the access to the different graph parts, projections of a triple graph are defined.

Definition 3 (Projection). *For a triple graph* $G = (S_G \overset{s_G}{\leftarrow} C_G \overset{t_G}{\rightarrow} T_G)$ *the* projections *of* G *to the source, connection, and target graph are defined by* $proj_S(G) = S_G$*,* $proj_C(G) = C_G$*, and* $proj_T(G) = T_G$*, respectively. In case of typed triple graphs, also the typing is projected.*

In the following, we give all the definitions based on triple graphs. They can be formulated analogously for typed triple graphs as used in Section 4.

A triple rule is a rule based on triple graphs as in standard DPO transformation. It constructs simultaneously source and target graphs as well as their connection graph. According to [1, 5], only non-deleting triple rules are allowed. This simplifies the definition since we do not need to consider an interface graph but only the left and right hand side of the rule.

Definition 4 (Triple Rule and Triple Transformation). *A* triple rule $tr = (L \xrightarrow{tr} R)$ *consists of triple graphs L and R, called left hand side and right hand side, respectively, and an injective triple graph morphism $tr : L \to R$.*

Given a triple rule tr, a triple graph G and a triple graph morphism $m : L \to G$, called match, a direct triple transformation $G \overset{tr,m}{\Longrightarrow} H$ *is given by the pushout* (1) *in* **TripleGraphs**, *which is the componentwise pushout* (2) *on the source, connection and target graphs in* **Graphs** *due to the comma category construction. The morphism p in pushout* (1) *is called* comatch.

A sequence of direct triple transformations is then called triple tranformation.

$$\begin{tikzcd}
L \arrow[dr, "tr"] \arrow[dd, "m"'] & \\
 & R \arrow[dd, "p"] \\
G \arrow[dr, "f"'] \arrow[ur, phantom, "(1)"] & \\
 & H
\end{tikzcd}
\qquad
\begin{tikzcd}
S_L \arrow[drr, "tr_S"] \arrow[dd, "m_S"'] & C_L \arrow[l, "s_L"'] \arrow[rr, "t_L"] \arrow[drrr, "tr_C"] \arrow[dd, "m_C"'] & & T_L \arrow[drr, "tr_T"] \arrow[dd, "m_T"] & & \\
 & & S_R \arrow[dd, "p_S"] & & C_R \arrow[ll, "s_R"'] \arrow[r, "t_R"] \arrow[dd, "p_C"] & T_R \arrow[dd, "p_T"] \\
S_G \arrow[drr, "f_S"'] & C_G \arrow[l, "s_G"'] \arrow[rr, "t_G"'] \arrow[drrr, "f_C"] & & T_G \arrow[drr, "f_T"] & & \\
(2) & & S_H & & C_H \arrow[ll, "s_H"] \arrow[r, "t_H"'] & T_H
\end{tikzcd}$$

Since we consider only nondeleting injective rules $tr : L \to R$ we can assume w.l.o.g. that $L \subseteq R$ and all derived triple graphs are included in each other, i.e. for a transformation sequence $G_0 \overset{tr_1,m_1}{\Longrightarrow} G_1 \overset{tr_2,m_2}{\Longrightarrow} \ldots \overset{tr_n,m_n}{\Longrightarrow} G_n$ we have that $G_i \subseteq G_j$ for $i \leq j$.

To extend the expressiveness of triple graph transformations we define negative application conditions which restrict the applicability of a triple rule.

Definition 5 (Negative Application Condition). *Given a triple rule $tr = (L \xrightarrow{tr} R)$, a* negative application condition *(NAC) (N, n) consists of a triple graph N and a triple graph morphism $n : L \to N$.*

A match $m : L \to G$ is NAC consistent *if there is no $q : N \to G$ such that $q \circ n = m$. A triple transformation $G \overset{*}{\Longrightarrow} H$ is* NAC consistent *if all matches are NAC consistent.*

Up to now, the triple rules simultaneously create the source, connection and target graphs. But for a model transformation, some source model is given that has to be transformed into the corresponding target model. For this purpose, we can derive source and forward rules from a given triple rule. The source rule only creates a part of the source model, and the forward rule describes the transformation of this part to the target model.

Definition 6 (Source and Forward Rule). *Given a triple rule* $tr = (L \xrightarrow{tr} R)$*, the* source rule tr_S *and the* forward rule tr_F *are defined by* $tr_S = ((S_L \overset{\varnothing}{\leftarrow} \varnothing \overset{\varnothing}{\rightarrow} \varnothing) \xrightarrow{(tr_S,\varnothing,\varnothing)} (S_R \overset{\varnothing}{\leftarrow} \varnothing \overset{\varnothing}{\rightarrow} \varnothing))$ *and* $tr_F = ((S_R \xleftarrow{tr_S \circ s_L} C_L \xrightarrow{t_L} T_L) \xrightarrow{(id_{S_R},tr_C,tr_T)} R)$*.*

Now a triple graph grammar consists of a set of triple rules and a start graph.

Definition 7 (Triple Graph Grammar). *A* triple graph grammar $GG = (TR, S)$ *consists of a set TR of triple rules and a triple graph S, the start graph.*

For the rule set TR, we get induced sets $TR_S = \{tr_S \mid tr \in TR\}$ and $TR_F = \{tr_F \mid tr \in TR\}$ of source and forward rules.

For the relationship of triple rules with their source and forward rules, match and source consistency are introduced. Match consistency describes that in a transformation sequence, the forward rule is always applied via the comatch of the corresponding source rule for the source graph. Source consistency of a forward transformation requires a match consistent transformation. In [5], it is shown that a triple transformation can be split into match consistent source and forward transformations via the same rule sequence, and vice versa.

Definition 8 (Match Consistency). *Consider a triple transformation tt : $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S \overset{tr_F^*}{\Longrightarrow} G$ where $tr_S^* = (tr_{i,S})_{i=1,\ldots,n}$ and $tr_F^* = (tr_{i,F})_{i=1,\ldots,n}$ are derived from the same triple rules $tr^* = (tr_i)_{i=1,\ldots,n}$, and we have matches $m_{i,S}$ and $m_{i,F}$, and comatches $p_{i,S}$ and $p_{i,F}$, respectively. Then tt is called* match consistent *if the source component of the match $m_{i,F}$ is completely determined by the comatch $p_{i,S}$, i.e. $(m_{i,F})_S = (p_{i,S})_S$, for $i = 1, \ldots, n$.*

Definition 9 (Source Consistency). *A forward triple transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ with $tr_F^* = (tr_{i,F})_{i=1,\ldots,n}$ is called* source consistent *if there exists a source triple transformation $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S$ such that $tr_S^* = (tr_{i,S})_{i=1,\ldots,n}$ and $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S \overset{tr_F^*}{\Longrightarrow} G$ is match consistent.*

According to [5], a source consistent transformation leads to a forward model transformation.

Definition 10 (Forward Model Transformation). *A forward triple transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ with $G'_S = proj_S(G_S)$ and $G'_T = proj_T(G)$ is called a* forward model transformation *from G'_S to G'_T if $G_S \overset{tr_F^*}{\Longrightarrow} G$ is source consistent.*

3 Example: From Activity Diagrams to CSP

In this section, we demonstrate the definitions from Section 2 on a model transformation from simple activity diagrams [9] with only actions, binary decisions and merges to communicating sequential processes (CSP) [10]. This transformation is a slightly smaller version of the case study proposed in [11]. Due to the

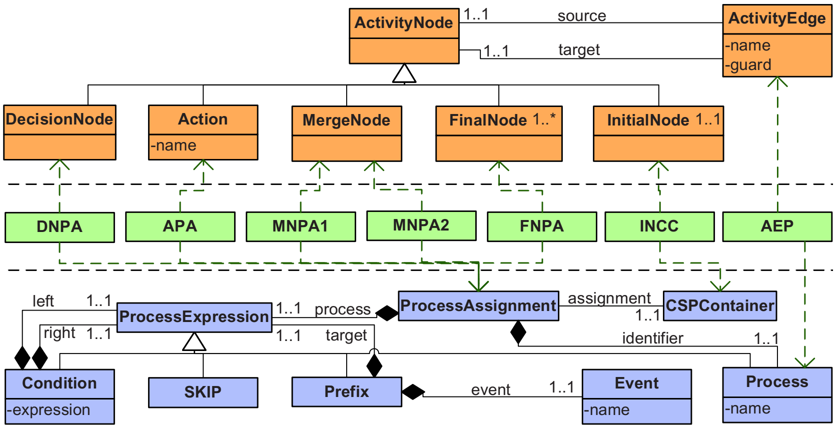


Fig. 1. The triple type graph

restrictions of the activity diagrams we can also restrict CSP to a SKIP process, prefix operations and conditions.

The triple type graph for the model transformation is shown in Fig. 1. In the upper part, the type graph for activity diagrams is shown. Basically, we have different kinds of activity nodes and activity edges, that are connected by source and target associations to the nodes. In the bottom of Fig. 1, the simplified type graph of CSP is depicted. Processes are defined via process assignments. A process expression can be a simple process, a prefix combining an event and a process, a condition, or a successful termination denoted by SKIP.

In Fig. 2, the triple rules for the consistent development of activity diagrams and the corresponding CSP models are depicted. We use a compact representation, where the stereotype ≪new≫ means that this element is created by the rule, and all other elements are already present in the left hand side. In the figure, on the left hand side of each rule the source graph is shown, followed by the connection graph and the target graph on the right hand side. The morphisms between these graphs are depicted by dashed arrows.

The triple rule $\mathsf{tr}^{\mathsf{InitialNode}}$ describes that an initial node corresponds to an CSP container where all other CSP elements are stored. With $\mathsf{tr}^{\mathsf{ActivityEdge}}$, an activity edge and its corresponding process are created. The other activity nodes correspond to different process assignments. With the triple rule $\mathsf{tr}^{\mathsf{Action}}$, an action and the corresponding prefix operation are created, while with $\mathsf{tr}^{\mathsf{FinalNode}}$ a final node and the corresponding SKIP process are defined. Finally, the rule $\mathsf{tr}^{\mathsf{DecisionNode}}$ handles the simultaneous creation of a binary decision and a condition, and $\mathsf{tr}^{\mathsf{MergeNode}}$ creates a binary merge and the corresponding identification of processes. Note that the rules $\mathsf{tr}^{\mathsf{ActivityEdge}}$ and $\mathsf{tr}^{\mathsf{Action}}$ have input parameters to define the attributes. To obtain a valid activity diagram, the rule $\mathsf{tr}^{\mathsf{InitialNode}}$ has to be applied exactly once, the rule $\mathsf{tr}^{\mathsf{FinalNode}}$ at least once, and each produced activity edge has to be connected by exactly one source and target association.

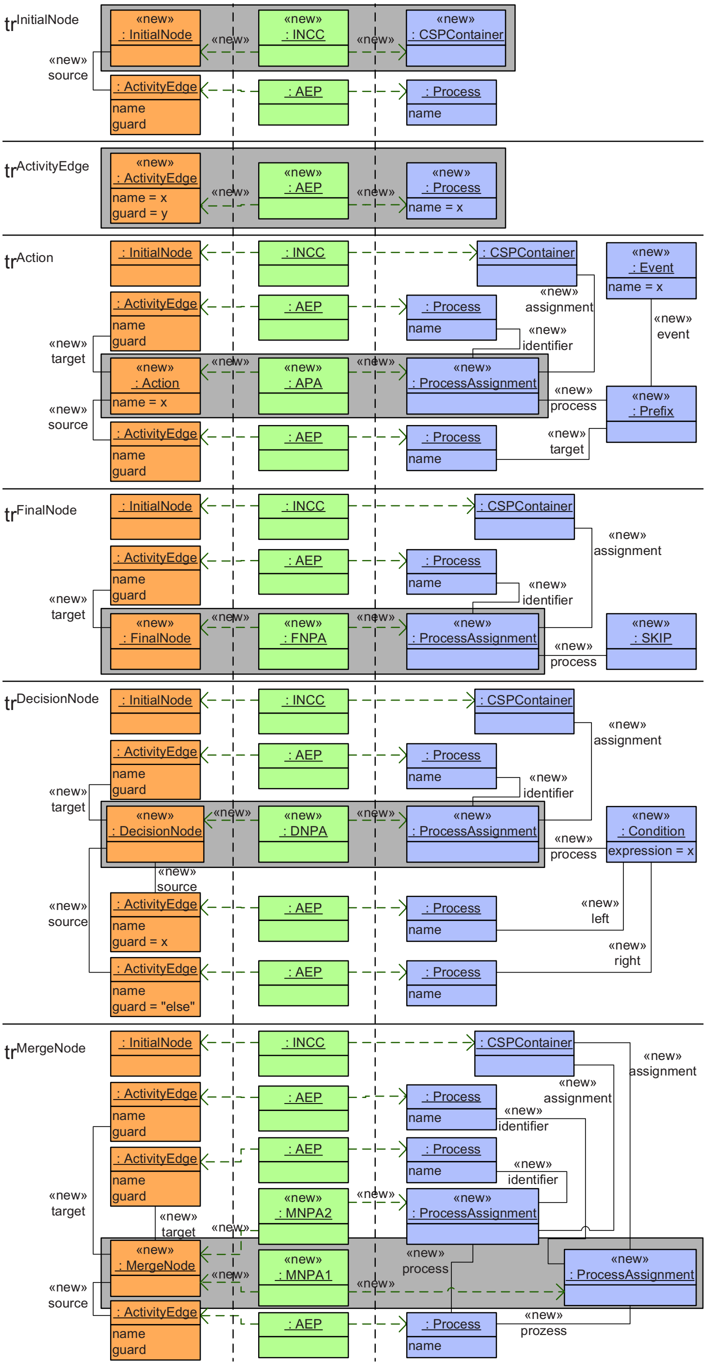
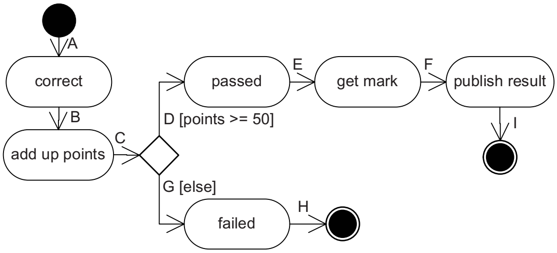


Fig. 2. The triple rules



$A :=$ correct $\rightarrow B$
$B :=$ add up points $\rightarrow C$
$C := D \not\prec$ points >= 50 $\not\succ G$
$D :=$ passed $\rightarrow E$
$E :=$ get mark $\rightarrow F$
$F :=$ publish result $\rightarrow I$
$G :=$ failed $\rightarrow H$
$H :=$ SKIP
$I :=$ SKIP

Fig. 3. Example model

In Fig. 3, an activity diagram and the corresponding process are depicted in concrete syntax, which are the results of the source and target projections of the transformation sequence $\emptyset \overset{tr^*}{\Longrightarrow} G$ with $tr^* = (\mathsf{tr}^{\mathsf{InitialNode}}, 9 \times \mathsf{tr}^{\mathsf{ActivityEdge}}, 6 \times \mathsf{tr}^{\mathsf{Action}}, \mathsf{tr}^{\mathsf{DecisionNode}}, 2 \times \mathsf{tr}^{\mathsf{FinalNode}})$ with suitable parameter values.

From the triple rules in Fig. 2, we can derive source and forward rules. For the source rules, we simply have to delete the connection and target graph parts of the rules. For the forward rules, the source graph of the left hand side of the forward rule is the source graph of the right hand side of the original rule, thus we only have to delete the ≪new≫-stereotypes of all elements in the source graph of a rule to obtain the corresponding forward rule. In Fig. 4, this is shown exemplarily for the rule $\mathsf{tr}^{\mathsf{FinalNode}}$ leading to the forward rule $\mathsf{tr}_{\mathsf{F}}^{\mathsf{FinalNode}}$.

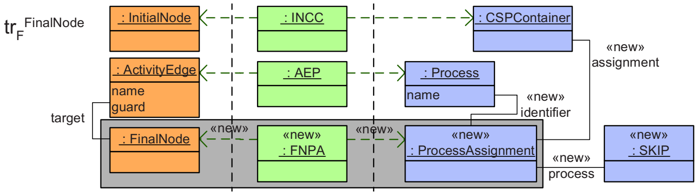


Fig. 4. A forward rule

Now the transformation $\emptyset \overset{tr^*}{\Longrightarrow} G$ from above can be decomposed into the transformations $\emptyset \overset{tr_S^*}{\Longrightarrow} G_S$ via the corresponding source rules and $G_S \overset{tr_F^*}{\Longrightarrow} G$ via the corresponding forward rules. In this case, $A = proj_S(G_S)$ is the activity diagram depicted in Fig. 3. The forward transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ is source consistent and leads to the forward model transformation from A to P, where $P = proj_T(G)$ is the CSP model in Fig. 3.

4 The Kernel Approach

In the kernel approach, we consider typed triple graphs and an empty start graph. For each rule, a distinguished kernel triple is selected. In this paper, we only consider the source and forward rules, but the theory can be done similarly for the target and backward rules.

The kernel of each rule is a triple of nodes, one from each graph part of a triple graph, that is connected and generated by the rule.

Definition 11 (Kernel). *For a triple graph G, a node triple $(x, r, y) \in S_G \times C_G \times T_G$ is called* connected *if $s_G(r) = x$ and $t_G(r) = y$.*

Given a triple graph grammar $GG = (TR, \varnothing)$, we define for each rule $tr = (L \xrightarrow{tr} R) \in TR$ the kernel *$k(tr) = (x, r, y) \in R \backslash L = (S_R \backslash S_L) \times (C_R \backslash C_L) \times (T_R \backslash T_L)$ which is a connected node triple. Then $k(tr_S) = x$ and $k(tr_F) = (x, r, y)$ are the corresponding kernels of tr_S and tr_F, respectively.*

For the source and forward rules we want to have distinguished kernel typing, which means that elements of kernel types cannot be created as non-kernel types by any other rule. In our example, we have distinguished kernel typing (see Section 6).

Definition 12 (Distinguished Kernel Typing). *Define the* source kernel types *$KTYPE_S = \{type(x) \mid tr_S \in TR_S, k(tr_S) = x\}$ and the* forward kernel types *$KTYPE_F = \{type(r) \mid tr_F \in TR_F, k(tr_F) = (x, r, y)\}$.*

TR_S has distinguished kernel typing *if for all $tr_S \in TR_S$ and x created by tr_S, i.e. $x \in S_R \backslash S_L$, we have that $x \neq k(tr_S)$ implies $type(x) \notin KTYPE_S$.*

TR_F has distinguished kernel typing *if for all $tr_F \in TR_F$ and connected triples (x, r, y) created by tr_F we have that $(x, r, y) \neq k(tr_F)$ implies $type(r) \notin KTYPE_F$, where $(x, r, y) \in R$ created by $tr_F : L \to R$ means that $(x, r, y) \notin L$.*

Moreover, TR_F is called type functional *if for all tr_F, $tr'_F \in TR_F$ with kernels $k(tr_F) = (x, r, y)$ and $k(tr'_F) = (x', r', y')$ we have that $type(x) = type(x')$ implies $type(r) = type(r')$.*

Remark 3. If s_{TG} of the type graph TG is injective then TR_F is type functional in any case. Moreover, $type(r) = type(r')$ implies $type(y) = type(y')$.

In a triple transformation, the images of the kernels under the comatches are called the kernel elements of the resulting graph.

Definition 13 (Kernel Elements). *Consider a triple transformation $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S$ with $tr_S^* = (tr_{i,S})_{i=1,\ldots,n}$ and comatches $p_{i,S}$. The* kernel elements *of G_S generated by tr_S^* are all elements $\overline{x_i} = p_{i,S}(x_i)$ for kernels $k(tr_{i,S}) = x_i$ and $i = 1, \ldots, n$.*

Consider a triple transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ with $tr_F^* = (tr_{i,F})_{i=1,\ldots,n}$ and comatches $p_{i,F}$. The kernel elements *of G generated by tr_F^* are all triples $(\overline{x_i}, \overline{r_i}, \overline{y_i}) = p_{i,F}(x_i, r_i, y_i)$ for kernels $k(tr_{i,F}) = (x_i, r_i, y_i)$ and $i = 1, \ldots, n$.*

In the following Facts 1 and 2, we show that for triple transformations with distinguished kernel typing, kernel elements are exactly the elements of kernel types.

Fact 1. *Consider a triple graph grammar $(TR, \varnothing)$ where TR_S has distinguished kernel typing and a transformation $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S$. Then we have that $\overline{x} \in G_S$ is a kernel element if and only if $type(\overline{x}) \in KTYPE_S$.*

Proof. For $\overline{x} \in G_S$ with $type(\overline{x}) \in KTYPE_S$ there is a rule $tr_{i,S} = (L \overset{tr_{i,S}}{\to} R)$ such that $\overline{x}$ has been created by $tr_{i,S}$, and there is some $x \in S_R \backslash S_L$ with $p_{i,S}(x) = \overline{x}$. Suppose $\overline{x}$ is no kernel element, i.e. $x \neq k(tr_{i,S})$. Since TR_S has distinguished kernel typing it follows that $type(\overline{x}) = type(x) \notin KTYPE_S$, which is a contradiction. Thus, $\overline{x}$ is a kernel element of G_S. Vice versa, if $\overline{x}$ is a kernel element generated by $k(tr_{i,S}) = x$ then $type(\overline{x}) = type(x) \in KTYPE_S$.

Fact 2. *Consider a triple graph grammar $(TR, \varnothing)$ where TR_F has distinguished kernel typing and a triple transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ with $proj_C(G_S) = \varnothing$. Then we have that a connected triple $(\overline{x}, \overline{r}, \overline{y}) \in G$ is a kernel element if and only if $type(\overline{r}) \in KTYPE_F$.*

Proof. For $(\overline{x}, \overline{r}, \overline{y}) \in G$ with $type(\overline{r}) \in KTYPE_F$ there is a rule $tr_{i,F} = (L \overset{tr_{i,F}}{\to} R)$ such that $\overline{r}$ has been created by $tr_{i,F}$, because $proj_C(G_S) = \varnothing$. Then there is a triple $(x, r, y) \in R$ with $p_{i,F}(x, r, y) = (\overline{x}, \overline{r}, \overline{y})$. It follows that $(x, r, y) \notin L$, otherwise $\overline{r}$ was created earlier. Suppose $(x, r, y) \neq k(tr_{i,S})$. Since TR_F has distinguished kernel typing it follows that $type(\overline{r}) = type(r) \notin KTYPE_F$, which is a contradiction. Thus we have that $k(tr_{i,F}) = (x, r, y)$ and $(\overline{x}, \overline{r}, \overline{y})$ is a kernel element of G. Vice versa, if $(\overline{x}, \overline{r}, \overline{y})$ is a kernel element generated by $k(tr_{i,S}) = (x, r, y)$ then $type(\overline{r}) = type(r) \in KTYPE_F$.

Kernel match and source consistency is the restriction of source and match consistency to the kernel elements. Kernel consistency is easier to verify since only one element for each direct transformation has to be considered.

Definition 14 (Kernel Match Consistency). *Consider a triple transformation $tt : \varnothing \overset{tr_S^*}{\Longrightarrow} G_S \overset{tr_F^*}{\Longrightarrow} G$ where $tr_S^* = (tr_{i,S})_{i=1,\dots,n}$ and $tr_F^* = (tr_{i,F})_{i=1,\dots,n}$ are derived from the same triple rules $tr^* = (tr_i)_{i=1,\dots,n}$ with kernels $k(tr_i) = (x_i, r_i, y_i)$, and we have matches $m_{i,S}$ and $m_{i,F}$, and comatches $p_{i,S}$ and $p_{i,F}$, respectively. The triple transformation tt is called* kernel match consistent *if $m_{i,F}(x_i) = p_{i,S}(x_i)$ for $i = 1, \dots, n$.*

Definition 15 (Kernel Source Consistency). *A forward triple transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ with $tr_F^* = (tr_{i,F})_{i=1,\dots,n}$ is called* kernel source consistent *if there exists a source triple transformation $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S$ such that $tr_S^* = (tr_{i,S})_{i=1,\dots,n}$ and $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S \overset{tr_F^*}{\Longrightarrow} G$ is kernel match consistent.*

For each kernel element, from a triple transformation a unique rule can be identified which has created this element.

Fact 3. *Given a triple transformation $tt : \varnothing \overset{tr_S^*}{\Longrightarrow} G_S \overset{tr_F^*}{\Longrightarrow} G$ with $tr_S^* = (tr_{i,S})_{i=1,\dots,n}$ and $tr_F^* = (tr_{i,F})_{i=1,\dots,n}$ derived from the same triple rules $tr^* = (tr_i)_{i=1,\dots,n}$, then we have that*

1. *For each kernel element $\overline{x}$ in G_S there is a unique $i \in \{1, \ldots, n\}$ such that $\overline{x}$ is generated by $tr_{i,S}$, i.e $\overline{x} = \overline{x_i}$ and $\overline{x_i} \neq \overline{x_j}$ for all $j \neq i$.*
2. *For each kernel element $(\overline{x}, \overline{r}, \overline{y})$ in G there is a unique $i \in \{1, \ldots, n\}$ such that $(\overline{x}, \overline{r}, \overline{y})$ is generated by $tr_{i,F}$, i.e. $(\overline{x}, \overline{r}, \overline{y}) = (\overline{x_i}, \overline{r_i}, \overline{y_i})$ and $\overline{r_i} \neq \overline{r_j}$, $\overline{y_i} \neq \overline{y_j}$ for all $j \neq i$.*
3. *If tt is kernel match consistent then in Item 2 also $\overline{x_i} \neq \overline{x_j}$ for all $j \neq i$.*

Proof. 1. For each $i = 1, \ldots, n$, when applying $tr_{i,S}$ a new kernel element $\overline{x_i} = p_{i,S}(x_i)$ is created in G_S such that $\overline{x_1}, \ldots, \overline{x_n}$ are pairwise disjoint.
2. For each $i = 1, \ldots, n$, when applying $tr_{i,F}$ a kernel element $(\overline{x_i}, \overline{r_i}, \overline{y_i}) = p_{i,F}(x_i, r_i, y_i)$ is created in G. Since r_i and y_i are newly created by $tr_{i,F}$ it follows that $\overline{r_1}, \ldots, \overline{r_n}$ and $\overline{y_1}, \ldots, \overline{y_n}$ are pairwise disjoint.
3. In the case of kernel match consistency, $\overline{x_i}$ of a kernel triple $(\overline{x_i}, \overline{r_i}, \overline{y_i})$ is the kernel element $\overline{x_i}$ generated by the kernel of $tr_{i,S}$ such that $\overline{x_1}, \ldots, \overline{x_n}$ are pairwise disjoint due to Item 1.

To allow the application of a forward rule to a source kernel element only once, kernel NACs for forward rules are defined.

Definition 16 (Kernel NAC). *For a forward rule $tr_F = (L \xrightarrow{tr_F} R)$ with kernel $k(tr_F) = (x, r, y)$ we define the* kernel NAC $NAC(tr_F) = (N, n)$ *with triple graph $N = L \cup k(tr_F)$, $s_N(r) = x$, $t_N(r) = y$ and inclusion $n : L \to N$.*

5 Results for Model Transformations in the Kernel Approach

In this section, we analyse how to achieve kernel source consistency and state the main results for forward transformations in the kernel approach. In Section 6, we apply these results to our example from Section 3.

A forward triple transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ with $tr_F^* = (tr_{i,F})_{i=1,\ldots,n}$ is kernel source consistent if G_S is generated by $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S$ with corresponding source rule $tr_S = (tr_{i,S})_{i=1,\ldots,n}$ leading to kernel elements $\overline{x_i}$ in G_S that determine the kernel match for the forward rule $tr_{i,F}$. In other words, each forward rule $tr_{i,F}$ is applied exactly once at the kernel element $\overline{x_i}$ generated by the source rule $tr_{i,S}$.

For a forward transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ to become kernel source consistent we have to construct first a source generating sequence $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S$ leading to kernel elements $\overline{x_1}, \ldots, \overline{x_n} \in G_S$. This is a parsing problem for G_S, which may lead to nondeterministic results. Then we have to apply the corresponding forward rules $tr_{i,F}$ without kernel NACs at the kernel elements $\overline{x_i}$. If this is successful we obtain a kernel source consistent forward transformation $tt : G_S \overset{tr_F^*}{\Longrightarrow} G$ and under the conditions of Thm. 1 tt is NAC consistent and terminating.

Obviously, source consistency of $G_S \overset{tr_F^*}{\Longrightarrow} G$ implies kernel source consistency. Vice versa, kernel source consistency implies source consistency if all matches are uniquely determined by the kernel matches (see Thm. 2).

Theorem 1 (NAC Consistency and Termination). *Consider a triple graph grammar* $(TR, \varnothing)$ *where* TR_S, TR_F *have distinguished kernel typing, a kernel source consistent forward triple transformation* $tt : G_S \overset{tr_F^*}{\Longrightarrow} G$, *and forward rules with kernel NACs. Then we have that:*

1. *tt is NAC consistent.*
2. *tt is terminating if* TR_F *is type functional.*

Proof. Let $tt : G_S = G_{S(0)} \overset{tr_{1,F}}{\Longrightarrow} G_{S(1)} \overset{tr_{2,F}}{\Longrightarrow} \ldots \overset{tr_{n,F}}{\Longrightarrow} G_{S(n)} = G$. Since tt is kernel source consistent there exists a triple transformation $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S$ such that $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S \overset{tr_F^*}{\Longrightarrow} G$ is kernel match consistent.

1. Suppose that tt is not NAC consistent. This means that there is a rule $tr_{i,F} = (L \overset{tr_{i,F}}{\to} R)$ with match $m_{i,F}$, comatch $p_{i,F}$, and kernel NAC (N_i, n_i) such that $G_{S(i-1)} \overset{tr_{i,F}}{\Longrightarrow} G_{S(i)}$ is not NAC consistent, i.e. there is a triple graph morphism $q : N_i \to G_{S(i-1)}$ such that $q \circ n_i = m_{i,F}$.
   For the kernel $k(tr_{i,F}) = (x_i, r_i, y_i)$ of $tr_{i,F}$ we have the kernel element $p_{i,F}(x_i, r_i, y_i) = (\overline{x_i}, \overline{r_i}, \overline{y_i}) \in G_{S(i)}$ and $q(x_i, r_i, y_i) = (\overline{x}, \overline{r}, \overline{y}) \in G_{S(i-1)}$ with $x_i \in S_L$ and $m_{i,F}(x_i) = \overline{x}$. The commutativity of (1) with horizontal inclusions and $x_i \in S_L$ imply that $\overline{x} = m_{i,F}(x_i) = p_{i,F}(x_i) = \overline{x_i}$.
   (x_i, r_i, y_i) is a connected triple and hence also $(\overline{x}, \overline{r}, \overline{y})$ is connected in $G_{S(i-1)}$. Since $type(\overline{r}) = type(r_i) \in KTYPE_F$, Fact 2 implies that $(\overline{x}, \overline{r}, \overline{y})$ is a kernel element of G, and by Fact 3 Item 2 there is a unique j such that $(\overline{x}, \overline{r}, \overline{y}) = (\overline{x_j}, \overline{r_j}, \overline{y_j})$ is generated by $tr_{j,F}$. Obviously, $j < i$ because $(\overline{x}, \overline{r}, \overline{y}) \in G_{S(i-1)}$. Now Fact 3 Item 3 implies that $\overline{x_i} \neq \overline{x_j} = \overline{x}$, which contradicts $\overline{x_i} = \overline{x}$. Hence tt is NAC consistent.

$$\begin{tikzcd}
N_i \arrow[dr, "q"'] & L \arrow[l, "n_i"'] \arrow[r, "tr_{i,F}"] \arrow[d, "m_{i,F}"'] \arrow[dr, phantom, "(1)"] & R \arrow[d, "p_{i,F}"] & N' \arrow[drr, "q'"'] & L' \arrow[l, "n'"'] \arrow[r, "tr'_F"] \arrow[d, "m'"'] \arrow[dr, phantom, "(2)"] & R' \arrow[d] \\
 & G_{S(i-1)} \arrow[r] & G_{S(i)} & & G \arrow[r] & G'
\end{tikzcd}$$

2. Suppose now that tt is not terminating, i.e. there is a direct triple transformation $G \overset{tr'_F, m'}{\Longrightarrow} G'$ for some triple rule $tr'_F = (L' \overset{tr'_F}{\to} R') \in TR_F$ with kernel $k(tr'_F) = (x', r', y')$ and kernel element $(\overline{x}', \overline{r}', \overline{y}') \in G'$ with $\overline{x}' \in G_S$ and $type(\overline{x}') = type(x') \in KTYPE_S$.
   By Fact 1, $\overline{x}'$ is a kernel element of G_S and by Fact 3 Item 1 there is a unique i such that $\overline{x}' = \overline{x_i}$ is generated by $tr_{i,S}$ with kernel $k(tr_{i,S}) = x_i$. Kernel match consistency of $\varnothing \overset{tr_S^*}{\Longrightarrow} G_S \overset{tr_F^*}{\Longrightarrow} G$ implies that the kernel $k(tr_{i,F}) = (x_i, r_i, y_i)$ implies a kernel element $(\overline{x_i}, \overline{r_i}, \overline{y_i}) \in G$ with $\overline{x_i} = \overline{x}'$. It follows that $type(x') = type(\overline{x}') = type(\overline{x_i}) = type(x_i)$, and by type functionality of TR_F also $type(r') = type(r_i) = type(\overline{r_i})$ and $type(y') = type(y_i) = type(\overline{y_i})$. For the kernel NAC $NAC(tr'_F) = (N', n')$, we define a morphism $q' \colon N' \to G$ by $q'|_{L'} = m'$ with $m'(x') = \overline{x}' = \overline{x_i}$ and $q'(x', r', y') =$

$(\overline{x_i}, \overline{r_i}, \overline{y_i})$. q' is a valid morphism because it preserves the typing, and we have that $q' \circ n' = m'$. Thus, $G \overset{tr'_F}{\Longrightarrow} G'$ is not NAC consistent, hence tt is terminating.

Remark 4. Since source consistency implies kernel source consistency this theorem also holds for source consistent forward triple transformations $tt : G_S \overset{tr^*_F}{\Longrightarrow} G$.

Thm. 1 and the following Thm. 2 concerning local confluence are applied to our example from Section 3 in Section 6.

Theorem 2 (Local Confluence). *Consider a triple graph grammar $(TR, \varnothing)$ where TR_F has distinguished kernel typing and kernel NACs. If we have that*

(1) The rules in TR_F are uniquely determined by the left hand sides, i.e. for rules $tr_F = (L \overset{tr_F}{\to} R)$ and $tr'_F = (L' \overset{tr'_F}{\to} R')$, $L \overset{\sim}{=} L'$ implies $tr_F = tr'_F$.
(2) The matches are uniquely determined by the kernel matches, i.e. given kernels $k(tr_F) = (x, r, y)$, $k(tr'_F) = (x', r', y')$ and matches $m : L \to G_0$, $m' : L' \to G_0$, $m(x) = m'(x')$ implies $m = m'$ with $L = L'$.

then the forward rules TR_F with kernel NACs are locally confluent. This means that given $G_0 \overset{tr_F,m}{\Longrightarrow} G_1$, $G_0 \overset{tr'_F,m'}{\Longrightarrow} G_2$ then we have either $G_1 \overset{\sim}{=} G_2$ or the direct transformations are parallel independent with NACs leading to the local Church-Rosser property, i.e. there are transformations $G_1 \overset{tr'_F, i\circ m'}{\Longrightarrow} G_3$, $G_2 \overset{tr_F, i'\circ m}{\Longrightarrow} G_3$.

$$\begin{tikzcd}
G_0 \arrow[r, Rightarrow, "tr_F{,}m"] \arrow[d, Rightarrow, "tr'_F{,}m'"'] & G_1 \arrow[d, Rightarrow, "tr'_F{,}i\circ m'"] \\
G_2 \arrow[r, Rightarrow, "tr_F{,}i'\circ m"'] & G_3
\end{tikzcd}$$

Proof. For given forward rules $tr_F, tr'_F \in TR_F$ and matches $m : L \to G_0$, $m' : L' \to G_0$ we have the following cases:

1. $m = m'$, which implies $L = L'$, and property (1) implies that also $tr_F = tr'_F$. Then the uniqueness of pushouts implies that $G_1 \overset{\sim}{=} G_2$.
2. $m \neq m'$. For the forward rules, we have kernels $k(tr_F) = (x, r, y)$ and $k(tr'_F) = (x', r', y')$ and $m(x) = \overline{x}$, $m'(x') = \overline{x}'$.

   Consider now the kernel NAC $NAC(tr'_F) = (N', n')$ with $N' = L' \cup k(tr'_F)$. Given the transformation $G_0 \overset{tr_F,m}{\Longrightarrow} G_1$ with pushout (1) we have to show that $i \circ m'$ is NAC consistent, i.e. there does not exist a morphism $q : N' \to G_1$ with $q \circ n' = i \circ m'$. Suppose that such a q exists, then using $k(tr'_F) = (x', r', y')$ there is a connected triple $q(x', r', y') = (\overline{x}', \overline{r}', \overline{y}')$ in G_1 with $type(x') = type(\overline{x}')$, $type(r') = type(\overline{r}')$ and $type(y') = type(\overline{y}')$. Since m' satisfies N', $(\overline{x}', \overline{r}', \overline{y}') \notin G_0$, but created by tr_F. Hence there is a connected triple $(x_2, r_2, y_2) \in R$, $(x_2, r_2, y_2) \notin L$ with $p(x_2, r_2, y_2) = (\overline{x}', \overline{r}', \overline{y}')$. Since $type(r_2) = type(\overline{r}') = type(r') \in KTYPE_F$ it follows that $k(tr_F) = (x, r, y) = (x_2, r_2, y_2)$ from distinguished kernel typing of TR_F. But this implies that $m(x) = m(x_2) = p(x_2) = \overline{x}' = m'(x')$ by commutativity of (1), and from property *(2)* it follows that $m = m'$, which is a contradiction.

Hence $i \circ m'$ is NAC consistent. Similarly, $i' \circ m$ is NAC consistent, and with pushout (3) the triple graph G_3 is the result of both transformations $G_1 \overset{tr'_F, i \circ m'}{\Longrightarrow} G_3$ and $G_2 \overset{tr_F, i' \circ m}{\Longrightarrow} G_3$. Thus we have parallel independence with NACs and the local Church-Rosser property leading to local confluence. □

$$\begin{tikzcd}
N' \arrow[rrd, dashed, "q"] & L \arrow[r, "tr_F"] \arrow[d, "m"] \arrow[rd, phantom, "(1)"] & R \arrow[d, "p"] \\
L' \arrow[u, "n'"] \arrow[r, "m'"] \arrow[d, "tr'_F"'] \arrow[rd, phantom, "(2)"] & G_0 \arrow[r, "i"'] \arrow[d, "i'"] \arrow[rd, phantom, "(3)"] & G_1 \arrow[d] \\
R' \arrow[r, "p'"'] & G_2 \arrow[r] & G_3
\end{tikzcd}$$

Since local confluence and termination imply confluence, we get the following sufficient conditions for functional behaviour of forward model transformations.

Theorem 3 (Functional Behaviour). *Under the assumptions of Thms. 1 and 2, forward model transformations have functional behaviour, i.e. they are terminating and confluent.*

6 Analysis of the Example Model Transformation

Now we want to analyse the model transformation described in Section 3. The kernels for the triple rules in Fig. 2 are the triples in a box shaded in gray. Thus we have that $KTYPE_S = \{\mathsf{InitialNode}, \mathsf{ActivityEdge}, \mathsf{Action}, \mathsf{FinalNode}, \mathsf{DecisionNode}, \mathsf{MergeNode}\}$ and $KTYPE_F = \{\mathsf{INCC}, \mathsf{AEP}, \mathsf{APA}, \mathsf{FNPA}, \mathsf{DNPA}, \mathsf{MNPA1}\}$, and it is easy to see that both TR_S and TR_F have distinguished kernel typing and TR_F is type functional. Moreover, the forward rules are uniquely determined by the left hand sides.

Now consider the forward triple transformation $G_S \overset{tr_F^*}{\Longrightarrow} G$ leading to the forward model transformation from A to P from Section 3, with A and P depicted in Fig. 3. Since this forward triple transformation is source consistent, it is also kernel source consistent. From Thm. 1 it follows that it is then NAC consistent and terminating if we consider forward rules with kernel NACs.

In a valid activity diagram without merge nodes, the matches for the forward rules are uniquely determined by the kernel matches. To see this we have to take a closer look at the triple rules. First we know that there is only one initial node. This means, whenever an initial node is present in the left hand side its match is uniquely determined. Moreover, the kernel element and its match induce the complete match because of the graph structure, and in case of the triple rule $\mathsf{tr}_\mathsf{F}^{\mathsf{DecisionNode}}$ also the value of the attributes. This means that for the triple rules of our forward model transformation from A to P the conditions of Thm. 2 are fulfilled and this model transformation is confluent. Thus, the target model P is unique for the source model A.

On the other hand, for the triple rule $\mathsf{tr}_\mathsf{F}^{\mathsf{MergeNode}}$ the matches are not uniquely determined by the kernel matches. This is easy to see, since for a valid match we can swap the matches of the both activity edges which have the merge node as a target. Thus we cannot apply Thm. 2. When applying the forward rule via both matches, we get two different triple graphs which only differ in the mappings of the nodes MNPA1 and MNPA2. Note, that these two direct triple transformations are not confluent, since no rule can be applied to this merge node due to the kernel NAC. Nevertheless, we have confluence concerning the target models. In fact, the resulting target models are already isomorphic, since the types of the connection nodes are not relevant for the target model.

7 Conclusion

In this paper, we have started a formal analysis of model transformations based on triple rules which have been introduced in [1] and applied to various application areas [2, 3, 7]. In [1], an important connection between the triple rules and the corresponding forward rules and transformations was given. This result was extended in [5] to a bijective correspondence between triple transformations $\varnothing \overset{tr^*}{\Longrightarrow} G$ based on triple rules tr^* and match consistent sequences $\varnothing \overset{tr^*_S}{\Longrightarrow} G_S \overset{tr^*_F}{\Longrightarrow} G$ based on corresponding source rules tr^*_S and target rules tr^*_F. This allows to define model transformations formally by source consistent forward transformation sequences $G_S \overset{tr^*_F}{\Longrightarrow} G$.

In order to analyse this kind of model transformations on a formal basis, we have defined the kernel of a forward rule and constructed a NAC based on this kernel. This allows to define kernel source consistency as source consistency restricted to kernel elements. Intuitively, this means that each forward rule is applied exactly once to the distinguished kernel element in the source graph generated by the corresponding source rule.

In our main results, we show that kernel source consistency implies NAC consistency and termination, and we give sufficient conditions for local confluence, which leads to functional behaviour of forward model transformations. Although the forward rules are non-deleting, this result is non-trivial because we have to ensure NAC consistency.

For a discussion of the relationship between model transformations based on triple and plain graph grammars we refer to [6].

At the moment, the conditions for distinguished kernel typing and type functionality are very restrictive, and only a subset of practical model transformations can be analyzed by our approach. In future work we want to extend our approach, in particular to forward rules that create either target or connection elements, but not both. As the discussion in Section 6 shows, the properties for local confluence in Thm. 2 are very restrictive. It would be interesting to analyse how these conditions and the concept for local confluence can be weakened, for example concerning confluence only on the target models. Moreover, we want to check under what conditions kernel source consistency is not only sufficient but also necessary for NAC consistency and termination. In addition to kernel NACs we want to consider also other NACs for forward rules. In this context, we want to apply the Critical Pair Lemma with NACs shown in [12] to forward transformations and verify confluence for other practical examples.

All our results dually hold for target and backward rules, which can be derived from triple rules similar to source and forward rules. This allows to analyze bidirectional model transformations between source and target languages, especially the problem of how to obtain functional inverse model transformations.

References

[1] 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)

[2] Aschenbrenner, N., Geiger, L.: Transforming Scene Graphs Using Triple Graph Grammars - A Practice Report. In: Proceedings of AGTIVE 2007 (2007)

[3] Guerra, E., de Lara, J.: Model View Management with Triple Graph Transformation Systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 351–366. Springer, Heidelberg (2006)

[4] Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)

[5] Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)

[6] Ehrig, H., Ehrig, K., Hermann, F.: From Model Transformation to Model Integration Based on the Algebraic Approach to Triple Graph Grammars. ECEASST (to appear, 2008)

[7] Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, University of Paderborn (2007)

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

[9] OMG: Unified Modeling Language, version 2.1.1 (2006)

[10] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)

[11] Bisztray, D., Ehrig, K., Heckel, R.: Case Study: UML to CSP Transformation. In: AGTIVE 2007 Graph Transformation Tool Contest (2007)

[12] Lambers, L.: Adhesive High-Level Replacement Systems with Negative Application Conditions. Technical Report 2007/14, TU Berlin (2007)

Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation

Hartmut Ehrig and Claudia Ermel

Department of Theoretical Computer Science and Software Technology
Technische Universität Berlin
ehrig@cs.tu-berlin.de, claudia.ermel@tu-berlin.de

Abstract. An important requirement of model transformations is the preservation of the behavior of the original model. A model transformation is *semantically correct* if for each simulation run of the source system we find a corresponding simulation run in the target system. Analogously, we have *semantical completeness*, if for each simulation run of the target system we find a corresponding simulation run in the source system.

In our framework of graph transformation, models are given by graphs, and graph transformation rules are used to define the operational behavior of visual models (called simulation rules). In order to compare the semantics of source and target models, we assume that in both cases operational behavior can be defined by simulation rules. The model transformation from source to target models is given by another set of graph transformation rules. These rules are also applied to the simulation rules of the source model. The main result in this paper states the conditions for model and rule transformations to be semantically correct and complete. The result is applied to analyze the behavior of a model transformation from a domain-specific visual language for production systems to Petri nets.

1 Introduction

In recent years, visual models represented by graphs have become very popular in model-based software development. The shift of paradigm from pure programming to visual modeling and model-driven development (MDD) led to a variety of domain-specific modeling languages (DSMLs) on the one hand, but also to a wide-spread use of general diagrammatic modeling languages such as UML and Petri nets. DSMLs provide an intuitive, yet precise way in order to express and reason about concepts at their natural level of abstraction. Starting with a domain-specific model, *model transformation* is the key technology of MDD and serves a variety of purposes, including the refinements of models, their mapping to implementations and/or semantic domains, consistency management and model evolution. For example, a complete design and analysis process involves designing the system using the design language, transforming it into the analysis language, and performing the verification and analysis on the analysis model.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 194–210, 2008.

In such a scenario, it is very important that the transformation preserves the semantics of the design model.

In this paper we study semantical correctness and completeness of model transformations provided that the source and the target languages have already a formal semantics. Approaches exist where semantic equivalence between one source model and its transformed target model is shown using bisimulation. In the approach of Karsai et al. [1] the particular transformation resulted in an output model that preserves the semantics of the input model with respect to a particular property. However, analogously to syntactical correctness proofs, it is desirable to have a more general concept for showing semantical correctness and completeness of a model transformation, independent of concrete source models.

This paper discusses an approach to verify semantical correctness of model transformations on the level of model transformation rules. Basically, semantical correctness of a model transformation means that for each simulation sequence of the source system we find a corresponding simulation sequence in the target system. Vice versa, we have semantical completeness, if for each simulation sequence in the target system there is a corresponding sequence simulating the source model. In order to compare the semantics of the source and target models, we assume that in both cases operational behavior can be defined by simulation graph rules. We then apply the model transformation to the simulation rules of the source model, leading to a so-called *rule transformation*. The resulting rules are compared to the given simulation rules of the target language.

The main result in this paper states the conditions for model and rule transformations to be semantically correct and complete. The paper generalizes and extends results from *simulation-to-animation model and rule transformation* (*S2A* transformation), which realizes a consistent mapping from simulation steps in a behavioral modeling language to animation steps in a more suitable domain-specific visualization [2,3,4]. The result is applied to analyze the behavior of a model and rule transformation from a domain-specific language for production systems to Petri nets.

The structure of the paper is as follows: In Section 2, our running example, a domain-specific visual language for production systems, is introduced. Section 3 reviews the basic concepts of model and rule transformation based on graph transformation. In Section 4, the notions *semantical correctness* and *semantical completeness* of model transformations are formally defined, and conditions are elaborated for correct and complete model transformations defined by graph rules. The main result is applied to our running example, showing that the model and rule transformation from production systems to Petri nets is semantically correct and complete. The result of the rule transformation is compared with the given simulation rules of the target language of Petri nets. Section 5 discusses related work, and Section 6 concludes the paper. Please note that the long version of this paper, the technical report [5], contains the complete case study as well as all definitions and full proofs of the theorems presented in this paper.

2 Example: Simulation of Production Systems

In this section we provide a description of a DSML for production systems and its operational semantics using graph transformation rules (a slightly simplified version of the DSML presented in [6]). Note that the rules are shown in concrete syntax, thus making the expression of operational semantics intuitive and domain-specific. Fig. 1 shows in the upper part a type graph for the production system language. The language contains different kinds of machines, which can be connected through conveyors. Human operators are needed to operate the machines, which consume and produce different types of pieces from/to conveyors. Conveyors can also be connected. The lower part of Fig. 1 shows a production system model (a graph typed over the type graph above) using a visual concrete syntax. The model contains six machines (one of each type), two operators, six conveyors and four pieces. Machines are represented as boxes, except generators, which are depicted as semi-circles with the kind of piece they generate written inside. Operators are shown as circles, conveyors as lattice boxes, and each kind of piece has its own shape. Two operators are currently operating a generator of cylindrical pieces and a packaging machine respectively.

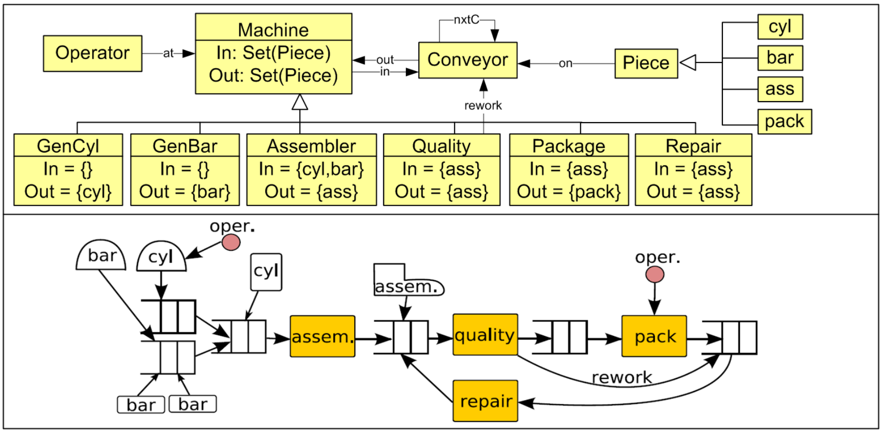


Fig. 1. Type Graph for Producer Systems and Instance Graph

Fig. 2 shows some of the graph transformation rules that describe the operational semantics for production systems. Rule assemble specifies the behaviour of an assembler machine, which converts one cylinder and a bar into an assembled piece. The rule can be applied if every specified element (except those marked as $\{new\}$) can be found in the model. When such an occurrence is found, then the elements marked as $\{del\}$ are deleted, and the elements marked as $\{new\}$ are created. Note that even if we depict rules using this compact notation, we use the DPO formalization in our graph transformation rules. In practice, this means that a rule cannot be applied if it deletes a node but not all its adjacent edges. In addition, we consider only injective matches. Rule genCylinder models

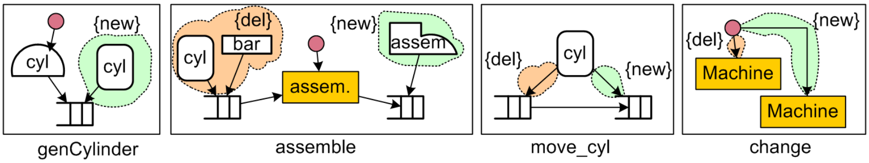


Fig. 2. Some Simulation Rules for Production Systems

the generation of a piece of kind *cylinder* which requires that the cylinder generator machine is attended by an operator and connected to a conveyor. Rule move_cyl describes the movement of cylinder pieces through conveyors. Finally, rule change models the movement of an operator from one machine (of any kind) to another one. Note that we may use abstract objects in rules (e.g., Machine is an abstract node type). In this case, the abstract objects in a rule are instantiated to objects of any concrete subclass [7]. Additional rules (not depicted) model the behaviour of the other machine types.

3 Basic Concepts of Model and Rule Transformation

In this section, we define model transformation by graph and rule transformation based on visual language specifications as typed graph transformation systems.

3.1 Visual Languages and Simulation

We use typed algebraic graph transformation systems (TGTS) in the double-pushout-approach (DPO) [8] which have proven to be an adequate formalism for visual language (VL) modeling. A VL is modeled by a type graph capturing the underlying visual alphabet, i.e. the symbols and relations which are available. Sentences or diagrams of the VL are given by graphs typed over the type graph. We distinguish abstract and concrete syntax in alphabets and models, where the concrete syntax includes the abstract symbols and relations, and additionally defines graphics for their visualization. Formally, a VL can be considered as a subclass of graphs typed over a type graph TG in the category $\mathbf{Graphs_{TG}}$.

For behavioral diagrams, an operational semantics can be given by a set of simulation rules P_S, using the abstract syntax of the modeling VL, defined by simulation type graph TG_S. A simulation rule $p = (L \leftarrow K \rightarrow R) \in P_S$ is a TG_S-typed graph transformation rule, consisting of a left-hand side L, an interface K, a right-hand side R, and two injective morphisms. In the case $L = K$, the rule is called *non-deleting*. Applying rule p to a graph G means to find a match of $L \xrightarrow{m} G$ and to replace the occurrence $m(L)$ of L in G by R leading to the target graph G'. Such a graph transformation step is denoted by $G \overset{(p,m)}{\Longrightarrow} G'$, or simply by $G \Rightarrow G'$. In the DPO approach, the deletion of $m(L)$ and the addition of R are described by two pushouts (a DPO) in the category $\mathbf{Graphs_{TG}}$ of typed graphs. A rule p may be extended by a set of *negative application conditions (NACs)* [8], describing situations in which the rule should not be applied to G. Formally, match $L \xrightarrow{m} G$ satisfies NAC $L \xrightarrow{n} N$ if there does not exist an injective graph morphism $N \xrightarrow{x} G$ with $x \circ n = m$. A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of graph transformation steps is called *transformation* and denoted as $G_0 \overset{*}{\Rightarrow} G_n$. A transformation $G_0 \overset{*}{\Rightarrow} G_n$, where rules from P are applied as long as possible, (i.e. as long as matches can be found satisfying the NACs), is denoted by $G_0 \overset{P\ !}{\Longrightarrow} G_n$. We regard a model's *simulation language* VL_S, typed over simulation alphabet TG_S, as a sublanguage of the modeling language VL, so that all diagrams $G_S \in VL_S$ represent different states of the same model during simulation. Based on VL_S, the operational semantics of a model is given by a *simulation specification.*

Definition 1 (*Simulation Specification*). *Given a visual language VL_S typed over TG_S, i.e. $VL_S \subseteq \mathbf{Graphs_{TG_S}}$, a* simulation specification $SimSpec_{VL_S} = (VL_S, P_S)$ *over VL_S is given by a typed graph transformation system (TG_S, P_S) so that VL_S is closed under simulation steps, i.e. $G_S \in VL_S$ and $G_S \Rightarrow H_S$ via $p_S \in P_S$ implies $H_S \in VL_S$. The rules $p_S \in P_S$ are called* simulation rules.

Example 1. The simulation specification $SimSpec_{VL_S} = (VL_S, P_S)$ for the production system consists of the visual language VL_S typed over TG_S, where TG_S is the type graph shown in the upper part of Fig. 1, P_S is the set of simulation rules partly shown in Fig. 2, and VL_S consists of all graphs that can occur in any production system simulation scenario, e.g. the instance graph shown in the lower part of Fig. 1 is one element of VL_S.

We divide a model and rule transformation from a source to a target simulation specification into two phases: in the first phase (called *S2I* transformation phase), non-deleting graph transformation rules are applied to the source model and to the source language simulation rules and add elements from the target language to the source model and rule graphs. The result is an *integrated simulation specification*, i.e. the resulting integrated model and simulation rules contain both source and target model elements. The second phase (called *I2T* transformation phase) restricts the integrated model and the integrated simulation rules to the type graph of the target language.

3.2 *S2I* Model and Rule Transformation

In order to transform a source simulation specification $SimSpec_{VL_S}$ to an integrated source-target simulation specification $SimSpec_{VL_I}$ where VL_I contains at least VL_S and VL_T, we define an *S2I* transformation $S2I = (S2I_M, S2I_R)$ consisting of a model transformation $S2I_M$, and a corresponding rule transformation $S2I_R$. The $S2I_M$ transformation applies model transformation rules from a rule set Q to each $G_S \in VL_S$ as long as possible (denoted by $G_S \overset{Q\ !}{\Longrightarrow} G_I$). The applications of the model transformation rules add symbols from the target language to the model state graphs. The resulting set of graphs G_I comprises the source-and-target integration language VL_I.

Definition 2 (*Model Transformation* $S2I_M$). *Given a simulation specification $SimSpec_{VL_S} = (VL_S, P_S)$ with VL_S typed over TG_S and a type graph TG_I,*

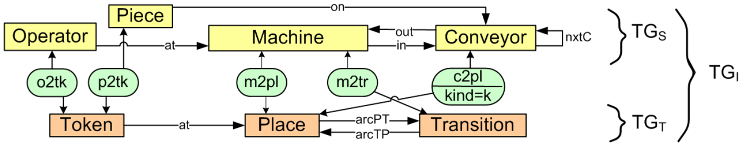


Fig. 3. Type Graph TG_I for the *ProdSystem2PetriNet* Model Transformation

called integration type graph, with $TG_S \subseteq TG_I$*, then a* model transformation $S2I_M : VL_S \rightarrow VL_I$ *is given by* $S2I_M = (VL_S, TG_I, Q)$ *where* (TG_I, Q) *is a typed graph transformation system with non-deleting rules* $q \in Q$*, and* $S2I_M$*-transformations* $G_S \overset{Q\ !}{\Longrightarrow} G_I$ *with* $G_S \in VL_S$*. The* integrated language VL_I *is defined by* $VL_I = \{G_I | \ \exists\ G_S \in VL_S \ \wedge\ G_S \overset{Q\ !}{\Longrightarrow} G_I\}$*. This means,* $G_S \overset{Q\ !}{\Longrightarrow} G_I$ *implies* $G_S \in VL_S$ *and* $G_I \in VL_I$*.*

Example 2. The integrated visual language VL_I for the model transformation from production systems to Petri nets is defined by the integrated type graph TG_I in Fig. 3. The subtypes of Machine and Piece are not depicted since they are not needed in our model transformation rules. Machines and conveyors are mapped to places; pieces and operators are elements moving from one place-like element to another and hence mapped to tokens. Connections between conveyors or between machines and conveyors which indicate the way token-like elements are transported, are mapped to transitions.

The model transformation rules Q are shown in Fig. 4.

Rules *mach2place* and *conv2place* generate places for machines and conveyors. A conveyor is transformed to four different places, realizing a flattening from our model with distinct piece types to a P/T net with indistinguishable tokens.

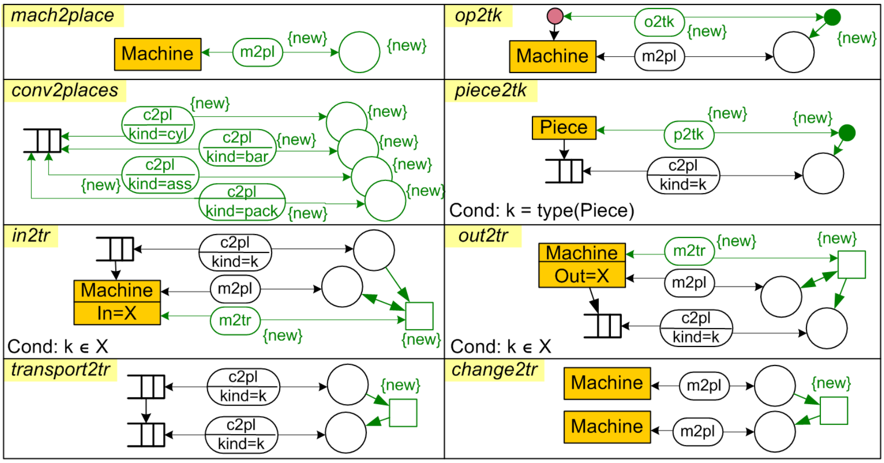


Fig. 4. *ProdSystem2PetriNet* Model Transformation Rules

Distinguishing the pieces is realized in the P/T net by placing them in distinct places. Rules *op2tk* and *piece2tk* generate tokens for operators and pieces on the places associated to their respective machines or conveyors. Transitions are generated for each connection between two conveyors (rule *transport2tr*) or between a machine and a conveyor (rules *in2tr* and *out2tr*). Two more rules (not depicted) exist which are applicable only if a machine is already connected to a transition and which just add an arc connecting the existing transition to an additional conveyor place. A machine's transition is always connected by a double arc to the machine's place to ensure that a machine is working only if an operator is present.

The result of an $S2I_M$-transformation is illustrated in Fig. 5, where a part from the model in Fig. 1 has been transformed, applying the model transformation rules in Fig. 4 as long as possible, but at most once at the same match. Our aim in this paper is not only to transform model states but to obtain a complete integrated simulation specification, including simulation rules, from the source simulation specification. In [5] we review a construction from [3,2], allowing us to apply the $S2I$ transformation rules from Q also to the simulation rules, resulting in a set of integrated simulation rules. Basically, the model transformation rules are applied to each graph of a simulation rule $p_S = (L_S \leftarrow K_S \rightarrow R_S)$ as long as possible, resulting in an integrated simulation rule $p_I = (L_I \leftarrow K_I \rightarrow R_I)$. In [5] we define rule transformation for the case without NACs. An extension to NACs is given in [3,2]. Based on this definition of rule transformation, we now define an $S2I_R$ transformation of rules, leading to an $S2I$ transformation $S2I = (S2I_M, S2I_R)$ from the source simulation specification $SimSpec_{VL_S}$ to the integrated simulation specification $SimSpec_{VL_I}$.

Definition 3 (*Rule Transformation* $S2I_R$). *Given a simulation specification $SimSpec_{VL_S} = (VL_S, P_S)$ and an $S2I_M$-transformation $S2I_M = (VL_S, TG_I, Q)$, then a* rule transformation $S2I_R : P_S \rightarrow P_I$ *is given by* $S2I_R = (P_S, TG_I, Q)$ *and* $S2I_R$ transformation sequence $p_S \overset{Q\ \ !}{\Longrightarrow} p_I$ *with $p_S \in P_S$, where rule transformation steps* $p_1 \overset{q}{\Longrightarrow} p_2$ *with $q \in Q$ (see [5]) are applied as long as possible. The* integrated simulation rules P_I *are defined by* $P_I = \{p_I | \ \exists\ p_S \in P_S \wedge p_S \overset{Q\ \ !}{\Longrightarrow} p_I\ \}$. *This means* $p_S \overset{Q\ \ !}{\Longrightarrow} p_I$ *implies $p_S \in P_S$ and $p_I \in P_I$.*

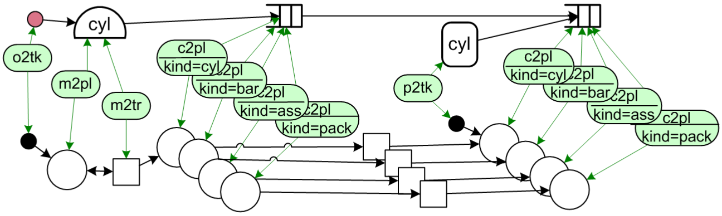


Fig. 5. *ProdSystem2PetriNet: $S2I_M$* Model Transformation Result

Definition 4 (*S2I* Transformation, Integrated Simulation Specification). *Given $SimSpec_{VL_S} = (VL_S, P_S)$, an $S2I_M$ transformation $S2I_M : VL_S \rightarrow VL_I$ and an $S2I_R$ transformation $S2I_R : P_S \rightarrow P_I$, then*

1. *$S2I : SimSpec_{VL_S} \longrightarrow SimuSpec_{VL_I}$, defined by $S2I = (S2I_M, S2I_R)$ is called $S2I$* transformation*.*
2. *$SimSpec_{VL_I} = (VL_I, P_I)$ is called* integrated simulation specification*, and each transformation step $G_I \overset{p_I}{\Longrightarrow} H_I$ with $G_I, H_I \in VL_I$ and $p_I \in P_I$ is called* integrated simulation step*.*

Example 3. Fig.6 shows three integrated simulation rules, the result of $S2I_R$ transformation, i.e. of applying the model transformation rules from Fig. 4 to the source simulation rules *genCylinder*, *move_cyl* and *change* from Fig. 2.

3.3 *I2T* Transformation

In the *I2T* transformation phase, we start with the integrated simulation specification $SimuSpec_{VL_I}$ and generate the target simulation specification $SimuSpec_{VL_T}$ by restricting the integrated model graph and the integrated simulation rules to the type graph of the target language.

Definition 5 (*I2T* Transformation and Target Simulation Specification). *Given an $S2I$ transformation $S2I : SimSpec_{VL_S} \rightarrow SimSpec_{VL_I}$, then*

1. *$I2T$: $SimSpec_{VL_I} \longrightarrow SimSpec_{VL_T}$, called $I2T$* transformation*, is defined by $I2T = (I2T_M : VL_I \rightarrow VL_T, I2T_R : P_I \rightarrow P_T)$ with*
   - *$I2T_M(G_I) = G_I|_{TG_T}$ (called $I2T_M$ transformation), and*
   - *$I2T_R(p_I) = p_I|_{TG_T}$ (called $I2T_R$ transformation).*
2. *$SimSpec_{VL_T} = (VL_T, P_T)$ with $VL_T = \{G_I|_{TG_T} \mid G_I \in VL_I\}$ and $P_T = \{p_I|_{TG_T} \mid p_I \in P_I\}$ is called* target simulation specification*, and each transformation step $G_T \overset{p_T}{\Longrightarrow} H_T$ with $G_T, H_T \in VL_T$ and $p_T \in P_T$ is called* target simulation step*.*

Example 4. Fig.7 shows the target simulation rules, the result of $I2T_R$ transformation, i.e. of restricting the integrated simulation rules from Fig. 6 to the type graph of TG_T of the target language from Fig. 3 (i.e. the Petri net type graph).

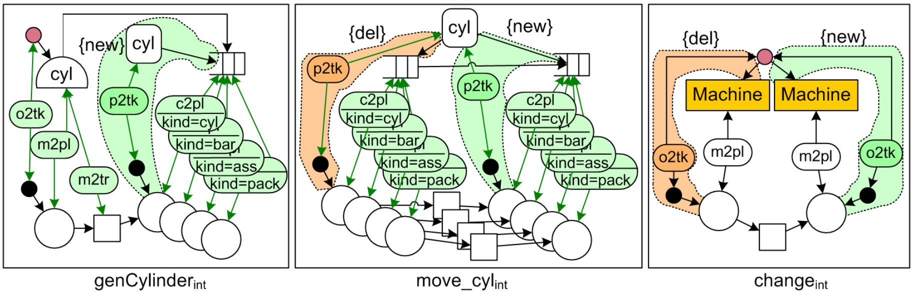


Fig. 6. Some Integrated Simulation Rules resulting from $S2I_R$ Transformation

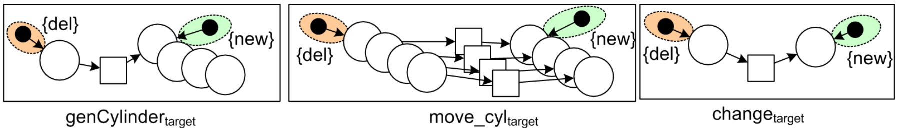


Fig. 7. Some Target Simulation Rules resulting from $I2T_R$ Transformation

We now can define the complete $S2T$ model and rule transformation by combining the two transformation phases $S2I$ and $I2T$.

Definition 6 ($S2T$ Transformation). *Given an S2I transformation* $S2I : SimSpec_{VL_S} \to SimSpec_{VL_I}$*, and an I2T transformation* $I2T\colon SimSpec_{VL_I} \to SimSpec_{VL_T}$*, then* $S2T\colon SimSpec_{VL_S} \to SimSpec_{VL_T}$*, called S2T* transformation*, is defined by* $S2T = (S2T_M : VL_S \to VL_T, S2T_R : P_S \to P_T)$ *with*

- $S2T_M = I2T_M \circ S2I_M$ *(called $S2T_M$ transformation), and*
- $S2T_R = I2T_R \circ S2I_R$ *(called $S2T_R$ transformation).*

4 Semantical Correctness and Completeness of Model and Rule Transformations

4.1 Semantical Correctness of *S2I* Transformations

In our case, semantical correctness of an $S2I$ transformation means that for each simulation step $G_S \overset{p_S}{\Longrightarrow} H_S$ there is a corresponding simulation step $G_I \overset{p_I}{\Longrightarrow} H_I$ where G_I (resp. H_I) are obtained by model transformation from G_S (resp. H_S), and p_I by rule transformation from p_S. Note that instead of a single step $G_I \overset{p_I}{\Longrightarrow} H_I$ we can also handle more general sequences $G_I \overset{*}{\Longrightarrow} H_I$ using concurrent rules and transformations. In [3], it is shown that the following properties have to be fulfilled by an $S2I$-transformation in order to be semantically correct:

Definition 7 (Termination of $S2I_M$ and Rule Compatibility of $S2I$)
An $S2I_M$ transformation $S2I_M : VL_S \to VL_I$ is terminating *if each transformation $G_S \overset{Q\ *}{\Longrightarrow} G_n$ can be extended to $G_S \overset{Q\ *}{\Longrightarrow} G_n \overset{Q\ *}{\Longrightarrow} G_m$ such that no $q \in Q$ is applicable to G_m anymore. An S2I-transformation $S2I = (S2I_M : VL_S \to VL_I, S2I_R : P_S \to P_A)$ with $S2I_M = (VL_S, TG_I, Q)$ is called* rule compatible*, if for all $p_I \in P_I$ and $q \in Q$ we have that p_I and q are parallel and sequential independent. More precisely, for each $G \overset{p_I}{\Longrightarrow} H$ with $G_S \overset{Q\ *}{\Longrightarrow} G$ and $H_S \overset{Q\ *}{\Longrightarrow} H$ for some $G_S, H_S \in VL_S$ and each $G \overset{q}{\Longrightarrow} G'$ (resp. $H \overset{q}{\Longrightarrow} H'$) we have parallel (resp. sequential) independence of $G \overset{p_I}{\Longrightarrow} H$ and $G \overset{q}{\Longrightarrow} G'$ (resp. $H \overset{q}{\Longrightarrow} H'$).*

Theorem 1 (Semantical Correctness of *S2I*)
Given an S2I-transformation $S2I : SimSpec_{VL_S} \to SimSpec_{VL_I}$ with $S2I = (S2I_M : VL_S \to VL_I, S2I_R : P_S \to P_I)$ which is rule compatible, and $S2I_M$

is terminating. Then, S2I is semantically correct in the sense that we have for each simulation step $G_S \overset{p_S}{\Longrightarrow} H_S$ *with* $G_S \in VL_S$ *and each* $S2I_R$*-transformation sequence* $p_S \overset{Q\ !}{\Rrightarrow} p_I$ *(see Def. 3):*

$$\begin{tikzcd}
G_S \arrow[rr, Rightarrow, "Q\ \ !"] \arrow[dd, Rightarrow] & & G_I \arrow[dd, Rightarrow] \\
p_S \arrow[rr, Rrightarrow, "Q\ \ !"] & & p_I \\
H_S \arrow[rr, Rightarrow, "Q\ \ !"] & & H_I
\end{tikzcd}$$

1. *$S2I_M$-transformation sequences* $G_S \overset{Q\ !}{\Longrightarrow} G_I$ *and* $H_S \overset{Q\ !}{\Longrightarrow} H_I$*, and*
2. *an integrated simulation step* $G_I \overset{p_I}{\Longrightarrow} H_I$

Proof. (See [5], similar to the proof of Semantical Correctness of *S2A* in [3]).

Example 5. Our *ProdSystem2PetriNet* model transformation is terminating, provided that all model transformation rules are applied at most once at each possible match. (For automatic model transformations, this can be ensured by using adequate NACs). Moreover, $S2I_R$ is rule compatible since all $p_I \in P_I$ are parallel and sequentially independent from the model transformation rules $q \in Q$. This is shown by considering all overlapping matches from a rule pair (q, p_I) into an integrated model $G_I : L_q \xrightarrow{h} G_I \xleftarrow{m} L_I$. Each overlap either is preserved by both rules, or $h(L_q)$ is completely included in $m(L_I)$. The first case is uncritical. In the second case, rule q is not applicable since it has been applied before at the same match. Hence this overlap cannot lead to a parallel dependency.

4.2 Semantical Correctness of *I2T* Transformations

We now consider the semantical correctness of the *I2T* transformation phase, which was defined in Def. 5 as the restriction of the integrated model graph and the integrated simulation rules to the type graph TG_T of the target VL.

Theorem 2 (Semantical Correctness of *I2T* Transformations). *Given an S2I transformation* $S2I = (S2I_M : VL_S \to VL_I, S2I_R : P_S \to P_I) : SimSpec_{VL_S} \to SimSpec_{VL_I}$*, and an I2T transformation* $I2T{:}SimSpec_{VL_I} \to SimSpec_{VL_T}$ *defined by* $I2T = (I2T_M, I2T_R)$ *according to Def. 5. Then, I2T is* semantically correct *in the sense that we have for each integrated simulation step* $G_I \overset{p_I}{\Longrightarrow} H_I$ *with* $G_I \in VL_I$ *and each* $I2T_R$*-transformation* $I2T_R(p_I) = p_I|_{TG_T} = p_T$*:*

$$\begin{tikzcd}
G_I \arrow[rr, "I2T_M"] \arrow[dd, Rightarrow] & & G_T \arrow[dd, Rightarrow] \\
p_I \arrow[rr, "I2T_R"] & & p_T \\
H_I \arrow[rr, "I2T_M"] & & H_T
\end{tikzcd}$$

1. $I2T_M(G_I) = G_T$ *and* $I2T_M(H_I) = H_T$*, and*
2. *a target simulation step* $G_T \overset{p_T}{\Longrightarrow} H_T$

Proof. See [5].

4.3 Semantical Completeness of *S2I* Transformations

In this section we consider the relation between an integrated simulation specification $SimSpec_{VL_I}$ and the corresponding source simulation specification $SimSpec_{VL_S}$. Similar to the construction of the target simulation specification

$SimSpec_{VL_T}$ by restriction of $SimSpec_{VL_I}$ to TG_T, the source simulation specification $SimSpec_{VL_S}$ can be re-constructed by restricting the integrated model graph and simulation rules to the type graph TG_S of the source language.

Definition 8 (*I2S* Backward Transformation). *Given an S2I transformation S2I* : $SimSpec_{VL_S} \rightarrow SimSpec_{VL_I}$, *then I2S*: $SimSpec_{VL_I} \rightarrow SimSpec_{VL_S}$, *called I2S* backward transformation, *is defined by I2S* $= (I2S_M : VL_I \rightarrow VL_S, I2S_R : P_I \rightarrow P_S)$ *with*

- $I2S_M(G_I) = G_I|_{TG_S}$ *(called* $I2S_M$ *backward transformation), and*
- $I2S_R(p_I) = p_I|_{TG_S}$ *(called* $I2S_R$ *backward transformation).*

The *S2I* transformation is called *faithful* if $S2I_M(G_S) = G_I$ implies $I2S_M(G_I) = G_S$ and $S2I_R(p_S) = p_I$ implies $I2S_M(p_I) = p_S$.

Remark 1. We call a rule $L \xrightarrow{q} R$ *faithful* if the restriction $q|_{TG_S}$ is the identity. It is straightforward (see [5]) to show that the *S2I* transformation is faithful if all rules $q \in Q$ are faithful.

Theorem 3 (Semantical Completeness of *S2I* Transformations). *Given a faithful S2I transformation* $S2I = (S2I_M, S2I_R) : SimSpec_{VL_S} \rightarrow SimSpec_{VL_I}$ *and its backward transformation* $I2S = (I2S_M, I2S_R) : VL_I \rightarrow VL_S$, *with* $I2S_M : VL_I \rightarrow VL_S$ *and* $I2S_R : P_I \rightarrow P_S$. *Then, S2I is* semantically complete *in the sense that we have for each integrated simulation step* $G_I \overset{p_I}{\Longrightarrow} H_I$ *with* $G_I, H_I \in VL_I$ *and* $p_I \in P_I$*:*

1. $I2S_M(G_I) = G_S$ *and* $I2S_M(H_I) = H_S$ *with* $S2I_M(G_S) = G_I$, $S2I_M(H_S) = H_I$, *and*
2. *a source simulation step* $G_S \overset{p_S}{\Longrightarrow} H_S$ *with* $I2S_R(p_I) = p_I|_{TG_S} = p_S$ *and* $S2I_R(p_S) = p_I$.

$$\begin{tikzcd}
G_I \arrow[r, "I2S_M"] \arrow[dd, Rightarrow] & G_S \arrow[dd, Rightarrow] \\
p_I \arrow[r, "I2S_R"] & p_S \\
H_I \arrow[r, "I2S_M"] & H_S
\end{tikzcd}$$

Proof. See [5].

Example 6. Our *ProdSystem2PetriNet* model transformation is faithful since all model transformation rules (see Fig. 4) add only language elements typed over $TG_I \setminus TG_S$. Hence, the rules are faithful, and the *ProdSystem2PetriNet S2I* transformation is semantically complete according to Thm. 3.

4.4 Semantical Completeness of *I2T* Transformations

Semantical completeness of *I2T* transformations means that for each simulation step in the target simulation specification we get a corresponding simulation step in the integrated simulation specification. We require the following property to be fulfilled for an *I2T* transformation in order to be semantically complete. (This property is discussed for our case study in Example 7.)

Definition 9 (*I2T* Completeness Condition)
Given a target simulation rule $p_T \in P_T$, *then due to the construction of* $SimSpec_{VL_T}$ *by restriction, there exists an integrated simulation rule* $p_I \in P_I$ *such that* $p_T =$

$p_I|_{TG_T}$. Then, for each target transformation $G_T \overset{p_T}{\Longrightarrow} H_T$ with $G_T \in VL_T$ and context graph D_T and morphism $K_T \to D_T$ we require that there exists a context graph D_I typed over TG_I and morphism $K_I \to D_I$ such that

1. *$K_T \to D_T$ is the restriction of $K_I \to D_I$ to TG_T, i.e. that we have two pullbacks in the diagonal squares in the diagram to the right.*
2. *For the pushout objects G_I and H_I in the front squares we have $G_I, H_I \in VL_I$.*

$$\begin{tikzcd}
 & L_T \arrow[dl] \arrow[dd] & & K_T \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] & & R_T \arrow[dl] \arrow[dd] \\
L_I \arrow[dd] & & K_I \arrow[ll] \arrow[rr] \arrow[dd] & & R_I \arrow[dd] & \\
 & G_T & & D_T \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] & & H_T \\
G_I & & D_I \arrow[ll] \arrow[rr] \arrow[dd] & & H_I & \\
 & & & TG_T \arrow[dl] & & \\
 & & TG_I & & &
\end{tikzcd}$$

Theorem 4 (Semantical Completeness of *I2T*). *Each I2T transformation $I2T = (I2T_M, I2T_R)$ which satisfies the I2T completeness condition (see Def. 9) is semantically complete in the sense that for each target transformation $G_T \overset{p_T}{\Longrightarrow} H_T$ with $G_T \in VL_T$ via simulation rule $p_T \in P_T$ with $p_T = p_I|_{TG_T}$ for some $p_I \in P_I$ there is an integrated transformation $G_I \overset{p_I}{\Longrightarrow} H_I$ such that*

– *$G_I, H_I \in VL_I$*
– *$G_T = G_I|_{TG_T}$ and $H_T = H_I|_{TG_T} \in VL_T$*

$$\begin{tikzcd}
G_I \arrow[r, "I2T_M"'] \arrow[dd, Rightarrow] & G_T \arrow[dd, Rightarrow] \\
p_I \arrow[r, "I2T_R"] & p_T \\
H_I \arrow[r, "I2T_M"] & H_T
\end{tikzcd}$$

Proof. See [5].

Example 7. We show that our *ProdSystem2PetriNet I2T* transformation does not fulfill the completeness condition and discuss an adaption of the model transformation rules in order to achieve satisfaction of the completeness condition.

Based on the set P_T of target rules resulting from the *ProdSystem2PetriNet I2T* transformation, we may apply more than one $p_T \in P_T$ to the same G_T. Consider for example the three target rules in Fig. 7. All of them are applicable to a target graph G_T if there exists a match from the "biggest" rule $move\_cyl_{target}$ to G_T. Thus, when applying any p_T from this set of applicable rules to G_T, we always get the same transformation span $G_T \leftarrow D_T \rightarrow H_T$, but the applied rule p_T might be the restriction of an integrated rule $p_I \in P_I$ such that the first part of the completeness condition is fulfilled, but not the second one: i.e., there exists a context graph D_I and morphism $K_I \to D_I$ such that the pushout objects G_I and H_I are *not* in VL_I. This might happen since our model transformation "forgets" information, i.e. given a target rule (typed over the Petri net language), we do not know from which integrated rule this target rule was constructed.

In order to avoid this situation, we propose a slight extension of the target type graph TG_T (Fig. 3) and the model transformation rules (Fig. 4). We introduce a suitable annotation of Petri net elements (transitions or places) by attributes keeping the information about the original role of the element. For example, we annotate each place originating from a machine by the machine type (e.g. *Assembler* or *GenCyl*), and each place originating from a conveyor by the piece type a token on this place would represent (e.g. *cyl* or *bar*). The annotation should establish a 1:1 correspondence between the integrated rules in P_I and the target rules in P_T, and between integrated models $G_I \in VL_I$ and their target models $G_T \in VL_T$. Hence, a target rule $p_T \in P_T$ which is a restriction of an integrated rule $p_I \in P_I$ now is applicable to a target model $G_T \in VL_T$ only if there exists $G_I \in VL_I$ to which p_I is applicable. Thus, the context graph D_I and the morphism $K_I \to D_I$ are unique and lead to pushouts in the front squares such that G_I and H_I are in VL_I, i.e. also the second part of the completeness condition is now satisfied. Note that the annotation does not affect the semantical correctness and completeness of *S2I* (shown in Examples 5 and 6) since *S2I* is still terminating, rule compatible and faithful.

4.5 Semantical Correctness and Completeness of *S2T* Transformations

Putting all steps together, we find that a source-to-target model transformation *S2T*: $SimSpec_{VL_S} \to SimSpec_{VL_T}$ with $S2T = I2T \circ S2I$ is semantically correct and complete if *I2T* and *S2I* are semantically correct and complete. In this case, we get for each source simulation step in $SimSpec_{VL_S}$ a corresponding target simulation step in $SimSpec_{VL_T}$, and vice versa.

Theorem 5 (Semantical Correctness and Completeness of *S2T*). *Each S2T transformation* $S2T = (S2T_M, S2T_R) : SimSpec_{VL_S} \to SimSpec_{VL_T}$ *with* $S2T = I2T \circ S2I$, *where* $S2I : SimSpec_{VL_S} \to SimSpec_{VL_I}$ *with S2I rule compatible,* $S2I_M$ *terminating (Def. 7) and S2I faithful, and* $I2T : SimSpec_{VL_I} \to SimSpec_{VL_T}$, *with I2T satisfying the completeness condition (Def. 9), is semantically correct and complete in the following sense:*

1. Semantical Correctness: *For each source simulation step* $G_S \overset{p_S}{\Longrightarrow} H_S$ *with* $G_S \in VL_S$ *and* $S2T_R$*-transformation sequence* $p_S \overset{Q\ \ !}{\Longrightarrow} p_I \overset{|_{TG_T}}{\longrightarrow} p_T$ *we have*

1. $S2T_M$*-trafo* $S2T_M(G_S) = G_T : G_S \overset{Q!}{\Longrightarrow} G_I \overset{|_{TG_T}}{\longrightarrow} G_T$,
   $S2T_M$*-trafo* $S2T_M(H_S) = H_T : H_S \overset{Q!}{\Longrightarrow} H_I \overset{|_{TG_T}}{\longrightarrow} H_T$, *and*
2. *a target simulation step* $G_T \overset{p_T}{\Longrightarrow} H_T$ *via target simulation rule* $p_T \in P_T$

2. Semantical Completeness: *For each target transformation step* $G_T \overset{p_T}{\Longrightarrow} H_T$ *with* $G_T \in VL_T$ *and* $p_T \in P_T$ *there is a source simulation step* $G_S \overset{p_S}{\Longrightarrow} H_S$ *with*

– $p_T = S2T_R(p_S)$,
– $G_T = S2T_M(G_S)$ *and*
  $H_T = S2T_M(H_S) \in VL_T$.

$$\begin{tikzcd}
G_S \arrow[rr, Rightarrow, "S2I_M"] \arrow[rrrrr, bend left, "S2T_M"] \arrow[dd, Rightarrow] & & G_I \arrow[rrr, "I2T_M"] \arrow[dd, Rightarrow] & & & G_T \arrow[dd, Rightarrow] \\
& p_S \arrow[rr, Rrightarrow, "S2I_R"] \arrow[rrr, bend left, "S2T_R"] & & p_I \arrow[r, "I2T_R"] & p_T & \\
H_S \arrow[rr, Rightarrow, "S2IM"] \arrow[rrrrr, bend left, "S2T_M"] & & H_I \arrow[rrr, "I2T_M"] & & & H_T
\end{tikzcd}$$

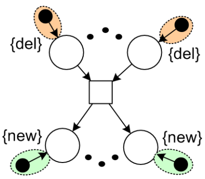
This means especially that the transformation step $G_T \overset{p_T}{\Longrightarrow} H_T$ *becomes a simulation step in* $SimSpec_{VL_T}$, *generated from the simulation step* $G_S \overset{p_S}{\Longrightarrow} H_S$.

Proof. By semantical correctness of *S2I* and *I2T* (Theorems 1 and 2), we get directly the semantical correctness of $S2T = I2T \circ S2I$. By semantical completeness of *S2I* and *I2T* (Theorems 3 and 4), we get directly the semantical completeness of $S2T = I2T \circ S2I$.

4.6 Relationship of $SimSpec_{VL_T}$ and Target Language Semantics

In the case that the target language has already an operational semantics given by simulation rules $P_{\bar{T}}$ (like in our running example, where the target language is the language of Petri nets), we may require for our model transformation *S2T* to be *behavior-preserving* in the sense that for each model in VL_T the simulations via rules in P_T correspond to simulations via rules in $P_{\bar{T}}$ and vice versa.

Example 8. As classical semantics of a P/T net (with fixed arc weight 1) we generate for each transition with i input places and o output places in a given Petri net model a corresponding firing rule [9]. Such firing rules belong to the rule schema depicted to the right. For a transition with i input places and o output places there is the graph rule $p_{\bar{T}} \in P_{\bar{T}}$ where the transition with its environment is preserved by the rule, all (and only the) input places are marked each by one token in the left-hand side, and all (and only the) output places are marked each by one token in the right-hand side.



Furthermore, the rules must not be applied to transitions with larger environment which can be ensured by suitable NACs (called environment-preserving). Considering the target simulation rules P_T which resulted from our extended *ProdSystem2Petri S2T* transformation (i.e. the rules in Fig. 7, extended by annotations as described in Example 7), we notice two differences to $P_{\bar{T}}$:

1. The target rules in P_T have no environment-preserving NACs,
2. The Petri net elements in the target rules in P_T are annotated,
3. The target rules in P_T in general contain *context* in addition to the environment of a single transition.

In case 1, we add environment-preserving NACs to each target rule without changing their applicability, since the annotations ensure that each target rule can be applied to a transition with fixed environment, anyway.

In case 2, we omit the annotations in the target rules and argue that the rules without annotations (but with environment-preserving NACs) lead to the same transformations as before. We find that all target rules without annotations which are applicable to G_T at matches overlapping in the enabled transition and its environment have the same transformation span $G_T \leftarrow D_T \rightarrow H_T$ (they are semantically equivalent) (like e.g. the target rules in Fig. 7 which are semantically equivalent for a match from the "biggest" rule $move\_cyl_{target}$ to G_T). It can be checked easily that we have a similar situation for all other target rules. The NACs prevent that target rules without annotations are applied to transitions with a larger environment. All semantically equivalent target rules without annotations which are applicable at matches containing the same enabled transition, correspond to exactly one application of an annotated target rule at this match. Thus, we can omit the annotations in the target rules without causing changes of the possible target transformation steps.

In case 3, the behavior is preserved only if the additional context in each rule $p_T \in P_T$ can always be found for each match into any model in $SimSpec_{VL_T}$, and if this context is not changed by the rule. We have additional context for instance in the rules $genCylinder_{target}$ and $move\_cyl_{target}$ (see Fig. 7). Here, the context was generated due to the flattening of conveyors to sets of four places. Since this flattening was also performed for each conveyor in the source model G_S, we know that each match at which the rule $genCylinder_{target}$ without the three additional context places is applicable, corresponds to a match of the rule with context. This is true in our example for all firing rules containing context in addition to the active transition's environment. Hence, we can conclude that the $ProdSystem2Petri_{annotated}$ model transformation is not only semantically correct and complete, but also behavior-preserving w.r.t. the Petri net semantics.

5 Related Work

Results concerning the correctness of model transformations have been published so far mainly on formally showing the *syntactical correctness* [10].

To ensure the semantical correctness of model transformations, Varró et al. [11] use graph transformation to specify the dynamic behavior of systems and generate a transition system for each model. A model checker verifies certain dynamic consistency properties by checking the source and target models. In [1], a method is presented to verify semantical equivalence for particular model transformations. It is shown by finding bisimulations that a target model preserves the semantics of the source model w.r.t. a particular property. This technique does not prove the correctness of the model transformation rules in general.

In [2,3,4], we consider $S2A$ transformation, realizing a consistent mapping from *simulation* steps in a behavioral modeling language to *animation* steps in a more suitable domain-specific visualization. The animation specification A in [2,3,4] corresponds to an integrated simulation specification in this paper. However, there is no $I2T$ transformation considered in [2,3,4]. This paper generalizes and extends the results from [2,3,4] to the more general case of $S2T$ model transformations.

6 Conclusion and Ongoing Work

We have considered the semantical correctness and completeness of model transformations based on simulation specifications (typed graph transformation systems). The main results show under which conditions an $S2T$ model transformation is semantically correct and complete. The theory has been presented in the DPO-approach for typed graphs, but it can also be extended to typed attributed graphs, where injective graph morphisms are replaced by suitable classes M and M' of typed attributed graph morphisms for rules and NACs, respectively [8]. The results have been used to analyze an *S2T* transformation of a production system (a domain-specific visual model) to Petri nets. We also discuss the requirement of a model transformation *S2T* to be *behavior-preserving* in the sense that for each target model in VL_T the simulations via rules in P_T correspond to simulations via rules in the target semantics, given by $P_{\bar{T}}$ (e.g. the Petri net firing rules) and vice versa. Work is in progress to establish formal criteria for semantically correct and complete *S2T* model transformations to be also behavior-preserving w.r.t. a given target language semantics.

Future work is planned to analyze in more detail our *I2T* completeness condition, to automatize our approach (e.g. check the correctness and completeness conditions automatically by a tool) and to apply the approach to triple graph grammars [12], nowadays widely used for model transformation specification.

References

1. Narayanan, A., Karsai, G.: Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations. In: Proc. Workshop on Graph and Model Transformation (GraMoT 2006). EC-EASST, EASST, vol. 4 (2006)
2. Ermel, C.: Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation. PhD thesis, Technische Universität Berlin,fak, IV, Books on Demand, Norderstedt (2006)
3. Ermel, C., Ehrig, H., Ehrig, K.: Semantical Correctness of Simulation-to-Animation Model and Rule Transformation. In: Proc. Workshop on Graph and Model Transformation (GraMoT 2006). EC-EASST, vol. 4, EASST (2006)
4. Ermel, C., Ehrig, H.: Behavior-preserving simulation-to-animation model and rule transformation. In: Proc. Workshop on Graph Transformation for Verification and Concurrency (GT-VC 2007). ENTCS, vol. 213(1), pp. 55–74. Elsevier Science, Amsterdam (2008)
5. Ehrig, H., Ermel, C.: Semantical Correctness and Completeness of Model Transformations using Graph and Rule Transformation: Long Version. Technical Report 2008-13, TU Berlin (2008), `http://iv.tu-berlin.de/TechnBerichte/2008/2008-13.pdf`
6. de Lara, J., Vangheluwe, H.: Translating Model Simulators to Analysis Models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 77–92. Springer, Heidelberg (2008)
7. Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed Graph Transformation with Node Type Inheritance. Theoretical Computer Science 376(3), 139–163 (2007)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science. Springer, Heidelberg (2006)
9. Kreowski, H.-J.: A Comparison between Petri Nets and Graph Grammars. In: Noltemeier, H. (ed.) WG 1980. LNCS, vol. 100, pp. 1–19. Springer, Heidelberg (1981)

10. Ehrig, H., Ehrig, K.: Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation. In: Proc. Workshop on Graph and Model Transformation (GraMoT 2005). ENTCS, vol. 152. Elsevier, Amsterdam (2005)
11. Varró, D.: Automated formal verification of visual modeling languages by model checking. Software and System Modeling 3(2), 85–113 (2004)
12. 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)

High-Level Programs and Program Conditions*

Karl Azab and Annegret Habel

Carl v. Ossietzky Universität Oldenburg, Germany
{azab,habel}@informatik.uni-oldenburg.de

Abstract. High-level conditions are well-suited for expressing structural properties. They can describe the precondition and the postcondition for a high-level program, but they cannot describe the relationship between the input and the output of a program derivation. Therefore, we investigate program conditions, a generalized type of conditions expressing properties on program derivations. Program conditions look like nested rules with application conditions. We present a normal form result, a suitable graphical notation, and conditions under which a satisfying program can be constructed from a program condition. We define a sequential composition on program conditions and show that, for a suitable type of program conditions with a complete dependence relation we have that: Whenever the original programs satisfy the original program conditions, then the composed program satisfies the composed program condition.

1 Introduction

Constraints, also called graph conditions, in the sense of [1,2,3,4] are a visual and intuitive, yet precise formalism, well suited to describe structural properties of system states. With these concepts we can express requirements for a program by a pair $\langle \mathrm{pre}, \mathrm{post} \rangle$ of a pre- and a postcondition. A relationship between the input and the output graph cannot be expressed. In imperative programming languages, the relationship between the input and the output is given by the names of variables: in general, the value of a variable after program execution depends on its value before. In this paper, we introduce so-called program conditions which allows us to express the relationship between the input- and output object for high-level programs. The concept is illustrated in the category of graphs by a simple example of repairs of broken routers in a computer network.

Example 1 (router repair). Consider a simple network consisting of computers (⊡) and routers (⊲) as nodes. Network links between routers and computers are shown as undirected edges (in reality two directed edges in the opposite direction). Inoperable routers have an incoming edge from a failure (⊗) node. Broken routers can be put on a priority list for quicker repair, which is modelled by an edge from a priority (!) node. Figure 1 shows a graph G which is transformed

[*] This work is supported by the German Research Foundation (DFG) under grant no. HA 2936/2 (Development of Correct Graph Transformation Systems).

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 211–225, 2008.

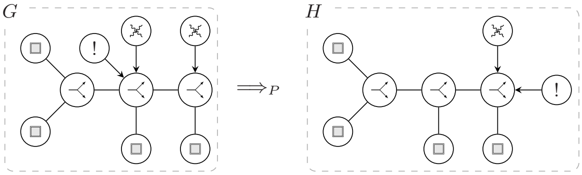


Fig. 1. Failing routers that were on the priority list should be repaired

into H by a graph program P. We will show how our program conditions can express temporal properties over such pairs, e.g. "Every router that were on the priority list has been repaired after the execution of P".

The paper is organized as follows: In Sect. 2, we review the definitions of conditions and rules. In Sect. 3, we introduce programs and program conditions, a generalized type of conditions expressing properties on program derivations. We also prove a normal form result for program conditions saying that, for every program condition, there is an equivalent program condition without pre- and postcondition. For this type of program conditions, a graphical notation is presented. In Sect. 4, for specific program conditions, a satisfying program is constructed. Moreover, for composed programs, a satisfying program condition is composed from the program conditions of the subprograms. The concepts are illustrated by examples in the category of graphs with the class of all injective graph morphisms. A conclusion including further work is given in Sect. 5.

2 Conditions and Rules

We use the framework of weak adhesive HLR categories [3,5] and introduce conditions and rules for high-level structures like Petri nets, (hyper)graphs, and algebraic specifications.

Assumption 1. We assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category with $\mathcal{M}$-initial object I satisfying the special pullback-decomposition property [6].

E.g. the category $\langle \text{Graphs}, \text{Inj} \rangle$ of graphs with class Inj of all injective morphisms is a weak adhesive HLR category satisfying the assumption.

(High-level) Conditions are defined as in [7,4]. Syntactically, the conditions may be seen a tree of morphisms equipped with certain logical symbols such as quantifiers and connectives.

Definition 1 (conditions). A *condition* over an object P is of the form true or $\exists(a, c)$, where $a\colon P \to C$ is a morphism and c is a condition over C. Moreover, Boolean formulas over conditions over P are conditions over P. $\exists a$ abbreviates $\exists(a, \text{true})$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$. Every morphism and every object *satisfies* true. A morphism $p\colon P \to G$ *satisfies* a condition $\exists(a, c)$ if there exists a morphism q in $\mathcal{M}$ such that $q \circ a = p$ and $q \models c$.

$$\begin{tikzcd}
\exists(\; P \arrow[rr, "a"] \arrow[dr, "p"'] & \arrow[d, phantom, "="] & C, \arrow[dl, hook', "q"] \arrow[r, phantom, "\models"] & c \;) \\
& G &
\end{tikzcd}$$

An object G *satisfies* a condition if all morphisms $p: P \to G$ in $\mathcal{M}$ satisfy the condition. The satisfaction of conditions over P by objects or morphisms with domain P is extended to Boolean formulas over conditions in the usual way. We write $p \models c$ $[G \models c]$ to denote that the morphism p [the object G] satisfies c.

Remark 1. In the context of objects, conditions are also called *constraints* [1] and, in the context of rules, conditions are also called *application conditions*. The definition generalizes those in [1,2,3].

Remark 2. We sometimes use a short notation for conditions: For a morphism $a: P \to C$ in a condition, we just depict C, if P can be unambiguously inferred, i.e. for conditions over the $\mathcal{M}$-initial object I. The graph condition $\forall(\emptyset \to$ ⊡$, \exists($⊡ $\hookrightarrow$ ⊡—⧀$))$ has the meaning "Every computer is connected to a router". In the short form, the condition is written as $\forall($⊡$, \exists($⊡—⧀$))$. A more complex example is the condition $\forall($⊡$, \exists($⊡—⧀$) \wedge \neg\exists($⧀—⊡—⧀$))$ which has the meaning "Every computer is connected to exactly one router".

(High-level) Rules are defined as in [3,4]. They are specified by a span of $\mathcal{M}$-morphisms $\langle L \hookleftarrow K \hookrightarrow R\rangle$ with a left and a right application condition. We consider the classical semantics based on the double-pushout construction [8,9] and restrict on $\mathcal{M}$-matching, i.e. matching morphisms for rules in $\mathcal{M}$.

Definition 2 (rules). A *rule* $\rho = \langle p, \mathrm{ac}_L, \mathrm{ac}_R\rangle$ consists of a plain rule $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$ with $K \hookrightarrow L$ and $K \hookrightarrow R$ in $\mathcal{M}$ and two application conditions ac_L and ac_R over L and R, respectively. L is called the left-hand side, R the right-hand side, and K the interface; ac_L and ac_R are the *left* and *right* application condition of p.

$$\begin{tikzcd}
\mathrm{ac}_L \arrow[r, phantom, "\models"] & L \arrow[d, hook, "m"'] \arrow[dr, phantom, "(1) \atop \mathrm{PO}"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, hook] \arrow[dr, phantom, "(2) \atop \mathrm{PO}"] & R \arrow[d, hook, "m^*"] & \mathrm{ac}_R \arrow[l, phantom, "\models"] \\
& G & D \arrow[l, hook'] \arrow[r, hook] & H &
\end{tikzcd}$$

Given a morphism $K \hookrightarrow D$ in $\mathcal{M}$, a *direct derivation* consists of two pushouts (1) and (2) such that $m \models \mathrm{ac}_L$ and $m^* \models \mathrm{ac}_R$. We write $G \Rightarrow_{p,m,m^*} H$ and say that $m: L \to G$ is the match, $m^*: R \to H$ is the comatch of p in H, and D is the *intermediate object*. We also write $G \Rightarrow_{\rho,m} H$ or $G \Rightarrow_{\rho} H$ to express that there is an m^* and there are m and m^*, respectively, such that $G \Rightarrow_{r,m,m^*} H$.

The well-known notions of parallel and sequential independence and the constructions of a parallel, concurrent, and amalgamated rule [10,9] partly have already been extended to rules with negative application conditions. In [11], a Local Church-Rosser, Parallelism, and Concurrency Theorem for rules with negative application conditions is given. There are generalized Local Church-Rosser, Parallelism, Concurrency and Amalgamation Theorems for rules with application conditions, see the long version of this paper [5].

3 Programs and Program Conditions

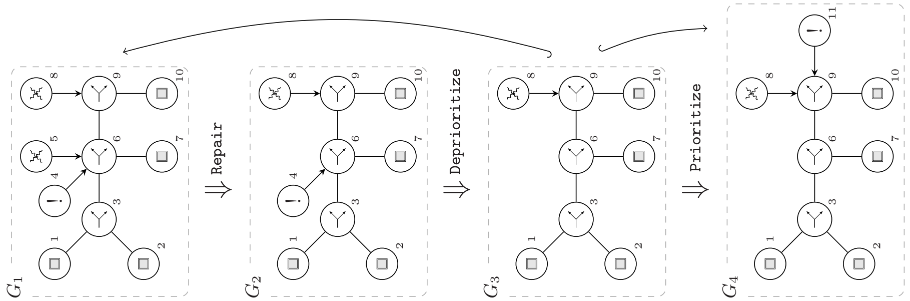
In this section, we recall the definition of programs and introduce program conditions, a generalized type of conditions expressing properties on program derivations. (High-level) Programs with input-output semantics are defined in [12,7]. The semantics presented here is based on the track morphism of [13].

Definition 3 (programs). Every rule p is a *program*. Every finite set $\mathcal{S}$ of programs is a program. If P and Q are programs, then $(P;Q)$, P^* and $P\downarrow$ are programs. The *semantics* of a program P is a set $[\![P]\!]$ of spans: For a rule p, $[\![p]\!] = \{\langle G \hookleftarrow D \hookrightarrow H\rangle \mid G \Rightarrow_p H$ with intermediate object $D\}$. For a finite set $\mathcal{S}$ of programs, $[\![\mathcal{S}]\!] = \cup_{P\in\mathcal{S}}[\![P]\!]$. For programs P and Q, $[\![(P;Q)]\!] = [\![P]\!] \circ [\![Q]\!]$, where the composition of spans is defined by $\langle G \hookleftarrow D_1 \hookrightarrow H\rangle \circ \langle H \hookleftarrow D_2 \hookrightarrow M\rangle = \langle G \hookleftarrow D \hookrightarrow M\rangle$, where D is the pullback object of $D_1 \hookrightarrow H \hookleftarrow D_2$, and (1) and (2) in the diagram below commutes.

$$\begin{tikzcd}
 & & D \arrow[dll, hook'] \arrow[dl] \arrow[dr] \arrow[drr, hook] \arrow[d, phantom, "(PB)"] & & \\
G & D_1 \arrow[l, hook'] \arrow[r, hook] \arrow[u, phantom, "(1)"] & H & D_2 \arrow[l, hook'] \arrow[r, hook] \arrow[u, phantom, "(2)"] & M
\end{tikzcd}$$

The composition is extended to sets of spans in the usual way. For a program P, $[\![P^*]\!]=[\![P]\!]^*$ and $[\![P\downarrow]\!]=\{\langle G \hookleftarrow D \hookrightarrow H\rangle\in[\![P]\!]^* | \neg\exists M.\langle H \hookleftarrow E \hookrightarrow M\rangle \in [\![P]\!]\}$.

Example 2. Consider the graph program `Repair`;`Deprioritize`;`Prioritize`, where `Repair` = ⟨≺—✕ ↩ ≺ ↪ ≺⟩, `Prioritize` = ⟨≺ ↩ ≺ ↪ ≺→!⟩, and `Deprioritize` = ⟨≺→! ↩ ≺ ↪ ≺⟩, and the derivation below.



The span $\langle G_1 \hookleftarrow G_3 \hookrightarrow G_4\rangle$ is in the semantics of that program. The trace of the nodes is visualized by indices: The nodes with the indices $1-3$ and $6-10$ are in the interface of the span, the nodes with the indices $4, 5$ are deleted, and the node with index 11 is inserted. The trace of the edges is given implicitly.

As in [14], we investigate spans and span morphisms. A *span* $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$ is a pair of $\mathcal{M}$-morphisms with common domain. A *(span) morphism* from $\mathrm{pc} = \langle L \hookleftarrow K \hookrightarrow R\rangle$ to $\mathrm{pc}' = \langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ is a triple of morphisms $\langle L \to L', K \to K', R \to R'\rangle$ such that (1) and (2) below commute. It is a *(span) $\mathcal{M}$-morphism* if morphisms in the triple are $\mathcal{M}$-morphisms. The class of such span $\mathcal{M}$-morphisms is denoted by $\mathcal{M}^3$.

$$\begin{tikzcd}
L \arrow[d, hook, "a"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(2)"] & R \arrow[d, hook, "a^*"] \\
L' & K' \arrow[l, hook'] \arrow[r, hook] & R'
\end{tikzcd}$$

The *composition* of morphisms is defined by the componentwise composition of the underlying morphisms. The *identity* on span morphisms is defined by the triple of identities on its components. In category theory, spans and $\mathcal{M}$-morphisms form a category.

(High-level) Conditions are well-suited for expressing structural properties. They can be used for describing the precondition and the postcondition for a high-level program, but they cannot be used for describing the relationship between the input and the output of a program. Therefore, we investigate program conditions, a generalized type of conditions expressing properties on program derivations.

Definition 4 (program conditions). A *program condition* (over p) is of the form $\langle p, \mathrm{pre}, \mathrm{post}\rangle$ and called *basic* if $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$ is a span of $\mathcal{M}$-morphisms and pre and post are conditions over L over R, respectively, or of the form true or $\exists(\bar{a}, \mathrm{pc})$ and called *nested* if $\bar{a}$ is a span morphism (with domain p) and pc is a program condition (over the codomain of $\bar{a}$). Moreover, Boolean formulas over program conditions yield program conditions. p abbreviates $\langle p, \mathrm{true}, \mathrm{true}\rangle$, $\exists\bar{a}$ abbreviates $\exists(\bar{a}, \mathrm{true})$, and $\forall(\bar{a}, \mathrm{pc})$ abbreviates $\neg\exists(\bar{a}, \neg\mathrm{pc})$. A morphism $\bar{m} = \langle m, d, m^*\rangle$ *satisfies* $\langle p, \mathrm{pre}, \mathrm{post}\rangle$ if (1) and (2) are pullbacks, $m \models \mathrm{pre}$, and $m^* \models \mathrm{post}$. The morphism $\bar{m}$ *satisfies* $\exists(\bar{a}, \mathrm{pc})$ if (1) and (2) are pullbacks and there is an $\mathcal{M}$-morphism $\bar{l}$ with $\bar{m} = \bar{l} \circ \bar{a}$ satisfying pc.

$$\begin{tikzcd}
\text{pre} \arrow[r] & L \arrow[d, hook, "m"'] \arrow[dl, phantom, "\models"] \arrow[dr, phantom, "(1)\ \text{PB}"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(2)\ \text{PB}"] & R \arrow[d, hook, "m^*"] \arrow[dr, phantom, "\models"] & \text{post} \arrow[l] \\
& G & D \arrow[l, hook'] \arrow[r, hook] & H &
\end{tikzcd}$$

$$\begin{tikzcd}
& L \arrow[dl, hook, "a"'] \arrow[dd, hook, "m"'] & & K \arrow[ll, hook'] \arrow[rr, hook] \arrow[dl, hook] \arrow[dd, hook] & & R \arrow[dl, hook] \arrow[dd, hook] \\
L' \arrow[dr, hook, "l"'] & & K' \arrow[ll, hook'] \arrow[rr, hook] \arrow[dr, hook] & & R' \arrow[dr, hook] & \\
& G & & D \arrow[ll, hook'] \arrow[rr, hook] & & H
\end{tikzcd}$$

A span $s = \langle G \hookleftarrow D \hookrightarrow H\rangle$ *satisfies* $\langle p, \mathrm{pre}, \mathrm{post}\rangle$ if for all $\mathcal{M}$-morphisms $m\colon L \to G$ satisfying pre, there is an $\mathcal{M}$-morphism $\bar{m} = \langle m, d, m^*\rangle$ satisfying $\langle p, \mathrm{pre}, \mathrm{post}\rangle$. The span s *satisfies* a program condition $\exists a$ $[\exists(a, c)]$ over p if all $\mathcal{M}$-morphisms from p to s satisfy the condition. The satisfaction of program conditions is extended to Boolean formulas over program conditions in the usual way. We write $\bar{m} \models \mathrm{pc}$ $[s \models \mathrm{pc}]$ to denote that the span morphism $\bar{m}$ [the span s] satisfies pc. Two program conditions pc_1 and pc_2 are *equivalent*, denoted by $\mathrm{pc}_1 \equiv \mathrm{pc}_2$, if $[\![\mathrm{pc}_1]\!] = [\![\mathrm{pc}_2]\!]$ where, for a program condition pc, $[\![\mathrm{pc}]\!]$ denotes the set $\{\bar{m} \mid \bar{m} \models \mathrm{pc}\}$. A program P *satisfies* the program condition pc, denoted by $P \models \mathrm{pc}$, if for all spans $s \in [\![P]\!]$ in the semantics of P, $s \models \mathrm{pc}$.

Remark 3. We sometimes use a short notation for program conditions: For a morphism $\bar{a}\colon p \to p'$ in a program condition, we just depict p', if p can be unambiguously inferred, i.e. for program conditions over the initial span $\langle I \hookleftarrow I \hookrightarrow I \rangle$.

Remark 4. Basic program conditions may be seen as ordinary rules with a left and a right application condition. Nested program conditions may be seen as nested rules in the sense of Rensink [15]. The main difference between double-pullback transitions and basic program condition satisfaction is informally that a span $\langle G \hookleftarrow D \hookrightarrow H \rangle$ satisfies the condition $\langle L \hookleftarrow K \hookrightarrow R \rangle$ if, for every instance of L in G, there corresponding instance of R in H. However, in the case of double-pullback transitions, $\langle G \hookleftarrow D \hookrightarrow H \rangle$ is in the semantics of $\langle L \hookleftarrow K \hookrightarrow R \rangle$ if, for a given instance of L in G, there is a corresponding instance of R in H. In a sense, program conditions are universal, while double-pullback transitions are existential conditions.

Fact 1. Program conditions generalize conditions in the sense of [1,2,3,4]: Every pair $\langle \mathrm{pre}, \mathrm{post} \rangle$ of conditions over the $\mathcal{M}$-initial object I constitutes a basic program condition $\langle i, \mathrm{pre}, \mathrm{post} \rangle$ with $i = \langle I \hookleftarrow I \hookrightarrow I \rangle$.

In the following, we present some examples for program conditions.

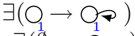
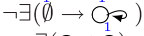
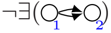
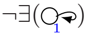
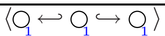
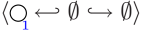
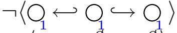
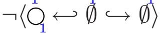
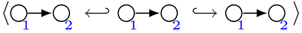
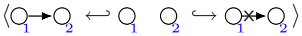
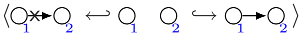
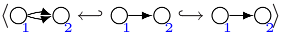
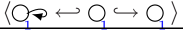
Example 3. The program condition with empty span $\langle \emptyset \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle$, precondition `simple`$= \neg\exists(\bigcirc_1 \rightleftarrows \bigcirc_2) \wedge \neg\exists(\bigcirc_1\!\circlearrowright)$, and postcondition `complete` $= \forall(\bigcirc_1\ \bigcirc_2, \exists(\bigcirc_1 \!\to\! \bigcirc_2))$ has the meaning "For simple graphs, the resulting graph has to be complete". (A graph is *simple* if it contains no parallel edges or loops and *complete* if every pair (v_1, v_2) of distinct nodes is connected by an edge from v_1 to v_2.) The program condition $\langle \bigcirc_1 \!\to\! \bigcirc_2 \hookleftarrow \bigcirc_1\ \bigcirc_2 \hookrightarrow \bigcirc_1 \!\leftarrow\! \bigcirc_2 \rangle$ has the meaning "Replace all edges with edges in opposite direction". Table 1 shows some pre- and postconditions and basic program conditions with their interpretation. The short notation $\bigcirc_1 \!\nrightarrow\! \bigcirc_2$ denotes the graph $\bigcirc_1\ \bigcirc_2$ together with the application condition $\neg\exists(\bigcirc_1\ \bigcirc_2 \to \bigcirc_1 \!\to\! \bigcirc_2)$, meaning "There does not exist an edge from the image of 1 to the image of 2.

The definition of program conditions combines pre- and postconditions with the concept of nesting. Pre- and post condition are well-known, well-understood, and natural; nesting is complicated and powerful. Every basic program condition can be transformed into an equivalent nested program condition *without pre- and postcondition*, i.e. in a program condition with subcondition of the form $\langle p, \mathrm{true}, \mathrm{true} \rangle$.

Theorem 1. *For every (basic) program condition, there is an equivalent (nested) program condition without pre- and postcondition.*

Proof. We define a transformation Nf from program conditions into an equivalent program conditions without pre- and postcondition. If we have a transformation

Table 1. Conditions and basic program conditions

condition	interpretation as pre/post condition
	All old/new nodes have a loop.
	No old/new node has a loop.
	Previously/afterwards, all pairs of distinct nodes are connected by at most one edge.
	Previously/afterwards, all nodes are loop-free.
program condition	**interpretation**
	All nodes are preserved.
	All nodes are deleted.
	At least one node is deleted.
	At least one node is preserved.
	All proper edges are preserved.
	All proper edges are deleted and no new ones are created.
	Whenever there were no proper edge, an edge is inserted.
	Proper parallel edges are deleted.
	All loops are deleted.

for basic program conditions, then the transformation can be extended to arbitrary program conditions by replacing basic subprogram conditions by equivalent ones without pre- and postcondition. For basic program conditions of the form $\langle p, \mathrm{pre}, \mathrm{post}\rangle$, we may use the fact that, by Definition 4, $\langle p, \mathrm{pre}, \mathrm{post}\rangle \equiv \langle p, \mathrm{pre}, \mathrm{true}\rangle \wedge \langle p, \mathrm{true}, \mathrm{post}\rangle$. For basic program conditions with true precondition, we may use the fact that $\langle p, \mathrm{true}, \mathrm{post}\rangle \equiv \langle p^{-1}, \mathrm{post}, \mathrm{true}\rangle^{-1}$, where, for a basic program condition $\mathrm{pc} = \langle p, \mathrm{pre}, \mathrm{post}\rangle$ with $p = \langle L \hookleftarrow K \hookrightarrow R\rangle$, pc^{-1} denotes the inverse basic program condition $\langle p^{-1}, \mathrm{post}, \mathrm{pre}\rangle$ with $p^{-1} = \langle R \hookleftarrow K \hookrightarrow L\rangle$. Without loss of generality, we can assume that pre and post are $\mathcal{M}$-conditions, i.e. for all subconditions of the form $\exists(a, c)$, the morphism a is in $\mathcal{M}$ (see [4]).

Construction. For basic program conditions of the form $\langle p, \mathrm{pre}, \mathrm{true}\rangle$, the transformation Nf is defined by induction on the structure of the condition pre:

$$\begin{aligned} \mathrm{Nf}(\langle p, \mathrm{true}, \mathrm{true}\rangle) &= \langle p, \mathrm{true}, \mathrm{true}\rangle \\ \mathrm{Nf}(\langle p, \exists(a, c), \mathrm{true}\rangle) &= \bigvee\nolimits_{\bar{a}\in A} \exists(\bar{a}, \mathrm{Nf}(\langle p', c, \mathrm{true}\rangle)) \end{aligned}$$

where $\bar{a} \in A$ ranges over all $\mathcal{M}$-morphisms $\bar{a} = \langle a, z, a^*\rangle$ such that (11) and (21) are pullbacks and $p' = \langle L' \leftarrow K' \rightarrow R'\rangle$ is the codomain of $\bar{a}$. For Boolean formulas over conditions, the transformation is straightforward: $\mathrm{Nf}(\langle p, \neg c, \mathrm{true}\rangle) = \neg(\mathrm{Nf}(\langle p, c, \mathrm{true}\rangle))$ and $\mathrm{Nf}(\langle p, c_1 \overset{\wedge}{\vee} c_2, \mathrm{true}\rangle) = \mathrm{Nf}(\langle p, c_1, \mathrm{true}\rangle \overset{\wedge}{\vee} \mathrm{Nf}(\langle p, c_2, \mathrm{true}\rangle)$.

We prove $\mathrm{Nf}(\langle p, \mathrm{pre}, \mathrm{true}\rangle) \equiv \langle p, \mathrm{pre}, \mathrm{true}\rangle$ by induction on the structure of the condition pre. **Basis.** Let pre = true. By definition of Nf, $\mathrm{Nf}(\langle p, \mathrm{true}, \mathrm{true}\rangle) =$

$\langle p, \mathrm{true}, \mathrm{true}\rangle$. **Induction.** Assume the hypothesis is true for conditions c, c_1, and c_2. Let pre $= \exists(a, c)$. For morphisms $\bar{m} = \langle m, d, m^*\rangle$, we have

$$\begin{array}{lll} & \bar{m} \models \langle p, \exists(a,c), \mathrm{true}\rangle & \\ \Leftrightarrow & (1),\ (2)\ \text{PB's} \wedge m \models \exists(a,c) & \text{(Definition 4)} \\ \Leftrightarrow & (1),\ (2)\ \text{PB's} \wedge \exists\, l \in \mathcal{M}.\ m = l \circ a \wedge l \models c & \text{(Def. of } \models\text{)} \\ \Leftrightarrow & \bar{m} \models p \wedge \exists \bar{a} \in A.\ \exists\, \bar{l} \in \mathcal{M}^3.\ \bar{m} = \bar{l} \circ \bar{a} \wedge \bar{l} \models \langle p', c, \mathrm{true}\rangle & (*) \\ \Leftrightarrow & \bar{m} \models p \wedge \exists \bar{a} \in A.\ \exists\, \bar{l} \in \mathcal{M}^3.\ \bar{m} = \bar{l} \circ \bar{a} \wedge \bar{l} \models \mathrm{Nf}(\langle p', c, \mathrm{true}\rangle) & \text{(Ind hyp)} \\ \Leftrightarrow & \bar{m} \models \bigvee_{\bar{a} \in A} \exists(\bar{a}, \mathrm{Nf}(\langle p', c, \mathrm{true}\rangle)) & \text{(Def. 4)} \\ \Leftrightarrow & \bar{m} \models \mathrm{Nf}(\langle p, \exists(a,c), \mathrm{true}\rangle) & \text{(Def. of Nf)} \end{array}$$

The equivalence (*) may be seen as follows: "$\Rightarrow$": Let (1) and (2) be pullbacks, $\mathrm{l} \in \mathcal{M}$, $m = l \circ a$, and $l \models c$. Then the decomposition of m induces a decomposition of the pullbacks (1) and (2) into pullbacks (11), (12) and (21), (22) as follows: Construct $L' \hookleftarrow K' \hookrightarrow D$ as a pullback of $L' \to G \hookleftarrow D$ (12). Then there is a unique morphism $K \to K'$ such that diagram (11) below commutes and $K \to K' \hookrightarrow D = K \hookrightarrow D$. Since $\mathcal{M}$ is closed under decompositions, the morphism $K \to K'$ is in $\mathcal{M}$. By the pullback-decomposition property, (11) is a pullback. Construct $K' \hookrightarrow R' \hookleftarrow R$ as a pushout of $K' \hookleftarrow K \hookrightarrow R$ (21). Then there is a unique morphism $R' \to H$ such that (22) commutes and $R \hookrightarrow R' \to H = R \hookrightarrow H$. By the special pullback-decomposition property, (21) and (22) are pullbacks. Since $\mathcal{M}$-morphisms are closed under decompositions, the morphism $R' \to H$ is in $\mathcal{M}$. Finally, let $p' = \langle L' \hookleftarrow K' \hookrightarrow R'\rangle$ and $\bar{a}$ and $\bar{l}$ be the triples of morphisms in the construction. Then $\bar{m} = \bar{l} \circ \bar{a}$. Now $l \models c$ implies $\bar{l} \models \langle p', c, \mathrm{true}\rangle$. "$\Leftarrow$": Let $\bar{m} \models p$, $\bar{m} = \bar{l} \circ \bar{a}$, $\bar{l} \models \langle p', c, \mathrm{true}\rangle$, Then the diagrams (1) and (2) are pullbacks, $m = l \circ a$, where m and l are the projections of $\bar{m}$ and $\bar{l}$ to the first component, and $l \models c$.

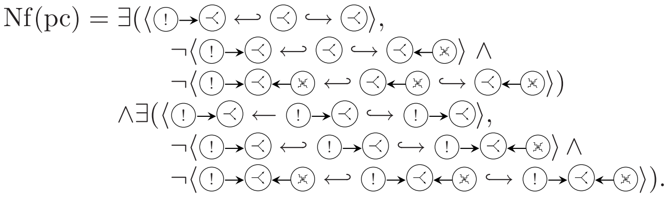
$$\begin{tikzcd} L \arrow[d, hook, "m"'] \arrow[r, phantom, ""] \arrow[dr, phantom, "(1)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(2)"] & R \arrow[d, hook] \\ G & D \arrow[l, hook'] \arrow[r, hook] & H \end{tikzcd}$$

$$\begin{tikzcd} L \arrow[d, hook, "a"'] \arrow[dd, bend right=60, "m"'] \arrow[dr, phantom, "(11)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(21)"] & R \arrow[d, hook] \\ L' \arrow[d, hook, "l"'] \arrow[dr, phantom, "(12)"] & K' \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(22)"] & R' \arrow[d, hook] \\ G & D \arrow[l, hook'] \arrow[r, hook] & H \end{tikzcd}$$

Let pre be a Boolean formula over conditions. By the definition of Nf and the inductive hypothesis, $\mathrm{Nf}(\langle p, \neg c, \mathrm{true}\rangle) = \neg\mathrm{Nf}(\langle p, c, \mathrm{true}\rangle) \equiv \neg\langle p, c, \mathrm{true}\rangle \equiv \langle p, \neg c, \mathrm{true}\rangle$ and $\mathrm{Nf}(\langle p, c_1 \overset{\wedge}{\vee} c_2, \mathrm{true}\rangle) = \mathrm{Nf}(\langle p, c_1, \mathrm{true}\rangle \overset{\wedge}{\vee} \mathrm{Nf}(\langle p, c_2, \mathrm{true}\rangle) \equiv \langle p, c_1, \mathrm{true}\rangle \overset{\wedge}{\vee} \langle p, c_2, \mathrm{true}\rangle \equiv \langle p, c_1 \overset{\wedge}{\vee} c_2, \mathrm{true}\rangle$. This completes the inductive proof.

Fact 2. The converse of Theorem 1 does not hold, e.g. see the program condition in Example 4.

Example 4. The property "Every router that were on the priority list has been repaired" can be expressed by the basic program condition pc with span ⟨≺⃝ $\hookleftarrow$ ≺⃝ $\hookrightarrow$ ≺⃝⟩, precondition ∃(!⃝→≺⃝), and postcondition ¬∃(≺⃝←✕⃝). The program condition can be transformed into a nested program condition Nf(pc), where



As illustrated by Example 4, nested program conditions can have a quite verbose form. Therefore, we introduce a shorter visual notation for program conditions.

Notation (graphical notation for program conditions). In the case of graphs, there is a nice short notation for program conditions, also used for representing rules in [16] and somewhat similar to the X and Y notations of [17]. The basic idea is that a graph span is represented as a single colored graph. The colored graph of the span $\langle L \hookleftarrow K \hookrightarrow R\rangle$ is structurally isomorphic to the pushout of $L \hookleftarrow K \hookrightarrow R$. Additionally, elements in the colored graphs are visually represented with dashed lines (deleted) if they are in $L - K$, solid lines (preserved) if in K, and with bold lines (created) if in $R - K$. Span morphisms are then visualized as morphisms on colored graphs, yielding a visualization of a graph program condition that is quite similar to the (standard) graph condition. A graph span with two deleted, one preserved, and two created elements, together with the program condition for "Every router that were on the priority list has been repaired" is shown in Fig. 2 using the visual notation.

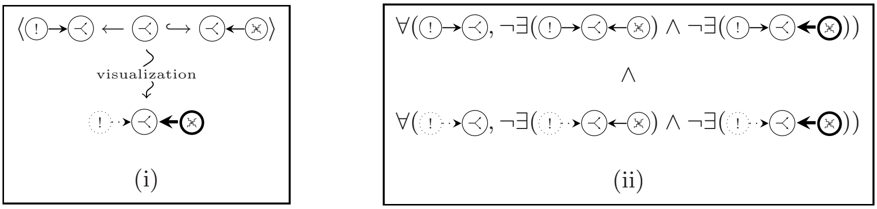


Fig. 2. Visual representation of (i) a relative graph and (ii) a graph program condition

4 Program Construction

We now investigate program construction. For a special kind of program conditions, a program can be constructed such that all spans in its semantics satisfy the program condition (and there exists a span). Based on sequential independence of direct derivations for rules with application conditions [5] and termination of rewrite systems [18], sequential self-independence and termination of basic program conditions are introduced.

Definition 5 (always applicable, self-independent, terminating). A basic program condition $\text{pc} = \langle p, \text{pre}, \text{post}\rangle$ is *always applicable* if, for every $\mathcal{M}$-morphism $m\colon L \hookrightarrow G$ satisfying pre, there exists a direct derivation $G \Rightarrow_{\text{pc},m,m^*}$

H such that m^* satisfies post. A basic program condition pc is *sequentially self-independent* if all derivation sequences $G \Rightarrow_{\mathrm{pc},m_1} H_1 \Rightarrow_{\mathrm{pc},m_2} M$ are sequentially independent and *terminating* if there is no infinite derivation sequence of the form $G_1 \Rightarrow_{\mathrm{pc}} G_2 \Rightarrow_{\mathrm{pc}} G_3 \ldots$.

Example 5. The program condition $\langle \bigcirc \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle$, meaning "Every node shall be deleted", is not always applicable; it is applicable if and only if the dangling condition [8] is satisfied. The program condition $\langle \bigcirc_1\!\to\!\bigcirc_2 \hookleftarrow \bigcirc_1\ \bigcirc_2 \hookrightarrow \bigcirc_1\!\leftarrow\!\bigcirc_2 \rangle$ is not sequentially self-independent, but the modified program condition $\langle \bigcirc_1\!\to\!\bigcirc_2 \hookleftarrow \bigcirc_1\ \bigcirc_2 \hookrightarrow \bigcirc_1\!\overset{*}{\leftarrow}\!\bigcirc_2 \rangle$ is. The (empty) program condition $\langle \emptyset \hookleftarrow \emptyset \hookrightarrow \emptyset \rangle$ is always applicable, sequentially self-independent, but not terminating. The program condition $\langle \bigcirc_1\!\not\to\!\bigcirc_2 \hookleftarrow \bigcirc_1\ \bigcirc_2 \hookrightarrow \bigcirc_1\!\to\!\bigcirc_2 \rangle$ is always applicable, sequentially self-independent, and terminating: the application condition "There does not exist an edge" guarantees termination because, for every finite graph with n nodes, at most 2^{n-1} proper edges can be inserted.

For basic program conditions, which are always applicable, sequentially self-independent, and terminating, a satisfying program can be constructed.

Theorem 2 (program construction). *For always applicable, sequentially self-independent, and terminating basic program conditions* pc*, the program* pc $\downarrow$ *satisfies* pc*.*

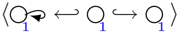
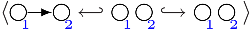
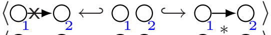
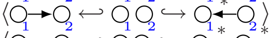
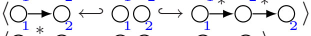
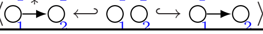
Proof. By the Parallelism Theorem for rules with application conditions [5]. Let $\mathrm{pc} = \langle p, \mathrm{pre}, \mathrm{post} \rangle$ be always applicable, sequentially self-independent, and terminating. Let $s = \langle G \hookleftarrow D \hookrightarrow H \rangle \in [\![\mathrm{pc} \downarrow\,]\!]$ and $m\colon L \hookrightarrow G$ be an $\mathcal{M}$-morphism such that $m \models \mathrm{pre}$. By the always applicability and termination, there is a derivation $G \Rightarrow_{\mathrm{pc},m} G_1 \Rightarrow_{\mathrm{pc}} \ldots \Rightarrow_{\mathrm{pc}} G_n = H$ such that there is no M with $\langle H \hookleftarrow E \hookrightarrow M \rangle \in [\![\mathrm{pc}]\!]$. By sequential self-independence and the Parallelism Theorem, there is a parallel derivation $G \Rightarrow H$ through the parallel rule $k$pc with $kp = \langle kL \hookleftarrow kK \hookrightarrow kR \rangle$ where, for $k \geq 0$, kA denotes the k-fold disjoint union of A. Then the diagrams (k1) and (k2) are pushouts. Since pushouts along $\mathcal{M}$-morphisms are pullbacks, (k1) and (k2) are also pullbacks.

$$\begin{tikzcd}
kL \arrow[d] \arrow[r, phantom, shift left=6, ""] & kK \arrow[l, hook'] \arrow[r, hook] \arrow[d] & kR \arrow[d] \\
G \arrow[u, phantom, shift right=0, ""] \arrow[ur, phantom, "\text{(k1)}"] & D \arrow[l, hook'] \arrow[r, hook] \arrow[ur, phantom, "\text{(k2)}"] & H
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d, hook] \arrow[dd, bend right=60, "m"'] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] & R \arrow[d, hook] \\
kL \arrow[d] \arrow[ur, phantom, "(1)"] & kK \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[ur, phantom, "(2)"] & kR \arrow[d] \\
G \arrow[ur, phantom, "\text{(k1)}"] & D \arrow[l, hook'] \arrow[r, hook] \arrow[ur, phantom, "\text{(k2)}"] & H
\end{tikzcd}$$

By the pullback decomposition property, (1) and (2) are pullbacks and, by the pullback composition property, the diagrams (1)+(k1) and (2)+(k2) are pullbacks. Thus, $m \models \mathrm{pc}$, Consequently $s \models \mathrm{pc}$. Therefore, $\mathrm{pc}\downarrow\, \models \mathrm{pc}$.

Example 6. In Table 2 we show a number of program conditions and their constructed programs that follow from Theorem 2.

Table 2. Program conditions and satisfying programs

program condition	program
`MakeLoopFree` = ⟨  ⟩	`MakeLoopFree`↓
`DeleteEdge` = ⟨  ⟩	`DeleteEdge`↓
`Complete` = ⟨  ⟩	`Complete`↓
`Converse`* = ⟨  ⟩	`Converse`*↓
`Subdivide`* = ⟨  ⟩	`Subdivide`*↓
`Relabel` = ⟨  ⟩	`Relabel`↓

Given satisfying program conditions for programs, a satisfying program condition for a set of programs can be constructed considering the disjunction of the original program conditions.

Fact 3. For programs P_1, P_2 and program conditions $\mathrm{pc}_1, \mathrm{pc}_2$

$$P_i \models \mathrm{pc}_i \ (i = 1, 2) \ \text{ implies } \ \{P_1, P_2\} \models \mathrm{pc}_1 \vee \mathrm{pc}_2.$$

Proof. Let $P_i \models \mathrm{pc}_i$ $(i = 1, 2)$. By the semantics of programs, the assumptions, and the definition of $\models$, $s \in [\![\{P_1, P_2\}]\!] = [\![P_1]\!] \cup [\![P_2]\!] \Leftrightarrow s \in [\![P_1]\!] \vee s \in [\![P_2]\!] \Leftrightarrow s \models \mathrm{pc}_1 \vee s \models \mathrm{pc}_2 \Leftrightarrow s \models \mathrm{pc}_1 \vee \mathrm{pc}_2$. Thus, for all $s \in [\![\{P_1, P_2\}]\!]$, $s \models \mathrm{pc}_1 \vee \mathrm{pc}_2$, i.e. $\{P_1, P_2\} \models \mathrm{pc}_1 \vee \mathrm{pc}_2$.

Given satisfying basic program conditions for programs, the concurrent program condition is a satisfying program condition for the sequence of programs.

Theorem 3 (composition of basic program conditions). *For programs P_1, P_2 and basic program conditions $\mathrm{pc}_1, \mathrm{pc}_2$ with dependence relation d for pc_1 and pc_2,*

$$P_i \models \mathrm{pc}_i \ (i = 1, 2) \ \text{ implies } \ P_1; P_2 \models \mathrm{pc}_1 *_d \mathrm{pc}_2,$$

where $\mathrm{pc}_1 *_d \mathrm{pc}_2$ is the d-concurrent program condition of pc_1 and pc_2 [5].

Proof. By the Concurrency Theorem for rules with application conditions [5]. Let $\mathrm{pc}_i = \langle p_i, \mathrm{pre}_i, \mathrm{post}_i \rangle$ with $p_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ $(i = 1, 2)$ be basic program conditions and $\mathrm{pc}_1 *_d \mathrm{pc}_2 = \langle p, \mathrm{pre}, \mathrm{post} \rangle$ with $p = \langle L^* \hookleftarrow K^* \hookrightarrow R^* \rangle$, $\mathrm{pre} = \mathrm{Shift}(k_1, \mathrm{pre}_1) \wedge \mathrm{L}(\mathrm{pc}_1^*, \mathrm{Shift}(k_2, \mathrm{pre}_2))$, and $\mathrm{post} = \mathrm{R}(\mathrm{pc}_2^*, \mathrm{Shift}(k_1^*, \mathrm{post}_1)) \wedge \mathrm{Shift}(k_2^*, \mathrm{post}_1)$ the d-concurrent program condition of pc_1 and pc_2, where Shift and L, and R are the shiftings of application conditions over morphisms and rules, from the right-hand side to the left-hand side, and from left-hand side to right-hand side, respectively [4]. Let $P_i \models \mathrm{pc}_i$ $(i = 1, 2)$. We will show that $P_1; P_2 \models \mathrm{pc}_1 *_d \mathrm{pc}_2$. Let $s = \langle G \hookleftarrow E \hookrightarrow M \rangle \in [\![P_1; P_2]\!]$ be an arbitrary span. By Definition 3, there are spans $\langle G \hookleftarrow E_1 \hookrightarrow H \rangle \in [\![P_1]\!]$ and $\langle H \hookleftarrow E_2 \hookrightarrow M \rangle \in [\![P_2]\!]$ such that $E_1 \hookleftarrow E \hookrightarrow E_2$ is the pullback of $E_1 \hookrightarrow H \hookleftarrow E_2$.

Let $l \colon L^* \hookrightarrow G$ be an $\mathcal{M}$-morphism satisfying pre. By the properties of Shift, the composed $\mathcal{M}$-morphism $m_1 = l \circ k_1$ satisfies pre_1. By assumption, we know that $P_1 \models \mathrm{pc}_1$, i.e., there are $\mathcal{M}$-morphisms $d_1 \colon K_1 \to E_1$ and $m_1^* \colon R_1 \to H$ such that the diagrams (1), (2) are pullbacks and m_1^* satisfies post_1. By the pullback decomposition property, the pullbacks (1) and (2) can be decomposed into pullbacks (11), (12), (21), (22). Now we have an $\mathcal{M}$-morphism $l'\colon H^* \to H$ such that $m_1^* = l' \circ k_1^*$. By the properties of Shift and L, the composed $\mathcal{M}$-morphism $m_2 = l' \circ k_2$ satisfies pre_2. By $P_2 \models \mathrm{pc}_2$, there are $\mathcal{M}$-morphisms $d_2\colon K_2 \to E_2$ and $m_2^*\colon R_2 \to M$ such that the diagrams (3), (4) are pullbacks and m_2^* satisfies post_2. By the pullback decomposition property, there is a decomposition of the pullbacks (3) and (4) into pullbacks (31), (32), (41), (42). Now we have an $\mathcal{M}$-morphism $l^*\colon H^* \to M$ such that $m_2^* = l^* \circ k_2^*$. Since the class of $\mathcal{M}$-morphisms is closed under decomposition, the morphism l^* is in M. Constructing E as the pullback object of $E_1 \hookrightarrow H \hookleftarrow E_2$, be the universal property of pushouts, there is a morphism $K \to E$ such that the diagrams (22') and (32') commute. By the (cube) pullback decomposition property, (22') and (32') are pullbacks. By the composition property of pullbacks, the diagrams (22')+(12) and (32')+(42) are pullbacks. Since $\mathcal{M}$-morphisms are closed under pullbacks, the morphisms $K \to E$ is in $\mathcal{M}$. By the properties of Shift and R, the $\mathcal{M}$-morphism $l^*\colon R^* \to M$ satisfies post. As a consequence, $l \models \mathrm{pc}_1 *_d \mathrm{pc}_2$. For every span $s \in [\![P_1; P_2]\!]$, $s \models \mathrm{pc}_1 *_d \mathrm{pc}_2$. Consequently, $P_1; P_2 \models \mathrm{pc}_1 *_d \mathrm{pc}_2$.

$$\begin{tikzcd}
 & & & S \arrow[dl, hook'] \arrow[dr, hook] & & & \\
\rhd L_1 \arrow[d, hook, "k_1"'] \arrow[d, dotted, bend right] \arrow[dddd, hook, bend right=60, "m_1"'] \arrow[dr, phantom, "(11)"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(21)"] & R_1 \lhd \arrow[dr, hook, "k_1^*"'] \arrow[dr, dotted, bend left] \arrow[rr, phantom, "(0)"] & & \rhd L_2 \arrow[dl, hook, "k_2"] \arrow[dl, dotted, bend right] \arrow[dr, phantom, "(31)"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(41)"] & R_2 \lhd \arrow[d, hook, "k_2^*"] \arrow[d, dotted, bend left] \\
\rhd L^* \arrow[dd, hook, "l"'] \arrow[ddr, phantom, "(12)"] & D_1 \arrow[l, hook'] \arrow[rr, hook] \arrow[dd, hook] \arrow[dr, phantom, "(22)"] & & H^* \arrow[lll, dotted, bend right] \arrow[rrr, dotted, bend left] \arrow[dd, hook, "l'"'] \arrow[ddr, phantom, "(32)"] & & D_2 \arrow[ll, hook'] \arrow[r, hook] \arrow[dd, hook] \arrow[ddr, phantom, "(42)"] & R^* \lhd \arrow[dd, hook, "l^*"] \\
 & \arrow[d, phantom, "(22')"] & K \arrow[ul, hook'] \arrow[urrr, hook] \arrow[dd, hook] & \arrow[d, phantom, "(32')"] & & & \\
G & E_1 \arrow[l, hook'] \arrow[rr, hook] & \arrow[d, phantom, "(52)"] & H & & E_2 \arrow[ll, hook'] \arrow[r, hook] & M \\
 & & E \arrow[ul, hook'] \arrow[urrr, hook] & & & &
\end{tikzcd}$$

Example 7. The program condition `Converse` $= \langle \underset{1}{\circ}\!\to\!\underset{2}{\circ} \hookleftarrow \underset{1}{\circ}\ \underset{2}{\circ} \hookrightarrow \underset{1}{\circ}\!\leftarrow\!\underset{2}{\circ} \rangle$is always applicable, but not sequentially self-independent; the program `Converse`$\downarrow$ is non-terminating. For avoiding sequentially self-dependence, the program condition could be modified into a sequentially self-independent program condition `Converse`$' = \langle \underset{1}{\circ}\!\to\!\underset{2}{\circ} \hookleftarrow \underset{1}{\circ}\ \underset{2}{\circ} \hookrightarrow \underset{1}{\circ}\!\overset{*}{\leftarrow}\!\underset{2}{\circ} \rangle$. For correcting the modification, a sequentially self-independent program condition `Relabel` $= \langle \underset{1}{\circ}\!\overset{*}{\to}\!\underset{2}{\circ} \hookleftarrow \underset{1}{\circ}\ \underset{2}{\circ} \hookrightarrow \underset{1}{\circ}\!\to\!\underset{2}{\circ} \rangle$ may be considered. By Theorem 3, the program `Converse`$'\downarrow$;`Relabel`$\downarrow$ satisfies the program condition `Converse`$' *_d$ `Relabel` $=$ `Converse` (where d is the "complete" dependence relation $\langle \underset{1}{\circ}\!\overset{*}{\to}\!\underset{2}{\circ} \hookleftarrow \underset{1}{\circ}\!\overset{*}{\to}\!\underset{2}{\circ} \hookrightarrow \underset{1}{\circ}\!\overset{*}{\to}\!\underset{2}{\circ} \rangle$).

Parallel self-independence and parallel independence is defined analogously to sequential self-independence and sequential independence of program conditions, see [5] for details. E.g., the program condition `Converse` is parallelly self-independent; a program `Converse` $\Downarrow$ based on a *parallel* composition of programs would satisfy `Converse`. Generally, it could be important to introduce a parallel composition of programs. and to show how a parallel composition can be expressed by the sequential composition. In this case, one could use the parallel composition as an abbreviation of a more complex sequential operation.

5 Conclusion

In this paper, we introduced the syntax and semantics of high-level program conditions. The syntax of program conditions is well-known: Basic program conditions syntactically may be seen as ordinary rules with left and right application conditions; nested program conditions may be seen as nested rules in the sense of Rensink [15] but with a categorical definition. Their semantic differences were discussed.

- **Normal form.** We showed a normal form result, relating basic program conditions with nested program conditions.
- **Program construction.** For special types of program conditions, we could create satisfying programs.
- **Sequential composition.** For basic program conditions, a composition is defined by constructing the concurrent program condition according to a dependence relation.

Further topics could be the following.

- **Expressiveness of program conditions.** As known by [4], graph conditions are expressively equivalent to first-order formulas on graphs. How powerful are graph program conditions?
- **Concurrency Theorem for nested rules.** It would be important to generalize the Concurrency Theorem to nested rules such that Theorem 3 can be formulated for program conditions instead of basic program conditions.
- **Comparison with first-order process logic.** In [19], the authors extend dynamic logic with the additional trace modalities *throughout* and *at least once*, which refer to all the states a program reaches and allow one to specify and verify invariants and safety constraints that have to be valid throughout the execution of a program.

References

1. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars – a constructive approach. In: SEGRAGRA 1995. ENTCS, vol. 2, pp. 95–104 (1995)
2. Koch, M., Mancini, L.V., Parisi-Presicce, F.: Graph-based specification of access control policies. Journal of Computer and System Sciences 71, 1–33 (2005)
3. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae 74(1), 135–166 (2006)

4. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. In: Mathematical Structures in Computer Science (to appear, 2008)
5. Azab, K., Habel, A.: High-level programs and program conditions (long version). Technical report, University of Oldenburg (2008)
6. Ehrig, H., Kreowski, H.J.: Pushout-properties: An analysis of gluing constructions for graphs. Mathematische Nachrichten 91, 135–149 (1979)
7. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer, Heidelberg (2006)
8. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Ng, E.W., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979)
9. 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 Transformation, pp. 163–245. World Scientific, Singapore (1997)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory of typed attributed graph transformation based on adhesive HLR-categories. Fundamenta Informaticae 74(1), 31–61 (2006)
11. Lambers, L., Ehrig, H., Prange, U., Orejas, F.: Parallelism and concurrency in adhesive high-level replacement systems with negative application conditions. In: Workshop on Applied and Computational Category Theory (ACCAT 2007). ENTCS. Elsevier, Amsterdam (to appear, 2008)
12. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 230–245. Springer, Heidelberg (2001)
13. 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)
14. Heckel, R., Ehrig, H., Wolter, U., Corradini, A.: Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. Applied Categorical Structures 9(1), 83–110 (2001)
15. Rensink, A.: Nested quantification in graph transformation rules. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 1–13. Springer, Heidelberg (2006)
16. Lambers, L.: A new version of GTXL: An exchange format for graph transformation systems. In: Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004). ENTCS, vol. 127, pp. 51–63 (2005)
17. Göttler, H.: Deriving productions from productions with an application to picasso's ceuvre. In: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 459–484. World Scientific, Singapore (1999)
18. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
19. Beckert, B., Schlager, S.: A sequent calculus for first-order dynamic logic with trace modalities. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 626–641. Springer, Heidelberg (2001)

A Shifting of Conditions

In the following, we review some improtant statements on the shifting of conditions over morphisms and rules investigated in detail in [4].

Fact 4 (shifting of conditions over morphisms [4]). There is a transformation Shift such that, for all conditions c over P, all morphisms $m \colon P \to P'$, and all $n \colon P' \hookrightarrow H$ in $\mathcal{M}$, $n \models \mathrm{Shift}(m, c) \Leftrightarrow n \circ m \models c$.

Right application conditions of a rule can be transformed into equivalent left application conditions.

Fact 5 (shifting of application conditions over rules [4]). There are transformations L and R such that, for every right application condition ac_R and every left application condition ac_L of a rule ρ and every direct derivation $G \Rightarrow_{\rho,m,m^*} H$, $m \models \mathrm{L}(\rho, \mathrm{ac}_R) \Leftrightarrow m^* \models \mathrm{ac}_R$ and $m \models \mathrm{ac}_L \Leftrightarrow m^* \models \mathrm{R}(\rho, \mathrm{ac}_L)$.

B Concurrent Rules

In the following we present the construction of a concurrent rule for rules with application conditions. It generalizes the well-known construction of concurrent rules for rules without application conditions [10] and makes use of shifting of application conditions over morphisms and rules (see Facts 4 and 5).

Definition 6 (d-concurrent rule). Let $\rho_i = \langle p_i, \mathrm{ac}_{Li}, \mathrm{ac}_{Ri} \rangle$ with $p_i = \langle L_i \hookleftarrow K_i \hookrightarrow R_i \rangle$ for $i = 1, 2$ be rules. A pair $d = \langle S \hookleftarrow R_1 \hookrightarrow L_2 \rangle$ is a *dependence relation* for rules ρ_1 and ρ_2, if (0) is the pushout of $R_2 \hookleftarrow S \hookrightarrow L_2$ and the pushout complements (1) and (2) of $K_1 \hookrightarrow R_1 \hookrightarrow H^*$ and $K_2 \hookrightarrow L_2 \hookrightarrow H^*$ exist. Given a dependence relation d for ρ_1 and ρ_2, the *d-concurrent rule* of ρ_1 and ρ_2 is the rule $\rho_1 *_d \rho_2 = \langle p, \mathrm{ac}_L, \mathrm{ac}_R \rangle$ with $p = \langle L^* \hookleftarrow K^* \hookrightarrow R^* \rangle$, where (3) and (4) are pushouts and (5) is a pullback, $\mathrm{ac}_L = \mathrm{Shift}(k_1, \mathrm{ac}_{L1}) \wedge \mathrm{L}(p_1^*, \mathrm{Shift}(k_2, \mathrm{ac}_{L2}))$, and $\mathrm{ac}_R = \mathrm{R}(p_2^*, \mathrm{Shift}(k_1^*, \mathrm{ac}_{R1})) \wedge \mathrm{Shift}(k_2^*, \mathrm{ac}_{R2})$, where $p_1^* = \langle L^* \hookleftarrow D_1 \hookrightarrow H^* \rangle$ and $p_2^* = \langle H^* \hookleftarrow D_2 \hookrightarrow R^* \rangle$ are the rules derived by p_1 and k_1 and p_2 and k_2, respectively.

$$\begin{tikzcd}
 & & & S \arrow[dl, hook] \arrow[dr, hook] & & & \\
\triangleright L_1 \arrow[d, hook, "k_1"] \arrow[d, dotted, bend right] \arrow[dr, phantom, "(3)"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(1)"] & R_1 \triangleleft \arrow[dr, hook, "k_1^*"'] & (0) & \triangleright L_2 \arrow[dl, hook, "k_2"] \arrow[dr, phantom, "(2)"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(4)"] & R_2 \triangleleft \arrow[d, hook, "k_2^*"'] \arrow[d, dotted, bend left] \\
\triangleright L^* & D_1 \arrow[l, hook'] \arrow[rr, hook] & & H^* \arrow[lll, dotted, bend right] \arrow[rrr, dotted, bend left] & & D_2 \arrow[ll, hook'] \arrow[r, hook] & R^* \triangleleft \\
 & & & \overset{(5)}{K^*} \arrow[lllu, hook] \arrow[llu, hook] \arrow[rru, hook] \arrow[rrru, hook] & & &
\end{tikzcd}$$

Fig. 3. Construction of a d-concurrent rule $\rho_1 *_d \rho_2$

Parallel and Sequential Independence for Borrowed Contexts[*]

Filippo Bonchi[1], Fabio Gadducci[1], and Tobias Heindel[2]

[1] Dipartimento di Informatica, Università di Pisa
[2] Institut für Informatik und Interaktive Systeme, Universität Duisburg-Essen

Abstract. Parallel and sequential independence are central concepts in the concurrency theory of the double pushout (DPO) approach to graph rewriting. However, so far those same notions were missing for DPO rewriting extended with borrowed contexts (DPOBC), a formalism used for equipping DPO derivations with labels and introduced for modeling open systems that interact with the environment.

In this work we propose the definition of parallel and sequential independence for DPOBC rewriting, and we prove that these novel notions allow generalizing the Church-Rosser and parallelism theorems holding for DPO rewriting. Most importantly, we show that the DPOBC version of these theorems still guarantees the local confluence and the parallel execution of pairs of independent DPOBC derivations.

1 Introduction

The dynamics of a computational device is often defined by a reduction system (RS): a set, representing the possible states of the device; and a relation among these states, representing the possible evolutions of the device. Most often, the states have a topological structure, and the relation is built from a finite set of reductions, in accordance to such a state structure. This construction could e.g. be performed by a matching mechanism, identifying which sub-component of a state is actually evolving, as well as denoting the successor state. This is the case for term and graph rewriting systems, where states are trees (graphs, respectively), and the reduction is built according to the DPO approach.

While RSs may convey the semantics with few rules, their drawback is poor compositionality: performing a computing step implicitly calls for a global choice of the subsystem where the reduction has to take place, making it difficult to model the dynamic behaviour of arbitrary standalone components. A solution is to express the behaviour of a computational device by a labelled transition system (LTS). Should the label associated to a component evolution faithfully express how that component might interact with the whole of the system, it would be possible to analyse on its own the behaviour of a single component, disregarding the larger systems it might occur in. Thus, a "well-behaved" LTS represents a fundamental step towards a compositional semantics of a computational device.

[*] Research partially supported by the EU FP6-IST IP 16004 SEnSOria.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 226–241, 2008.

Classically, graph transformation fits in the "global view" of the matching / "local application" of rules dichotomy that we mentioned before. The solution proposed by Ehrig and König [1] for DPO rewriting takes into account *graph with interfaces* for system representation. Now, a state is a morphism $J \to G$, where J represents the part of G that may interact with the environment. A reduction $J \to G \Rightarrow K \to H$ is built according to the DPO approach, but it is labelled with a *borrowed context* $J \to F \leftarrow K$, intuitively representing the amount of information required from the environment, before a reduction may take place.

Considered merely as a mechanism for label synthesis, the extension of DPO with borrowed contexts (DPOBC) is an instance of the so-called *relative pushouts* approach [2] for constructing an LTS from an RS, guaranteeing that the associated bisimilarity is automatically a congruence. This shows that the formalism has a sound theoretical basis, and further venues of work include deriving suitable inference rules (e.g. in the so-called SOS style) for building synthesized LTSs, as attempted in [3]. However, graphs with interfaces actually represent on their own an interesting formalism for modeling open systems, already used e.g for the graphical encoding of process calculi [4] or of spatial logics [5]. So, the construction via DPOBC of an LTS that has graphs with interfaces as both states and labels should just represent the starting point for further investigations. First of all, a thorough analysis of the labelling should be carried out, in order to understand which systems are bisimilar. So far, we are only aware of the correspondence with standard interleaving semantics that has been proved for the LTS obtained by the graphical encoding of CCS in [6]. Moreover, since DPOBC may represent a foundational mechanism for rewriting (when considering graphs with interfaces as main actors in system representation), it seems natural to ask under which conditions we may talk about concurrency for DPOBC derivations. As a start, suitable notions of parallel and sequential independence should be provided, as well as proving these definitions adequate, by establishing Church-Rosser and parallelism properties. As a further sanity check, it should be pivotal to ensure that in a LTS the labels of the square of reductions induced by two parallel independent derivations are somehow well-behaved.

The paper has the following structure. Section 2 recalls the basic definitions of DPOBC rewriting and illustrates our running example modeling a simple protocol for message broadcasting. Section 3 discusses the nature of labels on those LTSs induced by DPOBC, arguing, also on the basis of our example, which relationships should hold between the labels of a square of concurrent derivations. Section 4 introduces the notion of parallel and sequential independence for DPOBC rewriting, and it states the main results of our paper, i.e., the theorem concerning the (local) Church-Rosser and parallelism properties. Moreover, it shows that the labels of a square of derivations, as induced by the Church-Rosser property, enjoy what we call the *strict diamond property*, as introduced in Section 3. Section 5 further discusses about labels of parallel derivations, pointing out why they cannot be decomposed. Finally, Section 6 draws some conclusions, and it outlines venues and hypotheses for further work.

2 Background

In this section we recall the basics of the double pushout (DPO) approach to rewriting [7,8]. In particular, we consider its variant where rules and matches are monomorphisms; and, most importantly, we introduce it in full generality, assuming an arbitrary adhesive category [9] for system representation (yet aiming at a presentation suitable for those familiar with standard graph transformation).

Moreover, we also present the main concepts underlying DPO rewriting with *borrowed contexts* (DPOBC), as proposed in [1]. In the following, we fix a chosen adhesive category $\mathbb{C}$, where rules and matches live. For all notions concerning adhesive categories we refer the reader to [9].

2.1 Double Pushout Rewriting with Monic Matches

The idea of DPO rewriting dates back to the early Seventies, and has been shown to be a viable rewriting mechanism in any adhesive category [9]. In the paper we consider its restriction to monic matches, as presented in [8], since the original definition of DPO rewriting with borrowed contexts is based on this variant.

Definition 1 (Productions and rewriting). *A* DPO production p *is a span of monomorphisms* $p = L \xleftarrow{l} K \xrightarrow{r} R$, *and a* (monic) match of p *is a monomorphisms* $m\colon L \xrightarrow{m} G$.

A DPO direct derivation diagram *or* derivation *is a diagram consisting of two pushout squares as shown to the right. It witnesses that* p rewrites G into H at match m, *and we write* $G \models\langle p,m\rangle\Rightarrow H$.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "\mathrm{PO}"] & I \arrow[l, tail, "l"'] \arrow[r, tail, "r"] \arrow[d] \arrow[dr, phantom, "\mathrm{PO}"] & R \arrow[d] \\
G & C \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Note that, by virtue of the properties holding in adhesive categories, all arrows in the above right diagram are monic. Thus, in the remainder of the paper all mentioned arrows are assumed to be monic, unless stated otherwise.

Definition 2 (Parallel and sequential independence). *Let us consider two productions* $p_i = L_i \xleftarrow{l_i} I_i \xrightarrow{r_i} R_i$ *for* $i \in \{1, 2\}$, *and the two derivations* $H_1 \Leftarrow\langle m_1, p_1\rangle\Rightarrow G \models\langle p_2, m_2\rangle\Rightarrow H_2$ $(G \models\langle p_1, m_1\rangle\Rightarrow H_1 \models\langle p_2, m_2'\rangle\Rightarrow H)$ *shown below.*

$$\begin{tikzcd}
R_1 \arrow[d] & I_1 \arrow[l] \arrow[r] \arrow[d] & L_1 \arrow[d, "m_1"'] \arrow[rr, bend left, "s", to=2-5] & L_2 \arrow[dl, "m_2"] \arrow[to=2-2, bend left, "t"] & I_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \arrow[d] \\
H_1 & C_1 \arrow[l] \arrow[r] & G & & C_2 \arrow[ll] \arrow[r] & H_2
\end{tikzcd}$$

$$\left(\begin{tikzcd}
L_1 \arrow[d, "m_1"] & I_1 \arrow[l] \arrow[r] \arrow[d] & R_1 \arrow[d] \arrow[to=2-5, bend left, "s'"] & L_2 \arrow[dl, "m_2'"] \arrow[to=2-2, bend left, "t'"] & I_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \arrow[d] \\
G & C_1 \arrow[l] \arrow[r] & H_1 & & C_{1,2} \arrow[ll] \arrow[r] & H
\end{tikzcd}\right)$$

Then the two derivations are parallel independent (sequential independent) *if there exist morphisms* s *and* t *(*s' *and* t'*) such that they commute in the composed diagrams, as shown above.*

In the case of parallel independence, when thinking of the objects I_1, L_1, L_2, and I_2 as subobjects[1] of G, then the two derivations are independent if and only if both inclusions $L_1 \cap L_2 \subseteq I_1$ and $L_1 \cap L_2 \subseteq I_2$ hold, i.e. if $L_1 \cap L_2 = I_1 \cap I_2$; similarly for sequential independence: the illustrated derivations are independent if and only if $R_1 \cap L_2 = I_1 \cap I_2$.

As formulated precisely in Theorem 4, given two parallel independent derivations, it is possible to execute them in parallel, using a corresponding *parallel production*. As the latter concept is central to this paper, we define it in full detail. In its definition, $A_1 +_X A_2$ denotes pushout objects, e.g. $A_1 \xrightarrow{a'_1} (A_1 +_X A_2) \xleftarrow{a'_2} A_2$ would be the pushout of some span $A_1 \xleftarrow{a_1} X \xrightarrow{a_2} A_2$. Now, for any pair of arrows $g_1 \colon A_1 \to B$ and $g_2 \colon A_2 \to B$, $[g_1, g_2]_X \colon A_1 +_X A_2 \to B$ denotes the uniquely induced arrow. Further, for any span $B_1 \xleftarrow{f_1} A_1 \xleftarrow{a_1} X \xrightarrow{a_2} A_2 \xrightarrow{f_2} B_2$ with pushout $B_1 \xrightarrow{b'_1} (B_1 +_X B_2) \xleftarrow{b'_2} B_2$, $f_1 +_X f_2 \colon (A_1 +_X A_2) \to (B_1 +_X B_2)$ similarly denotes the uniquely induced arrow.

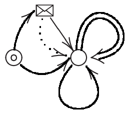
Definition 3 (Parallel productions). *Let* $p_1 = L_1 \xleftarrow{l_1} I_1 \xrightarrow{r_1} R_1$ *and* $p_2 = L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2$ *be productions, and let* $I_1 \xleftarrow{i_1} X \xrightarrow{i_2} I_2$ *be a span of morphisms. The* parallel production of p_1 and p_2 over $I_1 \leftarrow X \rightarrow I_2$ *is*

$$\begin{aligned} p_1 +_X p_2 &= (L_1 +_X L_2) \xleftarrow{l_1 +_X l_2} (I_1 +_X I_2) \xrightarrow{r_1 +_X r_2} (R_1 +_X R_2) \\ &= L_X \xleftarrow{l_X} I_X \xrightarrow{r_X} R_X \end{aligned}$$

$$= \begin{tikzcd} & & & R_1 \arrow[dll] & I_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[dlll, bend right] & L_1 \arrow[dlllll, bend right] \\ L_X & I_X \arrow[l, dashed, "l_X"'] \arrow[r, dashed, "r_X"] & R_X & & X \arrow[u, "i_1"] \arrow[d, "i_2"'] \arrow[ul] \arrow[ur] \arrow[dl] \arrow[dr] & \\ & & & R_2 \arrow[ull] & I_2 \arrow[l, "r_2"'] \arrow[r, "l_2"] \arrow[ulll, bend left] & L_2 \arrow[ulllll, bend left] \end{tikzcd}$$

Theorem 4 (Church-Rosser vs. parallelism). *For any pair of productions* $p_1 = L_1 \xleftarrow{l_1} I_1 \xrightarrow{r_1} R_1$ *and* $p_2 = L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2$*, the following statements are equivalent for certain* $m'_1 \colon L_1 \to H_2$ *and* $m'_2 \colon L_2 \to H_1$*.*

1. *There are parallel independent derivations* $H_1 \Leftarrow\!\langle m_1, p_1\rangle\!\dashv G \vDash\!\langle p_2, m_2\rangle\!\Rightarrow H_2$*.*
2. *There are sequential independent derivations* $G \vDash\!\langle p_1, m_1\rangle\!\Rightarrow H_1 \vDash\!\langle p_2, m'_2\rangle\!\Rightarrow H$*.*
3. *There are sequential independent derivations* $G \vDash\!\langle p_2, m_2\rangle\!\Rightarrow H_2 \vDash\!\langle p_1, m'_1\rangle\!\Rightarrow H$*.*
4. *There is a derivation* $G \vDash\!\langle (p_1 +_X p_2), [m_1, m_2]_X\rangle\!\Rightarrow H$ *with a parallel production* $p_1 +_X p_2$ *over the pullback* $I_1 \leftarrow X \rightarrow I_2$ *of the cospan* $I_1 \xrightarrow{m_1 \circ l_1} G \xleftarrow{m_2 \circ l_2} I_2$*.*



Example 5. Consider the category $\mathbf{Graph} \downarrow T$ of typed graphs over the graph T shown to the right. Circled nodes ⊚ represent users that want to send messages ✉ via a network where a network consists of nodes ○ and two different kinds of edges: a double arrow ⇒ represents a channel of unbounded capacity, while a single arrow → models a channel of capacity one.

[1] A suboject L of G is a monomorphism from L to G, and the intersection of two subobjets is achieved by taking pullbacks.

An edge from a message (user) to a node represent the fact that the message (user) is located at that node. The dotted arrow from a message to a node means that the message is addressed to that node, while an edge from a user to a message represents the fact that the user is the author of that message.

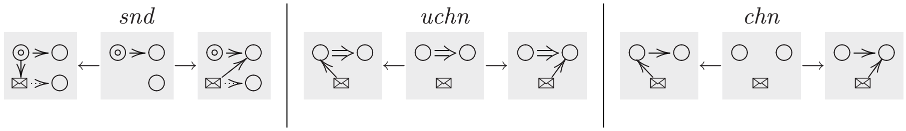
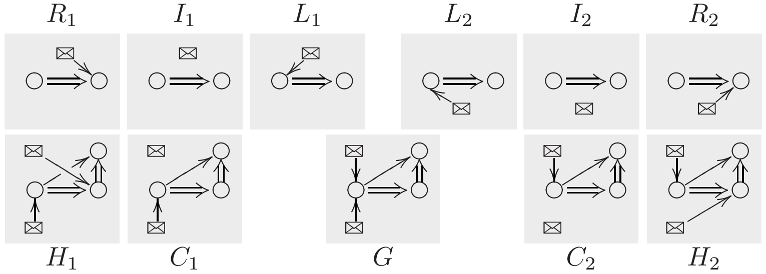
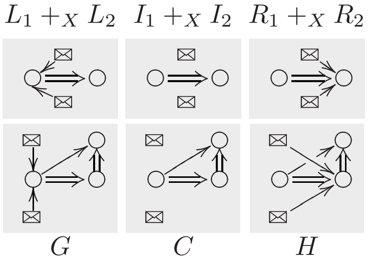


Fig. 1. Rewriting Rules

Consider the rule *snd* of the rewriting system that is shown in Fig. 1: if a user would like to deliver a message to a certain destination (singled out by the tip of the dotted arrow) while being located at another node, then the user can *send* the message by posting it via the network at the node he or she is located at. Then the message is *channeled* through the network, as modeled by the rules *chn* and *uchn*. When both edges of a message point to the same node, then the message has arrived at the destination.



As an example of parallel independent derivations, consider the graph G depicted above: it represents a small network with three nodes, two unbounded channels, one bounded channel and two messages on the same node. Both derivations use the production *uchn*, but in the first instance, on the left, the matching morphism $m_1 \colon L_1 \to G$ maps the message of L_1 into the upper one of G, while in the second instance, on the right, $m_2 \colon L_2 \to G$ maps the message into the lower one. Thus the first derivation moves the upper message, while the second moves the lower one. To see that the two derivations are parallel independent, consider the morphism $s \colon L_1 \to C_2$ that maps the message of L_1 into the upper message of C_2, and the morphism $t \colon L_2 \to C_1$ that maps the message of L_2 into the lower message of C_1.



We can also build a parallel rule by taking as common sub-object of I_1 and I_2 the graph $X = \bigcirc\!\Rightarrow\!\bigcirc$. The resulting production, namely $uchn +_X uchn$, is shown in the upper row to the right: in one single step, two messages can cross a channel of unbound capacity. The graph G can also be rewritten using this rule, which moves the two messages simultaneously.

A channel of capacity one cannot transmit two messages in one step, though.

2.2 Borrowed Contexts

The DPO rewriting formalism models the dynamics of a (closed) system in isolation, i.e., without considering its interactions with the environment. As introduced in [1], DPOBC is an extension of DPO such that, instead of manipulating single objects G, arrows $J \to G$ are considered, where the domain J represents the interface of the (open) system G.

Whereas in DPO derivations the left-hand side L of a production must occur completely in G, in a DPOBC derivation it is possible that only part of the object L might occur in G; the missing part of the left hand side can be supplied by the environment at J, thus completing the partial match of L to a total one. That part of the left hand side of the rule, that was provided by the environment to allow a total match of L, is called the *borrowed context*.

Definition 6 (DPOBC derivations, transformations and transitions) *Given a production $p = L \xleftarrow{l} I \xrightarrow{r} R$, a* DPO direct derivation diagram with borrowed context*, in short* BC derivation*, is a diagram of as shown to the right where the marked squares are either pushouts* (PO) *or pullbacks* (PB)*. If there exists a diagram of this form, then p* transforms $J \to G$ via the partial match $G \xleftarrow{a} D \xrightarrow{b} L$ to $K \to H$*, written*

$$\begin{tikzcd}
D \arrow[r, "b"] \arrow[d, "a"'] \arrow[dr, phantom, "\ulcorner\mathrm{PO}"] & L \arrow[d] \arrow[dr, phantom, "\mathrm{PO}\urcorner"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "\ulcorner\mathrm{PO}"] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[u] \arrow[r] \arrow[ur, phantom, "\llcorner\mathrm{PO}"] & F \arrow[u] \arrow[ur, phantom, "\ulcorner\mathrm{PB}"] & K \arrow[l] \arrow[u] \arrow[ur, dotted]
\end{tikzcd}$$

$$J \to G \models\langle p, a\text{-}D\text{-}b\rangle\Rightarrow K \to H.$$

If instead we want to focus on the interaction with the environment we say that $J \to G$ makes a transition *with borrowed context $J \to F \leftarrow K$ and becomes $K \to H$, written*

$$J \to G \xrightarrow{J \to F \leftarrow K} K \to H.$$

The roles of the squares in the diagram above can be thought of as follows: the upper left-hand pushout square ${}^{D}_{G}{\downarrow}\overset{\to}{\underset{\to}{\ulcorner}}{\downarrow}{}^{L}_{G^+}$ glues the left-hand side L and the object G together using the partial match $G \xleftarrow{a} D \xrightarrow{b} L$, which maps (part of) the left-hand side L onto G. The resulting pushout object G^+ allows for a total match of L and can be rewritten as in the standard DPO approach, which produces the two pushout squares ${}^{L}_{G^+}{\downarrow}\overset{\leftarrow}{\underset{\leftarrow}{\urcorner}}{\downarrow}{}^{I}_{C}$ and ${}^{I}_{C}{\downarrow}\overset{\to}{\underset{\to}{\ulcorner}}{\downarrow}{}^{R}_{H}$ in the upper

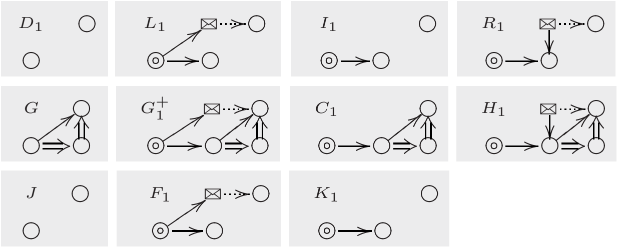


Fig. 2. A borrowed context derivation

row. The left-most bottom pushout $^{G}_{J}\!\uparrow\!\underset{\rightarrow}{\overset{\rightarrow}{\lrcorner}}\!\uparrow\!^{G+}_{F}$ glues the additional resources of the borrowed context $J \rightarrow F$, which are used to obtain the total match L into G^+, to the object G using the interface $J \rightarrow G$. Finally, the interface for the resulting object H and the second part of the label $J \rightarrow F \leftarrow K$ is obtained by intersecting the borrowed context F and the object C, i.e. by taking the pullback $^{G+}_{F}\!\uparrow\!\underset{\leftarrow}{\overset{\leftarrow}{\ulcorner}}\!\uparrow\!^{C}_{K}$. At the end, the new interface K includes what is preserved of the old interface and borrowed resources; finally, in the resulting $K \rightarrow H$ the object H additionaly contains everything produced by the "internal" DPO derivation.

Example 7. An example of a DPOBC derivation using the production *snd* can be found in Fig. 2. Here and in the remainder of the paper we do not explicitly describe the morphisms between graphs: we always assume that morphisms preserve the position of nodes and edges in the graphs. Moreover, we often denote an object in a specific derivation with the same (possibly indexed) identifier used in Def. 6. For example, the morphism $J \rightarrow G$ in Fig. 2 maps the leftmost and the topmost node of J into the leftmost and the topmost node of G, respectively.

The graph G represents a small network. Those nodes of this net occurring in the interface (the leftmost and the topmost one) are called *open*, since they are open to access from the environment, while each node that does not belong to the interface (e.g., the bottom right one in G) is called *closed*, as it has not been disclosed to the environment (yet).

Observe that the graph G does not contain any messages ⊠ and hence cannot perform any DPO derivation on its own. However, if there are users in the environment that want to send messages between the open nodes, messages are posted at the open nodes. This is what happens in the derivation of Fig. 2: a user wants to deliver one message to the upper open node and hence posts it at the lower open node. Note that (ordinary) users can only post at open nodes, which corresponds to the fact that every (ordinary) Internet user can only access the Internet via a provider.

In Section 4 we show how sequential and parallel independence can be lifted to the realm of DPOBC rewriting. In the next section we first present some fundamental observations about the nature of labels obtained via borrowed contexts.

3 A Discussion on Concurrency and Labels

In the standard DPO approach, two derivations are parallel independent if each one does not consume the resources needed by the other one. In the case of DPOBC derivations, due to the possibility to borrow resources from the environment, the situation is more complex. The notion of independence that we are looking for should guarantee that a pair of branching transitions may be executed simultaneously and can be joined to complete a Church-Rosser square, and also that the labels of the derivations are "well-related", which means that no superfluous interaction with the environment takes place.

In *transition systems with independence* [10], a formalism based on LTSs equipped with an independence relation among transitions, the labels represent events: in parallel and sequential independent transitions, the labels are related as shown to the right.

$$\begin{tikzcd}
& P \arrow[dl, Rightarrow, "a"'] \arrow[dr, Rightarrow, "b"] & \\
P_1 \arrow[dr, Rightarrow, "b"'] & & P_2 \arrow[dl, Rightarrow, "a"] \\
& Q &
\end{tikzcd}$$

In such a Church-Rosser square, the two transitions $P \Rightarrow P_1$ and $P_2 \Rightarrow Q$ are the same event, namely a, and similarly $P \Rightarrow P_2$ and $P_1 \Rightarrow Q$ are the event b. To get an analogy to DPOBC, we must think of events as productions. However this analogy is very rough, as the BC labels really are contexts, and, in general, the productions themselves are not part of the labels any more. Hence we cannot expect that the labels on parallel sides of the Church-Rosser square are actually identical. Indeed, the source and target state of a DPOBC derivation usually have different interfaces, and hence all labels of transitions originating from the former are different from those departing from the latter.

However, recall that (borrowed) contexts can be composed via pushouts (since they are arrows of a bicategory of cospans [11]). Hence one would expect that the composition of the two composable contexts on the left and on the right yields the same result, namely the diagonal: in other terms, a DPOBC Church-Rosser square should satisfy what we call the *weak diamond* property.

Definition 8 (Weak diamond property) *Given four transitions as shown in the commutative diagram to the right, then they satisfy the* weak diamond property *if there is a derivation*

$$J \to G \xrightarrow{J \to F \leftarrow K} K \to H$$

using a parallel rule and additionally the leftmost and rightmost square are pushouts.

$$\begin{tikzcd}
& & & G \arrow[dddlll, Rightarrow] \arrow[dddrrr, Rightarrow] & & & \\
& & & J \arrow[u] \arrow[dl] \arrow[dr] & & & \\
& & F_1 \arrow[dr] & & F_2 \arrow[dl] & & \\
H_1 \arrow[dddrrr, Rightarrow] & K_1 \arrow[l] \arrow[ur] \arrow[dr] & \mathrm{PO} & F & \mathrm{PO} & K_2 \arrow[r] \arrow[ul] \arrow[dl] & H_2 \arrow[dddlll, Rightarrow] \\
& & F_{1,2} \arrow[ur] & & F_{2,1} \arrow[ul] & & \\
& & & K \arrow[ul] \arrow[ur] \arrow[d] & & & \\
& & & H & & &
\end{tikzcd}$$

The latter requirement means that the composition of the labels of the two transitions on the left and the two transitions on the right are equal to $J \to F \leftarrow K$, which is the label of the parallel derivation along the diagonal of the square.

However, the weak diamond property does not ensure that the labels of the joining transitions and of the parallel one do not borrow "extra"-resources. As an example consider the pair of branching derivations in Fig. 3, witnessing transitions $J \to G \xrightarrow{J \to F_1 \leftarrow K_1} K_1 \to H_1$ and $J \to G \xrightarrow{J \to F_2 \leftarrow K_2} K_2 \to H_2$, where the former relays the upper message and the latter the lower one.

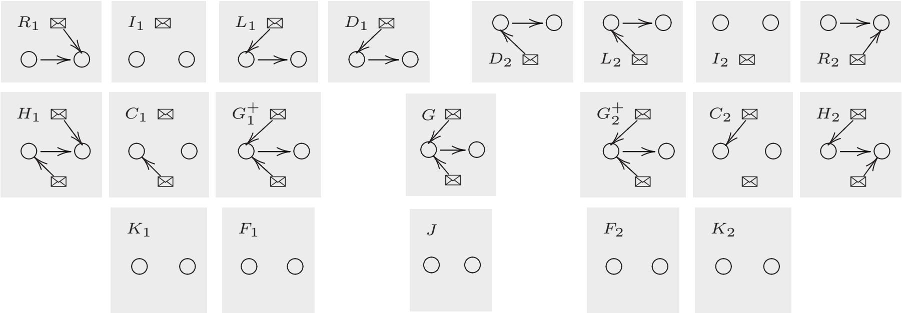
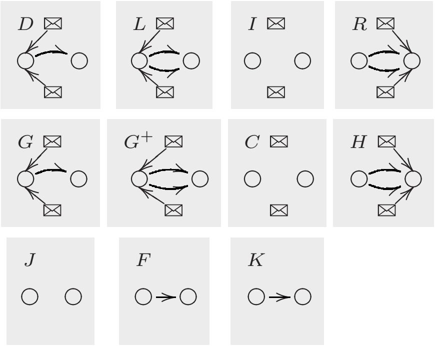


Fig. 3. Two non-parallel independent derivations

Note that both transitions use the same channel of capacity one, and this is also the reason why they turn out not to be parallel independent. In spite of this conflict we can construct the parallel derivation shown to the right, which borrows an extra channel from the environment. As shown in Fig. 4, the two derivations of Fig. 3 can be completed to a square satisfying the weak diamond property.



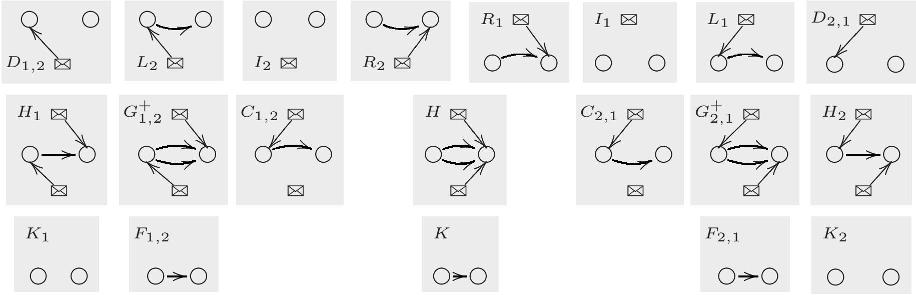


Fig. 4. The joining derivations of the weak diamond

Note that the label $K_2 \to F_{2,1} \leftarrow K$ in Fig. 4 is "bigger" than the corresponding $J \to F_1 \leftarrow K_1$ in Fig. 3 (and similarly for the other opposite labels of the square): the former consumes more environmental resources than the latter.

In order to guarantee that the joining derivations, as well as the parallel one, do not consume more resources than the two branching derivations, we have to require the following conditions.

Definition 9 (Strong diamond property). *The transitions in Def. 8 satisfy the* strong diamond property *if, additionally, the label $F_1 \to F \leftarrow F_2$ consists of a pair of jointly epic morphisms.*

Intuitively this means that all the resources that are borrowed by the parallel transition $J \to G \xrightarrow{J \to F \leftarrow K} K \to H$ have already been borrowed by one of the branching transitions. Since the composition of the contexts on the left (right) derivations must be equal to the parallel one, the joining derivations do not consume more than the branching ones.

However, also the strong diamond property is in some sense too weak. Indeed, it could be the case that the parallel derivation borrows less resources than the branching ones. In order to guarantee that the parallel transition consumes exactly the same resources of its two components, we have to require that the label F of the parallel derivation is the pushout of the two branching ones along the common interface, i.e., $F = F_1 +_J F_2$.

Definition 10 (Strict diamond property). *The transitions in Def. 8 satisfy the* strict diamond property *if, additionally, the upmost square of the diamond is a pushout.*

Note that the strict diamond property implies the strong one.

4 Parallel and Sequential Independence

In this section we introduce parallel and sequential independence for DPOBC rewriting, and we prove that these notions guarantee the satisfaction of the Church-Rosser property and the existence of a parallel transformation respecting the strict diamond property. In the next section we show that the vicevers is not always true, that is, a parallel transformation is not always decomposable into derivations satisfying the strict diamond property.

Definition 11 (Parallel independent derivations). *Let $J \to G$ be an object with interface, and moreover let $J \to G \vDash\langle p_1, a_1\text{-}D_1\text{-}b_1\rangle\Rightarrow K_1 \to H_1$ and $J \to G \vDash\langle p_2, a_2\text{-}D_2\text{-}b_2\rangle\Rightarrow K_2 \to H_2$ be two* DPOBC *derivations. These are* parallel independent *if for every pair of witnessing derivation diagrams as shown below there exist morphisms $i_{1,2} \colon D_1 \to C_2$ and $i_{2,1} \colon D_2 \to C_1$ which satisfy the equations $c_1 \circ i_{2,1} = g_1 \circ a_2$ and $c_2 \circ i_{1,2} = g_2 \circ a_1$.*

$$\begin{tikzcd}
R_1 \arrow[d] & I_1 \arrow[l] \arrow[r] \arrow[d] & L_1 \arrow[d] & D_1 \arrow[l, "b_1"'] \arrow[rd, "a_1"] \arrow[rrrrd, bend left=10, "i_{1,2}"] & & D_2 \arrow[r, "b_2"] \arrow[ld, "a_2"'] \arrow[llllld, bend right=10, "i_{2,1}"'] & L_2 \arrow[d] & I_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \arrow[d] \\
H_1 & C_1 \arrow[l, "h_1"'] \arrow[r, "c_1"] & G_1^+ & & G \arrow[ll, "g_1"'] \arrow[rr, "g_2"] & & G_2^+ & C_2 \arrow[l, "c_2"'] \arrow[r, "h_2"] & H_2 \\
& K_1 \arrow[u] \arrow[r] \arrow[lu, dotted] & F_1 \arrow[u] & & J \arrow[u] \arrow[ll] \arrow[rr] & & F_2 \arrow[u] & K_2 \arrow[u] \arrow[l] \arrow[ru, dotted] &
\end{tikzcd}$$

Intuitively, the existence of $i_{1,2}\colon D_1 \to C_2$ ensures that D_1 (the part of G that is needed to perform the first derivation) is left untouched by the second derivation. Thm. 16 shows that this condition is sufficient to guarantee the strict diamond property, but here we want to point out that it is also necessary for the strong diamond property. Indeed, if the first derivation would consume part of what is consumed by the second, then the parallel production could not be applied without borrowing more resources from the environment.

The above definition specializes to the standard DPO notion of parallel independence, since then b_1 and b_2 (and thus g_1 and g_2) would be identities.

Example 12. Consider the derivation in Fig.2. It is parallel independent with itself. Given another one copy of the derivation where all the graphs are labeled with D_2, L_2, $I_2 \ldots$ one can easily construct the morphism $i_{2,1}\colon D_2 \to C_1$ mapping the leftmost and the topmost node of D_2 (recall that D_2 is the same of D_1) onto the leftmost and the topmost node of C_1, respectively. Analogously for $i_{1,2}\colon D_1 \to C_2$. Intuitively, these derivations are parallel independent because they just add some structure to the nets, i.e., two users from the environment put two messages in the same node, and clearly this could happen concurrently. The derivations in Fig. 3 instead are not parallel independent, since the channel in D_1 does not occur in C_2.

Definition 13 (Sequential independent derivations). *Let $J \to G$ be an object with interface, and moreover let $J \to G \models\langle p_1,a_1\text{-}D_1\text{-}b_1\rangle\Rightarrow K_1 \to H_1$ and $K_1 \to H_1 \models\langle p_2,a_2\text{-}D_2\text{-}b_2\rangle\Rightarrow K \to H$ be two* DPOBC *derivations. These are* sequential independent *if for any witnessing pair of derivation diagrams as shown below, there exist morphisms $u\colon D_2 \to C_1$, $v\colon D_2 \to G$ and $w\colon R_1 \to C_{1,2}$ which satisfy the equations $c_1 \circ u = g_1 \circ v$, $c_2 \circ w = g_2 \circ o_1$ and $a_2 = x \circ u$.*

$$\begin{tikzcd}
D_1 \arrow[r, "b_1"] \arrow[d, "a_1"] & L_1 \arrow[d] & I_1 \arrow[l] \arrow[r] \arrow[d] & R_1 \arrow[d, "o_1"] \arrow[rrr, bend left, "w"] & D_2 \arrow[r, "b_2"] \arrow[dl, "a_2"] \arrow[dll, "u"'] \arrow[llll, bend right, "v"'] & L_2 \arrow[d] & I_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \arrow[d] \\
G \arrow[r, "g_1"] & G_1^+ & C_1 \arrow[l, "c_1"'] \arrow[r, "x"] & H_1 \arrow[rr, "g_2"] & & G_{1,2}^+ & C_{1,2} \arrow[l, "c_2"'] \arrow[r] & H \\
J \arrow[u] \arrow[r] & F_1 \arrow[u] & K_1 \arrow[l] \arrow[u] \arrow[ur, dotted] \arrow[rrr] & & & F_{1,2} \arrow[u] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}$$

Intuitively, the existence of v requires that what is needed by the second derivation (i.e., D_2) occurs in G and thus it is not added by the borrowed context. The existence of u requires that D_2 is not produced by the first derivation, while the existence of w guarantees that the second rule does not consume anything that the first one has produced.

Note that the definition above subsumes the notion of sequential independence in DPO rewriting, since for DPO derivations the object G_1^+ coincides with G, and thus the the existence of u implies the existence of v.

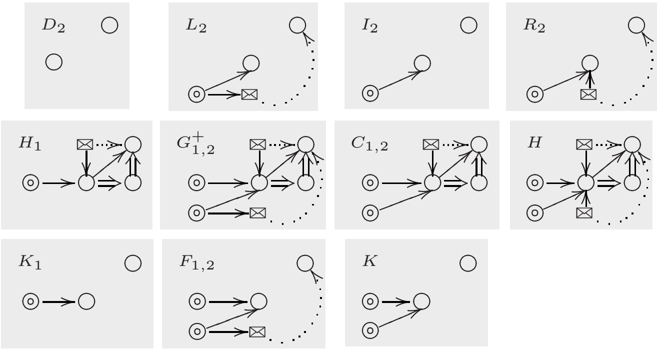


Fig. 5. A derivation that is sequential independent with that of Fig. 2

Example 14. The DPOBC derivations shown in Fig. 2 and Fig. 5 are sequential independent. One can safely take u, v, w as those morphisms preserving the position of nodes and edges into the respective graphs. Intuitively, the first derivation takes from the environment a user sending a message, and the second derivation takes another user sending another message. Clearly both derivations are sequential independent, and we can switch the order in which users are added (this is formally stated in Thm. 15).

Now, we can state our main result.

Theorem 15 (Church-Rosser and parallelism for Borrowed Contexts) *For any pair of productions $p_1 = L_1 \xleftarrow{l_1} I_1 \xrightarrow{r_1} R_1$ and $p_2 = L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2$, the following are equivalent for certain $a'_1 : D_1 \to H_2$ and $a'_2 : D_2 \to H_1$.*

1. *There are parallel independent derivations*

$$K_1 \longrightarrow H_1 \Leftarrow\!\langle b_1\text{-}D_1\text{-}a_1,p_1\rangle\!\dashv J \longrightarrow G \text{ and } J \longrightarrow G \models\!\langle p_2,a_2\text{-}D_2\text{-}b_2\rangle\!\Rightarrow K_2 \longrightarrow H_2.$$

2. *There are sequential independent derivations*

$$J \longrightarrow G \models\!\langle p_1,a_1\text{-}D_1\text{-}b_1\rangle\!\Rightarrow K_1 \longrightarrow H_1 \text{ and } K_1 \longrightarrow H_1 \models\!\langle p_2,a'_2\text{-}D_2\text{-}b_2\rangle\!\Rightarrow K \longrightarrow H.$$

3. *There are sequential independent derivations*

$$J \longrightarrow G \models\!\langle p_2,a_2\text{-}D_2\text{-}b_2\rangle\!\Rightarrow K_2 \longrightarrow H_2 \text{ and } K_2 \longrightarrow H_2 \models\!\langle p_1,a'_1\text{-}D_1\text{-}b_1\rangle\!\Rightarrow K \longrightarrow H.$$

Moreover, 1 implies that for $X = D_1 \cap D_2$,

4. *there is a parallel derivation*

$$J \longrightarrow G \models\!\langle p_1{+_X}p_2,[a_1,a_2]_X\text{-}(D_1{+_X}D_2)\text{-}b_1{+_X}b_2\rangle\!\Rightarrow K \longrightarrow H.$$

The main difference with respect to parallelism for DPO is that the fourth statement does not imply the others, as we discuss in the next section. Now, we want

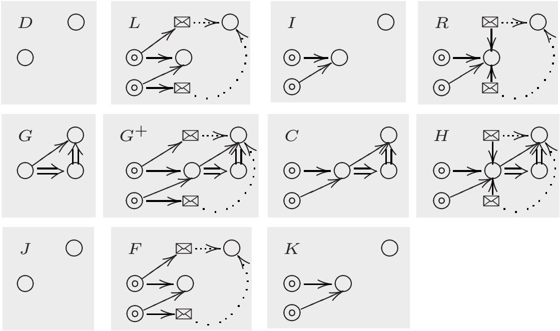


Fig. 6. Parallel derivation: two users put two messages in the same node

to point out that the opposite derivations in the Church-Rosser square employ exactly the same $b_1 : D_1 \to L_1$ and $b_2 : D_2 \to L_2$. Intuitively this means that the same resources of the system are used and thus the same are borrowed. This is analogous to LTSs with independence (discussed in Section 3), where opposite transitions in a concurrent diamond are labeled with the same event.

Theorem 16 (Church-Rosser and parallelism on labels). *The labels on the derivations described in the first three items of Thm. 15 form a diagram as the one shown in Def. 8 where the leftmost, the rightmost and the topmost squares are pushouts and the lowest is a pullback.*

Therefore our construction respects the strict diamond property. Moreover, since the lowest square of the diamond turns out to be a pullback, the diamond is a GRPO in the bicategory of cospans. Indeed, in [12] it is shown that every strong diamond where the lowest square is a pullback is a GRPO.

Example 17. In Ex. 12 we have shown that the derivation in Fig. 2 is parallel independent with itself. The joining derivations closing the Church-Rosser square

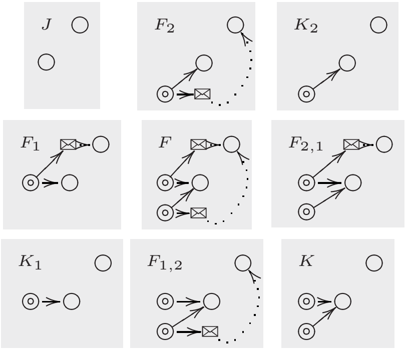


Fig. 7. A strict diamond

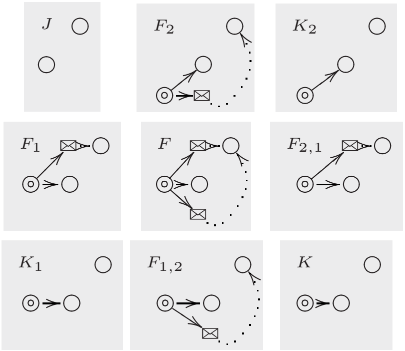


Fig. 8. A strong diamond

are two copies of the derivation shown in Fig. 5 (that is sequential independent with the one in Fig. 2), while the corresponding parallel derivation is shown in Fig. 6. This employs the parallel rule $\mathit{snd} +_X \mathit{snd}$ where X is the intersection of D_1 and D_2 over G, i.e., $X =$ ○ ○ (recall that D_2 is equal to D_1). This rule essentially states that two users can leave two messages at the same node in one single step. Thus the derivation borrows from the environment two users with two messages. These are exactly the same resources that are borrowed by performing sequentially first the derivation in Fig. 2 and then the one in Fig. 5, i.e., labels respect the strict diamond property as shown in Fig. 7.

5 From Parallel to Parallel Independent Derivations

The item 4 of Thm. 15 states that for any pair of parallel independent derivations there exists a parallel derivation whose labels enjoy the strict diamond property. Unfortunately, the converse does not hold: there exist derivations using a parallel rule that cannot be decomposed into derivations forming a strict diamond.

Consider the derivation in Fig. 9, for parallel production $\mathit{snd} +_Y \mathit{snd}$ over $Y =$ ⊚ ○ ○. This production allows a user to leave two messages in one step. Thus the derivation borrows one user with two messages as context. However, there exists no strict diamond with label $J \to F \leftarrow K$ of Fig. 9 as parallel label.

Indeed, any DPOBC derivation (by the rules in Fig. 1) contains at most one message. In order to have a user owning two messages as parallel label, we need to identify the users. Since the label F of the derivation (according to the strict diamond property) must be the pushout of F_1 and F_2 along J; and since the interface J does not contain any user, then users cannot be identified in F.

Intuitively, strict diamonds can not merge borrowed resources, while parallel derivations can. Indeed, given a pair of parallel independent transformations, we can construct a parallel transformation by gluing the productions over a graph containing resources common to both F_1 and F_2 that do not occur in G. Since these resources are merged in the left hand side of the parallel production, they are merged also in F. And since the resources are not in J, F is not the pushout of F_1 and F_2 along J. Thus, the resulting labels do not form a strict diamond.

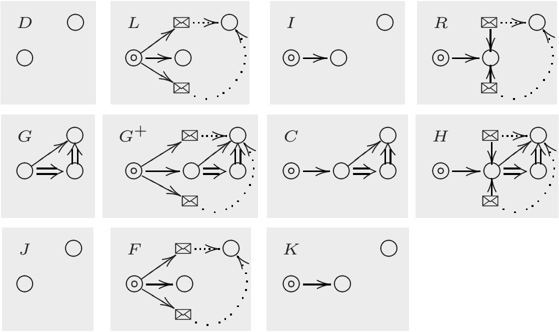


Fig. 9. Parallel derivation: one user sends two messages at the same node

More concretely, the parallel derivation in Fig. 9 is constructed by taking the two copies of the derivations in Fig. 2 (since it is parallel independent with itself) and by gluing the productions over J: this graph contains a user, that is an item occurring in F_1 and F_2, but not in G.

It seems thus clear that, in order to guarantee the converse of the parallelism theorem, we should move from strict to strong diamonds, since requiring that the morphisms $F_1 \to F$ and $F_2 \to F$ in the diamond are jointly epic allows to merge resources in F_1 and F_2 that are not in J. Indeed, it is worth noting that the labels of the derivation of the above example form a strong diamond (shown in Fig. 8) where the branching derivations are those of Fig. 2. So far, however, we were not able to prove such a correspondence, and we leave it for future work.

6 Conclusions and Future Work

Our work introduces the notions of parallel and sequential independence for DPOBC derivations, generalizing the corresponding concepts for DPO derivations. It then establishes a Church-Rosser theorem for independent derivations.

The crucial problem of lifting these properties and results from DPO to DPOBC rewriting is taking into account the borrowing of resources from the environment, and providing a precise description of the complex interactions between labels of independent transitions. Indeed, given two branching derivations, it is often the case that there exists a pair of joining derivations, even if these latter might differ with respect to resource usage when compared with the branching ones. With the strict diamond property we establish conditions to require in order to have that the joining derivations use neither more nor less environmental resources.

Our results provide the basis for further studies on the concurrent behaviour of open systems modeled via DPOBC rewriting. Since the standard DPOBC bisimilarity is interleaving, we plan to refine it to a true concurrent bisimilarity and compare it with similar behavioural equivalences proposed in the literature, e.g. history preserving bisimilarity [13,14] and related work on Petri nets [15,16].

For this, we plan to study the behavioural equivalence that arises by applying the standard DPOBC bisimilarity not to a given set of productions but to its *parallel closure*, which contains for every two rules also all their parallel compositions: using the standard results, this yields a bisimulation congruence.

As far as true concurrent semantics is concerned, we would like to explore the possibility to provide a notion of graph process for DPOBC rewriting, along the line of the theory developed for DPO rewriting [17]: in this respect, establishing the Church-Rosser property represents the cornerstone of such a development.

References

1. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Mathematical Structures in Computer Science 16(6), 1133–1163 (2006)
2. Leifer, J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)

3. Baldan, P., König, B., Ehrig, H.: Composition and decomposition of DPO transformations with borrowed context. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozemberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 153–167. Springer, Heidelberg (2006)
4. Gadducci, F.: Term graph rewriting and the π-calculus. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 37–54. Springer, Heidelberg (2003)
5. Gadducci, F., Lluch-Lafuente, A.: Graphical encoding of a spatial logic for the π-calculus. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 209–225. Springer, Heidelberg (2007)
6. Bonchi, F., Gadducci, F., König, B.: Process bisimulation via a graphical encoding. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozemberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 168–183. Springer, Heidelberg (2006)
7. 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, vol. 1, pp. 163–245. World Scientific, Singapore (1997)
8. Habel, A., Müller, J., Plump, D.: Double-pushout graph transformation revisited. Mathematical Structures in Computer Science 11, 637–688 (2001)
9. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. Theoretical Informatics and Applications 39, 511–545 (2005)
10. Winskel, G., Nielsen, M.: Models for concurrency. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S. (eds.) Handbook of Logic in Computer Science, vol. 4, pp. 1–148. Oxford University Press, Oxford (1992)
11. Sassone, V., Sobociński, P.: Reactive systems over cospans. In: Logic in Computer Science, pp. 311–320. IEEE Computer Society Press, Los Alamitos (2005)
12. Sobociński, P.: Deriving bisimulation congruences from reduction systems. PhD thesis, BRICS, Department of Computer Science, University of Aaurhus (2004)
13. Rabinovich, A.M., Trakhtenbrot, B.A.: Behaviour structures and nets. Foundamenta Informatica 11(4), 357–404 (1988)
14. van Glabbeek, R.J., Goltz, U.: Equivalence notions for concurrent systems and refinement of actions. In: Kreczmar, A., Mirkowska, G. (eds.) MFCS 1989. LNCS, vol. 379, pp. 237–248. Springer, Heidelberg (1989)
15. Baldan, P., Corradini, A., Ehrig, H., Heckel, R., König, B.: Bisimilarity and behaviour-preserving reconfiguration of open petri nets. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 126–142. Springer, Heidelberg (2007)
16. Nielsen, M., Priese, L., Sassone, V.: Characterizing behavioural congruences for Petri nets. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 175–189. Springer, Heidelberg (1995)
17. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae 26, 241–265 (1996)

Behavior Preservation in Model Refactoring Using DPO Transformations with Borrowed Contexts*

Guilherme Rangel[1], Leen Lambers[1], Barbara König[2], Hartmut Ehrig[1], and Paolo Baldan[3]

[1] Institut für Softwaretechnik und Theoretische Informatik,
Technische Universität Berlin, Germany
{rangel,leen,ehrig}@cs.tu-berlin.de

[2] Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany
barbara_koenig@uni-due.de

[3] Dipartimento di Matematica Pura e Applicata,
Università di Padova, Italy
baldan@math.unipd.it

Abstract. Behavior preservation, namely the fact that the behavior of a model is not altered by the transformations, is a crucial property in refactoring. The most common approaches to behavior preservation rely basically on checking given models and their refactored versions. In this paper we introduce a more general technique for checking behavior preservation of refactorings defined by graph transformation rules. We use double pushout (DPO) rewriting with borrowed contexts, and, exploiting the fact that observational equivalence is a congruence, we show how to check refactoring rules for behavior preservation. When rules are behavior-preserving, their application will never change behavior, i.e., every model and its refactored version will have the same behavior. However, often there are refactoring rules describing intermediate steps of the transformation, which are not behavior-preserving, although the full refactoring does preserve the behavior. For these cases we present a procedure to combine refactoring rules to behavior-preserving concurrent productions in order to ensure behavior preservation. An example of refactoring for finite automata is given to illustrate the theory.

1 Introduction

Model transformation [1] is concerned with the automatic generation of models from other models according to a transformation procedure which describes how a model in the source language can be "translated" into a model in the target language. Model refactoring is a special case of model transformation where the

* Research partially supported by DAAD (German Academic Exchange Service), DFG project Behavior-GT and MIUR project ART.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 242–256, 2008.

source and target are instances of the same metamodel. Software refactoring is a modern software development activity, aimed at improving system quality with internal modifications of source code which do not change the observable behavior. In object-oriented programming usually the observable behavior of an object is given by a list of public (visible) properties and methods, while its internal behavior is given by its internal (non-visible) properties and methods.

Graph transformation systems (GTS) are well-suited to model refactoring and, more generally, model transformation (see [2] for the correspondence between refactoring and GTS). Model refactorings based on GTS can be found in [3,4,5,6]. The left part of Fig. 1 describes schematically model refactoring via graph transformations. For a graph-based metamodel M, describing, e.g., deterministic finite automata or statecharts, the set $Refactoring^M$ of graph productions describes how to transform models which are instances of the metamodel M. A start graph G^M, which is an instance of the metamodel M, is transformed according to the productions in $Refactoring^M$ (using regular DPO transformations), thus producing a graph H^M which is the refactored version of G^M.

A crucial question that must be asked always is whether a given refactoring is behavior-preserving, which means that source and target models have the same observable behavior. In practice, proving behavior-preservation is not an easy task and therefore one normally relies on test suite executions and informal arguments in order to improve confidence that the behavior is preserved. On the other hand, formal approaches [7,8,9,10] have been also employed. A common issue is that behavior preservation is checked only for a certain number of models and their refactored versions. It is difficult though to foresee which refactoring steps are behavior-preserving for all possible instances of the metamodel. Additionally, these approaches are usually tailored to specific metamodels and the transfer to other metamodels would require reconsidering several details. A more general technique is proposed in [11] for analyzing the behavior of a graph production in terms of CSP processes and trace semantics which guarantees that the traces of a model are a subset of the traces of its refactored version.

In [3] we employed the general framework of borrowed contexts [12] to show that models are bisimilar to their refactored counterparts, which implies behavior preservation. The general idea is illustrated in the right-hand side of Fig. 1. We define a set $OpSem^M$ of graph productions describing the operational semantics of the metamodel M and use the borrowed context technique to check

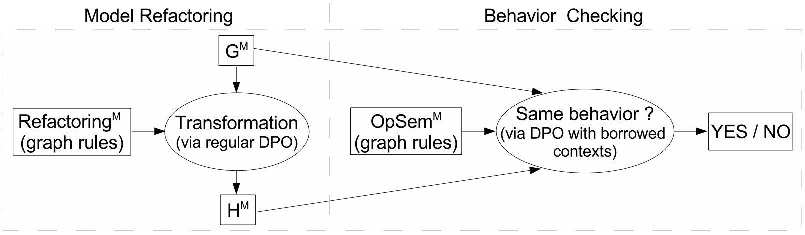

Model Refactoring
Behavior Checking
G^M
Refactoring^M (graph rules)
Transformation (via regular DPO)
H^M
OpSem^M (graph rules)
Same behavior ? (via DPO with borrowed contexts)
YES / NO

Fig. 1. Model refactoring via graph transformations and behavior preservation

whether the models G^M and H^M have the same behavior w.r.t. $OpSem^M$. In [3] we also tailored Hirschkoff's up-to bisimulation checking algorithm [13] to the borrowed context setting and thus equivalence checking can in principle be carried out automatically. The main advantage of this approach is that for every metamodel whose operational semantics can be specified in terms of finite graph transformation productions, the bisimulation checking algorithm can be used to show bisimilarity between models which are instances of this metamodel. However, this technique is also limited to showing behavior preservation only for a fixed number of instances of a metamodel.

In this paper we go a step further and employ the borrowed context framework in order to check refactoring productions for behavior preservation according to the operational semantics of the metamodel. We call a rule behavior-preserving when its left- and right-hand sides are bisimilar. Thanks to the fact that bisimilarity is a congruence, whenever all refactoring productions preserve behavior, so does every transformation via these rules. In this case, all model instances of the metamodel and their refactored versions exhibit the same behavior. However, refactorings very often involve non-behavior-preserving rules describing intermediate steps of the whole transformation. Given a transformation $G \overset{p_1}{\Rightarrow} H$ via a non-behavior-preserving rule p_1, the basic idea is then to check for the existence of a larger transformation $G \Rightarrow^* H'$ via a sequence $seq = p_1, p_2, \ldots, p_i$ of rule applications such that the concurrent production [14,15] induced by seq is behavior-preserving. Since the concurrent production p_c performs exactly the same transformation $G \overset{p_c}{\Rightarrow} H'$ we can infer that G and H' have the same behavior.

This paper is structured as follows. Section 2 briefly reviews how the DPO approach with borrowed contexts can be used to define the operational semantics of a metamodel. Section 3 defines the model refactorings we deal with. An example in the setting of finite automata is given in Section 4. In Section 5 we define a technique to check refactoring rules for behavior preservation and an extension to handle non-behavior-preserving rules in model refactoring. Finally, these techniques are applied to the automata example. The proofs of the results in this paper, which are omitted here because of space limitations, can be found in [16].

2 Operational Semantics Via Borrowed Contexts

In this section we recall the DPO approach with borrowed contexts [12,17] and show how it can be used to define the operational semantics of a metamodel M. In this paper we consider the category of labeled graphs, but the results would also hold for the category of typed graphs or, more generally, for adhesive categories. In standard DPO [18], productions rewrite graphs with no interaction with any other entity than the graph itself. In the DPO approach with borrowed contexts [17] graphs have interfaces and may borrow missing parts of left-hand sides from the environment via the interface. This leads to open systems which take into account interaction with the outside world.

Definition 1 (Graphs with Interfaces and Graph Contexts). *A* graph G with interface J *is a morphism* $J \to G$ *and a context consists of two morphisms* $J \to E \leftarrow \overline{J}$*. The* embedding *of a graph with interface* $J \to G$ *into a context* $J \to E \leftarrow \overline{J}$ *is a graph with interface* $\overline{J} \to \overline{G}$ *which is obtained by constructing* $\overline{G}$ *as the pushout of* $J \to G$ *and* $J \to E$*.*

$$\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & E \arrow[d] & \overline{J} \arrow[l] \arrow[dl, dotted] \\
G \arrow[r] & \overline{G} &
\end{tikzcd}$$

Observe that the embedding is defined up to isomorphism since the pushout object is unique up to isomorphism.

Definition 2 (Metamodel M and Model). *A* metamodel M *specifies a set of graphs with interface of the form* $J \to G$ *(as in Definition 1). An element of this set is called an instance of the metamodel* M*, or simply* model*.*

For example, the metamodel *DFA*, introduced in Section 4, describes deterministic finite automata. A model is an automaton $J \to G$, where G is the automaton and J specifies which parts of G may interact with the environment.

Definition 3 (Set of Operational Semantics Rules). *Given a metamodel* M *as in Definition 2, its operational semantics is defined by a set* $OpSem^M$ *of graph productions* $L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$*, where* l, r *are injective morphisms.*

Definition 4 (Rewriting with Borrowed Contexts). *Let* $OpSem^M$ *be as in Definition 3. Given a model* $J \to G$ *and a production* $p \colon L \leftarrow I \to R$ $(p \in OpSem^M)$*, we say that* $J \to G$ *reduces to* $K \to H$ *with transition label* $J \to F \leftarrow K$ *if there are graphs* D*,* G^+*,* C *and additional morphisms such that the diagram below commutes and the squares are either pushouts (PO) or pullbacks (PB) with injective morphisms. In this case a* rewriting step with borrowed context *(BC step) is called feasible:* $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$*.*

$$\begin{tikzcd}
D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[r] \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[ur, phantom, "PB"] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}$$

In the diagram above the upper left-hand square merges L and the graph G to be rewritten according to a partial match $G \leftarrow D \to L$. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with a morphism $J \to F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting graph H, which can be obtained by "intersecting" the borrowed context F and the graph C via a pullback. Note that the two pushout complements that are needed in Definition 4, namely C and F, may not exist. In this case, the rewriting step is not feasible. Furthermore, observe that for a given partial match $G \leftarrow D \rightarrow L$ the graphs G^+ and C are uniquely determined.

A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

Definition 5 (Bisimulation and Bisimilarity). *Let $OpSem^M$ be as in Definition 3 and let $\mathcal{R}$ be a symmetric relation containing pairs of models $(J \rightarrow G, J \rightarrow G')$. The relation $\mathcal{R}$ is called a* bisimulation *if, whenever we have $(J \rightarrow G)\,\mathcal{R}\,(J \rightarrow G')$ and a transition $(J \rightarrow G) \xrightarrow{J \rightarrow F \leftarrow K} (K \rightarrow H)$, then there exists a model $K \rightarrow H'$ and a transition $(J \rightarrow G') \xrightarrow{J \rightarrow F \leftarrow K} (K \rightarrow H')$ such that $(K \rightarrow H)\,\mathcal{R}\,(K \rightarrow H')$.*

We write $(J \rightarrow G) \sim_{OpSem^M} (J \rightarrow G')$ (or $(J \rightarrow G) \sim (J \rightarrow G')$ if the operational semantics is obvious from the context) whenever there exists a bisimulation $\mathcal{R}$ that relates the two instances of the metamodel M. The relation $\sim_{OpSem^M}$ is called bisimilarity*.*

When defining the operational semantics using the borrowed context framework, it should be kept in mind that rewriting is based on interactions with the environment, i.e., the environment should provide some information via F to the graph G in order to trigger the rewriting step. For instance, in our finite automata example in Section 4 the environment provides a letter to trigger the corresponding transition of the automaton.

An advantage of the borrowed context technique is that the derived bisimilarity is automatically a congruence, which means that whenever a graph with interface is bisimilar to another, one can exchange them in a larger graph without effect on the observable behavior. This is very useful for model refactoring since we can replace one part of the model by another bisimilar one, without altering its observable behavior.

Theorem 1 (Bisimilarity is a Congruence [12]). *Bisimilarity $\sim$ is a congruence, i.e., it is preserved by embedding into contexts as given in Definition 1.*

In [17] a technique is defined to speed up bisimulation checking, which allows us to take into account only certain labels. A label is considered superfluous and called *independent* if we can add two morphisms $D \rightarrow J$ and $D \rightarrow I$ to the diagram in Definition 4 such that $D \rightarrow I \rightarrow L = D \rightarrow L$ and $D \rightarrow J \rightarrow G = D \rightarrow G$. That is, intuitively, the graph G to be rewritten and the left-hand side L overlap only in their interfaces. Transitions with independent labels can be ignored in the bisimulation game, since a matching transition always exists.

3 Refactoring Transformations

Here we define refactoring transformations using DPO rules with negative application conditions (NAC).

Definition 6 (NAC, Rule with NAC and Transformation). *A* negative application condition *$NAC(n)$ on L is an injective morphism $n\colon L \to NAC$. An injective match $m\colon L \to G$ satisfies $NAC(n)$ on L if and only if there is no injective morphism $q\colon NAC \to G$ with $q \circ n = m$.*

$$\begin{tikzcd}
NAC \arrow[dr, "q"'] & L \arrow[l, "n"'] \arrow[d, "m"] \arrow[dl, phantom, "="] \\
 & G
\end{tikzcd}
\qquad
\begin{tikzcd}
NAC & L \arrow[l] \arrow[d, "m"'] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
 & G_0 & C_0 \arrow[l] \arrow[r] & G_1
\end{tikzcd}$$

A negative application condition $NAC(n)$ is called satisfiable *if n is not an isomorphism.*

A rule *$L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ (l, r injective)* with NACs *is equipped with a finite set of negative application conditions $NAC_L = \{NAC(n_i) \mid i \in I\}$. A* direct transformation *$G_0 \overset{p,m}{\Longrightarrow} G_1$ via a rule p with NACs and an injective match $m\colon L \to G_0$ consists of the double pushout diagram above, where m satisfies all NACs of p.*

Note that if $NAC(n)$ is satisfiable then the identity match $id : L \to L$ satisfies $NAC(n)$. We will assume that for any rule with NACs, the corresponding negative application conditions are all satisfiable, so that the rule is applicable to at least one match (the identity match on its left-hand side).

Definition 7 (Layered Refactoring System and Refactoring Rule). *Let metamodel M be as in Definition 2. A* refactoring rule *is a graph rule as in Definition 6. A* layered refactoring system *$Refactoring^M$ for the metamodel M consists of k sets $Refactoring_i^M$ ($0 \leq i \leq k-1$) of refactoring rules. Each set $Refactoring_i^M$ defines a transformation layer.*

Definition 8 (Refactoring Transformation). *Let $Refactoring^M$ be as in Definition 7. A* refactoring transformation *$t\colon (J \to G_0) \Rightarrow^* (J \to G_n)$ is a sequence $(J \to G_0) \overset{p_1}{\Rightarrow} (J \to G_1) \overset{p_2}{\Rightarrow} \cdots \overset{p_n}{\Rightarrow} (J \to G_n)$ of direct transformations (as in Definition 6) such that $p_i \in Refactoring^M$ and t preserves the interface J, i.e., for each i ($0 \leq i < n$) there exists an injective morphism $J \to C_i$ with $J \to G_i = J \to C_i \to G_i$ (see diagram below). Moreover, in t each layer applies its rules as long as possible before the rules of the next layer come into play.*

$$\begin{tikzcd}
NAC & L \arrow[l] \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
 & G_i & C_i \arrow[l] \arrow[r] & G_{i+1} \\
 & J \arrow[u] \arrow[ur, bend right] \arrow[urr, dotted, bend right] \arrow[u, phantom, shift left=4, "="] & &
\end{tikzcd}$$

Note that refactoring transformations operate only on the internal structure of G_i while keeping the original interface J.

4 Example: Deleting Unreachable States in *DFA*

In this section we present an example of refactoring in the setting of deterministic finite automata (DFA). The metamodel *DFA* describes finite automata

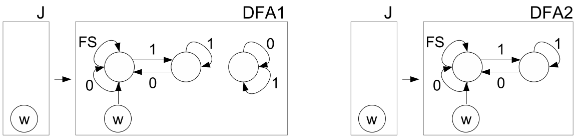


Fig. 2. Examples of DFA as graphs with interface

represented as graphs with interface as J → DFA1 and J → DFA2 in Fig. 2, where unlabeled nodes are states and directed labeled edges are transitions. An FS-loop marks a state as final. A W-node has an edge pointing to the current state and this edge points initially to the start state. The W-node is the interface, i.e., the only connection to the environment.

The operational semantics for DFA is given by a set $\mathsf{OpSem}^{\mathsf{DFA}}$ of rules containing Jump(a), Loop(a) and Accept depicted in Fig. 3. The rules Jump(a), Loop(a) must be defined for each symbol $a \in \Lambda$, where Λ is a fixed alphabet. According to $\mathsf{OpSem}^{\mathsf{DFA}}$ a DFA may change its state. The W-node receives a symbol (e.g. 'b') from the environment in form of a b-labeled edge connecting $\text{w} \xleftarrow{b} \text{w} \xleftarrow{c} \text{w}$ An acpt-edge between W-nodes marks the end of a string. When such an edge is consumed by a DFA, the string previously processed is accepted.

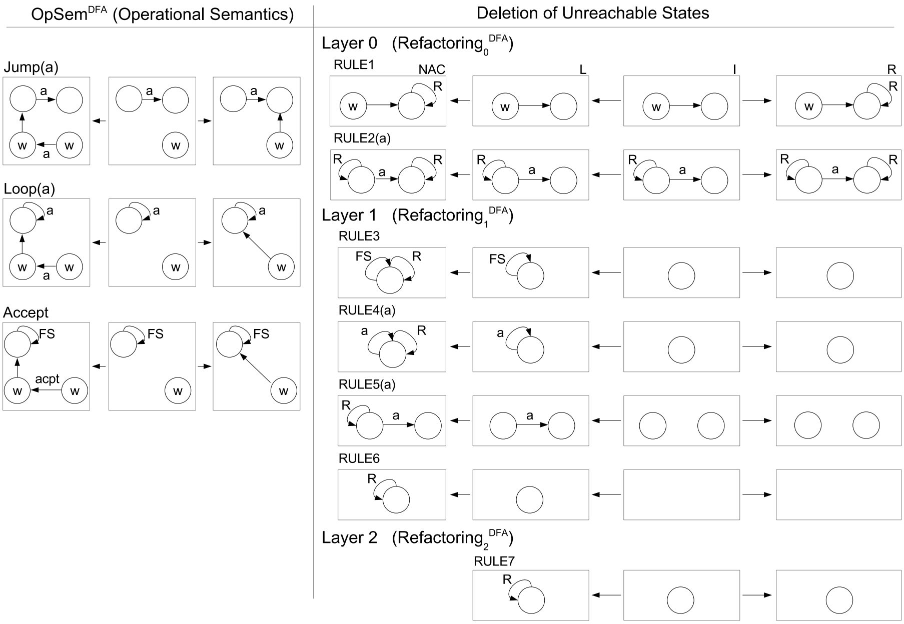

OpSem^DFA (Operational Semantics)
Deletion of Unreachable States
Jump(a)
a
w
a
w
a
w
a
w
Loop(a)
a
w
a
w
a
w
a
w
Accept
FS
acpt
w
w
FS
w
FS
w
Layer 0 (Refactoring_0^DFA)
RULE1
NAC
L
I
R
R
R
w
w
w
w
RULE2(a)
R
a
R
R
a
R
a
R
a
R
Layer 1 (Refactoring_1^DFA)
RULE3
FS
R
FS
RULE4(a)
a
R
a
RULE5(a)
R
a
a
RULE6
R
Layer 2 (Refactoring_2^DFA)
RULE7
R

Fig. 3. Operational semantics and a refactoring for DFA

A layered refactoring system for the deletion of unreachable states of an automaton is given in Fig. 3 on the right. To the left of each rule we depict the NAC (if it exists). The rules are spread over three layers. Rule1 marks the initial state as reachable with an R-loop. Rule2(a) identifies all other states that can be reached from the start state via a-transitions. Layer 1 deletes the loops and the edges of the unreachable states and finally the unreachable states. Layer 2 removes the R-loops.

Applying the refactoring rules above to the automaton $\mathsf{J} \to \mathsf{DFA1}$ we obtain $\mathsf{J} \to \mathsf{DFA2}$, where the rightmost state was deleted. By using the bisimulation checking algorithm of [3] we conclude that $\mathsf{J} \to \mathsf{DFA1}$ and $\mathsf{J} \to \mathsf{DFA2}$ are bisimilar w.r.t. $\mathsf{OpSem}^{\mathsf{DFA}}$. In our setting bisimilarity via the borrowed context technique corresponds to bisimilarity on automata seen as transition systems, which in turn implies language equivalence.

5 Behavior Preservation in Model Refactoring

Here we introduce a notion of behavior preservation for refactoring rules and, building on this, we provide some techniques for ensuring behavior preservation in model refactoring.

5.1 Refactoring Via Behavior-Preserving Rules

For a metamodel M as in Definition 2 we define behavior preservation as follows.

Definition 9 (Behavior-Preserving Transformation). *Let $OpSem^M$ be as in Definition 3. A refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ (as in Definition 8) is called behavior-preserving when $(J \to G) \sim_{OpSem^M} (J \to H)$.*

In order to check t for behavior preservation we can use Definition 4 to derive transition labels from $J \to G$ and $J \to H$ w.r.t. the rules in $OpSem^M$.

Observe that behavior preservation in the sense of Definition 9 is limited to checking specific models. This process is fairly inefficient and can never be exhaustive as behavior-preservation must be checked for each specific transformation. A more efficient strategy consists in focussing on the behavior-preservation property at the level of refactoring rules. The general idea is to check for every $p \in Refactoring^M$ whether its left and right-hand sides, seen as graphs with interfaces, are bisimilar, i.e., $(I \to L) \sim (I \to R)$ w.r.t. $OpSem^M$. Whenever this happens, since bisimilarity is a congruence, any transformation $(J \to G) \overset{p}{\Rightarrow} (J \to H)$ via p will preserve the behavior, i.e., $J \to G$ and $J \to H$ have the same behavior.

Definition 10 (Behavior-Preserving Refactoring Rule). *Let $OpSem^M$ be as in Definition 3. A refactoring production $p\colon L \leftarrow I \to R$ with NAC_L is* behavior-preserving *whenever $(I \to L) \sim (I \to R)$ w.r.t. $OpSem^M$.*

Now we can show a simple but important result that says that a rule is behavior-preserving if and only if every refactoring transformation generated by this rule is behavior-preserving.

Proposition 1. *Let $OpSem^M$ be as in Definition 3. Then it holds: $p\colon L \leftarrow I \rightarrow R$ (with NAC_L) is behavior-preserving w.r.t. $OpSem^M$ if and only if any refactoring transformation $(J \rightarrow G) \overset{p}{\Rightarrow} (J \rightarrow H)$ (as in Definition 8) is behavior-preserving, i.e., $(J \rightarrow G) \sim_{OpSem^M} (J \rightarrow H)$.*

Remark 1. The fact that the previous proposition also holds for rules with NACs, even though Definition 10 does not take NACs into account for behavior-preservation purposes, does of course not imply that negative application conditions for refactoring rules are unnecessary in general. They are needed in order to constrain the applicability of rules, especially of those rules that are not behavior-preserving, or rather, are only behavior-preserving when applied in certain contexts. As a direction of future work, we plan to study congruence results for restricted classes of contexts. This will help to better handle refactoring rules with NACs.

Theorem 2 (Refactoring via Behavior-Preserving Rules). *Let $OpSem^M$ and $Refactoring^M$ be as in Definitions 3 and 7. If each rule in $Refactoring^M$ is behavior-preserving w.r.t. $OpSem^M$ then any refactoring transformation $(J \rightarrow G_0) \Rightarrow^* (J \rightarrow G_n)$ via these rules is behavior-preserving.*

Example 1. We check the rules in $\mathsf{Refactoring}_{\mathsf{i}}^{DFA}$ $(i = 0, 1, 2)$ from Section 4 for behavior preservation. We begin with $\mathsf{Refactoring}_{\mathsf{0}}^{DFA}$ (Layer 0). For RULE1: $\mathsf{NAC} \leftarrow \mathsf{L} \leftarrow \mathsf{I} \rightarrow \mathsf{R}$ we derive the transition labels from $\mathsf{I} \rightarrow \mathsf{L}$ and $\mathsf{I} \rightarrow \mathsf{R}$ w.r.t. $\mathsf{OpSem}^{\mathsf{DFA}}$. On the left-hand side of Fig. 4 we schematically depict the first steps in their respective labeled transition systems (LTS), where each partner has three choices. Independent labels exist in both LTSs but are not illustrated below.

The derivation of label L_1 for $\mathsf{I} \rightarrow \mathsf{R}$ is shown on the right. Since $\mathsf{I} \rightarrow \mathsf{L}$ and $\mathsf{I} \rightarrow \mathsf{R}$ (and their successors) can properly mimic each other via a bisimulation we can conclude that $(\mathsf{I} \rightarrow \mathsf{L}) \sim_{\mathsf{OpSem}^{\mathsf{DFA}}} (\mathsf{I} \rightarrow \mathsf{R})$. The intuitive reason for this is that the R-loop, which is added by this rule, does not have any meaning in the operational semantics and is hence “ignored” by $\mathsf{OpSem}^{\mathsf{DFA}}$.

Analogously, RULE2(a) and the rule in Layer 2 are behavior-preserving as well. Hence, we can infer that every transformation via the rules of Layer 0 and Layer 2 preserves the behavior. On the other hand, all rules in Layer 1, except for RULE6, are not behavior-preserving. Note that RULE6 is only behavior-preserving because of the dangling condition. Thus, when a transformation is carried out via non-behavior-preserving rules of Layer 1 we cannot be sure whether the behavior has been preserved.

5.2 Handling Non-behavior-Preserving Rules

For refactoring transformations based on non-behavior-preserving rules the technique of Section 5.1 does not allow to establish if the behavior is preserved.

Very often there are refactoring rules representing intermediate transformations that indeed are not behavior-preserving. Still, when considered together with neighboring rules, they could induce a concurrent production [14,15] p_c, corresponding to a larger transformation, which preserves the behavior. For a

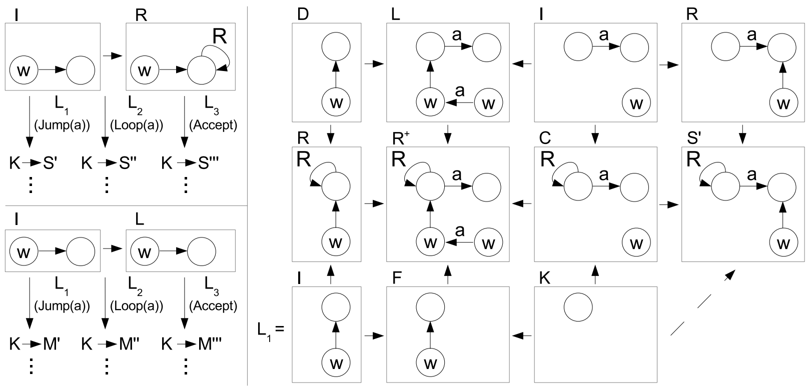


Fig. 4. Labeled transition systems for rule1 and a label derivation

transformation $t\colon (J \to G) \Rightarrow^* (J \to H')$ via a sequence $seq = p_1, p_2, \ldots, p_i$ the concurrent production $p_c\colon L_c \leftarrow I_c \to R_c$ with concurrent NAC_{L_c} induced by t performs exactly the same transformation $(J \to G) \overset{p_c}{\Rightarrow} (J \to H')$ in one step. Moreover, p_c can only be applied to $(J \to G)$ if the concurrent NAC_{L_c} is satisfied. This is the case if and only if every NAC of the rules in t is satisfied. The basic idea is now to check for a transformation $(J \to G) \overset{p_1}{\Rightarrow} (J \to H)$ based on a non-behavior-preserving rule p_1 whether there exists such a larger transformation $t : (J \to G) \Rightarrow^* (J \to H')$ via a sequence $seq = p_1, p_2, \ldots, p_i$ of rules such that the concurrent production induced by t is behavior preserving. Then we can infer that $J \to G$ and $J \to H'$ have the same behavior.

This is made formal by the notion of safe transformation and the theorem below.

Definition 11 (Safe Transformation). *Let $OpSem^M$ be as in Definition 3. A refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ (as in Definition 8) is called* safe *if it induces a behavior-preserving concurrent production w.r.t. $OpSem^M$.*

Theorem 3 (Safe Transformations preserve Behavior). *Let $OpSem^M$ and $Refactoring^M$ be as in Definitions 3 and 7, and let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation. If t is safe, then t is behavior-preserving, i.e., $(J \to G) \sim (J \to H)$.*

In order to prove that a refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ is safe (and thus behavior-preserving), we can look for a split $t^{sp}\colon G \Rightarrow^* H_1 \Rightarrow^* \cdots \Rightarrow^* H_n \Rightarrow^* H$ (interfaces are omitted) of t where each step $(\Rightarrow^*)$ induces a behavior-preserving concurrent production (see Definition 12). In fact, as shown below, if and only if such split exists we can guarantee that t preserves behavior (Theorem 4).

Definition 12 (Safe Transformation Split). *Let $OpSem^M$ be as in Definition 3 and let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation (as in*

Definition 8). A split *of t is obtained by cutting t into a sequence of subtransformations $t^{sp}\colon (J \to G) \Rightarrow^* (J \to H_1) \Rightarrow^* \cdots \Rightarrow^* (J \to H_n) \Rightarrow^* (J \to H)$. A transformation split t^{sp} is* safe *if each step ($\Rightarrow^*$) is safe.*

In Section 5.3 we present a simple search strategy for safe splits. More elaborate ones are part of future work.

Theorem 4. *Let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation. Then t is safe if and only if it admits a safe split.*

Observe that, instead, the following does not hold in general: if $t\colon (J \to G) \Rightarrow^* (J \to H)$ and $(J \to G) \sim_{OpSem^M} (J \to H)$ then t is safe. Consider for instance $\mathsf{RULE5(a)}$ in Fig. 3. As remarked, it is in general not behavior-preserving, but when, by coincidence, it removes a transition that is unreachable from the start state, the original automaton and its refactored version are behaviorally equivalent.

5.3 Ensuring Behavior Preservation

In this section we describe how the theory presented in this paper can be applied. Note that our results would allow us to automatically prove behavior preservation only in special cases, while, in general, such mechanized proofs will be very difficult. Hence here we will suggest a "mixed strategy", which combines elements of automatic verification and the search for behavior-preserving rules, in order to properly guide refactorings.

More specifically, a given model $J \to G$ can be refactored by applying the rules in $\mathit{Refactoring}^M$ in an automatic way, where the machine chooses nondeterministically the rules to be applied, or in a user-driven way, where for each transformation the machine provides the user with a list of all applicable rules together with their respective matches and ultimately the user picks one of them. The main goal is then to tell the user whether the refactoring is behavior-preserving.

The straightforward strategy to accomplish the goal above is to transform $J \to G$ applying only behavior-preserving rules. This obviously guarantees that the refactoring preserves behavior. However if a non-behavior-preserving rule p is applied we can no longer guarantee behavior preservation. Still, by proceeding with the refactoring, namely by performing further transformations, we can do the following: for each new transformation added to the refactoring we compute the induced concurrent production for the transformation which involves the first non-behavior-preserving rule p and the subsequent ones. If this concurrent production is behavior-preserving we can again guarantee behavior preservation for the refactoring since the refactoring admits a safe split (see Theorem 4).

The strategy above is not complete since behaviour preservation could be ensured by the existence of complex safe splits which the illustrated procedure is not able to find. We already have preliminary ideas for more sophisticated search strategies, but they are part of future work. Note however, that this strategy can reduce the proof obligations, since we do not have to show behavior preservation between the start and end graph of the refactoring sequence (which may be huge), but we only have to investigate local updates of the model.

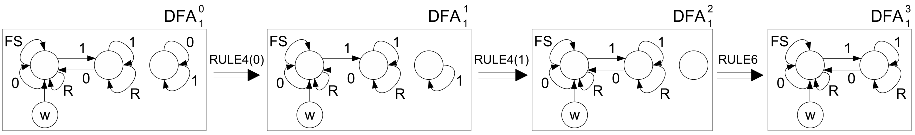


Fig. 5. Refactoring transformation

Example 2. Consider the automaton $\mathsf{J} \to \mathsf{DFA1}$ of Section 4. By applying the behavior-preserving rules of $\mathsf{Refactoring}_0^{DFA}$ ($\mathsf{Layer\ 0}$) we obtain $\mathsf{J} \to \mathsf{DFA}_1^0$ depicted in Fig. 5 (the interface J is omitted). Since $\mathsf{Refactoring}_0^{DFA}$ contains only behavior-preserving rules by Theorem 2 it holds that $(\mathsf{J} \to \mathsf{DFA1}) \Rightarrow^* (\mathsf{J} \to \mathsf{DFA}_1^0)$ preserves the behavior. No more rules in $\mathsf{Refactoring}_0^{DFA}$ can be applied, i.e., the computation of $\mathsf{Layer\ 0}$ terminates.

Now the rules of $\mathsf{Refactoring}_1^{DFA}$ ($\mathsf{Layer\ 1}$) come into play. Recall that all rules in $\mathsf{Refactoring}_1^{DFA}$ are non-behavior-preserving, except for $\mathsf{RULE6}$. This set contains $\mathsf{RULE4(0)}$ and $\mathsf{RULE4(1)}$ which are appropriate instantiations of $\mathsf{RULE4(a)}$. After the transformation $(\mathsf{J} \to \mathsf{DFA}_1^0) \overset{\mathsf{RULE4(0)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^1)$ we can no longer guarantee behavior-preservation since $\mathsf{RULE4(0)}$ has been applied. From now on we follow the strategy previously described to look for a behavior-preserving concurrent production. We perform the step $(\mathsf{J} \to \mathsf{DFA}_1^1) \overset{\mathsf{RULE4(1)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^2)$, build a concurrent production p_c induced by $(\mathsf{J} \to \mathsf{DFA}_1^0) \overset{\mathsf{RULE4(0)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^1) \overset{\mathsf{RULE4(1)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^2)$ and, by checking p_c for behavior-preservation, we find out that it is not behavior-preserving. We then continue with $(\mathsf{J} \to \mathsf{DFA}_1^2) \overset{\mathsf{RULE6}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^3)$, build p_c' (Fig. 6), induced by the transformation beginning at $\mathsf{J} \to \mathsf{DFA}_1^0$ and check it for behavior-preservation. Now p_c' is behavior-preserving and so we can once again guarantee behavior preservation (Theorem 3).

Finally, no more rules of $\mathsf{Refactoring}_1^{DFA}$ are applicable to $\mathsf{J} \to \mathsf{DFA}_1^3$. The behavior-preserving rule in $\mathsf{Refactoring}_2^{DFA}$ ($\mathsf{Layer\ 2}$) comes into play and performs a transformation $(\mathsf{J} \to \mathsf{DFA}_1^3) \overset{\mathsf{RULE6}}{\Longrightarrow}_2 (\mathsf{J} \to \mathsf{DFA2})$, where the final automaton is depicted in Section 4 ($\mathsf{DFA2}$). Concluding, since we have found a safe split for the transformation via non-behavior-preserving rules we can infer that $\mathsf{J} \to \mathsf{DFA1}$ and $\mathsf{J} \to \mathsf{DFA2}$ have the same behavior.

Intuitively, the concurrent production is behavior-preserving, since it deletes an entire connected component that is not linked to the rest of the automaton. Note that due to the size of the components involved it can be much simpler to check such transformation units rather than the entire refactoring sequence.

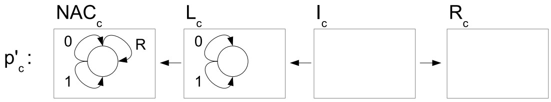


Fig. 6. Induced concurrent production p_c'

In addition, it would be useful if the procedure above could store the induced concurrent productions which are behavior-preserving into $\mathit{Refactoring}^M$ for later use. By doing so the user knows which combination of rules leads to behavior-preserving concurrent productions. Similarly, the user could also want to know which combination of rules leads to non-behavior-preserving concurrent productions. Of course, in the latter case the concurrent productions are just stored but do not engage in any refactoring transformation. It is important to observe that we store into $\mathit{Refactoring}^M$ only concurrent productions which are built with rules within the same layer (as in Example 2). For more complex refactorings, such as the flattening of hierarchical statecharts (see [19]), a behavior-preserving concurrent production p_c exists only when it is built from a transformation involving several layers. In this latter case, p_c is built and checked for behavior preservation but not stored for later use.

For the cases where a layer $\mathit{Refactoring}_i^M$ of $\mathit{Refactoring}^M$ is terminating and confluent it is then important to guarantee that adding concurrent productions to the refactoring layer does not affect these properties.

Theorem 5. *Let $\mathit{Refactoring}_i^M$ be as in Definition 7 and $R_i^{p_c}$ be a set containing concurrent productions p_c built from $p, q \in \mathit{Refactoring}_i^M \cup R_i^{p_c}$. Then whenever $\mathit{Refactoring}_i^M$ is confluent and terminating it holds that $\mathit{Refactoring}_i^M \cup R_i^{p_c}$ is also terminating and confluent.*

For the case where layer $\mathit{Refactoring}_i^M$ is terminating and confluent another interesting and useful fact holds: assume that we fix a start graph G_0 and we can show that *some* (terminating) transformation, beginning with G_0 allows a behavior-preserving split. Then clearly all transformations starting from G_0 are behavior-preserving since they result in the same final graph H.

6 Conclusions and Future Work

We have shown how the borrowed context technique can be used to reason about behavior-preservation of refactoring rules and refactoring transformations. In this way we shift the perspective from checking specific models to the investigation of the properties of the refactoring rules.

The formal techniques in related work [7,8,9,10] address behavior preservation in model refactoring, but are in general tailored to a specific metamodel and limited to checking the behavior of a fixed number of models. Therefore, the transfer to different metamodels is, in general, quite difficult.

Hence, with this paper we propose to use the borrowed context technique in order to consider any metamodel whose operational semantics can be given by graph productions. Furthermore, the bisimulation checking algorithm [3] for borrowed contexts provides the means for automatically checking models for behavior preservation. This can be done not only for a specific model and its refactored version, but also for the left-hand and right-hand sides of refactoring rules. Once we have shown that a given rule is behavior-preserving, i.e., its left- and right-hand sides are equivalent, we know that its application will always preserve the behavior, due to the congruence result. When rules are not behavior-preserving, they still can be combined into behavior-preserving concurrent productions. We believe that such a method will help the user to gain a better understanding of the refactoring rules since he or she can be told exactly which rules may modify the behavior during a transformation. An advantage of our technique over the one in [11] is that we work directly with graph transformations and do not need any auxiliary encoding. Furthermore, with our technique we can guarantee that a model and its refactored version have exactly the same observable behavior, while in [11] the refactored model "contains" the original model but may add extra behavior.

This work opens up several possible directions for future investigations. First, in some refactorings when non-behavior-preserving rules are applied, the search strategies for safe splits can become very complex. Here we defined only a simple search strategy, but it should be possible to come up with more elaborate ones.

Second, although we are working with refactoring rules with negative application conditions, these NACs do not play a prominent role in our automatic verification techniques, but of course they are a key to limiting the number of concurrent productions which can be built. In [20] the borrowed context framework and the congruence result has been extended to handle rules with NACs. However, this applies only to negative application conditions in the operational semantics. It is, nevertheless, also important to have similar results for refactoring rules with NACs, which would lead to a "restricted" congruence result, where bisimilarity would only be preserved by certain contexts (see also the discussion in Remark 1). Since model refactorings often use graphs with attributes it is useful to check whether the congruence results in [12,20] also hold for adhesive HLR categories (the category of attributed graphs is an instance thereof).

Acknowledgements. We would like to thank Gabi Taentzer and Reiko Heckel for helpful discussions on this topic.

References

1. Mens, T., Gorp, P.V.: A taxonomy of model transformation. ENTCS 152, 125–142 (2006)
2. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
3. Rangel, G., König, B., Ehrig, H.: Bisimulation verification for the DPO approach with borrowed contexts. In: Proc. of GT-VMT 2007. Electronic Communications of the EASST, vol. 6 (2007)
4. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: EMF model refactoring based on graph transformation concepts. In: SeTra 2006. Electronic Communications of EASST, vol. 3 (2006)
5. Hoffmann, B., Janssens, D., Eetvelde, N.V.: Cloning and expanding graph transformation rules for refactoring. ENTCS 152, 53–67 (2006)
6. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and Systems Modeling 6(3), 269–285 (2007)

7. van Kempen, M., Chaudron, M., Kourie, D., Boake, A.: Towards proving preservation of behaviour of refactoring of UML models. In: SAICSIT 2005, South African Institute for Computer Scientists and Information Technologists, pp. 252–259 (2005)
8. Pérez, J., Crespo, Y.: Exploring a method to detect behaviour-preserving evolution using graph transformation. In: Proceedings of the Third International ERCIM Workshop on Software Evolution, pp. 114–122 (2007)
9. Narayanan, A., Karsai, G.: Towards verifying model transformations. In: Bruni, R., Varró, D. (eds.) Proc. of GT-VMT 2006, Vienna. ENTCS, pp. 185–194 (2006)
10. Van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards automating source-consistent UML refactorings. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 144–158. Springer, Heidelberg (2003)
11. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 347–361. Springer, Heidelberg (2008)
12. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 151–166. Springer, Heidelberg (2004)
13. Hirschkoff, D.: Bisimulation verification using the up-to techniques. International Journal on Software Tools for Technology Transfer 3(3), 271–285 (2001)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, New York (2006)
15. Lambers, L.: Adhesive high-level replacement system with negative application conditions. Technical report, TU Berlin (2007)
16. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. Technical Report 12/08, TU Berlin (2008)
17. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Mathematical Structures in Computer Science 16(6), 1133–1163 (2006)
18. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Loewe, 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, Foundations, vol. 1, pp. 163–246. World Scientific, Singapore (1997)
19. Rangel, G.: Bisimulation Verification for Graph Transformation Systems with Borrowed Contexts. PhD thesis, TU Berlin (to appear, 2008)
20. Rangel, G., König, B., Ehrig, H.: Deriving bisimulation congruences in the presence of negative application conditions. In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 413–427. Springer, Heidelberg (2008)

Open Petri Nets: Non-deterministic Processes and Compositionality*

Paolo Baldan[1], Andrea Corradini[2], Hartmut Ehrig[3], and Barbara König[4]

[1] Dipartimento di Matematica Pura e Applicata, Università di Padova, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Institut für Softwaretechnik und Theoretische Informatik,
Technische Universität Berlin, Germany
[4] Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany

Abstract. We introduce ranked open nets, a reactive extension of Petri nets which generalises a basic open net model introduced in a previous work by allowing for a refined notion of interface. The interface towards the external environment of a ranked open net is given by a subset of places designated as open and used for composition. Additionally, a bound on the number of connections which are allowed on an open place can be specified. We show that the non-deterministic process semantics is compositional with respect to the composition operation over ranked open nets, a result which did not hold for basic open nets.

1 Introduction

Petri nets are a well-known model of concurrent and distributed systems, widely used both in theoretical and applicative areas ([14]). While the basic model is mainly aimed at representing closed, completely specified systems evolving autonomously through the firing of transitions, in recent years there has been an increasing attention to the development of reactive Petri net models, directly supporting certain features needed for modeling *open* systems, which can interact with the surrounding environment ([3], [9], [11], [12], [13], [15]).

In particular, open Petri nets, as introduced in ([1]), are a mild extension of basic nets with the possibility of interacting with the environment and of composing a larger net out of smaller open components. An open net is an ordinary net with a distinguished set of places, designated as open, through which the net can interact with the environment. As a consequence of such interaction, tokens can be freely generated and removed in open places. Open nets are endowed with a composition operation, characterised as a pushout in the corresponding category, suitable to model both interaction through open places and synchronisation of transitions.

It is very convenient if compositionality at the system level is reflected at the semantic level, i.e., if the behaviour of a system can be suitably expressed on the

* Supported by the MIUR Project ART, the DFG project Behaviour-GT and CRUI/DAAD Vigoni.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 257–273, 2008.

basis of the behaviour of its components. This allows for modular analysis of the systems and it helps in defining system reconfigurations (replacing a component by another) which keep the observable behaviour unchanged ([2, 4]).

In particular, as non-sequential processes of a Petri net can be fruitfully used as a representation of possible scenarios in the execution of a system (see, e.g., the work on workflows and the encodings of web-service description languages like OWL or BPEL as Petri nets ([4, 16, 17, 18])), it can be interesting to relate the processes of a Petri net with those of its components. Specifically, one should understand under which conditions processes of the subcomponent nets can be combined into a consistent process of their composition and vice versa, how processes of the full system can be decomposed into processes of the components.

Results in this direction have been provided for open nets in ([1]), by showing compositionality for a semantics based on *deterministic* processes *à la* Goltz-Reisig. Unfortunately, as noticed in the same paper, the result does not extend to non-deterministic processes. To get a rough intuition of what fails, consider the open nets in the Fig. 3(b) (ignoring, for the moment, the labels 2 and ω attached to dangling arcs). The representation of the nets is standard; only open places have ingoing and/or outgoing dangling arcs, meaning that transitions of the environment could be attached and thus put and/or remove tokens in these places. The nets Z_i are simple enough to be considered processes themselves. For instance, Z_1 represents a process in which a token can be consumed either by t_1 or by the environment. When joining Z_1 and Z_2 along the net Z_0, in the result Z_3 place s is still open, the intuition being that each open place allows for an unbounded number of connections, hence adding one connection does not affect its openness. There is no way of specifying that, as a result of the composition, the open port of each of the two components is occupied by the other component, thus producing a net where place s is closed. This is problematic since a net identical to Z_3, but where place s is closed, is a valid process of Z_3 (specifying a computation having no interactions with the environment). However there is no way to obtain it as the composition of two processes of Z_1 and Z_2.

In order to overcome this problem, we introduce *ranked open nets*, a refined model of open nets where besides specifying the open places, which can be used for composition with other nets, we also specify the maximum number of allowed (input and output) connections. This provides a more expressive model, properly subsuming basic open nets (which can be seen as special ranked open nets, where open places always allow for an unbounded number of connections).

A mechanism for composing ranked open nets is defined which generalises the one for basic open nets. In this case the composition operation cannot be characterised as a colimit. Instead, it can be seen as an abstraction of a pushout in a more concrete category where ports are made explicit.

The composition operation is extended to non-deterministic processes and we prove the desired compositionality result: if a net Z_3 is the composition of Z_1 and Z_2, then any process of Z_3 can be obtained as the composition of processes of the component nets and vice versa, the composition of processes of Z_1 and Z_2, which agree on the common interface, always provides a process of Z_3.

The paper is organised as follows. In § 2 we introduce the categories of ranked open nets, and an operation of composition for such nets is defined in § 3. In § 4 we introduce non-deterministic processes for ranked open nets. In § 5 we prove the main result, i.e., compositionality for non-deterministic processes. Finally, in § 6 we draw some conclusions and directions of future investigation.

2 Ranked Open Nets

An *open net*, as introduced in ([1]), is an ordinary P/T Petri net with a distinguished set of places. These places are intended to represent the interface of the net towards the environment, which, interacting with the net, can "freely" add or remove some tokens in the open places. Rather than simply distinguishing between *input* and *output* places, here, for every place we specify the largest number of allowed incoming and outgoing new connections. A place is closed if it does not allow for any new connection.

Given a set X we will denote by $X^\oplus$ the free commutative monoid generated by X, with identity 0, and by $\mathbf{2}^X$ its powerset. Furthermore given a function $h : X \to Y$ we denote by $h^\oplus : X^\oplus \to Y^\oplus$ its monoidal extension, while the same symbol $h : \mathbf{2}^X \to \mathbf{2}^Y$ denotes the extension of h to sets.

A *P/T Petri net* is a tuple $N = (S, T, \sigma, \tau)$ where S is the set of places, T is the set of transitions ($S \cap T = \emptyset$) and $\sigma, \tau : T \to S^\oplus$ assign source and target to each transition. In this paper we will consider only *finite* Petri nets. We will denote by ${}^\bullet(\cdot)$ and $(\cdot)^\bullet$ the monoidal extensions of the functions σ and τ to functions from $T^\oplus$ to $S^\oplus$. Furthermore, given a place $s \in S$, the pre- and post-set of s are defined by ${}^\bullet s = \{t \in T \mid s \in t^\bullet\}$ and $s^\bullet = \{t \in T \mid s \in {}^\bullet t\}$.

Let N_0 and N_1 be Petri nets. A *Petri net morphism* $f : N_0 \to N_1$ is a pair of total functions $f = \langle f_T, f_S \rangle$ with $f_T : T_0 \to T_1$ and $f_S : S_0 \to S_1$, such that for all $t_0 \in T_0$, ${}^\bullet f_T(t_0) = {f_S}^\oplus({}^\bullet t_0)$ and $f_T(t_0)^\bullet = {f_S}^\oplus(t_0{}^\bullet)$. The category of P/T Petri nets and Petri net morphisms will be denoted by **Net**.

We use $\mathbb{N}$ for the set of natural numbers and $\mathbb{N}^\omega$ for the same set extended with infinity, i.e., $\mathbb{N} \cup \{\omega\}$. Operations and relations on $\mathbb{N}^\omega$ are defined in the expected way, i.e., $n \leq \omega$ for each $n \in \mathbb{N}$, $\omega - n = \omega + n = \omega + \omega = \omega$ for each $n \in \mathbb{N}$, while $\omega - \omega$ is undefined. The same operators will be applied, pointwise, to functions over natural numbers. E.g., given $f, g : X \to \mathbb{N}^\omega$ we denote by $f + g : X \to \mathbb{N}^\omega$ the function defined by $(f + g)(x) = f(x) + g(x)$ for any $x \in X$.

Definition 1 (ranked open net). *A* (ranked) open net *is a pair* $Z = (N_Z, o_Z)$, *where* $N_Z = (S_Z, T_Z, \sigma_Z, \tau_Z)$ *is an ordinary P/T Petri net (called the* underlying net*) and* $o_Z = (o_Z^+, o_Z^-) : S_Z \to \mathbb{N}^\omega$. *We define* $O_Z^x = \{s \in S_Z : o_Z^x(s) > 0\}$, *for* $x \in \{+, -\}$ *and call them the sets of* input *and* output open places *of the net.*

As mentioned above, the functions o_Z^+ and o_Z^- intuitively specify for each place in S_Z the maximum number of allowed new ingoing/outgoing connections, also referred to as the *ranks* of s. In ([1]) whenever a place was open, intuitively there was no limit to the number of new connections. Hence the open nets of ([1]) can be seen as special ranked open nets, where $o_Z^x(s) \in \{0, \omega\}$ for any place s.

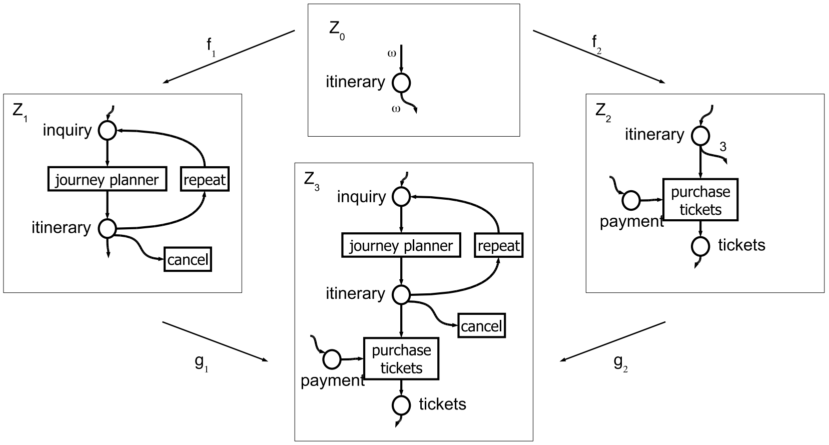


Fig. 1. Composing ranked open nets

As an example of ranked open nets, consider net Z_3 in Fig. 1, intuitively modelling the booking of a ticket in a travel agency. In the graphical representation an input (resp. output) open place s has a dangling ingoing (resp. outgoing) arc, marked by the corresponding rank. When the rank is 1 it is omitted.

Conceptually, we can think that every place of an open net has a set of *attaching points*, which can either be used by an existing transition connected to the place, or can be free and thus usable for connecting new transitions. Sometimes, as in the definition of concrete morphisms below (Definition 6), we need to consider an explicit identity of such attaching points, that we call *ports*. A port used by a transition is identified with the transition itself, while free ports are identified by a progressive number. Most often, however, we will be interested only in their number, i.e., in the *degree* of a place. Given $n \in \mathbb{N}$, let $[n]$ denote the set $\{0 \ldots, n-1\}$. For all considered Z we assume that $T \cap \mathbb{N}^\omega = \emptyset$.

Definition 2 (input and ouput ports and degree). *Let Z be an open net. For any place $s \in S$ we define the sets of* input *and* output ports *of s as follows:*

$$p^+(s) = [o_Z^+(s)] \cup {}^\bullet s \quad \textit{and} \quad p^-(s) = [o_Z^-(s)] \cup s^\bullet$$

The ports in $[o_Z^+(s)]$ and $[o_Z^-(s)]$ are called open ports.

Furthermore, we define the input degree *of s as $deg^+(s) = |p^+(s)|$, and, similarly, the* output degree *of s as $deg^-(s) = |p^-(s)|$.*

The token game of open nets. The notion of enabledness for transitions is the usual one, but, besides the changes produced by the firing of the transitions of the net, we consider also the interaction with the environment which is modelled by events, denoted by $+_s$ and $-_s$, which produce or consume a token in an open place s. For an open net Z, the *set of extended events*, denoted $\bar{T}_Z$, is defined as

$$\bar{T}_Z = T_Z \cup \{+_s : s \in O_Z^+\} \cup \{-_s : s \in O_Z^-\}.$$

Pre- and post-set functions are extended by defining ${}^{\bullet}{+_s} = 0$ and ${+_s}^{\bullet} = s$, and symmetrically, ${}^{\bullet}{-_s} = s$ and ${-_s}^{\bullet} = 0$.

Definition 3 (firing). *Let Z be an open net. A* firing *in Z consists of the execution of an extended event $\epsilon \in \bar{T_Z}$, i.e., $u \oplus {}^{\bullet}\epsilon\ [\epsilon\rangle\ u \oplus \epsilon^{\bullet}$.*

A firing can be (i) the execution of a transition $u \oplus {}^{\bullet}t\ [t\rangle\ u \oplus t^{\bullet}$, with $t \in T_Z$; (ii) the creation of a token by the environment $u\ [+_s\rangle\ u \oplus s$, with $s \in O_Z^+$; (iii) the deletion of a token by the environment $u \oplus s\ [-_s\rangle\ u$, with $s \in O_Z^-$.

Morphisms of open nets. Morphisms of open nets will be defined as standard net morphisms satisfying suitable conditions on the place ranks. Intuitively, a morphism $f : Z_1 \to Z_2$ "inserts" net Z_1 into a larger net Z_2, allowing a place s of Z_1 to be connected to "new" transitions, i.e., transitions in $Z_2 \setminus f(Z_1)$. The condition we impose guarantees that each new connection of s and each open port of $f(s)$ can be mapped to an open port of s.

For reasons discussed in § 3, we define two kinds of morphisms. In the more abstract ones, we impose only a cardinality constraint, while in the more concrete ones we require an explicit mapping relating, for each place s of Z_1, the ports of s to those of $f(s)$. We next formalise the idea of "new connections" of a place.

Definition 4 (in-set and out-set of a place along a morphism). *Given open nets Z_1 and Z_2 and a Petri net morphism $f : N_{Z_1} \to N_{Z_2}$, for each place $s_1 \in S_1$ the* in-set of s_1 along f *is defined as $\mathsf{in}(f)(s_1) = \{{}^{\bullet}f_S(s_1) - f_T({}^{\bullet}s_1)\}$, and similarly the* out-set *is $\mathsf{out}(f)(s_1) = \{f_S(s_1)^{\bullet} - f_T(s_1{}^{\bullet})\}$. This defines the functions $\mathsf{in}(f), \mathsf{out}(f) : S_1 \to \mathbf{2}^{T_2}$.*

The functions $\#\mathsf{in}(f), \#\mathsf{out}(f) : S_1 \to \mathbb{N}$ are defined, respectively, as $\#\mathsf{in}(f)(s_1) = |\mathsf{in}(f)(s_1)|$ and $\#\mathsf{out}(f)(s_1) = |\mathsf{out}(f)(s_1)|$.

Definition 5 (open net morphisms). *An* open net morphism $f : Z_1 \to Z_2$ *is a Petri net morphism $f : N_{Z_1} \to N_{Z_2}$ such that*

$$\textit{(i)}\ \#\mathsf{in}(f) + o_2^+ \circ f_s \le o_1^+\ \textit{and} \qquad \textit{(ii)}\ \#\mathsf{out}(f) + o_2^- \circ f_s \le o_1^- .$$

A morphism f is called an open net embedding *if both f_T and f_S are injective.*

Intuitively, condition (i) requires that the number of new incoming transitions added to $s \in S_1$ in the target net Z_2 plus the input connections which are still allowed for $f_S(s)$ in Z_2 must be bounded by the maximum number of allowed input connections for s. Examples of open net embeddings can be found in Fig. 1. The mappings are those suggested by the labelling of the nets.

Definition 6 (concrete morphisms). *Let Z_1 and Z_2 be open nets. A* concrete open net morphism $\mathbf{f} : Z_1 \to Z_2$ *is a pair $\mathbf{f} = \langle f, \{f_{s_1}\}_{s_1 \in S_1}\rangle$, where $f : N_{Z_1} \to N_{Z_2}$ is a Petri net morphism and for any $s_1 \in S_1$, f_{s_1} consists of a pair of partial surjections $f_{s_1}^x : p^x(s_1) \to p^x(f(s_1))$ for $x \in \{+, -\}$, consistent with f, i.e., satisfying, for any $t \in {}^{\bullet}s_1$, $f_{s_1}^+(t) = f(t)$ and for any $t \in s_1{}^{\bullet}$, $f_{s_1}^-(t) = f(t)$.*

A morphism $\mathbf{f}$ *is called an* open net embedding *if all components are injective.*

As anticipated, concrete morphisms explicitly relate, for each place $s \in S_1$, the ports of $f(s)$ and of s, using the component f_s. In the sequel, instead of f_s^+ and f_s^-, when place s is clear from the context, we will often write f^+ and f^-. Moreover, when defining f_s^x we will only specify its values on the open ports $[o^x_{Z_1}(s)]$, which, as f_s^x must be consistent with f, completely determines f_s^x.

As expected, the two notions of morphism just introduced determine two categories related by an obvious forgetful functor. In fact, given a concrete morphism $\mathbf{f} = \langle f, \{f_{s_1}\}_{s_1 \in S_1} \rangle \colon Z_1 \to Z_2$ it is straightforward to check that the Petri net morphism f satisfies conditions (i) and (ii) of Definition 5.

Definition 7 (open nets categories). *We denote by* $\mathbf{ONet}_r$ *the category of ranked open nets and open net morphisms, and by* $\mathbf{ONet}_c$ *the category having the same objects and concrete open net morphisms as arrows.*

Furthermore, we denote by $U \colon \mathbf{ONet}_c \to \mathbf{ONet}_r$ *the forgetful functor which is the identity on objects, and acts on an arrow* $\mathbf{f} = \langle f, \{f_s\} \rangle$ *as* $U(\mathbf{f}) = f$.

Sometimes, categories $\mathbf{ONet}_c$ and $\mathbf{ONet}_r$ will be referred to as the *concrete* and the *abstract* category of (ranked) open nets, respectively.

The category of basic open nets introduced in [1] is (isomorphic to) the full subcategory of $\mathbf{ONet}_r$ including all the nets Z such that for any place s we have $o^x_Z(s) \in \{0, \omega\}$, i.e., either s is closed or it allows for an unbounded number of connections. In the following this subcategory will be referred to as $\mathbf{ONet}$.

3 Composing Open Nets

Intuitively, two open nets Z_1 and Z_2 are composed by specifying a common subnet Z_0, and then by joining the two nets along Z_0. Composition will be characterised as a pushout in the concrete category of open nets $\mathbf{ONet}_c$. But since for specification purposes the abstract category $\mathbf{ONet}_r$ is often more appropriate and easier to deal with, next we will focus on the notion of composition induced on such category by the colimit based composition in $\mathbf{ONet}_c$.

Composition is possible if it respects the interface of the involved nets. This is formalised by the notion of composability of a span of embeddings in $\mathbf{ONet}_c$.

Definition 8 (composable span in $\mathbf{ONet}_c$). *A span of embeddings* $\mathbf{f}_1 : Z_0 \to Z_1$ *and* $\mathbf{f}_2 : Z_0 \to Z_2$ *in* $\mathbf{ONet}_c$ *is called* composable *if, for any* $s_0 \in S_0$

1. *for all* $i \in [o^+_{Z_0}(s_0)]$, *if* $f_1^+(i) \in \mathsf{in}(f_1)(s_0)$ *then* $f_2^+(i) \in [o^+_{Z_2}(f_2(s_0))]$
2. *for all* $i \in [o^-_{Z_0}(s_0)]$, *if* $f_1^-(i) \in \mathsf{out}(f_1)(s_0)$ *then* $f_2^-(i) \in [o^-_{Z_2}(f_2(s_0))]$

plus the analogous conditions, exchanging the roles of Z_1 *and* Z_2.

Intuitively, condition (1) says that, given a place s_0 and an open input port $i \in [o^+_{Z_1}(s_0)]$, if according to $\mathbf{f}_1$ the transition $f_1^+(i) \in \mathsf{in}(f_1)(s_0)$ is going to be attached to this port, then the corresponding port in Z_2 must be open, i.e., $f_2^+(i) \in [o^+_{Z_2}(f_2(s_0))]$. The other conditions are analogous.

Given a concrete composable span $Z_1 \overset{\mathbf{f}_1}{\leftarrow} Z_0 \overset{\mathbf{f}_2}{\rightarrow} Z_2$, the *composition* of Z_1 and Z_2 along Z_0 is the open net Z_3 (see Fig. 2) obtained as the pushout of $\mathbf{f}_1$ and $\mathbf{f}_2$, which exists by the next result.

$$\begin{tikzcd}
& Z_0 \arrow[dl, "f_1"'] \arrow[dr, "f_2"] & \\
Z_1 \arrow[dr, "g_1"'] & & Z_2 \arrow[dl, "g_2"] \\
& Z_3 &
\end{tikzcd}$$

Fig. 2. Composition of ranked open nets

Proposition 9 (pushout in $\mathbf{ONet}_c$). *A span of embeddings $\mathbf{f}_1 : Z_0 \to Z_1$, $\mathbf{f}_2 : Z_0 \to Z_2$ in $\mathbf{ONet}_c$ is composable if and only if it has a pushout $Z_1 \xrightarrow{\mathbf{g}_1} Z_3 \xleftarrow{\mathbf{g}_2} Z_2$ in $\mathbf{ONet}_c$, whose underlying diagram is a pushout in* **Net**.

The construction of the pushout of a composable span in $\mathbf{ONet}_c$ turns out to be quite complex and it is not reported for space limitations, but the intuition is simple. Firstly, the underlying net N_{Z_3} is obtained as the pushout of N_{Z_1} and N_{Z_2} along N_{Z_0} in **Net**. Next, if a place is not in Z_0, then in the pushout it maintains exactly its ports. Instead, for a place s in Z_0, in the pushout the ports of the image of s are obtained by taking the pushout of the ports of the images of s in Z_1 and Z_2. Since mappings between ports can be partial, open ports can disappear. A port is open only if it is open in both nets Z_1 and Z_2.

The notions of composability of spans and of composition between nets can be transferred to the abstract category via the forgetful functor $U \colon \mathbf{ONet}_c \to \mathbf{ONet}_r$. More interestingly, these notions can be defined also directly at the abstract level, by referring only to the ranks of places of the involved nets. Thanks to this fact, in the rest of the paper we will be able to work in the abstract category only, which provides a simpler and natural framework to be used for specification purposes. Still, we stress here that we defined the composition of nets in the concrete category first, because the corresponding notion in the abstract category cannot be characterized by a universal property as a pushout.

Given a pair of embeddings $f_1 : Z_0 \to Z_1$ and $f_2 : Z_0 \to Z_2$ in $\mathbf{ONet}_r$, we say that they are *composable* if there exists a composable span of embeddings $\mathbf{f}_1 : Z_0 \to Z_1$ and $\mathbf{f}_2 : Z_0 \to Z_2$ in $\mathbf{ONet}_c$ such that $U(\mathbf{f}_1) = f_1$ and $U(\mathbf{f}_2) = f_2$.

Fact 10 (composable span in $\mathbf{ONet}_r$). A span of embeddings $f_1 : Z_0 \to Z_1$ and $f_2 : Z_0 \to Z_2$ in $\mathbf{ONet}_r$ is composable if and only if

1. $\#\mathsf{in}(f_1) \leq o^+_{Z_2} \circ f_2$ and $\#\mathsf{out}(f_1) \leq o^-_{Z_2} \circ f_2$;
2. $\#\mathsf{in}(f_2) \leq o^+_{Z_1} \circ f_1$ and $\#\mathsf{out}(f_2) \leq o^-_{Z_1} \circ f_1$.

Intuitively, the first half of condition (1) requires that the number of input connections which are added to each place s of Z_0 by f_1, namely $\#\mathsf{in}(f_1)(s)$, is bounded by the number of additional input connections allowed for $f_2(s)$ in Z_2, i.e., $o^+_{Z_2}(f_2(s))$. The remaining conditions are similar.

Now, given a composable span of embeddings $f_1 : Z_0 \to Z_1$ and $f_2 : Z_0 \to Z_2$ in $\mathbf{ONet}_r$, let $\langle \mathbf{f}_1, \mathbf{f}_2 \rangle$ be any pair of composable embeddings in $\mathbf{ONet}_c$ such that $U(\mathbf{f}_1) = f_1$ and $U(\mathbf{f}_2) = f_2$. Then the *composition* of Z_1 and Z_2 along Z_0

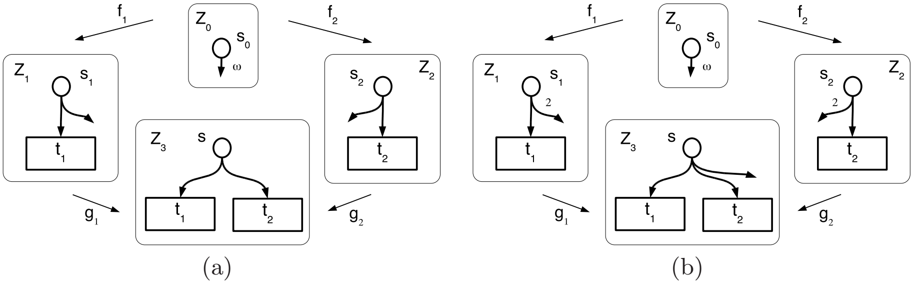


Fig. 3. Composing ranked open nets

in $\mathbf{ONet}_r$ is defined exactly as their composition in $\mathbf{ONet}_c$, i.e., as the pushout object of $\mathbf{f}_1$ and $\mathbf{f}_2$ in $\mathbf{ONet}_c$. It can be shown that this definition is well given, and that it can be characterized as follows.

Fact 11 (composition in $\mathbf{ONet}_r$). Let $f_1 : Z_0 \to Z_1$ and $f_2 : Z_0 \to Z_2$ be a span of embeddings in $\mathbf{ONet}_r$. Compute the pushout of the corresponding diagram in the category **Net** obtaining the net N_{Z_3} and the morphisms g_1 and g_2. For $i \in \{1, 2\}$, define $res_i^+(s_3) = o_{Z_i}^x(s_i) - \#\mathsf{in}(g_i)(s_i)$ if there is some $s_i \in S_i$ such that $g_i(s_i) = s_3$ and $res_i^+(s_3) = \omega$, otherwise.[1] The function res_i^- is defined in a dual way. Then take, for $x \in \{+, -\}$

$$o_{Z_3}^x = \min\{res_1^x, res_2^x\}$$

Then Z_3 (with morphisms g_1 and g_2) is the composition along Z_0 of f_1 and f_2.

Intuitively, for a place $s_3 = g_i(s_i)$, the value $res_i^+(s_i)$ is obtained by subtracting from the number of connections allowed for s_i, i.e., $o_{Z_i}^+(s_i)$, the number of connections which have been added as an effect of the composition, i.e., $\#\mathsf{in}(g_i)(s_i)$. In other words $res_i^+(s_i)$ is the residual number of allowed connections. When joining two places, the number of allowed connections for the resulting place will be the minimum among the residuals of the two original places.

Two simple examples of composition can be found in Fig. 3. It is worth explaining why, for example, diagram (a) is not a pushout in $\mathbf{ONet}_r$. In fact, since Z_1 and Z_2 are isomorphic, we can close the span $Z_1 \overset{f_1}{\leftarrow} Z_0 \overset{f_2}{\rightarrow} Z_2$ with arrows $Z_1 \overset{id}{\rightarrow} Z_1 \overset{\cong}{\leftarrow} Z_2$ obtaining a commutative square in $\mathbf{ONet}_r$, but there is no mediating morphism $Z_3 \to Z_1$ because the counter-image of an open place cannot be closed. For a more complex example see Fig. 1, where two nets Z_1 and Z_2 representing the planning of a trip and the buying of the ticket, respectively, are composed. Note, e.g., that place itinerary in Z_2 is output open with rank 3 and input open with rank 1, as needed for adding the connections in Z_1.

[1] Observe that res_i^+ is well-defined since g_i is injective.

4 Processes of Open Nets

A process of an open net is an open net itself, satisfying suitable acyclicity and conflict freeness requirements, together with a mapping to the original net.

The open net underlying a process is an open *occurrence* net, namely an open net K such that the underlying net N_K is an ordinary occurrence net, with some additional conditions on open places. Fig. 6 shows some examples of occurrence nets. The open places in the occurrence net are intended to represent occurrences of tokens which are produced or consumed by the environment in the considered computation. Hence, input open places must satisfy $o^+(s) = 1$ and additionally they must be minimal. In fact, an input open place with $o^+(s) > 1$ would represent a token possibly produced by two different transitions in the environment; similarly an input open place in the post-set of some transition would represent a token which can be produced either internally or by some transition in the environments. In both cases the situation would correspond to a backward conflict and it would prevent one to interpret the place as a token occurrence. Instead, an output open place can be in the pre-set of a transition, as it happens for place itinerary in the open occurrence nets K_1 and K_2 of Fig. 6, and it might be that $o^-_Z(s) > 1$. The idea is that the token occurrence represented by place s can be consumed either by transition t or by two or more occurrences of transitions in the environment.

For a Petri net $N = (S, T, \sigma, \tau)$ the *causality relation* $<_N \subseteq (S \cup T)^2$ is the least transitive relation such that $x <_N y$ if $y \in x^\bullet$. Moreover, the *conflict relation* $\#_N \subseteq (S \cup T)^2$ is the least symmetric relation generated by the rules:

$$\frac{{}^\bullet t \cap {}^\bullet t' \neq \emptyset \qquad t \neq t' \qquad t, t' \in T}{t \,\#_N\, t'} \qquad\qquad \frac{x \,\#_N\, y \qquad y <_N y'}{x \,\#_N\, y'} \ \text{(hereditarity)}$$

These definitions lift to open nets by considering the underlying net. We will omit the subscripts when clear from the context.

Definition 12 (open occurrence net). *An* open occurrence net *is an open net K such that*

1. *${}^\bullet t$ and $t^\bullet$ are sets rather than proper multisets, for each transition $t \in T$;*
2. *the causality relation $<_K$ is a finitary strict partial order;*
3. *the conflict relation $\#_K$ is irreflexive;*
4. *there are no* backward conflicts, *i.e., $deg^+(s) \leq 1$ for each place $s \in S$.*

Notice that the net N_K underlying an open occurrence net is an occurrence net according to the standard definition.

We next introduce the notion of process for open nets.

Definition 13 (open net process). *A* process *of an open net Z is a mapping $\pi : K \to Z$ where K is an open occurrence net and $\pi : N_K \to N_Z$ is a Petri net morphism, such that $\pi_S(O^+_K) \subseteq O^+_Z$ and $\pi_S(O^-_K) \subseteq O^-_Z$.*

Note that the mapping from the occurrence net K to the original net Z, is *not* an open net morphism in general. In fact, the process mapping, differently from open net morphisms, must be a simulation, i.e., it must preserve the behaviour. To this aim the image of an open place in K must be an open place in Z, since tokens can be produced (consumed) by the environment only in input (output) open places of Z. Instead, there is no relation between the rank of open places in the occurrence net and in the net Z since a token in an open place can be consumed by distinct occurrences of the same transition in the environment.

We next introduce the category of processes, where objects are processes and arrows are pairs of open net morphisms.

Definition 14 (category of processes). *We denote by* **Proc** *the category where objects are processes and given two processes $\pi_0 : K_0 \to Z_0$ and $\pi_1 : K_1 \to Z_1$, an arrow $\psi : \pi_0 \to \pi_1$ is a pair of open net morphisms $\psi = \langle \psi_Z : Z_0 \to Z_1, \psi_K : K_0 \to K_1 \rangle$ such that the diagram on the right (indeed the underlying diagram in* **Net***) commutes.*

$$\begin{tikzcd}
K_0 \arrow[r, "\psi_K"] \arrow[d, "\pi_0"'] & K_1 \arrow[d, "\pi_1"] \\
Z_0 \arrow[r, "\psi_Z"'] & Z_1
\end{tikzcd}$$

4.1 Projecting Behaviours along Embeddings

Since open net morphisms are designed to capture the idea of "insertion" of a net into a larger one, they are expected to "reflect" the behaviour in the sense that given $f : Z_0 \to Z_1$, the behaviour of Z_1 can be projected along the morphism to the behaviour of Z_0. As in [1], this intuition can be formalised for open net embeddings by showing how a process of Z_1, as defined before, can be projected along f giving a process of Z_0. Intuitively, each possible computation in Z_1 can be "projected" to Z_0, by considering only the part of the computation of the larger net which is visible in the smaller net. Ranks are defined correspondingly.

Definition 15 (projection of a process). *Let $f : Z_0 \to Z_1$ be an open net embedding and let $\pi_1 : K_1 \to Z_1$ be a process of Z_1. A* projection of π_1 along f, *is a pair $\langle \pi_0, \psi \rangle$ where $\pi_0 : K_0 \to Z_0$ is a process of Z_0 and $\psi : \pi_0 \to \pi_1$ is an arrow in* **Proc**, *constructed as follows. Consider the pullback of π_1 and f in* **Net**, *thus obtaining the net morphisms π_0 and ψ_K (see the diagram on the right). Then K_0 is obtained by taking N_{K_0} as underlying net, and defining*

$$\begin{tikzcd}
N_{K_1} \arrow[r, "\pi_1"] & N_{Z_1} \\
N_{K_0} \arrow[u, dotted, "\psi_K"] \arrow[r, dotted, "\pi_0"'] & N_{Z_0} \arrow[u, "f"']
\end{tikzcd}$$

$$o^+_{K_0} = o^+_{K_1} \circ \psi_K + \#\mathsf{in}(\psi_K) \quad \textit{and} \quad o^-_{K_0} = o^-_{K_1} \circ \psi_K + \#\mathsf{out}(\psi_K)$$

(i.e., by opening the places as least as possible to make $\psi_K : K_0 \to K_1$ an open net morphism) and $\psi = \langle \psi_K, f \rangle$.

5 Composing Non-deterministic Processes

Consider a composition diagram in $\mathbf{ONet}_r$, as in Fig. 2, where f_1 and f_2 are open net embeddings. One would like to establish a clear relationship among the

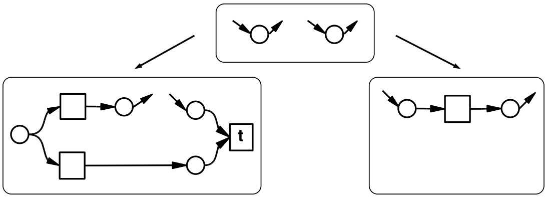


Fig. 4. Transition t would be in self-conflict in the composition

behaviours of the involved nets. Roughly, we would like that the behaviour of Z_3 could be constructed "compositionally" out of the behaviours of Z_1 and Z_2.

In ([1]) we have shown that in the setting of basic open nets this can be done only for deterministic processes. Here we show how, in the setting of ranked open nets, the result extends to general, possibly non-deterministic processes. Given two processes π_1 of Z_1 and π_2 of Z_2 which "agree" on Z_0, one can construct a process π_3 of Z_3 by amalgamating π_1 and π_2. Vice versa, each process π_3 of Z_3 can be projected over two processes π_1 and π_2 of Z_1 and Z_2, which can be amalgamated to produce π_3 again. Hence, all and only the processes of Z_3 can be obtained by amalgamating the processes of the components Z_1 and Z_2.

5.1 Composition of Non-deterministic Occurrence Open Nets

A basic step towards the composition operation is the formalisation of the intuitive idea of processes of different nets which "agree" on a common part. Concretely, this amounts to identify suitable conditions which ensure that the composition of occurrence open nets exists and produces a net in the same class.

First, given a span $K_1 \overset{f_1}{\leftarrow} K_0 \overset{f_2}{\rightarrow} K_2$ we introduce the notion of causality relation induced by K_1 and K_2 over K_0. When the two nets are composed their causality relations get "fused". Hence, to ensure that the resulting net is again an occurrence net, the induced causality must be a strict partial order.

Definition 16 (induced causality). *Let $K_1 \overset{f_1}{\leftarrow} K_0 \overset{f_2}{\rightarrow} K_2$ be a span of embeddings in $\mathbf{ONet}_r$, where K_i $(i \in \{0, 1, 2\})$ are occurrence open nets. The relation of causality $<_{1,2}$ induced over K_0 by K_1 and K_2, through f_1 and f_2 is the least transitive relation such that for any x_0, y_0, if $f_1(x_0) <_{K_1} f_1(y_0)$ or $f_2(x_0) <_{K_2} f_2(y_0)$ then $x_0 <_{1,2} y_0$.*

When composing non-deterministic occurrence nets, which can include mutual exclusive branches of computation, we must also avoid that transitions becomes non-firable due to the creation of self-conflicts. For example, Fig. 4 shows a span where the induced causality is a strict partial order, but there would be a self-conflict on t in the composed occurrence net. Hence t would not be firable in any computation of the net.

To this aim, we introduce new relations, called *anti-causality* and *anti-conflict.* Intuitively, two items x and y in K are related by anti-causality (anti-conflict) if, to ensure the firability of each transition in the net, x and y *must* remain causally unrelated (not in conflict, resp.) when K is composed with other nets. Then the idea is to avoid compositions which can lead to situations in which two items are related both by a relation and by the corresponding anti-relation.

Definition 17 (anti-relations). *Let K be an occurrence open net. The* anti-causality $\neg{<_K}$ *and* anti-conflict $\neg\#_K$ *relations over $(S \cup T)^2$ are defined by the following rules (subscripts are omitted as clear from the context):*

$$\frac{}{x \neg\# x \quad x \neg< x}\ (\mathsf{anti1}) \qquad \frac{x \neg\# y \quad x' < x}{x' \neg\# y}\ (\mathsf{anti2})$$

$$\frac{x \neg\# y \quad x \# y'}{y' \neg< y}\ (\mathsf{anti3}) \qquad \frac{x \neg\# y}{y \neg\# x}\ (\mathsf{anti4})$$

The rules have a clear interpretation. Rule (anti1) states that the each single item must remain concurrent, while rules (anti2) and (anti3) are obtained by "reverting" the rule which expresses hereditarity of conflict w.r.t. causality. Finally, (anti4) states that $\neg\#$ is symmetric.

Given an open net morphism $f_1 : K_0 \to K_1$, where K_0 and K_1 are occurrence nets, in the following we will use the symbols $<_1$, $\#_1$, $\neg\#_1$ and $\neg<_1$ to denote the *projection over* K_0 of the corresponding relations over K_1, i.e., for any $r \in \{<, \#, \neg\#, \neg<\}$ and x_0, y_0 in K_0 we will write

$$x_0\ r_1\ y_0 \quad \text{iff} \quad f_1(x_0)\ r_{K_1}\ f_1(y_0)$$

Given a span of occurrence open nets $K_1 \xleftarrow{f_1} K_0 \xrightarrow{f_2} K_2$ we next define the conflict relation and the anti-relations induced over the net K_0 by K_1 and K_2, through f_1 and f_2. This has been already done for causality in Definition 16, where induced causality $<_{1,2}$ is defined as the transitive closure of $<_1 \cup <_2$.

Definition 18 (induced relations). *Let $K_1 \xleftarrow{f_1} K_0 \xrightarrow{f_2} K_2$ be a span in* $\mathbf{ONet}_r$*, where K_i ($i \in \{0,1,2\}$) are occurrence open nets. The conflict relation and the anti-relations induced over K_0 by K_1 and K_2, through f_1 and f_2 are as follows.*

For x_0, y_0 in K_0, let $x_0 \searrow_1 y_0$ be a shortcut for $x_0 <_1 y_0$ and there is no z_0 such that $x_0 <_{K_0} z_0 \leq_1 y_0$. Observe that in this case x_0 must be a place, connected to y_0 through a chain of transitions in K_1, but not in K_0. The notation $x_0 \searrow_2 y_0$ is defined in the dual way.

- ***induced conflict** $\#_{1,2}$: The relation $\#_{1,2}$ over K_0 is the least relation, hereditary w.r.t. $<_{1,2}$ such that, for any x_0, y_0,*
  1) if $x_0 \#_1 y_0$ or $x_0 \#_2 y_0$ then $x_0 \#_{1,2} y_0$.
  2) if $x_0 \searrow_1 y_0$ and $x_0 \searrow_2 z_0$ then $y_0 \#_{1,2} z_0$.
- ***induced anti-relations** $\neg<_{1,2}$ **and** $\neg\#_{1,2}$: The relations $\neg\#_{1,2}$ and $\neg<_{1,2}$ over K_0 are defined as the least relations such that for x_0, y_0, for $i \in \{1,2\}$, if $x \neg\#_i y$ then $x \neg\#_{1,2} y$, and similarly, if $x \neg<_i y$ then $x \neg<_{1,2} y$, and closed under rules* (anti1) – (anti4)*.*

Now we can identify the conditions which guarantee that the composition of two occurrence open nets is still an occurrence open net.

Definition 19 (consistent span). *A span $K_1 \overset{f_1}{\leftarrow} K_0 \overset{f_2}{\rightarrow} K_2$ of occurrence open nets is* consistent *if it is composable in* $\mathbf{ONet}_r$ *and for any x_0, y_0 in K_0*

1. $x_0 \neg{<_{1,2}}\, y_0 \;\Rightarrow \neg(x_0 <_{1,2} y_0)$ and $x_0 \neg\#_{1,2}\, y_0 \;\Rightarrow \neg(x_0 \#_{1,2}\, y_0)$;
2. for $i, j \in \{1, 2\}$, $i \neq j$, we have that $x_0 \neg\#_i\, y_0$ implies $\neg(x_0 \searrow_j y_0)$.

Condition (1) just requires that each anti-relation does not intersect the corresponding relation. Condition (2), instead, just imposes that two anti-conflictual places in K_1 are never connected by a chain of transitions in K_2 (and vice versa), otherwise in the composition one would get a self-conflict.

We can now show that the composition in $\mathbf{ONet}_r$ of a consistent span of occurrence nets produces an occurrence net. We first need a preliminary result.

Lemma 20. *Let $K_1 \overset{f_1}{\leftarrow} K_0 \overset{f_2}{\rightarrow} K_2$ be a composable span of embeddings in $\mathbf{ONet}_r$, where K_i ($i \in \{0, 1, 2\}$) are occurrence open nets, and let $K_1 \overset{g_1}{\rightarrow} K_3 \overset{g_2}{\leftarrow} K_2$ be the composition. Then for any x_0, y_0 in K_0, if we let $x_3 = g_1(f_1(x_0)) = g_2(f_2(x_0))$ and $y_3 = g_1(f_1(y_0)) = g_2(f_2(y_0))$, we have*

1. $x_0 <_{1,2} y_0$ iff $x_3 <_{K_3} y_3$;
2. $x_0 \#_{1,2}\, y_0$ iff $x_3 \#_{K_3}\, y_3$;
3. $x_0 \neg\#_{1,2}\, y_0$ iff $x_3 \neg\#_{K_3}\, y_3$;
4. $x_0 \neg{<_{1,2}}\, y_0$ iff $x_3 \neg{<_{K_3}}\, y_3$.

Proposition 21. *In the hypotheses of Lemma 20 above, $K_1 \overset{f_1}{\leftarrow} K_0 \overset{f_2}{\rightarrow} K_2$ is a consistent span* iff *the composition K_3 is an occurrence open net.*

5.2 Amalgamating Non-deterministic Processes

For the rest of this section we refer to a fixed composition in $\mathbf{ONet}_r$, as in Fig. 2, where f_1 and f_2 are composable open net embeddings. Two processes π_1 of Z_1 and π_2 of Z_2 can be amalgamated when they agree on the common subnet Z_0.

Definition 22 (agreement of non-deterministic processes). *We say that two non-deterministic processes $\pi_1 : K_1 \to Z_1$ and $\pi_2 : K_2 \to Z_2$* agree *on Z_0 if there are projections $\langle \pi_0, \psi_K^i \rangle$ along f_i of π_i for $i \in \{1, 2\}$ such that the span $K_1 \overset{\psi_K^1}{\leftarrow} K_0 \overset{\psi_K^2}{\rightarrow} K_2$ is consistent and, for any s_0 in K_0, if $s_3 = f_i(g_i(\pi_0(s_0)))$ is the corresponding place in Z_3, the following holds:*

$$\text{if} \quad \#\mathsf{out}(\psi_K^1)(s_0) + \#\mathsf{out}(\psi_K^2)(s_0) < o^-_{K_0}(s_0) \quad \text{then} \quad s_3 \in O^-_{Z_3}. \qquad (1)$$

In this case $\langle \pi_0, \psi_K^1 \rangle$, $\langle \pi_0, \psi_K^2 \rangle$ are called agreement projections *for π_1 and π_2.*

Intuitively, the two processes agree if they have the same projection over Z_0. Additionally, as required by condition (1), if, for a place s_0 in K_0, the number of external events that can consume the token in s_0 exceeds the events provided by Z_1 and Z_2 then the corresponding place in Z_3 must be open.

$$\begin{tikzcd}
 & & K_0 \arrow[dll, "\psi^1_K"'] \arrow[drr, "\psi^2_K"] \arrow[d, "\pi_0"] & & \\
K_1 \arrow[d, "\pi_1"] \arrow[drr, dotted, "\phi^1_K"' near end] & & Z_0 \arrow[dll, "f_1"'] \arrow[drr, "f_2"] & & K_2 \arrow[d, "\pi_2"] \arrow[dll, dotted, "\phi^2_K" near end] \\
Z_1 \arrow[drr, "g_1"'] & & K_3 \arrow[d, dotted, "\pi_3"] & & Z_2 \arrow[dll, "g_2"] \\
 & & Z_3 & &
\end{tikzcd}$$

Fig. 5. Amalgamation of open net processes

Definition 23 (process amalgamation). *Let $\pi_i : K_i \to Z_i$ ($i \in \{0,1,2,3\}$) be non-deterministic processes and let $\langle \pi_0, \psi^1_K \rangle$ and $\langle \pi_0, \psi^2_K \rangle$ be agreement projections of π_1 and π_2 along f_1 and f_2 (see Fig. 5). We say that π_3 is an* amalgamation *of π_1 and π_2, written $\pi_3 = \pi_1 +_{\psi^1_K, \psi^2_K} \pi_2$, if there are projections $\langle \pi_1, \phi^1 \rangle$ and $\langle \pi_2, \phi^2 \rangle$ of π_3 over Z_1 and Z_2, respectively, such that the upper square is a composition in* $\mathbf{ONet}_r$.

We next give a more constructive characterisation of process amalgamation, which also proves that the result is unique up to isomorphism.

Lemma 24 (amalgamation construction). *Let $\pi_1 : K_1 \to Z_1$ and $\pi_2 : K_2 \to Z_2$ be non-deterministic processes that agree on Z_0, and let $\langle \pi_0, \psi^1_K \rangle$ and $\langle \pi_0, \psi^2_K \rangle$ be corresponding agreement projections. Then the* amalgamation *$\pi_1 +_{\psi^1_K, \psi^2_K} \pi_2$ is a process $\pi_3 : K_3 \to Z_3$, where net K_3 is obtained as the composition in* $\mathbf{ONet}_r$ *of $\psi^1_K : K_0 \to K_1$ and $\psi^2_K : K_0 \to K_2$ and the process mapping $\pi_3 : K_3 \to Z_3$ is uniquely determined by the universal property of the underlying pushout diagram in* **Net** *(see Fig. 5). Hence $\pi_1 +_{\psi^1_K, \psi^2_K} \pi_2$ is unique up to isomorphism.*

As an example, in Fig. 6 a process for the net Z_3 of Fig. 1 is obtained as the amalgamation of processes of the component nets. The process for Z_1 represents a reservation activity, which can succeed after two attempts or can be finally cancelled. In the process for Z_2 two possible itineraries are visible: the first one can only be discarded (used by the environment) while the second one can also trigger a payment, thus resulting in a ticket. Composing the two processes one gets a full booking process for net Z_3.

We next show that each non-deterministic process of a composed net arises as the amalgamation of non-deterministic processes of the components.

Lemma 25 (process decomposition). *Let $\pi_3 : K_3 \to Z_3$ be a process of Z_3 and, for $i \in \{1,2\}$, let $\langle \pi_i, \phi^i \rangle$ be projections of π_3 along g_i. Then there are agreement projections $\langle \pi_0, \psi^1_K \rangle$, $\langle \pi_0, \psi^2_K \rangle$ of π_1, π_2 such that $\pi_3 \cong \pi_1 +_{\psi^1_K, \psi^2_K} \pi_2$.*

As a consequence we finally have our main result.

Theorem 26 (compositionality for non-deterministic processes). *All and only the non-deterministic processes of Z_3 can be obtained as amalgamations of processes of Z_1 and Z_2 which agree on Z_0.*

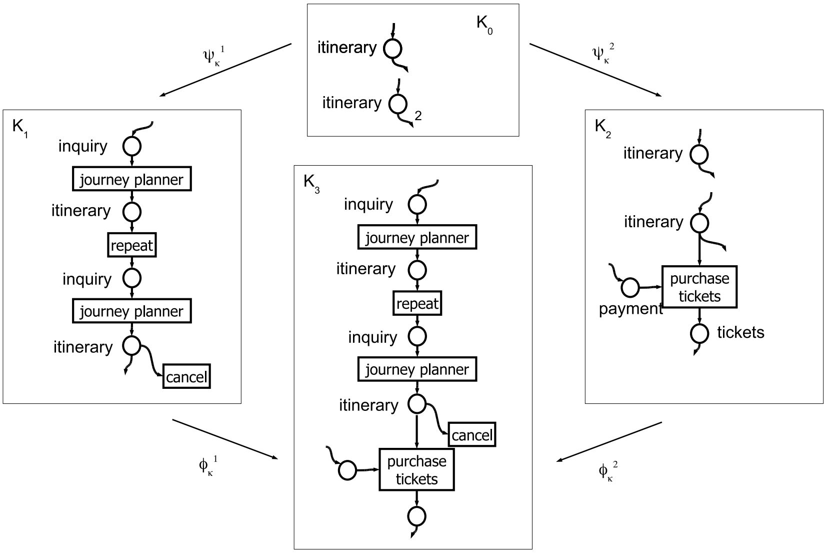


Fig. 6. An example of process amalgamation

6 Conclusions and Future Work

We have introduced a compositional semantics based on non-deterministic processes for ranked open nets, an extension of the basic open net model of ([1]) where it is possible to specify, for open places, the maximum number of allowed connections. The composition operation is characterised as a pushout in a category of ranked open nets with concrete morphisms. The notion of agreement between processes of different sub-components, which is a requirement for process composition, builds upon a theory of anti-relations (i.e., anti-causality and anti-conflict) which could have an interest for Petri nets in general.

We believe that a theory of non-deterministic processes for open nets can represent a starting point for a modular verification of open nets based on finite prefixes of the unfolding ([10]). There are obvious difficulties, e.g., the fact that open nets are always infinite state (whenever they have at least one input open place). However the "regularity" of the state space suggests the possibility of undertaking a symbolic approach, for which analogous work for standard Petri nets, like ([6]), could provide an inspiration.

We foresee also potential outcomes in the setting of graph transformation systems. In fact graph transformation systems can be seen as generalisation of Petri nets, and it has been often productive to focus first in the latter simpler setting. The notion of openness ([7], [8]) as well as the notion of processes ([5]) have already been studied in the setting of graph transformation, however until now there have been no attempts to combine them. The present work can be a first step in this direction.

References

1. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. Mathematical Structures in Computer Science 15(1), 1–35 (2005)
2. Baldan, P., Corradini, A., Ehrig, H., Heckel, R., König, B.: Bisimilarity and behaviour-preserving reconfigurations of open Petri nets. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 126–142. Springer, Heidelberg (2007)
3. Basten, T.: In terms of nets: System design with Petri nets and process algebra. PhD thesis, Eindhoven University of Technology (1998)
4. Bonchi, F., Brogi, A., Corfini, S., Gadducci, F.: A behavioural congruence for web services. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 240–256. Springer, Heidelberg (2007)
5. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae 26, 241–265 (1996)
6. Desel, J., Juhás, G., Neumair, C.: Finite unfoldings of unbounded Petri nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 157–176. Springer, Heidelberg (2004)
7. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 151–166. Springer, Heidelberg (2004)
8. Heckel, R.: Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems. PhD thesis, Technische Universität Berlins (1998)
9. Kindler, E.: A compositional partial order semantics for Petri net components. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 235–252. Springer, Heidelberg (1997)
10. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–174. Springer, Heidelberg (1993)
11. Milner, R.: Bigraphical reactive systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 16–35. Springer, Heidelberg (2001)
12. Nielsen, M., Priese, L., Sassone, V.: Characterizing Behavioural Congruences for Petri Nets. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 175–189. Springer, Heidelberg (1995)
13. Priese, L., Wimmel, H.: A uniform approach to true-concurrency and interleaving semantics for Petri nets. Theoretical Computer Science 206(1–2), 219–256 (1998)
14. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (1985)
15. Sassone, V., Sobociński, P.: A congruence for Petri nets. In: Proc. of PNGT 2004. ENTCS, pp. 107–120. Elsevier, Amsterdam (2005)

16. van der Aalst, W.: The application of Petri nets to workflow management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
17. van der Aalst, W.: Interorganizational workflows: An approach based on message sequence charts and Petri nets. System Analysis and Modeling 34(3), 335–367 (1999)
18. Zhang, G., Meng, F., Jiang, C., Pang, J.: Using Petri Net to Reason with Rule and OWL. In: Proc. of CIT 2006, IEEE Computer Society Press, Los Alamitos (2006)

Attributed Graph Constraints

Fernando Orejas*

Dpt. L.S.I., Universitat Politècnica de Catalunya,
Campus Nord, Mòdul Omega,
Jordi Girona 1-3,
08034 Barcelona, Spain
orejas@lsi.upc.edu

Abstract. Graph constraints were introduced in the area of graph transformation, in connection with the notion of (negative) application conditions, as a form to limit the applicability of transformation rules. However, in a previous paper, we showed that graph constraints may also play a significant role in the area of visual software modelling or in the specification and verification of semi-structured documents or websites (i.e. HTML or XML sets of documents). In that paper we present a sound and complete proof system for reasoning with this kind of constraints. Those results apply, in principle, to any category satisfying some given properties, but the category of (typed) attributed graphs does not satisfy these properties. In particular, the proof rules introduced for reasoning with standard graph constraints allow us to infer infinitary formulas, making the logic incomplete. In addition, using the straightforward generalization of standard graph constraints, there is no obvious way of stating properties about the attributes of the given graphs.

In this paper we introduce a new formulation for attributed graph constraints. More precisely, the idea is to see these constraints as standard graph constraints whose attributes are just variables, together with a logic formula that expresses properties that must be satisfied by these attributes. Then a proof system, which extends the one introduced in the previous paper, is presented and it is shown to be sound and complete.

1 Introduction

Graph constraints were introduced in the area of graph transformation, in connection with the notion of (negative) application conditions, as a form to limit the applicability of transformation rules [5,7,10,4,8,9]. However, in a previous paper [15], we showed that graph constraints may also play a significant role in the area of visual software modelling or in the specification and verification of semi-structured documents or websites (i.e. HTML or XML sets of documents). For instance, in the area of software modelling graph constraints can be complementary to the use of OCL when defining systems constraints. On the other

[*] This work has been partially supported by the CICYT project (ref. TIN2007-66523) and by the AGAUR grant to the research group ALBCOM (ref. 00516).

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 274–288, 2008.

hand, we consider that graph constraints can be more adequate for the specification of websites than approaches like [1,6], based on extending a fragment of first-order logic with XPath notation, which makes specifications very verbose and, as a consequence, unpleasant to read and to write. In addition, in these approaches, given a specification of a class of documents we can check if a given document satisfies the specification, but it is not studied how to reason about the specifications themselves, for instance for checking their consistency. The reason is probably that they would need to use associative-commutative unification which can be very costly and difficult to implement (in general, two arbitrary terms may have a doubly exponential amount of most general unifiers).

Using graph constraints to describe classes of documents is an approach in a way similar to Schematron [12], a language and a tool that is part of an ISO standard (DSDL: Document Schema Description Languages). The language allows us to specify constraints on XML documents by describing directly XML patterns (using XML) and expressing properties about these patterns. Then, the tool allows us to check if a given XML document satisfies these constraints. Unfortunately, this approach has no proper foundations. Actually, in some sense our work could serve to provide these foundations and to extend it by allowing the use of more general kinds of patterns and by providing deduction capabilities.

Unfortunately, the results in [15] were not defined for attributed graphs, but for standard graphs (without attributes). Those results apply, in principle, to any category satisfying some given properties. However, the category of (typed) attributed graphs does not satisfy these properties. In particular, the proof rules introduced for reasoning with standard graph constraints, when dealing with attributed graphs, allow us to infer infinitary formulas, making the logic incomplete. In addition, using the straightforward generalization of standard graph constraints, there is no obvious way of stating properties about the attributes of the given graphs. This issue is discussed in more detail in Section 3.

In this paper we introduce a new formulation for attributed graph constraints. The new notion of attributed constraint combines a (standard) graph constraint with a formula describing a condition on the attributes of the graphs involved in the constraint. Then a proof system, which extends the one introduced in the previous paper, is presented and it is shown to be sound and complete.

The work that we present is not the first logic to reason about graphs. In particular, with different aims, Courcelle in a series of papers has studied in detail the use of monadic second-order logic (MSOL) to express graph properties (for a survey, see [2]). That logic is quite more powerful than the one that we study in this paper. For instance, we cannot express global properties about graphs (e.g that a graph is connected), but using MSOL we can. Actually, we think that MSOL is too powerful for the kind of applications that we have in mind.

The paper is organized as follows. In the second section we present some basic concepts that are used along the paper. In particular we present the concepts of labeled and attributed graphs and the notions of (standard) graph constraints. Then, in section 3 we introduce attributed graph constraints and present a small example to motivate their use in connection with visual modelling or website specification. In section 4 we describe the framework that we use for checking satisfiability of sets of constraints, which is based on the use of inference rules. Then, in section 5 we present inference rules for the classes of constraints considered, showing their soundness and completeness. Finally, in the conclusion we briefly discuss the results presented.

2 Graphs and Graph Constraints

In this section we present the basic notions that are used in this paper. First we present the basic notions related to attributed graphs, and then we introduce the kind of (non-attributed) graph constraints that we consider.

2.1 Attributed Graphs

We present attributed graphs following [3]. More precisely, first we introduce a notion of labeled graphs (called E-graphs in [3]) as a kind of graphs where the labels are just seen as special nodes and where we associate a label to a (standard) node (resp. to an edge) by having a special kind of edge from the node (resp. from the edge) to its label. Then, we define attributed graphs as labeled graphs where the labels are elements of a given (many-sorted) algebra. We could have also extended these definitions to deal with their typed versions, however to simplify the overall presentation we have preferred to deal with untyped graphs.

Definition 1. ***(Labeled Graphs and morphisms).*** *A* labeled graph *over the set of labels L is a tuple $G = (V, L, E_G, E_{NL}, E_{EL}, \{s_j, t_j\}_{j \in \{G,NL,EL\}})$ consisting of:*

- *V and L, which are the sets of* graph nodes *and of* label nodes*, respectively.*
- *E_G, E_{NL}, and E_{EL}, which are the sets of* graph edges*,* node label edges*, and* edge label edges*, respectively.*

and the source and target functions:

- $s_G : E_G \to V_G$ *and* $t_G : E_G \to V_G$
- $s_{NL} : E_{NL} \to L$ *and* $t_{NL} : E_{NL} \to L$
- $s_{EL} : E_{EL} \to L$ *and* $t_{EL} : E_{EL} \to L$

Given the graphs G and G', a labeled graph morphism *$f : G \to G\,'$ is a tuple, $\langle f_{V_G} : V_G \to V'_G, f_L : L \to L', f_{E_G} : E_G \to E'_G, f_{E_{NL}} : E_{NL} \to E'_{NL}, f_{E_{EL}} : E_{EL} \to E'_{EL}\rangle$ such that f commutes with all the source and target functions.*

As said above, an attributed graph is a labeled graph such that the labels are elements of a given algebra:

Definition 2. ***(Attributed Graphs and Morphisms).*** *Given a signature Σ, an* attributed graph *$AG = (G, \mathcal{A})$ consists of a Σ-algebra $\mathcal{A}$ and of a labeled graph $G = (V_G, L, E_G, E_{NL}, E_{EL}, \{s_j, t_j\}_{j \in \{G,NL,EL\}})$, such that L is the disjoint union of all the carrier sets $\mathcal{A}_s$ for every sort $s \in S$. Given the attributed graphs $AG = (G, \mathcal{A})$ and $AG' = (G', \mathcal{A}')$, an* attributed graph morphism

$f : AG \to AG\,'$ is a pair, $\langle f^G : G \to G', f^{\mathcal{A}} : \mathcal{A} \to \mathcal{A}' \rangle$ such that f^G is a labeled graph morphism, $f^{\mathcal{A}}$ is a Σ-homomorphism and such that, for every sort $s \in S$ and every $a \in \mathcal{A}_s$, $f^G_L(a) = f^{\mathcal{A}}_s(a)$.

In several results of the paper, given two graphs $AG, AG\,'$ we need to put them together in all possible ways. This is done using the construction $AG \otimes AG\,'$:

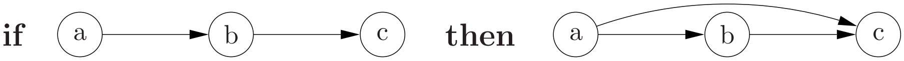
Definition 3. ***(Jointly surjective morphisms).*** *Two attributed graph morphisms $m_1 : AG_1 \to AG$ and $m_2 : AG_2 \to AG$ are* jointly surjective *if for every element a in AG (i.e. a is a node or an edge of any kind) there is an element a_1 in AG_1 or an element a_2 in AG_2 such that $m_1(a_1) = a$ or $m_2(a_2) = a$.*

Given two graphs AG and $AG\,'$, the set of all pairs of jointly surjective monomorphisms from AG and $AG\,'$ is denoted $AG \otimes AG\,'$, that is, $AG \otimes AG\,' = \{m : AG \to H \leftarrow AG\,' : m' \mid \text{ m and m}' \text{ are jointly surjective monomorphisms}\}$.

Note that, in general, $AG \otimes AG\,'$ is an infinite set, unless the sets of labels of AG and $AG\,'$ are both finite.

2.2 Graph Constraints

The underlying idea of a graph constraint is that it should specify that certain structures must be present (or must not be present) in a given graph. For instance, the simplest kind of graph constraint, $\exists C$, specifies that a given graph G should include (a copy of) C. Obviously, $\neg\exists C$ specifies that a given graph G should not include C. A slightly more complex kind of graph constraints are atomic constraints of the form $\forall(c : X \to C)$ where c is a monomorphism (or, just, an inclusion). This constraint specifies that whenever a graph G includes (a copy of) the graph X it should also include (a copy of) its extension C. However, in order to enhance readability (the monomorphism arrow may be confused with the edges of the graphs), in our examples we will display this kind of constraints using an **if - then** notation, where the two graphs involved have been labeled to implicitly represent the given monomorphism. For instance, the constraint:



specifies that a graph must be transitive, i.e. the constraint says that for every three nodes, a, b, c if there is an edge from a to b and an edge from b to c then there should be an edge from a to c.

Graph constraints can be combined using the standard connectives $\vee$ and $\neg$ (as usual, $\wedge$ can be considered a derived operation). In [4,16] a more complex kind of constraints, called nested constraints, is defined, but we do not consider them in this paper.

It may be noted that the constraint $\exists\emptyset$, where $\emptyset$ denotes the empty graph, is satisfied by any graph, i.e. $\exists\emptyset$ may be considered the trivial *true* constraint.

3 Attributed Graph Constraints

The notions and examples presented in the previous subsection deal with (standard) graphs and graph morphisms, but these concepts could, in principle, be used to define a notion of constraint in any arbitrary category. Actually, in [15] it is shown that the results presented in that paper apply to a large class of categories. However, this class does not include the category of attributed graphs as presented above. The main problem is related to the fact that for arbitrary attributed graphs the construction $AG \otimes AG'$ is an infinite set, unless the sets of labels of AG and AG' are both finite. In that context the completeness result stated in that paper would not hold.

In addition, there are other problems with more practical implications. For instance, suppose that we want to state the basic constraint $\exists AC$ over some class of attributed graphs. To state this constraint we would need to define the algebra $\mathcal{A}$ that defines the attributes in the graph AC. But this implies that we need a specification language to define such algebras. Obviously, having to write a complete specification for each constraint may be considered a bit cumbersome.

Moreover, if we want to define constraints over attributed graphs we will probably want to state conditions over the values included in the given graph. If these conditions are just conditions that can be expressed as equalities, then we can specify these conditions by an adequate definition of the corresponding algebra. However, if they cannot be expressed as a simple equality (for instance because a condition involves the use of quantifiers) then there may be no obvious way to express it by an adequate definition of the corresponding algebra $\mathcal{A}$.

The above problems have a simple solution which is partially inspired on the definition of programs in Constraint Logic Programming [11,13]. We consider that an atomic attributed graph constraint over a given algebraic signature Σ, including predicate and function symbols, is a triple $\langle \mathcal{V}, \forall(c : X \to C), \alpha \rangle$, where $\mathcal{V}$ is a finite set of variables, $c : X \to C$ is a monomorphism between the graphs X and C that are labeled over $\mathcal{V}$ (i.e, c is a standard graph constraint) and α is a first-order Σ-formula over the variables in $\mathcal{V}$. Then, we say that a given attributed graph $AG = (G, \mathcal{A})$ satisfies the constraint $\langle \mathcal{V}, \forall(c : X \to C), \alpha \rangle$ if, whenever AG includes a copy of the graph X (for a given assignment of values to the variables in $\mathcal{V}$), AG also includes a copy of its extension C such that the formula α holds in $\mathcal{A}$ for that assignment:

Definition 4. ***(Syntax and satisfaction of atomic attributed graph constraints).*** *An* atomic attributed graph constraint *over a signature Σ is a triple $\langle \mathcal{V}, \forall(c : X \to C), \alpha \rangle$, where $\mathcal{V}$ is a finite set of variables, X and C are labeled graphs over $\mathcal{V}$, $c : X \to C$ is a labeled graph monomorphism, which is the identity over the variables and α is a $\Sigma(\mathcal{V})$-formula. A constraint $\langle \mathcal{V}, \forall(c : X \to C), \alpha \rangle$ where $X = \emptyset$, is called a basic constraint and is denoted $\langle \mathcal{V}, \exists C, \alpha \rangle$. Given a constraint $\langle \mathcal{V}, \forall(c : X \to C), \alpha \rangle$, we call $\forall(c : X \to C)$ its associated* graph constraint *and α its associated* attribute condition.

An attributed graph $AG = (G, \mathcal{A})$ satisfies a constraint $\langle \mathcal{V}, \forall(c : X \to C), \alpha \rangle$, if for every monomorphism $h : X \to G$ there is a monomorphism $f : C \to G$

such that $h = f \circ c$ and $\mathcal{A} \models_{f_\mathcal{V}} \alpha$, where $\mathcal{A} \models_{f_\mathcal{V}} \alpha$ denotes, as usual, that the algebra $\mathcal{A}$ satisfies the formula α when the variables in α are replaced by the their corresponding values according to $f_\mathcal{V}$.

As in the case of standard graph constraints, we can define general attributed constraints using the logical connectives $\vee$ and $\neg$. Then, satisfaction may be extended accordingly. Anyhow, in this paper, for simplicity, we will assume that our specifications consist only of positive and negative basic constraints and positive atomic constraints. However, in our inference systems, in addition, we may use as premises and infer arbitrary clauses of the form:

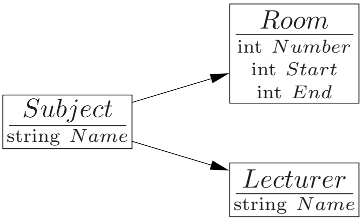
$$L_1 \vee \cdots \vee L_n$$

where each *literal* L_i is either a positive or negative basic constraint.

In this paper we have not considered the case where the given set of constraints includes negative atomic constraints, because this case poses additional difficulties. However, we think that using the same technique that was used in [15], based on a notion of contextual constraints, the extension should be straightforward. As a consequence, it should also be relatively simple to extend our approach to deal with specifications consisting of general clauses. Actually, this would mean that we could deal with arbitrary formulas since they could always be transformed into clausal form.

It may be noted that using only variables as labels in a given constraint is not a limitation. For instance, if we would like to state that a given node should have a certain value v as an attribute, we can label this node with the variable X and, then, state $X = v$ in the associated condition.

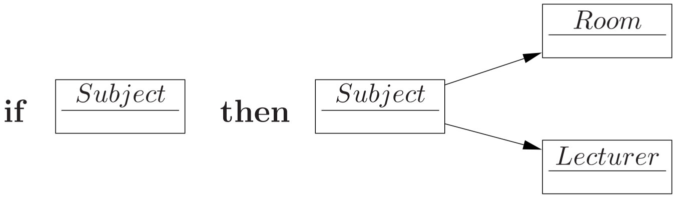
Example 1. Let us suppose that we want to describe an information system modeling the lecturing organization of a department. Then the type graph of (part of) our system could be the following one:



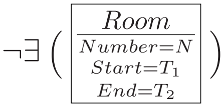
This means that in our system we have three types of nodes. Rooms including three attributes, the room number and a time slot denoted by the attributes Start and End, and Subjects and Lecturers, having its name as an attribute. We also have two types of edges. In particular, an edge from a Subject S to a Lecturer L means, obviously, that L is the lecturer for S. An edge from a Subject S to to a Room means that the lecturing for S takes place in that room for the given time slot. Now for this system we could include the following constraint:

$$\textbf{(1)}\quad \exists \left(\boxed{\frac{Subject}{Name{=}X}} \quad \boxed{\frac{Subject}{Name{=}Y}} \right) \quad \textbf{with} \quad X = CS1 \wedge Y = CS2$$

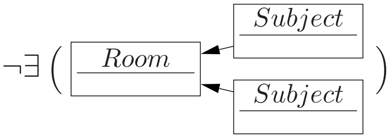
meaning that the given system must include the compulsory subjects Computer Science 1 and Computer Science 2. Moreover we may have a constraint, like constraint **(2)**, saying that every subject included in the system must have some lecturer assignment and some room assignment:

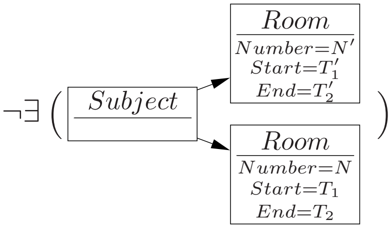
(2) 

We may also have constraints expressing some negative conditions. For instance, that there cannot be a room node with a negative time slot (constraint **(3)**). Or that a room is not assigned at the same time to two subjects (constraint **(4)**) or that two different rooms are not assigned with overlapping time slots to the same subject (constraint **(5)**):

(3) 

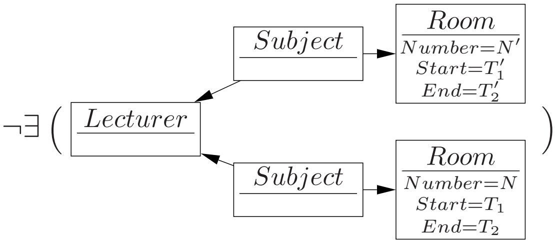
with $T_1 > T_2$

(4) 

(5) 

with $(T_1 < T'_1) \wedge (T'_2 < T_2)$

or, finally (constraint **(6)**), that a lecturer does not have to lecture on two different subjects in two different rooms at the same time:

(6) 

with $(T_1 < T'_1) \wedge (T'_2 < T_2)$

The system that we are describing with these graphical constraints may be not an information system, but the set of web pages of a department, where an arrow from a node of type t_1 to a node of type t_2 may mean that there is a link between two web pages, or it may mean that the information of type t_2 is a subfield of the information of type t_1. In this case, we could have displayed our constraints not in terms of graphs, but as HTML or XML expressions.

4 Refutation Procedures

In this section we describe the framework that we use to present our refutation procedure. We follow an approach which is quite standard in the area of automated deduction (e.g. this is the approach followed to describe resolution, or paramodulation theorem proving). The procedure is defined by means of some inference rules. Then, a refutation procedure can be seen as a (possibly non-terminating) nondeterministic computation where the current state is given by the set of formula that have been inferred until the given moment and where a computation step means adding to the given state the result of applying an inference rule to that state. The procedure terminates when no new inference can be applied or when the false formula (which is represented by the empty clause, denoted □) is inferred. In the latter case, we conclude that the given set of formulas is unsatisfiable.

We assume that a first-order specification SP is given which characterizes the algebras that can be used for defining the class of attributed graphs of interest. This means that our procedure checks if there exists an attributed graph $AG = (G, \mathcal{A})$, where $\mathcal{A} \in Mod(SP)$ that satisfies the given set of constraints, where $Mod(SP)$ denotes the class of models of the specification SP. In this sense, our refutation procedure is parameterized by SP. Actually, some inference rules check, as a side condition, if certain formulas are satisfied by $Mod(SP)$. Thus a proof tool implementing this procedure would need to be built on top of a deductive tool for first-order specifications.

In our case, we assume that the inference rules have the form:

$$\frac{\Gamma_1 \qquad \alpha}{\Gamma_2}$$

where Γ_1 and Γ_2 are clauses including only positive literals and where α is an atomic constraint. Moreover, Γ_1 is assumed to belong to the current set of inferred clauses and α is assumed to belong to the original set of constraints. Then a *refutation procedure* for a set of constraints $\mathcal{C}$ is a sequence of inferences:

$$\mathcal{C}_0 \Rightarrow_{\mathcal{C}} \mathcal{C}_1 \Rightarrow_{\mathcal{C}} \cdots \Rightarrow_{\mathcal{C}} \mathcal{C}_i \Rightarrow_{\mathcal{C}} \dots$$

where the initial state just includes the *true* clause (i.e. $\mathcal{C}_0 = \{\langle \emptyset, \exists\emptyset, \mathbf{true}\rangle\}$) and where we write $\mathcal{C}_i \Rightarrow_{\mathcal{C}} \mathcal{C}_{i+1}$ if there is an inference rule as the one above such that $\Gamma_1 \in \mathcal{C}_i$, $\alpha \in \mathcal{C}$, and $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{\Gamma_2\}$. Moreover, we assume that $\mathcal{C}_i \subset \mathcal{C}_{i+1}$, i.e. $\Gamma_2 \notin \mathcal{C}_i$, to avoid useless inferences.

Since the application of rules is nondeterministic, there is the possibility that we never apply some key inference. To avoid this problem we assume that procedures are *fair*, which means that if at any moment i, there is a possible inference $\mathcal{C}_i \Rightarrow_{\mathcal{C}} \mathcal{C}_i \cup \{\Gamma\}$, then at some moment j we have that $\Gamma \in \mathcal{C}_j$. Then, a refutation procedure for $\mathcal{C}$ is *sound* if whenever the procedure infers the empty clause we have that $\mathcal{C}$ is unsatisfiable. And a procedure is *complete* if, whenever $\mathcal{C}$ is unsatisfiable, we have that the procedure infers the empty clause.

5 Proof Rules for Basic Constraints and Positive Atomic Constraints

In this section we provide inference rules for the kind of constraints considered in this paper, i.e. when specifications include (positive or negative) basic constraints and positive atomic constraints. This means that the given specifications are assumed to consist of literals of the form $\langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle$, $\neg\langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle$, or $\langle \mathcal{V}_2, \forall(c : X \to C_2), \alpha_2 \rangle$. Moreover, as explained in the previous section, we assume that a specification SP is given and we can check if a given formula is satisfiable in $Mod(SP)$. In addition, without loss of generality (we can do some variable renaming if necessary), we assume that the sets of variables involved in the premises of the rules are disjoint.

Satisfiability is based on the four rules below.

$$\frac{\langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle \vee \Gamma \qquad \neg\langle \mathcal{V}_2, \exists C_2, \alpha_2 \rangle}{\langle \mathcal{V}_1, \exists C_1, (\alpha_1 \wedge \neg\alpha_2 \wedge eq(m)) \rangle \vee \Gamma} \tag{$R1$}$$

if there exists a monomorphism $m : C_2 \to C_1$ and if the equivalence of $(\alpha_1 \wedge \neg\alpha_2 \wedge eq(m))$ and α_1 is not a logical consequence of SP, and where $eq(m)$ denotes the formula $\bigwedge_{X \in \mathcal{V}_2} X = m(X)$

$$\frac{\langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle \vee \Gamma \qquad \langle \mathcal{V}_2, \exists C_2, \alpha_2 \rangle}{(\bigvee_{\langle f_1 : C_1 \to G \leftarrow C_2 : f_2 \rangle \in (C_1 \otimes C_2)} \langle \mathcal{V}, \exists G, \alpha_1 \wedge \alpha_2 \wedge eq(f_1) \wedge eq(f_2) \rangle \vee \Gamma} \tag{$R2$}$$

if there is no monomorphism $m : C_2 \to C_1$ and where $\mathcal{V}$ is the set of labels of the graph G.

$$\frac{\langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle \vee \Gamma \qquad \langle \mathcal{V}_2, \forall(c : X \to C_2), \alpha_2 \rangle}{(\bigvee_{\langle f_1 : C_1 \to G \leftarrow C_2 : f_2 \rangle \in \mathcal{G}} \langle \mathcal{V}, \exists G, \alpha_1 \wedge \alpha_2 \wedge eq(f_1) \wedge eq(f_2) \rangle \vee \Gamma} \tag{$R3$}$$

if there is a monomorphism $m : X \to C_1$ and there is no monomorphism $h : C_2 \to C_1$ such that $m = h \circ c$ and where $\mathcal{V}$ is the set of labels of the graph G and where $\mathcal{G}$ is the set consisting of all cospans $(f_1 : C_1 \to G \leftarrow C_2 : f_2) \in (C_1 \otimes C_2)$ such that the diagram below commutes:

$$\begin{tikzcd}
 & C_1 \arrow[dr, "f_1"] & \\
X \arrow[ur, "m"] \arrow[dr, "c"'] & & G \\
 & C_2 \arrow[ur, "f_2"'] &
\end{tikzcd}$$

Finally, the fourth rule is:

$$\frac{\langle \mathcal{V}, \exists C, \alpha \rangle \vee \Gamma}{\Gamma} \qquad (R4)$$

if α is not satisfiable in $\mathit{Mod}(SP)$.

The first rule just says that if a given graph AG should include a copy of C_1 such that α_1 holds, but should not include a copy of C_2 such that α_2 holds, and if, in addition, C_2 is included in C_1, then we can infer that AG should include a copy of C_1 such that α_1 holds but α_2 does not hold. The requirement in the side condition that the formulas $(\alpha_1 \wedge \neg\alpha_2 \wedge eq(m))$ and α_1 should not be a logical consequence from $\mathit{Mod}(SP)$ is just to avoid the repeated application of the same inference. The second rule can be seen as a rule that, given two constraints, builds a new constraint that subsumes them. More precisely, the graphs involved in the new literals in the conclusion of the inference rule, i.e. the graphs G, satisfy both graph constraints $\exists C_1$ and $\exists C_2$ and, in addition, the associated attribute condition $\alpha_1 \wedge \alpha_2 \wedge eq(f_1) \wedge eq(f_2)$ represents the combination of the conditions of the atomic constraints involved in the rule. This means that if we apply this rule repeatedly, using all the positive constraints in the original set $\mathcal{C}$, we would build (minimal) graphs that satisfy the graph part of all the positive basic constraints in $\mathcal{C}$. If, in addition, we find a substitution that satisfies the associated attribute conditions, we would have (minimal) attributed graphs that satisfy all the positive basic constraints in $\mathcal{C}$. The third rule is similar to rule (R2) in the sense that, given a positive basic constraint and a positive atomic constraint, it builds a disjunction of literals representing graphs that try to satisfy both constraints. However, in this case the satisfaction of the graph constraint $\forall(c : X \to C_2)$ is not ensured. In particular, the idea of the rule is that if we know that X is included in C_1 then we build all the possible extensions of C_1 which also include C_2 (each G would be one of such extensions). But in this case we cannot be sure that G satisfies $\langle \mathcal{V}_2, \forall(c : X \to C_2), \alpha_2 \rangle$, because G may include more instances of X, which perhaps were not included in C_1. Finally, the last rule just says that if a given clause includes a literal with an unsatisfiable attribute condition then we can delete this literal.

The rules (R1), (R2), (R3) and (R4) are sound and complete. The soundness of the first rule is quite obvious. If an attributed graph AG satisfies both premises and in addition we know that C_2 is included in C_1, then either AG satisfies Γ or AG includes C_1 and it satisfies α_1 but not α_2 (with respect to the renaming defined by m). The soundness of the second rule is based on the *pair factorization* *property*: Given two (mono)morphisms, $g_1 : C_1 \to H$, $g_2 : C_2 \to H$, with the same codomain H there exists a graph G and monomorphisms $f_1 : C_1 \to G$, $f_2 : C_2 \to G$ and $h : G \to H$ such that f_1 and f_2 are jointly surjective and the diagram below commutes:

$$\begin{tikzcd}
C_1 \arrow[d, "f_1"'] \arrow[dr, "g_1"] & \\
G \arrow[r, "h"] & H \\
C_2 \arrow[u, "f_2"] \arrow[ur, "g_2"'] &
\end{tikzcd}$$

Notice that, in that case, $(f_1 : C_1 \to G \leftarrow C_2 : f_2) \in (C_1 \otimes C_2)$. Then, if an attributed graph $AH = (H, \mathcal{A})$ satisfies $\langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle$ and $\langle \mathcal{V}_2, \exists C_2, \alpha_2 \rangle$ then we can see that AH also satisfies the constraint $\langle \mathcal{V}, \exists G, \alpha_1 \wedge \alpha_2 \wedge eq(f_1) \wedge eq(f_2) \rangle$. On one hand, we have that, according to the pair factorization property, $h : G \to H$. On the other, $\mathcal{A} \models_h \alpha_1$ because $\mathcal{A} \models_{g_1} \alpha_1$ and $g_1 = h \circ f_1$ (and similarly for α_2) and $\mathcal{A} \models_h eq(f_1) \wedge eq(f_2)$ because the above diagram commutes.

The proof of soundness of (R3) is very similar to the proof for rule (R2). If $AH = (H, \mathcal{A})$ satisfies $\langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle$ and $\langle \mathcal{V}_2, \forall (c : X \to C_2), \alpha_2 \rangle$, using pair factorization we get the diagram below:

$$\begin{tikzcd}
 & C_1 \arrow[dr, "f_1"] \arrow[drr, "h_1"] & & \\
X \arrow[ur, "m"] \arrow[dr, "c"'] & & G \arrow[r] & H \\
 & C_2 \arrow[ur, "f_2"] \arrow[urr, "h_2"'] & &
\end{tikzcd}$$

then, AH will also satisfy $\langle \mathcal{V}, \exists G, \alpha_1 \wedge \alpha_2 \wedge eq(f_1) \wedge eq(f_2) \rangle$.

Finally, the soundness of (R4) is trivial since no attributed graph $AG = (G, \mathcal{A})$ can satisfy a constraint $\langle \mathcal{V}, \exists C, \alpha \rangle$ if α is unsatisfiable in $\mathcal{A}$.

Let us now sketch the proof of completeness of these rules. The underlying idea is inspired by a technique called *model construction* used for proving completeness of some inference systems for first-order logic with equality [14]. According to this technique, we see the inference rules as steps for building a model of the given set of constraints $\mathcal{C}$, in case it is satisfiable. In our case, first we see that if our constraints are satisfiable then we can build a labeled graph which satisfies the graph part of the constraints in $\mathcal{C}$. Then, we see that this labeled graph may be transformed into an attributed graph over some algebra $\mathcal{A}$ in $Mod(SP)$ that satisfies all the constraints in $\mathcal{C}$.

The construction is as follows. We consider sequences of basic constraints $\langle \emptyset, \exists \emptyset, \mathbf{true} \rangle \prec \langle \mathcal{V}_1, \exists C_1, \alpha_1 \rangle \prec \cdots \prec \langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle \prec \ldots$, where $C_i \prec C_{i+1}$ if (a) $\langle \mathcal{V}_{i+1}, \exists C_{i+1}, \alpha_{i+1} \rangle$ is a "new" literal in the clause inferred after applying rules (R1), (R2) or (R3) to $\langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle$ and to some positive constraint in $\mathcal{C}$

(this means that the graph C_i is included in C_{i+1}) ; and (b) α_i is satisfiable in $Mod(SP)$. Each of these sequences can be seen as a path for building a possible model of the graph part for the constraints in $\mathcal{C}$. Note that, if condition (b) does not hold then it is useless to continue this path. We require these sequences to be fair, which means that if we can have $\langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle \prec \langle \mathcal{V}'_{i+1}, \exists C'_{i+1}, \alpha'_{i+1} \rangle$ via some inference, where $\alpha'_{i+1} = \alpha_i \wedge \alpha \wedge eq(f_1) \wedge eq(f_2)$ and the sequence is infinite, then there would be some $j > i$ such that C_j includes $C\ '_{i+1}$ and $\alpha'_j = \alpha_{j-1} \wedge \alpha \wedge eq(f'_1) \wedge eq(f'_2)$, for some adequate monomorphisms f'_1 and f'_2.

Before continuing with the proof sketch it is important to notice that if $\langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle$ is an element of one of these sequences then the condition α_i is satisfiable in $\mathcal{A} \in Mod(SP)$ for some substitution $h : \mathcal{V}_i \to \mathcal{A}$. This implies that the attributed graph $AC_i = (C'_i, \mathcal{A})$, where C'_i is obtained from C_i by replacing each variable $X \in \mathcal{V}_i$ by $h(X)$, satisfies the constraint $\langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle$. Moreover AC_i also satisfies all the basic constraints in $\mathcal{C}$ which are used in inferences in the sequence $\langle \emptyset, \exists \emptyset, \mathbf{true} \rangle \prec \langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle$.

Now, resuming the proof sketch, we have three cases:

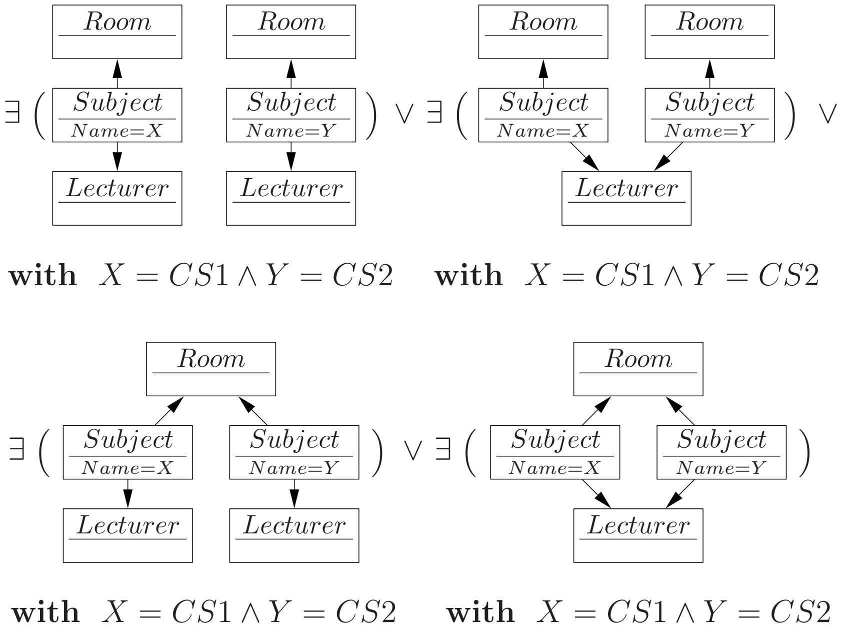
- All maximal sequences of this kind are finite and their last element $\langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle$ are such that α_i is unsatisfiable in $\mathcal{A}$. This means that no path is useful for building a model. Then it can be shown that a fair procedure would generate the empty clause for $\mathcal{C}$.
- There is a finite sequence whose last element is $\langle \mathcal{V}_i, \exists C_i, \alpha_i \rangle$ where α_i is satisfiable in $\mathcal{A} \in Mod(SP)$ for some substitution $h : \mathcal{V}_i \to \mathcal{A}$. Then any attributed graph $AC_i = (C'_i, \mathcal{A})$ satisfies all the constraints in $\mathcal{C}$, where C'_i is obtained from C_i by replacing each variable $X \in \mathcal{V}_i$ by $h(X)$.
- There is an infinite fair sequence. Then we can prove that the graph C, which is the union of all the graphs in the sequence (the colimit of the sequence of graphs), is a model of the graph part of all the positive constraints in $\mathcal{C}$. In addition, we know that each attribute condition in the sequence, α_i, is satisfiable in $Mod(SP)$ and, moreover, we know by construction that for every i, α_{i+1} implies α_i. Then, by compactness of first-order logic, we know that the set consisting of all the constraints α_i is satisfiable for some $\mathcal{A} \in Mod(SP)$ via some substitution $h : \mathcal{V}_i \to \mathcal{A}$. As a consequence, as in the previous case, any attributed graph $AC = (C', \mathcal{A})$ satisfies all the constraints in $\mathcal{C}$, where C' is obtained from C by replacing each variable $X \in \mathcal{V}$ by $h(X)$, and where $\mathcal{V}$ is the set of labels of the graph C.

As a consequence, we have:

Theorem 1. ***(Soundness and Completeness)*** *Given an attribute specification SP, let $\mathcal{C}_0 \Rightarrow_{\mathcal{C}} \mathcal{C}_1 \Rightarrow_{\mathcal{C}} \cdots \Rightarrow_{\mathcal{C}} \mathcal{C}_k \ldots$ be a fair refutation procedure defined over a set of basic constraints and positive atomic constraints $\mathcal{C}$, based on the rules (R1), (R2), (R3) and (R4). Then, $\mathcal{C}$ is unsatisfiable if and only if there is a j such that the empty clause is in $\mathcal{C}_j$.*

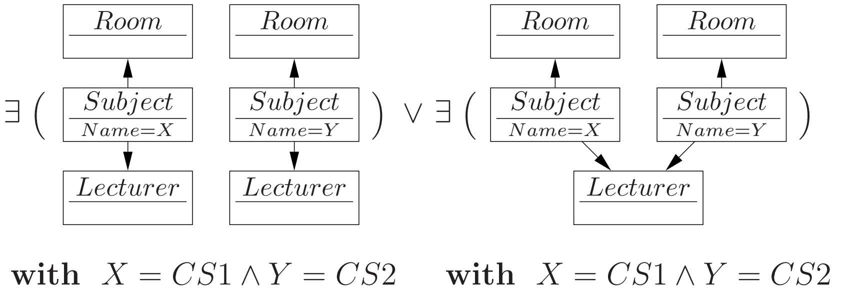
Example 2. Let us consider the constraints that are included in Example 1 (i.e. the constraints (1), (2), (3), (4), (5), and (6)). If we apply the third rule on constraints (1) and (2), and again on the resulting clause and on constraint (3) then we would infer the following clause:

(7)



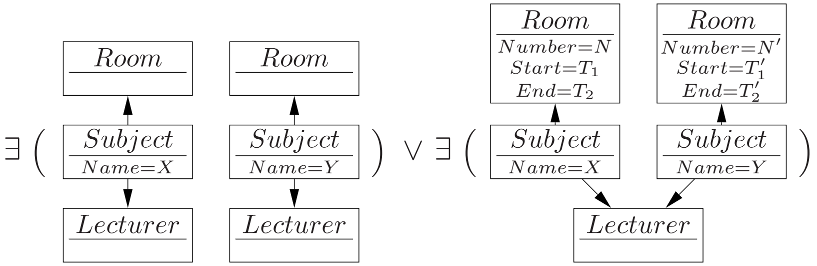
This clause states that the graph should include two subjects (CS1 and CS2) and these subjects may be assigned to two different rooms and to either two different lecturers, or to the same lecturer, or they may be assigned to the same room, and to either different lecturers, or the same lecturer. Obviously, the last two constraints in this clause violate constraint (4), which means that we can eliminate them using twice rule (R1), yielding the following clause:

(8)



Finally, we can apply rule (R1) to clause (8) and to constraint (6) and then, again, rule (R1) to the resulting clause and to constraint (6). The result would be the clause:

(9)



$$\textbf{with} \quad X = CS1 \wedge Y = CS2 \qquad \textbf{with} \quad X = CS1 \wedge Y = CS2 \wedge$$

$$\wedge\neg(T_1 < T'_1 \wedge T'_2 < T_2)\wedge$$

$$\wedge\neg(T'_1 < T_1 \wedge T_2 < T'_2)$$

where the second literal states that if the two subjects are lectured by the same lecturer then there should be no overlapping between the time slots associated to the two subjects. Then no further inference can be applied, which means that the given set of constraints is satisfiable

6 Conclusion

In this paper we have extended the notion of graph constraint to deal with attributed graphs. The new notion of attributed constraint combines a (standard) graph constraint with a formula describing a condition on the attributes of the graph. In addition, we have shown that this new kind of constraint can be an adequate visual formalism for specifying classes of semi-structured documents (as websites) or to define constraints associated to visual modelling formalisms.

We have also shown how can we reason with this new formalism. We have provided inference rules that are sound and complete for the class of basic and positive atomic constraints. Contrary to [15] we have not dealt with negative atomic constraints. Nevertheless, we believe that the techniques that we used in that paper for dealing with that kind of constraints can be extended in a straightforward manner to deal with the attributed case.

We have not yet implemented these techniques, although it would not be too difficult to implement them on top of the AGG system, given that the basic construction that we use in our inference rules (i.e. building $G_1 \otimes G_2$) is already implemented there. As a consequence, it would be difficult to compare the performance of this approach with an obvious possible approach based on coding this constraints into standard first-order logic. Obviously, this comparison would depend on the coding chosen. However, the coding considered in [1], defined just for XML documents and not for arbitrary graphs, needs to use associative-commutative matching for checking satisfaction of constraints. This means that to implement deduction they would probably need to use associative-commutative unification which is very costly.

References

1. Alpuente, M., Ballis, D., Falaschi, M.: Automated Verification of Web Sites Using Partial Rewriting. Software Tools for Technology Transfer 8, 565–585 (2006)
2. Courcelle, B.: The expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. In: [17], pp. 313–400 (1997)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
4. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H.: Constraints and Application Conditions: From Graphs to High-Level Structures. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 287–303. Springer, Heidelberg (2004)
5. Ehrig, H., Habel, A.: Graph Grammars with Application Conditions: From Graphs to High-Level Structures. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 87–100. Springer, Heidelberg (1986)
6. Ellmer, E., Emmerich, W., Finkelstein, A., Nentwich, C.: Flexible Consistency Checking. ACM T. on Soft. Eng. and Methodology 12(1), 28–63 (2003)
7. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Fundam. Inform. 26(3/4), 287–313 (1996)
8. 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)
9. Habel, A., Pennemann, K.-H.: Satisfiability of High-Level Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 430–444. Springer, Heidelberg (2006)
10. Heckel, R., Wagner, A.: Ensuring Consistency of Conditional Graph Grammars - A Constructive Approach. In: Proceedings SEGRAGRA 1995. Electr. Notes Theor. Comput. Sci, vol. 2, pp. 118–126 (1995)
11. Jaffar, J., Maher, M., Marriot, K., Stukey, P.: The semantics of constraint logic programs. The Journal of Logic Programming (37), 1–46 (1998)
12. Jelliffe, R.: "Schematron", Internet Document (May 2000), http://xml.ascc.net/resource/schematron/
13. Lucio, P., Orejas, F., Pasarella, E., Pino, E.: A Functorial Framework for Constraint Normal Logic Programming. Abstract Categorical Structures 3, 421–450 (2008)
14. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, Elsevier Science and MIT Press (2001)
15. Orejas, F., Ehrig, H., Prange, U.: A Logic of Graph Constraints. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 179–198. Springer, Heidelberg (2008)
16. 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)
17. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific, Singapore (1997)

Resolution-Like Theorem Proving for High-Level Conditions*

Karl-Heinz Pennemann

University of Oldenburg, Germany
pennemann@informatik.uni-oldenburg.de

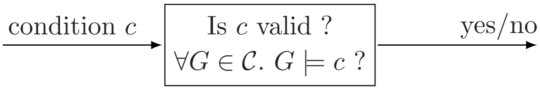
Abstract. The tautology problem is the problem to prove the validity of statements. In this paper, we present a calculus for this undecidable problem on graphical conditions, prove its soundness, investigate the necessity of each deduction rule, and discuss practical aspects concerning an implementation. As we use the framework of weak adhesive HLR categories, the calculus is applicable to a number of replacement capable structures, such as Petri-Nets, graphs or hypergraphs.

Keywords: first-order tautology problem, high-level conditions, theorem proving, resolution, weak adhesive HLR categories.

1 Introduction

(High-level) Conditions are a graphical formalism to specify valid objects as well as morphisms, i.e., they can be used to describe system or program states as well as specify matches for transformation rules. They provide an intuitive formalism for structural properties and are well suited for reasoning about the behavior of transformation systems.

Our goal is to decide the correctness of graphical specifications consisting of a precondition, a program [HP01, HPR06] and a postcondition. A classical approach [DS89] to this problem is the proof or refutation that the precondition implies the weakest precondition [HPR06] of the program and the postcondition. To decide such an implication is a special instance of the tautology problem: the decision whether or not a claimed statement is valid for all possible objects.



For the category of finite, directed, labeled graphs, conditions are expressively equivalent [Ren04, HP08] to first order logic on graphs [Cou90]. Consequently, the tautology problem for arbitrary conditions over arbitrary categories is not decidable, i.e., there does not exist an algorithm that decides the validity of

[*] This work is supported by the German Research Foundation (DFG), grants GRK 1076/1 (Graduate School on Trustworthy Software Systems) and HA 2936/2 (Development of Correct Graph Transformation Systems).

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 289–304, 2008.

arbitrary conditions over arbitrary categories. In the context of finite graphs, the tautology problem is not even semi-decidable: while it is possible to search for proofs, one is not guaranteed to find one, even if there is no (finite) graph that does not satisfy the given condition.

In the case of graphs, the translation of conditions into first order logic [HP08] enables to solve the tautology problem using existing first-order theorem provers such as VAMPIRE [RV02], DARWIN [BFT06] or PROVER9 [McC08]. However, 7 out of 74 example tautologies generated from correct program specifications of the "access control" example in [HPR06] cannot be solved by any of the aforementioned tools, given 1 hour time per tautology (INTEL T5600, 1.83GHz). One reason for this is that first-order theorem provers need to be restricted to graphs via axioms that become part of the problem to be solved. In contrast, a theorem prover based on conditions would be restricted to the considered category in a natural, constructive way. This property in combination with other advantages makes it worthwhile to investigate a theorem prover dedicated to conditions.

In this paper, we present a calculus for conditions over adhesive high-level replacement categories. Taking resolution [Rob65], the most successful approach to first-order theorem proving, as an ideal, we present six deduction rules able to refute conditions over graphs and graph-like structures in conjunctive normal form. We show that every rule application corresponds to a logical deduction, and investigate if omission of any rule leads to an incomplete calculus. We discuss practical aspects concerning an implementation such as filtering out structurally equivalent conditions, and briefly compare our results with related work, e.g. Koch et. al. [KMP05] and Orejas et. al. [OEP08].

The paper is organized as follows. In Section 2, the definition of conditions is reviewed and examples are given. In Section 3, the calculus is presented, and its soundness is shown. In Section 4, we discuss practical aspects concerning an implementation. We briefly relate our results to other work in Section 5. A conclusion including further work is given in Section 6.

2 Conditions

In this section, we recall the definition of conditions on graphs and graph-like structures. To abstract from a specific structure, we use the framework of weak adhesive HLR categories. A detailed introduction can be found in [EEPT06].

Assumption 1. *Assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category consisting of a category $\mathcal{C}$ of objects and a class $\mathcal{M}$ of monomorphisms. Additionally, we require*

- *an* $\mathcal{M}$-initial object I*, i.e., an object $I \in \mathcal{C}$ such that, for every object $G \in \mathcal{C}$, there exists a unique morphism $i_G \colon I \to G$ and i_G is in $\mathcal{M}$,*
- epi-$\mathcal{M}$-factorization*, i.e., for every morphism there is an epi-mono-factorization with monomorphism in $\mathcal{M}$,*

- *a* finite number of $\mathcal{M}$-morphisms, *i.e. for every objects G, H, there exists only a finite number of morphisms $G \hookrightarrow H$ in $\mathcal{M}$ (up to isomorphism),*
- *a* finite number of epimorphisms *for any domain G, i.e. for every object G, there is only a finite number of epimorphisms $e: G \to H$ (up to isomorphism).*

The last two requirements ensure the effectiveness of the constructions in this paper. From now on, morphisms in $\mathcal{C}$ are simply referred to as morphisms.

Example 1. The category **Graph** of finite, directed, labeled graphs together with the class $\mathcal{M}$ of all injective graph morphisms constitutes a weak adhesive HLR category [EEPT06] satisfying the assumptions. The empty graph $\emptyset$ is the $\mathcal{M}$-initial object.

Notation. A morphism m with domain A and codomain B is denoted by $m: A \to B$. We write "$\hookrightarrow$" instead of "$\to$" to indicate that the morphism is in $\mathcal{M}$. For graph morphisms, the mapping of nodes is depicted by indices, if necessary.

For a certain rule in our calculus, we require the notion of $\mathcal{M}$-pushout. An $\mathcal{M}$-pushout is a special pushout for which it can be guaranteed that the unique morphism u is in $\mathcal{M}$, if the commutative morphisms p, q are in $\mathcal{M}$.

Definition 1 ($\mathcal{M}$-pushout). *A pushout $c \circ a = d \circ b$ with $c, d \in \mathcal{M}$ is called $\mathcal{M}$-*pushout*, if for all morphisms p, q with $p \circ a = q \circ b$, the unique existing morphism u with $p = c \circ u$ and $q = d \circ u$ is in $\mathcal{M}$.*

$$\begin{tikzcd}
\bullet \arrow[r, "b"] \arrow[d, "a"'] \arrow[rd, phantom, "(PO)"] & \bullet \arrow[d, hook, "d"] \arrow[rdd, hook, bend left, "q"] & \\
\bullet \arrow[r, hook, "c"'] \arrow[rrd, hook, bend right, "p"'] & \bullet \arrow[rd, hook, "u"] & \\
 & & \bullet
\end{tikzcd}$$

We use the following characterization of $\mathcal{M}$-pushouts.

Fact 1 ($\mathcal{M}$-pushout). *A pushout $c \circ a = d \circ b$ with $c, d \in \mathcal{M}$ is an $\mathcal{M}$-*pushout*, if and only if for all epimorphisms e with* $\mathrm{dom}(e) = \mathrm{codom}(d)$ *we have $e \notin \mathcal{M}$ implies $e \circ c \notin \mathcal{M}$ or $e \circ d \notin \mathcal{M}$.*

Proof. Via epi-$\mathcal{M}$-factorization using the converse statement.

$$\begin{array}{ll}
\quad e \notin \mathcal{M} \text{ implies } (e \circ c \notin \mathcal{M} \text{ or } e \circ d \notin \mathcal{M}) & \text{(characterization)} \\
\Leftrightarrow \text{not } (e \circ c \notin \mathcal{M} \text{ or } e \circ d \notin \mathcal{M}) \text{ implies not } e \notin \mathcal{M} & \text{(converse)} \\
\Leftrightarrow (e \circ c \in \mathcal{M} \text{ and } e \circ d \in \mathcal{M}) \text{ implies } e \in \mathcal{M} & \text{(deMorgan)} \\
\Leftrightarrow (m \circ e \circ c \in \mathcal{M} \text{ and } m \circ e \circ d \in \mathcal{M}) \text{ implies } m \circ e \in \mathcal{M} & \left(\begin{smallmatrix} m \in \mathcal{M},\ \mathcal{M} \text{ closed} \\ \text{under comp./decomp.} \end{smallmatrix}\right) \\
\Leftrightarrow (u \circ c \in \mathcal{M} \text{ and } u \circ d \in \mathcal{M}) \text{ implies } u \in \mathcal{M} & \text{(epi-}\mathcal{M}\text{-factorization)} \\
\Leftrightarrow (p \in \mathcal{M} \text{ and } q \in \mathcal{M}) \text{ implies } u \in \mathcal{M} & \text{(commutativity)}
\end{array}$$

Example 2 (access control graphs). In the following, we present state graphs of a simple access control for computer systems, which abstracts authentication and models user and session management in a simple way. We use this example solely for illustrative purposes. A more elaborated, role-based access control model is considered in [KMP05]. The basic items of our model are users 👤, sessions ▭, logs 📄, computer systems 🖥, and directed edges between those items. An edge between a user and a system represents that the user has the right to access the system, i.e. to establish a session with the system. Every user node is connected with one log node, while an edge from a log to the system represents a

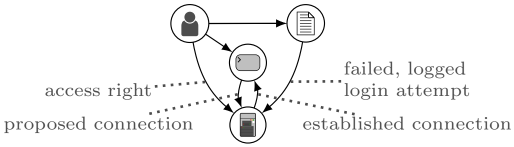


Fig. 1. The type graph of the access control system

failed (logged) login attempt. Every session is connected to a user and a system. The direction of the latter edge differentiates between sessions that have been proposed (an outgoing edge from a session node to a system) and sessions that have been established (an incoming edge to a session node from a system).

Conditions are nested constraints and application conditions generalizing the corresponding notions in [HW95, EEHP06] along the lines of [Ren04].

Definition 2 (conditions). *A* (nested) condition *over an object P is of the form* true *or* $\exists(a, c)$, *where* $a\colon P \to C$ *is a morphism and* c *is a condition over* C. *Moreover, Boolean formulas over conditions over* P *yield conditions over* P, *i.e.,* $\neg c$ *and* $\wedge_{j\in J} c_j$ *are (Boolean) conditions over* P, *where* J *is a finite index set and* c, $(c_j)_{j\in J}$ *are conditions over* P. *Additionally,* $\exists a$ *abbreviates* $\exists(a, \text{true})$, $\forall(a, c)$ *abbreviates* $\neg\exists(a, \neg c)$, false *abbreviates* $\neg$true, $\vee_{j\in J} c_j$ *abbreviates* $\neg\wedge_{j\in J} \neg c_j$ *and* $c \Rightarrow d$ *abbreviates* $\neg c \vee d$.

Every object and morphism satisfies true. *A morphism* p satisfies *a condition* $\exists(a, c)$, *if there exists a morphism* q *in* $\mathcal{M}$ *such that* $q \circ a = p$ *and* q *satisfies* c.

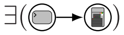
$$\begin{tikzcd} \exists(\ P \arrow[rr, "a"] \arrow[dr, "p"'] & = & C, \arrow[dl, hook', "q"] \arrow[r, phantom, "\models"] & c\ ) \\ & G & & \end{tikzcd}$$

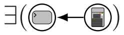
An object G satisfies *a condition* $\exists(a, c)$, *if the condition is over the initial object* I *and the initial morphism* $i_G\colon I \to G$ *satisfies the condition. The satisfaction of conditions by objects and morphisms is extended onto Boolean conditions in the usual way. We write* $G \models c$ *resp.* $p \models c$ *to denote that the object* G *resp. the morphism* p *satisfies* c. *For two conditions* c, d *over* C, d *is a* consequence *or* logical deduction *of* c, *written* $c \models d$, *if for all morphisms* p *in* $\mathcal{M}$ *with domain* C, $p \models c$ *implies* $p \models d$. *Two conditions* c *and* c' *are* equivalent, *denoted by* $c \equiv c'$, *if for all morphisms* p *in* $\mathcal{M}$, $p \models c$ *iff* $p \models c'$.

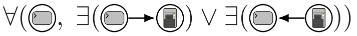
In the context of objects, conditions (over the initial object I) are also called *constraints*.

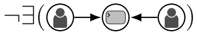
Notation. For every morphism $a\colon P \to C$ in a condition, we just depict the codomain C, if the domain P can be unambiguously inferred. This is the case for constraints, which are by definition conditions over I. For instance, the constraint $\forall(\emptyset \to \circ_1, \exists(\circ_1 \to \circ_1 \!\to\! \circ_2))$ with the meaning "Every node has an outgoing edge to another distinct node" can be represented by $\forall(\circ_1, \exists(\circ_1 \!\to\! \circ_2))$.

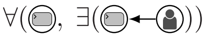
Example 3 (access control conditions). Consider the access control graphs introduced in Example 2. Conditions allow to formulate statements on the graphs of the access control and can be combined to form more complex statements. The following conditions are over the empty graph:

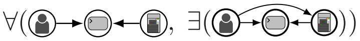

A session is proposed

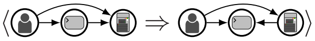
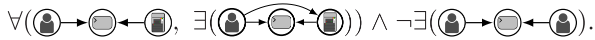

A session is established

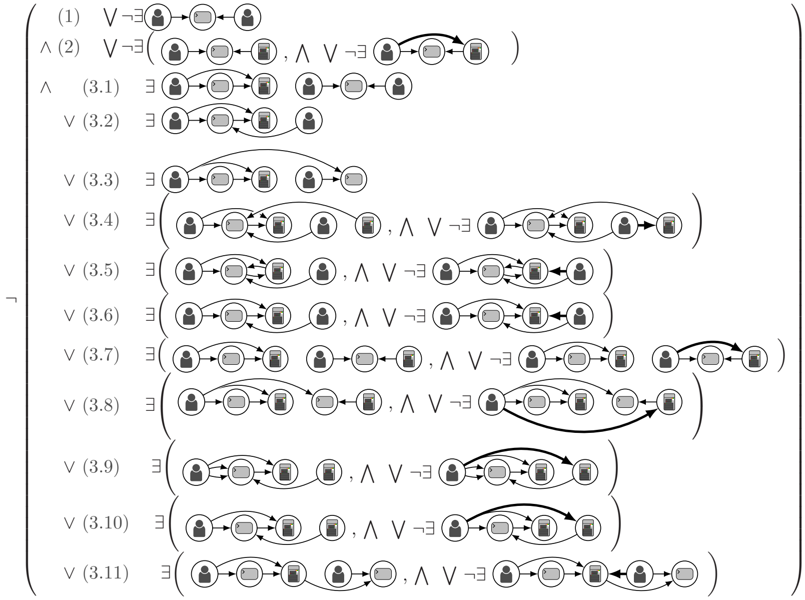

Every session is either proposed or established


No session is shared between two users


Every session is associated to a user


Every user that is logged into a system, has an access right.

Example 4. Consider the access control graphs introduced in Example 2. The dynamic part of the access control system is the reflexive, transitive closure of a non-deterministic choice of programs such as the addition and removal of users, the grant and revocation of access rights and a login and logout procedure. See [HPR06] for a complete overview. We exemplarily consider the transformation rule `Access` =  which is a part of the login procedure. Such a rule consists of a left-hand side expressing the prerequisites "If a user proposes a session to a system for which he has the appropriate access right" and the local effect of the rule's application "Then this proposed session is accepted and becomes established". Our goal is to show that `Access` preserves the satisfiability of the condition . By construction of a weakest precondition [HPR06], the problem reduces to prove that



is a tautology.

For rules in our calculus such as (Lift), we assume conditions to be in $\mathcal{M}$*-normal form* ($\mathcal{M}$NF), i.e. if for all subconditions $\exists(a,c)$ the morphism a is in $\mathcal{M}$.

Fact 2 ($\mathcal{M}$-normal form). *Every condition c over P can be transformed into a condition c' in $\mathcal{M}$NF such that $c \equiv c'$.*

Proof. $\exists(a,c) \equiv \exists a \equiv$ false, if $a \notin \mathcal{M}$ (see [HP08]).

We now define the notion of a non-negated subcondition, which we use later to restrict the applicability of deduction rules.

Definition 3 (non-negated subcondition). *A condition c is a* non-negated subcondition *of a condition d, if $c = d$ or if d is of the form $\exists(a,e)$ or $(e \wedge e')$ or $(e \vee e')$ and c is a non-negated subcondition of e or e'.*

3 Proving High-Level Conditions

The tautology or validity problem is the fundamental problem of deciding whether or not a claimed statement is true for all possible objects.

Definition 4 (tautology problem). *Given a category $\mathcal{C}$, the* tautology problem *is the problem to decide for any condition c, whether or not forall $G \in \mathcal{C}$, $G \models c$.*

We write "$\models c$" if c is a tautology and "$\not\models c$" if c is not a tautology. A straightforward approach to answer the tautology problem for a condition c is to prove "true $\models c$". This can be done by constructing a proof chain "true $\models \ldots \models c$", starting without any assumptions (true), yielding in logical deductions the given condition c. Instead of constructing such a proof top-down, resolution follows a more target-oriented view and considers the complementary problem of refuting the negated condition "$\neg c$". In this case, the goal is to find a refutation "$\neg c \models \ldots \models$ false".

After negation of an input F, a resolution-based algorithm on formulas would transform the negated statement $\neg F$ into prenex normal form and skolemize to yield clauses. However neither does there exists a comparable normal form for conditions, nor is skolemization possible for a given category such as **Graph**: Skolemization requires the introduction of fresh function symbols of unbounded arity, for which there seems no equivalent operation for a fixed structure. Nevertheless, it is possible to transform the condition $\neg c$ into conjunctive normal form.

Definition 5 (conjunctive normal form). *Condition* true *is in* conjunctive normal form (CNF). *Every condition $\wedge_{j \in J} \vee_{k \in K_j} c_k$ is in CNF, if for every $j \in J$ and every $k \in K_j$, $c_k = \exists(a_k, d_k)$ or $c_k = \neg\exists(a_k, d_k)$ for some morphism a and some condition d_k in CNF.*

Given a condition in CNF, the actual resolution process begins and adds derived facts (disjunctions) to the conjunction. The goal is the addition of false as conjunct. If a true resolution calculus were possible for conditions, each refutation step "$\models$" would be of the form:

- *Select* two disjunctions $(\neg\exists(a,c)\vee c_1)$ and $(\exists(b,d)\vee c_2)$ from the conjunction such that $\exists(b,d)\models\exists(a,c)$.
- *Add* the *resolvent* $(c_1\vee c_2)$ to the conjunction.

Special case: if c_1 and c_2 do not exist, or equivalently, are false, the resolvent is false and the negated condition $\neg c$ is refuted (the goal).

However, to decide $\exists(b,d)\models\exists(a,c)$ is as hard as the original problem $\neg c\models$ true, as we can not dissolve nested subconditions. Therefore we need additional deduction rules that cope with this situation and create, manage and (hopefully) solve subproblems of the form $\exists(b,d)\models\exists(a,c)$. Formally, these deduction rules are defined as follows.

Definition 6 (deduction rules). *Let $c_1,\ldots,c_n,e$ be conditions. A* (deduction) rule *R has the form*

$$\boxed{\begin{array}{c} c_1\\ \vdots\\ c_n\\ \hline e \end{array}}\qquad \textit{if } \alpha$$

and is shortly denoted by $R=[c_1,\ldots,c_n/e]\alpha$. The conditions $c_1,\ldots,c_n$ are called premises*, e is the* resolvent *and α is an (informal)* side condition*. A rule may be applied to a condition c in CNF, if there is exists a non-negated subcondition c' in c such that $c'=\wedge_{j\in J}d_j$ is a conjunction of disjunctions $(d_j)_{j\in J}$ that contains all premises of R, i.e. for all $1\leq k\leq n$, there is a $j\in J$ with $c_k=d_j$, and the side condition α is satisfied. Application of R yields a new condition d that is derived from c by adding the resolvent e to the conjunction c'. We write $c\vdash_R d$ to denote such a derivation step, whereas we write $c\vdash_{\mathcal{K}} d$ to denote a derivation sequence $c\vdash_R\ldots\vdash_Q d$ with rules $R,\ldots,Q$ in $\mathcal{K}$.*

The deduction rules of our calculus contain variables for morphisms and conditions. Prior to a rule application, these variables must be matched in an unification process, as usual, to yield an applicable instance of the rule. For the formal definition of our calculus, we require the following theorem stating the possibility of combining two conditions $\exists p$ and c conjunctively:

Theorem 1 ([HP08, HP05]). *There is a transformation* A_2*, such that for every morphism p in $\mathcal{M}$ and every condition c over* $\mathrm{dom}(p)$ *the following holds: For all $p''\in\mathcal{M}$ with $\mathrm{dom}(p'')=\mathrm{codom}(p)$, $p''\models\mathrm{A}_2(p,c)\ \Leftrightarrow\ p''\circ p\models c$.*

The transformation A_2 is described in [HP08] as follows:

Construction 1. *For morphisms p in $\mathcal{M}$ and conditions over* $\mathrm{dom}(p)$*, let*

$$\begin{tikzcd}
\bullet \arrow[d, "a'"'] \arrow[dd, bend right=60, "b"'] \arrow[dr, phantom, "(1)"] & \bullet \arrow[l, hook', "p"'] \arrow[d, "a"] \\
\bullet \arrow[d, "e"'] & \bullet \arrow[l, hook', "q"] \arrow[dl, "r"] \\
\bullet &
\end{tikzcd}$$

$$\begin{aligned}\mathrm{A}_2(p,\,\mathrm{true}) &= \mathrm{true}\\ \mathrm{A}_2(p,\,\exists(a,c)) &= \vee_{e\in\mathcal{E}}\,\exists(b,\,\mathrm{A}_2(r,c)).\end{aligned}$$

Construct the pushout (1) of p and a leading to morphisms a' and q. The disjunction $\vee_{e\in\mathcal{E}}$ ranges over all epimorphisms e with domain $\mathrm{dom}(a')$ *such that both $b=e\circ a'$ and $r=e\circ q$ are in $\mathcal{M}$.*

Furthermore, $\mathrm{A}_2(p,\,\neg c)=\neg\mathrm{A}_2(p,\,c)$ *and* $\mathrm{A}_2(p,\,\wedge_{j\in J}c_j)=\wedge_{j\in J}\mathrm{A}_2(p,\,c_j)$*.*

3.1 A Calculus for High-Level Conditions

In the following, we introduce a calculus $\mathcal{K}$ for high-level conditions representing the possible actions a theorem prover based on $\mathcal{K}$ may perform.

Definition 7 (calculus $\mathcal{K}$). *The calculus $\mathcal{K}$ for high-level conditions consists of the following six rules: (Descent), (Resolve), (Partial resolve), (Partial lift), (Lift) and (Supporting lift). Let a, b, m be morphisms and let c, d, c_1, c_2 be conditions.*

(Descent)

$$\boxed{\dfrac{\exists(a,\ \text{false} \wedge c) \vee c_1}{c_1}}$$

(Resolve)

$$\boxed{\dfrac{\begin{array}{c}\neg\exists(a,\ \text{true}) \vee c_1\\ \exists(b,\ d) \vee c_2\end{array}}{c_1 \vee c_2}}$$

if $\exists m \in \mathcal{M}.\ m \circ a = b$ and $d \neq \text{false}$

$$\begin{tikzcd}\bullet \arrow[r, "a"] \arrow[dr, "b"'] & \bullet \arrow[d, hook, "m"] \\ & \bullet\end{tikzcd}$$

(Partial resolve)

$$\boxed{\dfrac{\begin{array}{c}\neg\exists(a,\ \text{true}) \vee c_1\\ \exists(b,\ d) \vee c_2\end{array}}{\neg\exists(m^*, \text{true}) \vee c_1 \vee c_2}}$$

if $\exists m \in \mathcal{M}.\ m \circ b = a$ and $\langle m^*, b^* \rangle$ is the $\mathcal{M}$-pushout complement of $\langle b, m \rangle$ and $d \neq \text{false}$

$$\begin{tikzcd}\bullet \arrow[r, "b"] \arrow[d, "m^*"'] \arrow[dr, "a"] & \bullet \arrow[d, hook, "m"] \\ \bullet \arrow[r, hook, "b^*"'] & \bullet\end{tikzcd}$$

(Partial lift)

$$\boxed{\dfrac{\begin{array}{c}\neg\exists(a, c) \vee c_1\\ \exists(b, d) \vee c_2\end{array}}{\exists(b,\ d \wedge \mathrm{A}_2(m, \neg c)) \vee c_1 \vee c_2}}$$

if $c \neq \text{true}$ and $\exists m \in \mathcal{M}.\ m \circ a = b$ and $d \neq \text{false}$

$$\begin{tikzcd}\bullet \arrow[r, "a"] \arrow[dr, "b"'] & \bullet \arrow[d, hook, "m"] \\ & \bullet\end{tikzcd}$$

(Lift)

$$\boxed{\dfrac{\begin{array}{c}\neg\exists(a, c) \vee c_1\\ \exists(b, d) \vee c_2\end{array}}{\exists(b,\ d \wedge \mathrm{A}_2(b, \neg\exists(a, c))) \vee c_1 \vee c_2}}$$

if $c \neq \text{true}$ and $b \in \mathcal{M}$ and $d \neq \text{false}$

(Supporting lift)

$$\boxed{\dfrac{\begin{array}{c}\exists(a, c) \vee c_1\\ \exists(b, d) \vee c_2\end{array}}{\exists(b,\ d \wedge \mathrm{A}_2(b, \exists(a, c))) \vee c_1 \vee c_2}}$$

if $b \in \mathcal{M}$ and $c \neq \text{false}$ and $d \neq \text{false}$

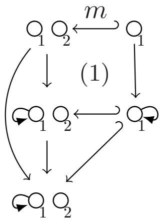
The rule (Descent) is used to carry over a successful nested refutation into an outer refutation. The rule (Resolve) is the core of our calculus and represents a straightforward case for which the problem $\exists(b, d) \models \exists(a, c)$ is decidable. The rules (Descent) and (Resolve) are the only ones that (may) reduce the number of elements in a disjunction. The rule (Partial resolve) is necessary for proving the validity of conditions outside the decidable $\forall$-free fragment of conditions. The rules (Partial lift), (Lift) and (Supporting lift) are similar in the sense that they create additional facts to find nested refutations by combining information. Outstandig is the rule (Partial lift) which moves the negation from a condition towards the nested subcondition. Note that (Supporting lift) is the only rule for which repeated applications on its own resolvent may be necessary. For graphs, unbounded applications of (Supporting lift) are the only reason a theorem prover based on $\mathcal{K}$ does not terminate, assuming that the deduction of structural equivalent conditions is surpressed, as discussed in Section 4.

Example 5. Consider the following tautology $\forall(\circ_1,\ \exists \circlearrowright_1) \Rightarrow \forall(\circ_1\,\circ_2,\ \exists \circlearrowright_1\,\circlearrowright_2)$ expressing "Every node has a loop implies every two nodes each have a loop". A transformation into CNF yields

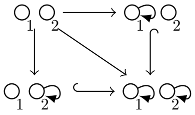
$$\begin{array}{ll} (1) & \neg\exists(\circ_1,\ \neg\exists \circlearrowright_1) \\ (2) & \wedge\ \exists(\circ_1\,\circ_2,\ \neg\exists \circlearrowright_1\,\circlearrowright_2) \end{array}$$

and a proof of the statement's validity is as follows:

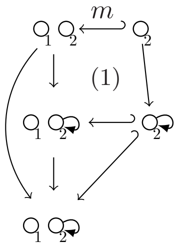
$$\begin{array}{|lll|}\hline (1) & \neg\exists(\circ_1,\ \neg\exists \circlearrowright_1) & \\ (2) & \exists(\circ_1\,\circ_2,\ \neg\exists \circlearrowright_1\,\circlearrowright_2) & \\ \hline (3) & \exists(\circ_1\,\circ_2,\ (3.1) & \neg\exists \circlearrowright_1\,\circlearrowright_2\ ) \\ & \qquad\quad (3.2) & \wedge\ \exists \circlearrowright_1\,\circ_2 \\ \hline \end{array}\ \text{(Partial lift)}$$



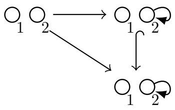
$$\begin{array}{|ll|}\hline (3.1) & \neg\exists \circlearrowright_1\,\circlearrowright_2 \\ (3.2) & \exists \circlearrowright_1\,\circ_2 \\ \hline (3.3) & \neg\exists \circ_1\,\circlearrowright_2 \\ \hline \end{array}\ \text{(Partial resolve)}$$



$$\begin{array}{|lll|}\hline (1) & \neg\exists(\circ_1,\ \neg\exists \circlearrowright_1) & \\ (3) & \exists(\circ_1\,\circ_2,\ (3.1) \wedge \ldots \wedge (3.3)) & \\ \hline (4) & \exists(\circ_1\,\circ_2,\ (4.1) & \neg\exists \circlearrowright_1\,\circlearrowright_2\ ) \\ & \qquad\quad (4.2) & \wedge\ \exists \circlearrowright_1\,\circ_2 \\ & \qquad\quad (4.3) & \wedge\ \neg\exists \circ_1\,\circlearrowright_2 \\ & \qquad\quad (4.4) & \wedge\ \exists \circ_1\,\circlearrowright_2 \\ \hline \end{array}\ \text{(Partial lift)}$$



$$\begin{array}{|ll|}\hline (4.3) & \neg\exists \circ_1\,\circlearrowright_2 \\ (4.4) & \exists \circ_1\,\circlearrowright_2 \\ \hline (4.5) & \text{false} \\ \hline \end{array}\ \text{(Resolve)}$$



$$\begin{array}{|ll|}\hline (4) & \exists(\circ_1\,\circ_2,\ (4.1) \wedge \ldots \wedge (4.5)) \\ \hline (5) & \text{false} \\ \hline \end{array}\ \text{(Descent)}$$

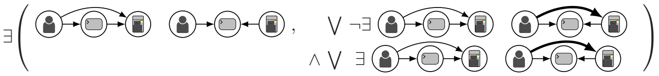
Example 6. Consider the condition stated in Example 4. Given the rules of $\mathcal{K}$, our goal is to refute the disjunction (3) with the help of the facts (1) and (2). The rule (Resolve) can be applied with argument (1) to resolve (3.1)-(3.6), e.g.,

$$\boxed{\frac{\begin{array}{l}(1)\\ (3.1)\lor((3.2)\lor\ldots\lor(3.11))\end{array}}{(3.2)\lor\ldots\lor(3.11)}}\ \text{(Resolve)}$$

Subconditions (3.7)-(3.11) are resolved by applying rule (Partial lift) with argument (2) and subsequent application of (Resolve) on the nested subconditions, and (Descent), e.g.,

$$\boxed{\frac{\begin{array}{l}(2)\\ (3.7)\lor((3.8)\lor\ldots\lor(3.11))\end{array}}{(3.7')\lor(3.8)\lor\ldots\lor(3.11)}}\ \text{(Partial lift)}$$

with (3.7')



Eventually, we yield an empty disjunction, or equivalently, false as an element of the outer conjunction, thus the input condition is refuted and the condition proved.

3.2 Soundness

In this section, we prove the soundness of the calculus $\mathcal{K}$. We show that every application of a rule R in $\mathcal{K}$ corresponds to a logical deduction.

Theorem 2 (soundness of $\mathcal{K}$). *The calculus $\mathcal{K}$ for high-level conditions is sound, i.e., for every conditions c, d over C in CNF the following holds:*

$$c \vdash_{\mathcal{K}} d \quad \textit{implies} \quad c \models d.$$

The proof is done in three steps: first, we establish that we can investigate the soundness of deduction rules independently of disjunctive context. In the following, let r, p_j, q_j be conditions for $1 \leq j \leq n$.

Fact 3. *For every rule $R = [(p_1 \lor q_1), \ldots, (p_n \lor q_n)/(r \lor q_1 \lor \ldots \lor q_n)]\alpha$ we have $(p_1 \land \ldots \land p_n) \models r$ implies $((p_1 \lor q_1) \land \ldots \land (p_n \lor q_n)) \models (r \lor q_1 \lor \ldots \lor q_n)$.*

Second, we can prove the soundness of each individual rule R in $\mathcal{K}$.

Lemma 1. *For every rule $R = [c_1, \ldots, c_n/d]\alpha$ in $\mathcal{K}$, if α holds then $(c_1 \land \ldots \land c_n) \models d$.*

Proof. For the rule (Descent), we have $\exists(a,\ \text{false} \land c) \equiv \exists(a,\ \text{false}) \equiv \text{false}$. For every rule of the form $R = [(p_1 \lor q_1), \ldots, (p_n \lor q_n)/(r \lor q_1 \lor \ldots \lor q_n)]\alpha$ with $c_j = (p_j \lor q_j)$ for $1 \leq j \leq n$, we first show $(p_1 \land \ldots \land p_n) \models r$:

(Resolve). First, we transform the proof obligation:

$(\neg\exists(a,\ \text{true}) \land \exists(b,\ d)) \Rightarrow \text{false}$

$\equiv \neg(\neg\exists(a,\ \text{true}) \land \exists(b,\ d)) \lor \text{false}$ (Def. $\Rightarrow$)

$\equiv \exists(a,\ \text{true}) \lor \neg\exists(b,\ d) \lor \text{false}$ (De Morgan)

$\equiv \exists(a,\ \text{true}) \lor \neg\exists(b,\ d)$ (($c \lor \text{false}) \equiv c$)

$\equiv \exists(a,\ \text{true}) \Leftarrow \exists(b,\ d)$ (Def. $\Rightarrow$)

We show $\exists(b,\, d) \models \exists(a,\, \text{true})$:

$\quad\exists(b,\, d)$
$\models \exists(b,\, \text{true})$ $\quad (d \models \text{true})$
$\models \exists(a,\, \text{true})$ $\quad (\exists m \in \mathcal{M}.\ m \circ a = b$, Def. 2)

$$\begin{tikzcd}
\bullet \arrow[r, "a"] \arrow[dr, "b"] \arrow[ddr, bend right, "p"'] & \bullet \arrow[d, hook, "m"] \arrow[dd, hook, bend left=60, "q"] \\
& \bullet \arrow[d, hook, "r"] \\
& \bullet
\end{tikzcd}$$

(Partial resolve). First, we transform the proof obligation:

$\quad(\neg\exists(a,\, \text{true}) \wedge \exists(b,\, d)) \Rightarrow \neg\exists(m^*,\ \text{true})$
$\equiv \neg(\neg\exists(a,\, \text{true}) \wedge \exists(b,\, d)) \vee \neg\exists(m^*,\ \text{true})$ $\quad$ (Def. $\Rightarrow$)
$\equiv \exists(a,\, \text{true}) \vee \neg\exists(b,\, d) \vee \neg\exists(m^*,\ \text{true})$ $\quad$ (De Morgan)
$\equiv \exists(a,\, \text{true}) \Leftarrow \neg(\neg\exists(b,\, d) \vee \neg\exists(m^*,\ \text{true}))$ $\quad$ (Def. $\Rightarrow$)
$\equiv \exists(a,\, \text{true}) \Leftarrow (\exists(b,\, d) \wedge \exists(m^*,\ \text{true}))$ $\quad$ (De Morgan)

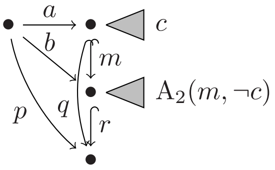
We show $(\exists(b,\, d) \wedge \exists(m^*,\ \text{true})) \models \exists(a,\, \text{true})$:

$\quad p \models (\exists(b,\, d) \wedge \exists(m^*,\ \text{true}))$
$\Leftrightarrow \exists q \in \mathcal{M}.\ q \circ b = p$ and $q \models d$
$\quad$ and $\exists r \in \mathcal{M}.\ r \circ m^* = p$ and $r \models \text{true}$ $\quad$ (Def. 2)
$\Rightarrow \exists s \in \mathcal{M}.\ r \circ m^* = s \circ b^* \circ m^* = s \circ a = p$
$\quad$ and $s \models \text{true}$ $\quad$ ($\mathcal{M}$-Pushout)

$$\begin{tikzcd}
\bullet \arrow[r, "b"] \arrow[d, "m^*"'] \arrow[dr, "a"] \arrow[ddr, bend right=70, "p"'] & \bullet \arrow[d, hook, "m"] \arrow[dd, hook, bend left=60, "q"] \\
\bullet \arrow[r, hook, "b^*"] \arrow[dr, hook, "r"'] & \bullet \arrow[d, hook, "s"] \\
& \bullet
\end{tikzcd}$$

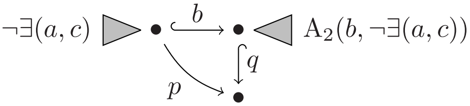
(Partial lift). We show $\exists(b, d) \wedge \neg\exists(a, c) \models \exists(b,\ d \wedge \mathrm{A}_2(m, \neg c))$:

$\quad p \models \exists(b, d) \wedge \neg\exists(a, c)$
$\Leftrightarrow p \models \exists(b, d)$ and $p \models \neg\exists(a, c)$ $\quad$ (Def. 2)
$\Leftrightarrow \exists r \in \mathcal{M}.\ r \circ b = p$ and $r \models d$ and $r \circ b \models \neg\exists(a, c)$ $\quad$ (Def. 2)
$\Rightarrow \exists r \in \mathcal{M}.\ r \circ b = p$ and $r \models d$ and
$\quad \exists r \circ m \in \mathcal{M}.\ r \circ m \circ a = p$ and $r \circ m \models \neg c$ $\quad$ ($m \circ a = b$, Def. 2)
$\Leftrightarrow \exists r \in \mathcal{M}.\ r \circ b = p$ and $r \models d$ and
$\quad \exists r \circ m \in \mathcal{M}.\ r \circ m \circ a = p$ and $r \models \mathrm{A}_2(m, \neg c)$ $\quad$ (Thm. 1)
$\Leftrightarrow \exists r \in \mathcal{M}.\ r \circ b = p$ and $r \models d$ and $r \models \mathrm{A}_2(m, \neg c)$ $\quad$ ($m \circ a = b$, Def. 2)
$\Leftrightarrow p \models \exists(b,\ d \wedge \mathrm{A}_2(m, \neg c))$ $\quad$ (Def. 2)



(Lift). We show $\exists(b, d) \wedge \neg\exists(a, c) \models \exists(b,\ d \wedge \mathrm{A}_2(b, \neg\exists(a, c)))$:

$\quad p \models \exists(b, d) \wedge \neg\exists(a, c)$
$\Leftrightarrow \exists q \in \mathcal{M}.\ q \circ b = p$ and $q \models d$ and $q \circ b \models \neg\exists(a, c)$ $\quad$ (Def. 2)
$\Leftrightarrow \exists q \in \mathcal{M}.\ q \circ b = p$ and $q \models d$ and $q \models \mathrm{A}_2(b, \neg\exists(a, c))$ $\quad$ (Thm. 1)
$\Leftrightarrow p \models \exists(b,\ d \wedge \mathrm{A}_2(b, \neg\exists(a, c)))$ $\quad$ (Def. 2)



(Supporting lift). The proof is analogous to (Lift) except $\neg\exists(a, c)$ is replaced with $\exists(a, c)$.

By Fact 3, we can lift any statement $(p_1 \wedge \ldots \wedge p_n \models r)$ for any disjunctive context $q_1, \ldots, q_n$ and yield $(p_1 \vee q_1) \wedge \ldots \wedge (p_n \vee q_n) \models (r \vee q_1 \vee \ldots \vee q_n)$. This concludes the soundness proof for the deduction rules in $\mathcal{K}$.

Third, we show that deductions concerning non-negated subconditions within a condition in CNF can be lifted to the whole condition.

Fact 4. *For any non-negated condition c' within a condition c over C in CNF, with d derived from c by replacing c' with d', we have $c' \models d'$ implies $c \models d$.*

Proof. By induction over the structure of conditions.
Basis: $c = c' \models d' = d$.
Step: We show, for all morphisms m in $\mathcal{M}$ with domain C:
Case $\exists(a, c)$: $m \models \exists(a, c)$ iff $(\exists q \in \mathcal{M}.\ m = q \circ a$ and $q \models c)$ implies $(\exists q \in \mathcal{M}.\ m = q \circ a$ and $q \models d)$ iff $m \models \exists(a, d)$.
Case $(c \wedge e)$: $m \models (c \wedge e)$ iff $(m \models c$ and $m \models e)$ implies $(m \models d$ and $m \models e)$ iff $m \models (d \wedge e)$.
Case $(c \vee e)$: analogous to $(c \wedge e)$.
The case $\neg c$ is excluded by the assumption that c' is a non-negated subcondition.

Finally, we can prove the soundness of $\mathcal{K}$.

Proof of Theorem 2. Let c, d be arbitrary conditions over C in CNF. A deduction $c \vdash_{\mathcal{K}} d$ is a sequence of deductions $c \vdash_R \ldots \vdash_Q d$ for rules $R, \ldots, Q$ in $\mathcal{K}$. Using induction over the length of the deduction, we can reduce the proof obligation to "$c \vdash_R d$ implies $c \models d$", where c, d are arbitrary conditions over C in CNF and $R = [c_1, \ldots, c_n/e]\alpha$ is an arbitrary deduction rule in $\mathcal{K}$. Assume, $c \vdash_R d$. By Definition 6, there is a non-negated subcondition c' which is a conjunction $(c_1 \wedge \ldots \wedge c_n \wedge q)$ and d is derived from c by adding e to the conjunction, i.e. $(c_1 \wedge \ldots \wedge c_n \wedge q) \vdash (e \wedge c_1 \wedge \ldots \wedge c_n \wedge q)$. By Lemma 1, we have $(c_1 \wedge \ldots \wedge c_n) \models e$. Consequently, $(c_1 \wedge \ldots \wedge c_n \wedge q) \models (e \wedge q)$. By Fact 4, we conclude $c \models d$.

3.3 Necessity

In the following, we investigate whether or not a rule is necessary, for every rule of the calculus $\mathcal{K}$. A rule R is necessary, if there exists a tautology which cannot be proven anymore if R is omitted. We show that (Resolve) could be omitted, if the artificial restriction $c \neq \text{true}$ is omitted from (Partial lift), and show that all other rules are necessary. However, our considerations do not exclude the existence of a smaller calculus with similar or different rules.

Fact 5. *For every deduction $c' \vdash_{(Resolve)} f'$, there is a sequence of deductions*

$$c' \vdash_{(Partial\ lift')} d' \vdash_{(\neg \text{true} \equiv \text{false})} e' \vdash_{(Descent)} f'$$

where (Partial lift')=$[\exists(b, d) \wedge \neg\exists(a, c)/\exists(b,\ d \wedge \mathrm{A}_2(m, \neg c))]\alpha$ and $\alpha = (\exists m \in \mathcal{M}.\ m \circ a = b$ and $d \neq \text{false})$.

Proof. Let $c' = (\neg\exists(a, \text{true}) \wedge \exists(b, d))$. Then $c' \vdash_{(\text{Partial lift'})} d' \vdash_{(\neg\text{true} \equiv \text{false})} e' \vdash_{(\text{Descent})} f'$ where $d' = \exists(b,\ d \wedge \mathrm{A}_2(m, \neg\text{true})) = \exists(b,\ d \wedge \neg\text{true})$ and

$e' = \exists(b,\ d \wedge \mathrm{false})$ and $f' = \mathrm{false}$. Using Fact 3 and Lemma 1, our considerations can be lifted to arbitrary c',f' with $c' \vdash_{(\mathrm{Resolve})} f'$.

We propose to let shortcut (Resolve) remain in $\mathcal{K}$, as the side conditions of (Resolve) and (Partial lift) prevent both rules from being applicable simultaneously.

Fact 6. *For every rule $R \in \mathcal{K} \setminus \{$(Resolve)$\}$, there is a condition c such that "$\neg c \vdash_{\mathcal{K}}$ false" and not "$\neg c \vdash_{\mathcal{K}\setminus\{R\}}$ false".*

Proof. **(Descent)** Negation of the tautology $\neg\exists(\circ,\ \mathrm{false})$ with the meaning "No node satisfies false" yields $\exists(\circ,\ \mathrm{false})$. No other rule is applicable.

(Partial resolve) Negation of the tautology $\forall(\circ_1\,\circ_2,\ \exists\circlearrowright_1\,\circlearrowright_2 \vee \neg\exists\circ_1\,\circlearrowright_2 \vee \neg\exists\circlearrowright_1\,\circ_2)$ with the meaning "For every pair of nodes, either both have loops or the first node has no loop or the second node has no loop" yields $\exists(\circ_1\,\circ_2,\ \neg\exists\circlearrowright_1\,\circlearrowright_2 \wedge \exists\circ_1\,\circlearrowright_2 \wedge \exists\circlearrowright_1\,\circ_2)$. Only the rule (Partial resolve) can derive the intermediate fact $\exists\circ_1\,\circlearrowright_2$, or alternatively $\exists\circlearrowright_1\,\circ_2$, required for a refutation.

(Partial lift) Negation of the tautology $\exists(\circ_1,\ \neg\exists\circlearrowright_1) \Rightarrow \neg\forall(\circ_1,\ \exists\circlearrowright_1)$ with the meaning "There is a node without a loop implies not every node has a loop" yields $\exists(\circ_1,\ \neg\exists\circlearrowright_1) \wedge \neg\exists(\circ_1,\ \neg\exists\circlearrowright_1)$ and only the use of (Partial lift) leads to a successful refutation as it negates the subcondition $\neg\exists\circlearrowright_1$:

$$\boxed{\begin{array}{l} \neg\exists(\circ_1,\ \neg\exists\circlearrowright_1\ ) \\ \exists(\circ_1,\ \neg\exists\circlearrowright_1\ ) \\ \hline \exists(\circ_1,\ \neg\exists\circlearrowright_1\ \wedge \exists\circlearrowright_1\ ) \end{array}} \quad \text{(Partial lift)}$$

(Lift) Negation of the tautology $\neg\exists\circlearrowright \Rightarrow \forall(\circ_1,\ \neg\exists\circlearrowright_1 \vee \exists\circ_1\,\circ_2)$ with the meaning "There is no node with a loop implies for all nodes, there is no loop or there is a second node" yields $\neg\exists\circlearrowright \wedge \exists(\circ_1,\ \exists\circlearrowright_1 \wedge \neg\exists\circ_1\,\circ_2)$ and no rule other than (Lift) is applicable. Note, partial resolve is not applicable because the ($\mathcal{M}$-)pushout complement is non-existent.

$$\begin{tikzcd} \emptyset \arrow[r, "b"] \arrow[d] \arrow[dr, "a"] & \circ \arrow[d, hook, "m"] \\ \circlearrowright \arrow[r, hook] & \circlearrowright \end{tikzcd}$$

(Supporting lift) Negation of the tautology $\exists\circlearrowright_1 \Rightarrow \forall(\circ_1,\ \exists\circlearrowright_1 \vee\exists\ \circ_1\,\circlearrowright_2)$ with the meaning "There is a node with a loop implies for all nodes, the node has a loop or there is a second node with a loop" yields $\exists\circlearrowright_1 \wedge \exists(\circ_1,\ \neg\exists\circlearrowright_1 \wedge \neg\exists\circ_1\,\circlearrowright_2)$ and no rule other than (Supporting lift) is applicable.

4 Implementation

In this section, we discuss practical aspects of a theorem prover based on $\mathcal{K}$. The deduction rules represent the main computation steps a theorem prover based on $\mathcal{K}$ will perform. Besides an implementation of those deduction rules, one requires a method that transforms any condition into conjunctive $\mathcal{M}$-normal form.

The equivalences depicted in Figure 2, strictly read from left to right, can be applied as long as possible to transform any condition into an (optimized) condition in conjunctive $\mathcal{M}$-normal form. In [Pen04], the equivalences are proven and it is shown that an as long as possible application yields the desired normal form. Another implementational aspect is the prevention of redundancy of rule applications with the intent to contain non-termination as far as possible. For example, any of the rules (Partial lift), (Lift) or (Supporting lift) may add subconditions to a conjunction that are already present anyway. Repeated application of such a rule on its own resolvent would lead into an infinite redundant branch of the search space. In theses cases, a notion of structural equivalence can help to filter out double subconditions and to prevent unnecessary deductions: Two conditions c, d are said to be *structurally equivalent*, denoted by $c \mathrel{\widehat{=}} d$, if $c = \text{true} = d$, or if $c = \neg c'$, $d = \neg d'$ and c', d' are structurally equivalent, or if $c = (c_1 \wedge c_2)$, $d = (d_1 \wedge d_2)$ and at least $(c_1, d_1$ and $c_2, d_2)$ or $(c_1, d_2$ and $c_2, d_1)$ are structurally equivalent (case $\vee$ analogous), or if $c = \exists(a, c')$, $d = \exists(a, d')$ and c', d' are structurally equivalent. The applicability of deduction rules may then be restricted to those cases for which the resolvent is not structurally equivalent to already existing conditions. Except for the rule (Supporting lift), this effectively prevents recursive application of rules to derived conditions.

$$
\begin{array}{rlrl}
\exists a & \equiv \exists(a, \text{true}) & \exists(\text{id},\ c) & \equiv c \\
\forall(a, c) & \equiv \neg\exists(a, \neg c) & \exists(a,\ \text{false}) & \equiv \text{false} \\
\exists(a, c) & \equiv \text{false} \quad \text{if } a \notin \mathcal{M} & \exists(a,\ \exists(b, c)) & \equiv \exists(b \circ a,\ c) \\
\neg\neg c & \equiv c & \exists(a,\ \vee_{j \in J}\ c_j) & \equiv \vee_{j \in J}\ \exists(a,\ c_j) \\
\neg\text{true} & \equiv \text{false} & \vee_{j \in J}\, c_j & \equiv \text{true} \quad \text{if } \exists k \in J.\ c_k = \text{true} \\
\neg\text{false} & \equiv \text{true} & \vee_{j \in J}\, c_j & \equiv \vee_{j \in J \setminus \{k\}}\, c_j \quad \text{if } \exists k \in J.\ c_k = \text{false} \\
\neg(\vee_{j \in J}\ c_j) & \equiv (\wedge_{j \in J}\ \neg c_j) & \wedge_{j \in J}\, c_j & \equiv \wedge_{j \in J \setminus \{k\}}\, c_j \quad \text{if } \exists k \in J.\ c_k = \text{true} \\
\neg(\wedge_{j \in J}\ c_j) & \equiv (\vee_{j \in J}\ \neg c_j) & \wedge_{j \in J}\, c_j & \equiv \text{false} \quad \text{if } \exists k \in J.\ c_k = \text{false} \\
((\wedge_{j \in J}\ c_j) \vee c) & \equiv (\wedge_{j \in J}\ (c_j \vee c)) & \vee_{j \in \emptyset}\, c_j & \equiv \text{false} \\
 & & \wedge_{j \in \emptyset}\, c_j & \equiv \text{true}
\end{array}
$$

Fig. 2. Equivalences for conjunctive $\mathcal{M}$-normal form

5 Related Work

In this section, we briefly relate our results to other work. Earliest attempts to find deduction rules for graphical conditions are made by Koch et. al. [KMP05], but remain incomplete. They investigate the notion of conflicting conditions of the form $\forall(I \to X,\ \exists(X \to C))$ and state prerequisites under which a conjunction of two graph conditions of this form is unsatisfiable.

Independently to our work, Orejas et. al. [OEP08] investigate sound and complete calculi for three fragments of graph conditions: the fragment of Boolean conditions over basic existential conditions $\exists(I \to C)$, the fragment of Boolean conditions over basic existential conditions $\exists(I \to C)$ and non-negated "atomic" conditions of the form $\forall(I \to X,\ \exists(X \to C))$, and the fragment of Boolean Conditions over "atomic" conditions of the form $\forall(I \to X,\ \exists(X \to C))$. Their deduction rules relate to our own as follows: (R1) is a special case of (Resolve), (R2) is comparable to the rule (Supporting Lift), and (R3) is comparable to the rule (Partial lift), although (R2), (R3) do not lift (parts of) the resolvent (as this is neither necessary nor possible for the considered fragments of conditions). The operator $\oplus$ is a special instance of A_2 restricted to basic existential conditions. In [Ore08], a sound and complete calculus for the fragment of "basic" and "positive atomic" attributed graph constraints is presented. Attributed graph constraints are conditions over attributed graphs combined with a formula expressing conditions on the attributes such as "$(x > y)$".

An alternative approach to apply theorem proving to graph transformation is a translation into logical formulas [Cou90]. Following this idea, Strecker [Str08] models graph transformation in the proof assistant Isabelle. His approach supports the manual verification of formulas of "a fragment of first-order logic enriched by transitive closure".

The relation to our previous work is as follows: In [Pen08], a correct and complete satisfiability algorithm named `SeekSat` is described. For the fragment of Boolean conditions over basic existential conditions $\exists a$, `SeekSat` is shown to terminate, thus is able to decide. While `SeekSat` covers contradictions with finite counterexamples and tautologies in the decidable fragment of conditions, the presented calculus $\mathcal{K}$ is intended to cover all tautologies with finite proofs.

6 Conclusion

In this paper, we presented a calculus for conditions over adhesive high-level replacement categories. We took resolution [Rob65] as an ideal and postulated six deduction rules able to refute conditions in conjunctive normal form. We proved that every rule application corresponds to a logical deduction, and investigated whether or not omission of any rule leads to an incomplete calculus. We discussed practical aspects concerning an implementation such as filtering out structural equivalent conditions, and briefly compared our results with related work. An implementation of $\mathcal{K}$ is currently under development. Future topics include

– a proof of the completeness of the calculus,
– a systematic evaluation of the implementation. Currently, all 74 example tautologies generated from correct program specifications of the "access control" example in [HPR06] can be proved in average 9.1 seconds (median 0.1s) (Intel T5600, 1.83GHz).

Acknowledgment. Many thanks to the referees for thoroughly reviewing the paper and suggesting several improvements.

References

[BFT06] Baumgartner, P., Fuchs, A., Tinelli, C.: Implementing the model evolution calculus. Int. Journal on Artificial Intelligence Tools 15(1), 21–52 (2006)

[Cou90] Courcelle, B.: Graph rewriting: An algebraic and logical approach. In: Handbook of Theoretical Computer Science, vol. B, pp. 193–242. Elsevier, Amsterdam (1990)

[DS89] Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1989)

[EEHP06] Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae 74, 135–166 (2006)

[EEPT06] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science. Springer, Berlin (2006)

[HP01] Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 230–245. Springer, Heidelberg (2001)

[HP05] 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)

[HP08] Habel, A., Pennemann, K.-H.: Correctness of high-level transformation systems relative to nested conditions. In: MSCS 2008 (Accepted for publication, 2008)

[HPR06] Habel, A., Pennemann, K.-H., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer, Heidelberg (2006)

[HW95] Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars. In: SEGRAGRA 1995. ENTCS, vol. 2, pp. 95–104 (1995)

[KMP05] Koch, M., Mancini, L.V., Parisi-Presicce, F.: Graph-based specification of access control policies. JCSS 71, 1–33 (2005)

[McC08] McCune, W.: Homepage of Prover9 (2008), http://www.prover9.org/

[OEP08] Orejas, F., Ehrig, H., Prange, U.: A logic of graph constraints. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 179–199. Springer, Heidelberg (2008)

[Ore08] Orejas, F.: Attributed graph constraints. In: Ehrig, H., et al. (eds.) Proc. ICGT 2008. LNCS, vol. 5214, Springer, Heidelberg (2008)

[Pen04] Pennemann, K.-H.: Generalized constraints and application conditions for graph transformation systems. Master's thesis, Department of Computing Science, University of Oldenburg, Oldenburg (2004)

[Pen08] Pennemann, K.-H.: An algorithm for approximating the satisfiability problem of high-level conditions. In: Proc. GT-VC 2007. ENTCS, vol. 213, pp. 75–94. Elsevier, Amsterdam (2008)

[Ren04] Rensink, A.: Representing first-order logic by graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004)

[Rob65] Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM 12, 23–41 (1965)

[RV02] Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Communications 15(2-3), 91–110 (2002)

[Str08] Strecker, M.: Modeling and verifying graph transformations in proof assistants. In: Proc. Termgraph 2007. ENTCS, vol. 203, pp. 135–148. Elsevier, Amsterdam (2008)

Towards the Verification of Attributed Graph Transformation Systems*

Barbara König and Vitali Kozioura

Abteilung für Informatik und Angewandte Kognitionswissenschaft
Universität Duisburg-Essen, Germany

Abstract. We describe an approach for the verification of attributed graph transformation systems (AGTS). AGTSs are graph transformation systems where graphs are labelled over an algebra. We base our verification procedure on so-called approximated unfoldings combined with counterexample-guided abstraction refinement. Both techniques were originally developed for non-attributed systems. With respect to refinement we focus especially on detecting whether the spurious counterexample is caused by structural over-approximation or by an abstraction of the attributes which is too coarse. The technique is implemented in the verification tool Augur 2 and a leader election protocol has been successfully verified.

1 Introduction

For practical purposes modelling languages are usually extended with the possibility of adding data types and suitable operations. This is for instance done in coloured Petri nets [12] and attributed graph transformation systems (AGTSs) [18,9]. Extending a GTS with attributes allows one to combine the intuitive graphical aspects of the modelled systems with the natural data structures, which makes such extended GTSs more suitable for practical applications. In some cases attributes can be simulated artificially by encoding them into the graph structure (since GTSs are Turing-complete), but specifying attributes directly leads to more compact models. This is an advantage with respect to over-approximation techniques since we have more control over what is abstracted and in what way it is abstracted.

In the last years we have developed a verification technique for non-attributed graph transformation systems (GTSs) [3], which allowed us to successfully verify several case studies [7,2,16]. The technique approximates GTSs by Petri graphs (which are Petri nets with additional hypergraph structure) and refines the obtained Petri graph via counterexample-guided abstraction refinement (CEGAR) when necessary [15]. CEGAR is a standard program analysis technique which refines overly coarse approximations by looking for a spurious run, i.e., a run which violates the property to be verified, but which has no counterpart in the original system. Then the approximation is refined in such a way that the spurious run

* Supported by the DFG project SANDS and CRUI/DAAD Vigoni "Models based on Graph Transformation Systems: Analysis and Verification".

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 305–320, 2008.

disappears. This procedure can be repeated, but due to the undecidability of the verification problem it is not guaranteed that it will eventually give a definitive yes/no-answer.

In this paper we apply this technique, including abstraction refinement, to AGTSs. We describe our view on AGTSs as graph transformation systems labelled over an algebra and approximate AGTSs by attributed Petri graphs which are basically coloured Petri nets [12] or algebraic high-level nets [8] equipped with a hypergraph structure. After performing the approximation described in [3] attributes are added to the resulting Petri graph, which can then be analyzed as a coloured Petri net. Since the carrier sets underlying data types are often infinite, we additionally need attribute abstraction, which is standard in the framework of abstract interpretation [5]. In the conclusion we will discuss how the approach might be extended to predicate abstraction [10,11]. The verification technique for AGTSs presented here was implemented in Augur 2[1] [14] and we introduce a case study concerning a leader election protocol and describe how it has been verified with Augur 2.

2 Attributed Graph Transformation Systems

2.1 Algebras

In this section we describe attributed graph transformation systems (AGTSs). After introducing the (standard) notion of algebra and the (non-standard) notion of Boolean algebra, we show how to define and rewrite attributed graphs.

Definition 1 (signature, algebra). *A* signature Σ *is a pair* $\langle \mathcal{S}, \mathcal{F} \rangle$ *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 will also be called* types.

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* $\sigma(f) = (s_1 \dots s_n, s)$.

For a Boolean Σ-algebra *we require that* $\mathcal{S}$ *contains the sort Bool and that we have two subsets* $T_{\mathcal{A}}, F_{\mathcal{A}} \subseteq \mathcal{A}_{Bool}$ *representing the truth values.*

By $T(\Sigma, X)$ *we denote the usual* Σ-term algebra, *where* X *is a set of variables, each equipped with a fixed 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 1. In our implementation we use an algebra denoted by $\mathcal{C}$ with sorts *Bool*, *Int*, *Str*, *Unit* (which have as carrier sets the standard truth values, integers, strings and one-element set respectively) and tuples over the first three sorts. We consider standard operations, for instance $+, -, *, /$ for the integers and comparison operators $<, \leq, =$ in order to obtain truth values. Operators can also be extended to functions operating on tuples.

We will now define a specific type of algebra needed in the following.

[1] The tool is available at `http://www.ti.inf.uni-due.de/research/augur/`.

Definition 2 (powerset algebra). *For a given Σ-algebra $\mathcal{A}$ we will denote by $\mathcal{P}(\mathcal{A})$ its* powerset algebra *which is an algebra over the same signature. The carrier sets of $\mathcal{P}(\mathcal{A})$ are the powersets of the original carrier sets, i.e., $\mathcal{P}(\mathcal{A})_s = \mathcal{P}(\mathcal{A}_s)$ and function symbols f with $\mathcal{F}(f) = (s_1 \dots s_n, s)$ are interpreted as:*

$$f^{\mathcal{P}(\mathcal{A})}(A_1, \dots, A_n) = \{f^{\mathcal{A}}(a_1, \dots, a_n) \mid a_i \in A_i\},$$

where $A_i \in (\mathcal{P}(\mathcal{A}))_{s_i}$. In the case of a Boolean algebra we set $T_{\mathcal{P}(\mathcal{A})} = \{A' \subseteq \mathcal{A}_{Bool} \mid A' \cap T_{\mathcal{A}} \neq \emptyset\}$ and similarly for $F_{\mathcal{P}(\mathcal{A})}$.

Note that in the case of our example algebra $\mathcal{C}$ we have four truth values in $\mathcal{P}(\mathcal{C})$ where $T_{\mathcal{P}(\mathcal{C})} = \{\{\mathit{true}\}, \{\mathit{true}, \mathit{false}\}\}$, $F_{\mathcal{P}(\mathcal{C})} = \{\{\mathit{false}\}, \{\mathit{true}, \mathit{false}\}\}$. Going to powersets is a necessary step since the concretization of abstract values, which will be introduced later, provides us with an entire set of values, as opposed to a single value. However if we only work with single values, i.e., one-element sets, we will get exactly the same results as in the original algebra.

Finally we need a notion of algebra homomorphism.

Definition 3 (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 $\sigma(f) = (s_1 \dots s_n, s_{n+1})$ we have*

$$h_{s_{n+1}}(f^{\mathcal{A}}(a_1, \dots, a_n)) = f^{\mathcal{B}}(h_{s_1}(a_1), \dots, h_{s_n}(a_n)).$$

2.2 Attributed Graphs

We will now define the notion of graphs we are working with. We consider a fixed set of labels Λ and we start with the definition of hypergraphs and their morphisms.

Definition 4 (hypergraph). *A* hypergraph *G is a tuple (V_G, E_G, c_G, l_G), where V_G is a finite set of nodes, E_G is a finite set of edges, $c_G : E_G \to V_G^*$ is a connection function, and $l_G : E_G \to \Lambda$ is the labeling function.*

We consider a fixed typing function $\mathit{ltype} : \Lambda \to \mathcal{S}$ which associates a sort to each label. The theory could be easily extended to associating several (named) attributes to each label and this is how it is handled in our implementation.

We are now ready to introduce attributed hypergraphs. Note that here we choose a different representation of attributed graphs than in [9] where the focus is on viewing attributed graphs in the framework of adhesive HLR categories and where graphs include specific data nodes. One of our main concerns is to fully separate the graph structure and the attributes for verification purposes.

Definition 5 (attributed hypergraph). *Let $\mathcal{A}$ be a Σ-algebra. An $\mathcal{A}$-*attributed hypergraph *is a tuple $G = (V_G, E_G, c_G, l_G, \mathit{attr}_G)$, where (V_G, E_G, c_G, l_G) is a labelled hypergraph and $\mathit{attr}_G : E_G \to \mathcal{A}_{\mathcal{S}}$ is a function such that for each $e \in E_G$ it holds that $\mathit{attr}_G(e) \in \mathcal{A}_{\mathit{ltype}(l_G(e))}$.*

We consider nodes of a hypergraph as unlabelled (and without attributes). Attributes can be added by providing nodes with unary hyperedges which contain the attribute for that node.

Definition 6 (hypergraph morphisms). *Let G_1, G_2 be two hypergraphs. A* (hypergraph) morphism $\varphi : G_1 \to G_2$ *consists of two total functions* $\varphi_V : V_{G_1} \to V_{G_2}$ *and* $\varphi_E : E_{G_1} \to E_{G_2}$ *such that for every* $e \in E_{G_1}$ *it holds that* $l_{G_1}(e) = l_{G_2}(\varphi_E(e))$ *and* $\varphi_V(c_{G_1}(e)) = c_{G_2}(\varphi_E(e))$. *A morphism is called* edge-bijective *(*edge-injective*) whenever it is bijective (injective) on edges. (We will in the following drop the subscripts V and E.)*

Definition 7 (morphisms of attributed hypergraphs). *Let G_1, G_2 be two attributed hypergraphs (where G_1 is attributed over $\mathcal{A}$ and G_2 over $\mathcal{B}$). An* attributed hypergraph morphism $\varphi = (\psi, h) : G_1 \to G_2$ *consists of a hypergraph morphism ψ and an algebra homomorphism $h : \mathcal{A} \to \mathcal{B}$ such that*

$$\forall e \in E_{G_1} : attr_{G_2}(\psi(e)) = h(attr_{G_1}(e)).$$

2.3 Rewriting of Attributed Graphs

Attributed hypergraphs can be transformed using rewriting rules which we define in the following. Our approach follows essentially the presentation in [18], but without using category theory. Furthermore we use the same restrictions on rules as in [3] since this greatly simplifies verification.

Definition 8 (attributed rewriting rule). *We fix a signature Σ and a set X of variables. An* attributed rewriting rule *r is a quadruple (L, R, α, g), where L and R are $T(\Sigma, X)$-attributed hypergraphs, called left-hand side and right-hand side respectively, $\alpha : V_L \to V_R$ is an injective mapping, indicating how nodes are preserved, and $g \in T(\Sigma, X)$ is a guard condition of sort Bool.*

We demand that each of the term attributes of L is a single variable of X (such that each variable appears only once). The set of all variables in the left-hand side is denoted by X'. The right-hand side R may be attributed with arbitrary terms from $T(\Sigma, X')$. Each rule r is associated with a guard expression $g(r) \in T(\Sigma, X')_{Bool}$.

We demand also that there are no isolated nodes in the left-hand side L and no isolated nodes in $V_R - \alpha(V_L)$. Additionally E_L must not be empty and there can not be two edges with the same label in the left-hand side of a rule.

If an instance of the left-hand side is found in the current state of the system, then this rule can be applied and the instance of the left-hand side of the rule will be replaced by its right-hand side. We are now ready to define the notion of attributed graph transformation systems.

Definition 9 (attributed graph transformation system (AGTS)). *An* attributed graph transformation system (AGTS) $\mathcal{G} = (\mathcal{R}, G_0)$ *over an algebra $\mathcal{A}$ is a finite set of attributed rewriting rules $\mathcal{R}$ together with an $\mathcal{A}$-attributed start hypergraph G_0 (also called initial graph).*

We now describe in a set-based notation how rules can be applied to attributed graphs. This could also be done categorically.

Definition 10 (rule application). *A* match *of a rewriting rule $r = (L, R, \alpha, g)$ in an $\mathcal{A}$-attributed graph G is a morphism $\psi = (\varphi, h) : L \to G$ which is injective on edges. We can apply r to a match in G obtaining a new graph H, written $G \overset{r}{\Rightarrow} H$, whenever the guard expression is satisfied, i.e., $h(g) \in T_{\mathcal{A}}$. The target graph H is defined as follows*

$$V_H = V_G \uplus (V_R - \alpha(V_L)) \qquad E_H = (E_G - \varphi(E_L)) \uplus E_R$$

and, defining $\overline{\varphi} : V_R \to V_H$ by $\overline{\varphi}(\alpha(v)) = \varphi(v)$ if $v \in V_L$ and $\overline{\varphi}(v) = v$ otherwise, the source, target, labelling and attribute functions are given by

$$\begin{aligned} e \in E_G - \varphi(E_L) \quad &\Rightarrow \quad c_H(e) = c_G(e), \quad l_H(e) = l_G(e), \quad \mathit{attr}_H(e) = \mathit{attr}_G(e) \\ e \in E_R \quad &\Rightarrow \quad c_H(e) = \overline{\varphi}(c_R(e)), \quad l_H(e) = l_R(e), \quad \mathit{attr}_H(e) = h(\mathit{attr}_R(e)) \end{aligned}$$

That is, a left-hand side is found and replaced by the corresponding right-hand side. We use a restricted version of the DPO (double-pushout) approach where we only allow discrete interfaces. Merging as well as deletion of nodes is forbidden. Edges, however, can be deleted. The new attributes in the right-hand side are obtained by using h, the *binding* of the set of *free variables* X' of the left-hand side.[2]

Example 2. We use the simple AGTS shown in Fig. 1 as a running example. Edges labelled B and C have integer attributes. The attribute in B is increased by one whenever a new edge is created, whereas the attribute in C is multiplied with the corresponding attribute in B when C crosses B. The edges A and Error have no attributes. The property we want to verify is that no Error edge will ever be created. Note that intuitively this holds since no edge labelled 7 will ever be created and hence rule "Cross Backward" will never be applied, since C will always contain a even attribute value.

3 Approximation of Attributes

In Example 1 we considered an algebra with infinite carrier sets. In order to analyse the systems thus obtained we need a mechanism of attribute approximation. Hence we work in the framework of abstract interpretation [5] and start with the notion of a Galois connection, which is basically a pair of adjoints.

Definition 11 (Galois connection on algebras). *Let $\Sigma = \langle \mathcal{S}, \mathcal{F} \rangle$ be a signature and let $\mathcal{A}$, $\mathcal{B}$ be two algebras over this signature, where each carrier set is lattice-ordered via $\sqsubseteq$.*[3]

[2] Note that in the case of a powerset algebra some elimination of over-approximation could be useful, by removing attribute values in the right-hand side that did not satisfy the guard expression. In order to be able to represent the theory in a compact way we choose not to follow this path at the moment.

[3] The partial order $\sqsubseteq$ stands for the information ordering. Intuitively whenever $a \sqsubseteq b$, then a is considered to be more exact, i.e., a conveys more information about the system state.

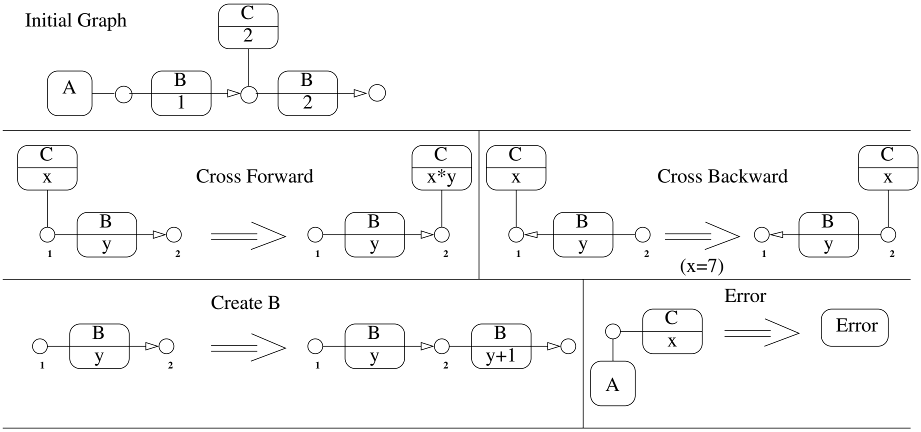


Fig. 1. Example of an attributed graph transformation system

A family of functions $(\alpha_s : \mathcal{A}_s \to \mathcal{B}_s,\ \gamma_s : \mathcal{B}_s \to \mathcal{A}_s)_{s\in\mathcal{S}}$ *is called* Galois connection on algebras *if they are monotone with respect to* $\sqsubseteq$ *and if for all* $s \in \mathcal{S}$*:* $\forall a \in \mathcal{A}_s : a \sqsubseteq \gamma_s(\alpha_s(a))$ *and* $\forall b \in \mathcal{B}_s : \alpha_s(\gamma_s(b)) \sqsubseteq b$*.*

Finally we require that for each function symbol f *with* $\sigma(f) = (s_1 \dots s_n, s)$ *the function* $f^{\mathcal{B}}$ *is a safe over-approximation of* $f^{\mathcal{A}}$*, i.e., for all* $a_1, \dots, a_n$ *with* $a_i \in \mathcal{A}_{s_i}$ *it holds that:* $\alpha_s(f^{\mathcal{A}}(a_1, \dots, a_n)) \sqsubseteq f^{\mathcal{B}}(\alpha_{s_1}(a_1), \dots, \alpha_{s_n}(a_n))$*. Note that this condition says that* α *is an algebra homomorphism "up to"* $\sqsubseteq$*. Such mappings will also be called* $\sqsubseteq$*-homomorphisms.*

Furthermore if $\mathcal{A}$*,* $\mathcal{B}$ *are Boolean algebras we require that both use the same carrier set for Bool, that* $\alpha_{Bool}, \gamma_{Bool}$ *are identities,* $T_{\mathcal{A}} = T_{\mathcal{B}}$*,* $F_{\mathcal{A}} = F_{\mathcal{B}}$ *and that furthermore truth values respect the information ordering* $\sqsubseteq$ *in the following sense:*

$$\forall v, v' \colon (v' \sqsubseteq v \wedge v' \in T_{\mathcal{A}} \Rightarrow v \in T_{\mathcal{A}}) \qquad \textit{(and similarly for } F_{\mathcal{A}}\textit{).}$$

Example 3. The algebra $\mathcal{C}$ that is used by our implementation allows several possible abstractions via algebras with finite carrier sets, some of which are already predefined. For instance, for the integers we use modulo abstraction modulo base b (each integer k is abstracted by $(k \bmod b)$) and interval abstraction with boundaries m, n (k is abstracted by one of "$< -m$",$-m, \dots, n-1, n$,"$> n$"). We also defined suitable operators on the abstract values which safely over-approximate the original functions in the sense of Definition 11.

However, we can not use directly the algebra $\mathcal{C}$ for a Galois abstraction since it is not lattice-ordered. Hence we work with $\mathcal{P}(\mathcal{C})$ where the lattice-order is set inclusion. Then every set of concrete values is mapped to set of abstract values via α_s, whereas γ_s is the corresponding concretization.

4 Analysis of Attributed Graph Transformation Systems

Since GTSs are in general Turing-powerful, over-approximation techniques are needed for their analysis. In our case we abstract AGTSs by coloured Petri nets, which are a conceptually simpler formalism which is easier to analyse. In [3] an approximated unfolding technique for GTSs was presented, in which—compared to standard unfolding techniques—additional folding steps are used, which over-approximate but guarantee a finite approximation. The resulting over-approximation is a so-called Petri graph which is a Petri net with an additional hypergraph structure, i.e., the hyperedges are at the same time the places of the net. Our idea here is to construct an *attributed* Petri graph which over-approximates an AGTS: an attributed Petri graph consists of an attributed (or coloured) Petri net and a hypergraph structure over it. Our notation is oriented on coloured Petri nets [12] and algebraic high-level nets [8].

4.1 Attributed Petri Graphs

We now formally define attributed Petri nets and attributed Petri graphs. We consider a fixed set of labels Λ and a function $ltype : \Lambda \to S$.

By $A^\oplus$ we denote the free commutative monoid over A with monoid operation $\oplus$, whose elements are also called *multisets*. A multiset $M \in A^\oplus$ can be written as a formal sum $M = \bigoplus_{a \in A} m_a \cdot a$ and given M we write $M(a)$ to denote the coefficient m_a. A function $f : A \to B$ can be extended to a function $f : A^\oplus \to B^\oplus$ on multisets as follows: For $M \in A^\oplus$ we define $M' = f(M)$ with $M'(b) = \sum_{a \in f^{-1}(b)} M(a)$ for every $b \in B$. Besides $\oplus$ we also use *difference* $M \ominus M'$, where $M, M' \in A^\oplus$ and *inclusion*, defined by $M \leq M'$, when there exists $M'' \in A^\oplus$ such that $M \oplus M'' = M'$.

We will now introduce attributed Petri nets which imitate coloured nets [12] in their graphical representation, and which are basically algebraic high-level nets [8] with small variations. For instance, compared to [8], we only allow variables, but not arbitrary terms in the preset of a transition.

Definition 12 (attributed Petri net). *Let $\mathcal{A}$ be a Σ-algebra. An $\mathcal{A}$-*attributed Petri net *is a tuple $\mathcal{N} = (S, T, l, {}^\bullet(), ()^\bullet, guard, m_0)$, where S is a set of places, T is a set of transitions, $l : S \to \Lambda$ is a labelling function, ${}^\bullet(), ()^\bullet : T \to (S \to (T(\Sigma, X)_\mathcal{S})^\oplus)$ are pre- and postset functions, $guard : T \to T(\Sigma, X)_{Bool}$ is a guard function, and m_0 is the initial marking of the net. A* marking *of an attributed Petri net is a function $m : S \to \mathcal{A}_\mathcal{S}^\oplus$. We also require that:*

(1) Each element of the multisets ${}^\bullet t(s)$, $t^\bullet(s)$ and $m(s)$ is of sort $ltype(l(s))$.
(2) The multiset $\bigoplus_{s \in S} {}^\bullet t(s) = X'$ contains only variables, each with multiplicity 1. Furthermore, the elements of $t^\bullet(s)$ are contained in $T(\Sigma, X')$ and $guard(t) \in T(\Sigma, X')_{Bool}$.

Elements of $m(s)$ (which are elements of the carrier sets) are also called *tokens*. For a marking m define $|m| : S \to \mathbb{N}$ as $|m|(s) = |m(s)|$, i.e., each place is associated with the number of tokens it contains.

A transition t is *enabled* for the marking m if there exists a binding $h : T(\Sigma, X) \to \mathcal{A}_\mathcal{S}$ such that $h(guard(t)) \in T_\mathcal{A}$ and for each place s it holds that $m(s) \geq h({}^\bullet t(s))$. An enabled transition with a given binding h can be fired and the marking m of the net will be transformed into m', denoted by $m\,[t, h\rangle\, m'$:

$$m'(s) = m(s) \ominus h({}^\bullet t(s)) \oplus h(t^\bullet(s)).$$

We consider a Petri graph as consisting of an attributed Petri net and a non-attributed hypergraph structure over it.

Definition 13 (attributed Petri graph). *Let $\mathcal{G} = (\mathcal{R}, G_0)$ be an AGTS. An* $\mathcal{A}$-attributed Petri graph *(over $\mathcal{R}$) is a tuple $P = (G, \mathcal{N}, p_N, \mu)$, where G is a (non-attributed) hypergraph, $\mathcal{N}$ is an $\mathcal{A}$-attributed Petri net where the places are the edges of G, p_N associates to each transition t a rule $p_N(t) = (L, R, \alpha, g) \in \mathcal{R}$ such that $guard_P(t) = g$ and μ associates to each transition t from $\mathcal{N}$ with $p_N(t)$ as above a (non-attributed) hypergraph morphism $\mu(t) : L \cup R \to G$ such that ${}^\bullet t(s) = \bigoplus_{\mu(t)(e)=s, e\in E_L} attr_L(e)$ and $t^\bullet(s) = \bigoplus_{\mu(t)(e)=s, e\in E_R} attr_R(e)$.*

An attributed Petri graph for $\mathcal{G}$ *is a pair (P, ι), where $P = (G, N, p_N, \mu)$ is an attributed Petri graph over $\mathcal{R}$ and $\iota : G_0 \to G$ is a (non-attributed) graph morphism such that $m_0(s) = \bigoplus_{\iota(e)=s} attr_{G_0}(e)$ for each edge $e \in E_{G_0}$.*

Note that the edges of the graph are at the same time the places of the net and that the transitions are labelled with rules of the AGTS.

For each marking m of an attributed Petri graph we define an attributed graph $graph(m)$ as follows: first we take the subgraph G' of G with edge set $E' = \{e \mid m(e) \neq \emptyset\}$ and with all nodes adjacent to some edge in E'. Assume that $m(e) = \bigoplus_{i=1}^{k} a_i$ is the marking of $e \in E'$. Now we replace in G' each e by k edges $e_1, \ldots, e_k$ with $l_G(e_i) = l_G(e)$, $c_G(e_i) = c_G(e)$ and $attr_G(e_i) = a_i$.

4.2 Approximated Unfolding

We now describe how to obtain an attributed Petri graph from a given AGTS. First, we unfold the underlying GTS in an approximative way as it is described in [3] without taking attributes into consideration. This is done by starting with the initial graph and applying unfolding steps that "simulate" rule applications by adding transitions, as well as folding steps that merge left-hand sides which are causally dependent. Since the approximated unfolding procedure supplies us with morphisms ι and $\mu(t)$ as described in Definition 13 there is a unique way of adding attributes to the Petri graph after the approximated unfolding. This means that attributes do not affect the unfolding procedure itself in any way.

Still, it is necessary to show that the resulting Petri graph is a valid over-approximation.

Proposition 1. *Let P be an attributed Petri graph for a GTS $\mathcal{G}$ obtained as described above. Then, there exists a simulation relation[4] $\mathcal{R}$ between the reachable graphs in $\mathcal{G}$ and the reachable markings in P such that: $(G_0, m_0) \in \mathcal{R}$ and for every pair (G, m) there exists an edge-bijective attributed hypergraph morphism (with the identity as algebra homomorphism) $G \to graph(m)$. Specifically this means that every graph reachable in $\mathcal{G}$ is over-approximated by a reachable marking of P.*

[4] In the simulation game every application of a rule r must be answered by a transition labelled r.

We extended Augur 2 [14] to construct and analyse over-approximations of AGTSs. Fig. 2 depicts the coarsest over-approximation for the AGTS in Fig. 1 computed by Augur 2. Places, which coincide with the edges, are depicted as boxes with rounded corners, with circle-shaped tokens inside. Transitions are represented by thin black rectangles with guard conditions and preset/postset annotations. For instance $1'x$ on an arc leaving a place means that one token is removed and its value bound to x. Note that the over-approximation below is too coarse since the error edge can be covered. Hence abstraction refinement is necessary.

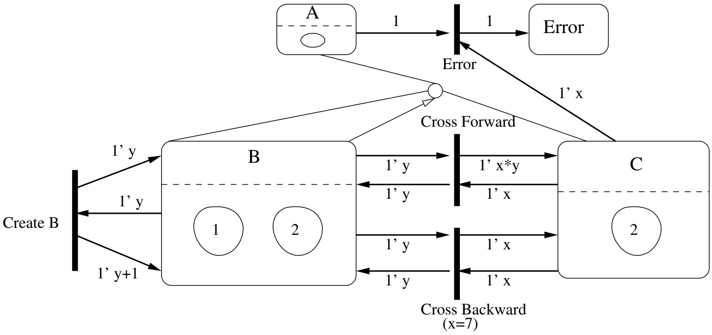


Fig. 2. Petri graph approximating the GTS (first approximation)

4.3 Analysing Petri Graphs

The obtained Petri graphs are basically coloured Petri nets [12] and can be analyzed with techniques developed for such nets. In particular we want to check that certain edges (or places), called error edges, can not be covered. Due to Proposition 1 we can infer that if this holds for the approximation, it is also true for the original system. However, we still have to handle infinite carrier sets, which is done by attribute abstraction. We show here that if the attributes are correctly abstracted, then the abstract version of a Petri graph correctly over-approximates the concrete version.

In the following we assume that AGTSs are attributed over an algebra $\mathcal{A}$, which will be abstracted by an algebra $\mathcal{B}$ via a Galois connection (α_s, γ_s) (see Definition 11). If we take a Petri graph $\mathcal{P}$ attributed over $\mathcal{A}$ this can be easily seen as a Petri graph attributed over $\mathcal{B}$ by applying α_s to all elements of the initial marking. The (abstract) Petri graph obtained in this way is denoted by P^a.

The following proposition shows how the abstract Petri graph P^a can be used in order to analyse P. But let us first fix some notation: For two multisets M_1, M_2 we write $M_1 \sqsubseteq M_2$ if there is a bijection from M_1 to M_2 such that each element of M_1 is smaller than or equal to its image in M_2 (with respect to the information ordering $\sqsubseteq$). For two markings we write $\hat{m}_1 \sqsubseteq \hat{m}_2$ whenever $\hat{m}_1(s) \sqsubseteq \hat{m}_2(s)$ for each place s.

Proposition 2. *For the attributed Petri graphs P and P^a it holds that there is a simulation relation $\mathcal{R}$ on the reachable markings such that $(m_0, m_0^a) \in \mathcal{R}$ and for each pair $(m, \hat{m}) \in \mathcal{R}$ we have $m^a \sqsubseteq \hat{m}$.*

To analyse attributed Petri graphs we need to check whether certain markings or places can be covered by a reachable marking. Hence we adapted two such techniques, coverability graphs [19] and backward reachability [1], to attributed Petri graphs with finite carrier sets and implemented them in Augur 2. We also extended both methods to provide us with a trace (= counterexample) to a given coverable marking.

4.4 Abstraction Refinement

This section generalizes the abstraction refinement technique from [15]. We adapt the technique of abstraction refinement for AGTS and attributed Petri graphs. If the analysis of the Petri net gives us a spurious counterexample for the property to verify then we can try to eliminate it using counterexample-guided abstraction refinement [15]. In our case there are two possible ways to refine the obtained over-approximation: either we can refine the graph structure of the obtained over-approximation or the attribute abstraction. One of the challenges is to determine which of the two cases applies.

First we define a notion of (abstract) runs and their correspondence.

Definition 14 (abstract run of an AGTS). *An* abstract run *of an AGTS* $(\mathcal{R}, G_0)$ *is a sequence of attributed hypergraphs* $\mathcal{J} = (J_0 \Rightarrow_{r_1} J_1 \Rightarrow_{r_2} \cdots \Rightarrow_{r_n} J_n)$*, where* r_i *is a rule name, together with (attributed) morphisms* $\varphi_i : L_{i+1} \to J_i$ *for each* $i = 1, \ldots, n{-}1$*, where* L_i *is the left-hand side of rule* $r_i \in \mathcal{R}$*.*

Note that we do not demand that J_i *can be derived from* J_{i-1} *by applying rule* r_i *at match* φ_i *(hence the name* abstract*). If an abstract run is derivable it will be called a* real run*. The* j*-th prefix of* $\mathcal{J}$ *is the run* $pr_j(\mathcal{J}) = (J_0 \Rightarrow_{r_1} J_1 \Rightarrow_{r_2} \cdots \Rightarrow_{r_j} J_j)$ *together with the morphisms* φ_i*.*

Let $\mathcal{J}' = (J_0' \Rightarrow_{r_1} J_1' \Rightarrow_{r_2} \ldots \Rightarrow_{r_n} J_n')$ *be another abstract run with morphisms* $\varphi_i' : L_{i+1} \to J_i'$ *for each* $i = 1, \ldots, n{-}1$*. We say that* $\mathcal{J}'$ weakly corresponds *to* $\mathcal{J}$ *(in symbols* $\mathcal{J}' \ll \mathcal{J}$*) if for each* $i = 1, \ldots, n{-}1$ *there exist edge-bijective (attributed) morphisms* $\xi_i : J_i' \to J_i$ *for* $i = 0, \ldots, n$*. If furthermore the following diagram commutes we say that* $\mathcal{J}'$ corresponds *to* $\mathcal{J}$ *and write* $\mathcal{J}' \lll \mathcal{J}$*.*

$$\begin{tikzcd}
L_{i+1} \arrow[r, "\varphi_i'"] \arrow[rr, bend right, "\varphi_i"'] & J_i' \arrow[r, "\xi_i"] & J_i
\end{tikzcd}$$

In both cases, we require that the attributed morphisms are equipped with identity homomorphisms. If they have only $\sqsubseteq$*-homomorphisms (as defined in Definition 11) we talk about* (weak) $\sqsubseteq$-correspondence *and write* $\ll_{\sqsubseteq}$ *and* $\lll_{\sqsubseteq}$*.*

For later use we need following construction (cf. [15]): Let G be a hypergraph and m a marking of the underlying Petri net, specifically $m \in E_G^{\oplus}$. That is, there exists a (non-attributed) morphism $\psi : \mathit{graph}(m) \to G$. Now let $\varphi : G' \to G$ be a morphism such that $\varphi^{\oplus}(E_{G'}) \leq |m|$. Then there exists an edge-injective morphism $e_{m,\varphi} : G' \to \mathit{graph}(m)$ such that $\psi \circ e_{m,\varphi} = \varphi$.

We will mainly use this construction for the special case where $\varphi = \mu(t)|_L : L \to G$, i.e., φ is a match of the left-hand side in the Petri graph (see Definition 13), and m is a marking that allows to fire transition t.

Petri graphs can, as mentioned above, be seen as symbolic representations of graph transition systems and also as representations of sets of abstract runs.

Definition 15 (abstract runs of an attributed Petri graph). *Let (P, ι) with $P = (G, N, p_N, \mu)$ be an attributed Petri graph for an AGTS $(\mathcal{R}, G_0)$. Furthermore let $m_0[t_1, h_1\rangle \dots [t_n, h_n\rangle m_n$ be a firing sequence of the net N and let $r_i = p_N(t_i)$ be the rules corresponding to the transitions. We consider (non-attributed) morphisms $\nu_{i+1} = e_{m_i,\mu(t_{i+1})|_{L_{i+1}}} : L_{i+1} \to \mathit{graph}(m_i)$, where L_{i+1} is the left-hand side of rule r_{i+1} and extend them in the canonical way to attributed morphisms by adding bindings. It is easy to see that the sequence $\mathit{graph}(m_0) \Rightarrow_{r_1} \mathit{graph}(m_1) \Rightarrow_{r_2} \dots \Rightarrow_{r_n} \mathit{graph}(m_n)$ together with the morphisms $\varphi_i = (\nu_i, h_i)$ is an* abstract run*.*

Each real run $\mathcal{J}_R = (G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} G_n)$ of the AGTS $(\mathcal{R}, G_0)$ can be considered as an abstract run where the $\varphi_i : L_{i+1} \to G_i$ represent the matches of the left-hand sides of the rules r_i.

Now let $\mathcal{G}$ be an AGTS, let P be an attributed Petri graph approximating $\mathcal{G}$ and let P^a be the abstract Petri graph derived from P. That is, P over-approximates the (graph) structure, whereas P^a additionally abstracts attributes.

Then, for every real run $\mathcal{J}_R$ of $\mathcal{G}$ there exists an abstract run $\mathcal{J}_A$ of P, such that $\mathcal{J}_R \lll \mathcal{J}_A$. And furthermore for every abstract run $\mathcal{J}_A$ of P there exists an abstract run $\hat{\mathcal{J}}_A$ of P^a such that $\mathcal{J}_A \lll_{\sqsubseteq} \hat{\mathcal{J}}_A$. This is a direct consequence of the simulation property (see Propositions 1 and 2). Since correspondence is transitive this means that every real run $\mathcal{J}_R$ of $\mathcal{G}$ can be associated with an abstract run $\hat{\mathcal{J}}_A$ of P^a such that $\mathcal{J}_R \lll_{\sqsubseteq} \hat{\mathcal{J}}_A$.

We start abstraction refinement with an attributed Petri graph $\mathcal{P}^a$ which is obtained by unfolding an AGTS $\mathcal{G}$ and interpreting the resulting Petri graph in $\mathcal{B}$ (as described in the previous section). If the property we want to verify is violated, we obtain a counterexample of the following form:

$$\hat{m}_0[t_1, \hat{h}_1\rangle \dots [t_n, \hat{h}_n\rangle \hat{m}_n,$$

where the t_i are transitions and the $\hat{h}_i$ are the corresponding bindings. Usually the AGTSs that we consider have an error rule and the property we want to verify is that this rule is not applicable. Hence an error trace includes a firing of the corresponding error transition as the last step. It can be seen as an abstract run (with abstracted attributes) of the following form:

$$\hat{\mathcal{J}}_A = (\mathit{graph}(\hat{m}_0) \Rightarrow_{r_1} \mathit{graph}(\hat{m}_1) \Rightarrow_{r_2} \dots \Rightarrow_{r_n} \mathit{graph}(\hat{m}_n)),$$

where $r_j = p_N(t_j)$ and $(\nu_j, \hat{h}_j) : L_j \to \mathit{graph}(\hat{m}_{j-1})$ are the corresponding morphisms from the left-hand side of r_j to $\mathit{graph}(\hat{m}_{j-1})$ for $j = 1, \dots, n$.

After analysing the Petri graph P^a and searching for counterexamples there could be the following four possibilities:

(1) The property is successfully verified, i.e., no counterexample was found in P^a.
(2) A real (non-spurious) counterexample $\hat{\mathcal{J}}_A$ is found. That is, we have $\mathcal{J}_R \lll_{\sqsubseteq} \hat{\mathcal{J}}_A$ for a real run $\mathcal{J}_R$ of $\mathcal{G}$. In this case we have found an error.

(3) The detected counterexample is *spurious*. This means that no real run $\mathcal{J}_R$ with $\mathcal{J}_R \lll_{\sqsubseteq} \hat{\mathcal{J}}_A$ exists. However, there could be real runs $\mathcal{J}'_R$ shorter than $\hat{\mathcal{J}}_A$ that correspond to a prefix $pr_i(\hat{\mathcal{J}}_A)$ of the counterexample, i.e., $\mathcal{J}'_R \lll_{\sqsubseteq} pr_i(\hat{\mathcal{J}}_A)$. Let k be the maximal length of such a run. The set of all such maximal real runs (there could be several of them) is denoted by $\mathcal{H}$.

For a given $\mathcal{J}'_R \in \mathcal{H}$ there always exists a (unique) run $\mathcal{J}'_A$ of the attributed Petri graph P (with concrete attributes) with morphisms $(\nu_j, h_j) : L_j \to \mathit{graph}$ (m_{j-1}) (morphisms ν_j as above) such that $\mathcal{J}'_R \lll \mathcal{J}'_A$ (see Fig. 3). It is easy to see that also $\mathcal{J}'_A \lll_{\sqsubseteq} pr_i(\hat{\mathcal{J}}_A)$.

$$\begin{tikzcd}
\mathcal{J}'_A\colon & \mathit{graph}(m_0) & \Longrightarrow_{r_1} & \mathit{graph}(m_1) & \Longrightarrow_{r_2} \cdots & \mathit{graph}(m_i) \\
\mathcal{J}'_R\colon & G_0 \arrow[u, "\xi_0"] \arrow[d, "\hat{\xi}_0"'] & \Longrightarrow_{r_1} & G_1 \arrow[u, "\xi_1"] \arrow[d, "\hat{\xi}_1"'] & \Longrightarrow_{r_2} \cdots & G_i \arrow[u, "\xi_i"] \arrow[d, "\hat{\xi}_i"'] \\
pr_i(\hat{\mathcal{J}}_A)\colon & \mathit{graph}(\hat{m}_0) & \Longrightarrow_{r_1} & \mathit{graph}(\hat{m}_1) & \Longrightarrow_{r_2} \cdots & \mathit{graph}(\hat{m}_i) \\
 & L_1 \arrow[u, "(\nu_1, \hat{h}_1)"'] \arrow[uu, bend right, "\varphi_0"'] \arrow[uuu, bend left, "(\nu_1, h_1)"] & & L_2 \arrow[u, "(\nu_2, \hat{h}_2)"'] \arrow[uu, bend right, "\varphi_1"'] \arrow[uuu, bend left, "(\nu_2, h_2)"] & &
\end{tikzcd}$$

Fig. 3. Counterexample (abstract and real runs with corresponding left-hand sides)

We now distinguish the following two cases:

(3a) We say that the over-approximation is *structurally too coarse* if for some $\mathcal{J}'_R \in \mathcal{H}$ the corresponding run $\mathcal{J}'_A$ can be extended to a run $\mathcal{J}''_A$ of length $k+1$ with a morphism $(\nu_{k+1}, h_{k+1}) : L_{k+1} \to \mathit{graph}(m_k)$ in such a way that $\mathcal{J}''_A \lll_{\sqsubseteq} pr_{k+1}(\hat{\mathcal{J}}_A)$. The set of such prefixes of $\mathcal{H}$ is denoted by $\mathcal{H}_S$. Below we describe a technique based on the one proposed in [15] which allows us to eliminate the obtained counterexample in this case.

(3b) In the last case for each run $\mathcal{J}'_R \in \mathcal{H}$ such that $\mathcal{J}'_R \lll_{\sqsubseteq} pr_k(\hat{\mathcal{J}}_A)$, the corresponding run $\mathcal{J}'_A$ can not be extended as in the previous case, i.e., $\mathcal{H}_S = \emptyset$. If this holds then we say that in the over-approximation P^a *the attribute abstraction is too coarse*.

In the first two cases we have solved the problem either with a positive or a negative outcome. If the obtained over-approximation is structurally too coarse (case (3a)) and does not allow us to verify the property, a counterexample-guided abstraction refinement technique [15] for refining the approximation is available. It uses the set $\mathcal{H}_S$ of prefixes of the counterexample and refines the structure of the Petri graph. This is done by identifying which nodes have previously been merged or folded erroneously and by restarting the approximated unfolding from scratch, but making sure that those node are now kept separate. The technique described in [15] for non-attributed GTSs can be applied here without modification. As in [15], we will eliminate not only the spurious run $\hat{\mathcal{J}}_A$ but all other abstract runs corresponding to it and at the same time having a weak correspondence to some run in $\mathcal{H}_{\mathcal{S}}$.

Proposition 3. *The structurally refined Petri graph P'^a constructed above does not contain any run $\hat{\mathcal{J}}'_A$ which $\sqsubseteq$-corresponds to the spurious run $\hat{\mathcal{J}}_A$ of P^a and has a weak $\sqsubseteq$-correspondence to some run in $\mathcal{H}_{\mathcal{S}}$. Furthermore if P'^a contains a spurious run $\hat{\mathcal{J}}'_A$, then it $\sqsubseteq$-corresponds to some run $\hat{\mathcal{J}}_A$ in P^a.*

To refine the approximation in the last case we make our abstraction of attributes more exact in a predefined way. For example for the modulo abstraction we can increase the modulo base b (we usually multiply it by two), and for the interval abstraction we can increase the interval bounds m and/or n. However in this case we have no guarantee that the spurious counterexample will be eliminated. In the implementation the attribute abstraction is refined a predefined number of times and if spurious counterexample are then still reproducible, we terminate with the answer “don’t know”. Future work is the integration of the predicate abstraction which will be discussed in the conclusion.

So our results for the refinement of attribute abstraction are weaker than in the case of structure refinement. But we can still show that whenever we refine the attribute abstraction in a certain way, no new spurious runs will appear.

Proposition 4. *Let $(\alpha_s : \mathcal{A}_s \to \mathcal{B}_s,\ \gamma_s : \mathcal{B}_s \to \mathcal{A}_s)_{s\in\mathcal{S}}$ be the Galois connection between algebras $\mathcal{A}$, $\mathcal{B}$ which was originally used for attribute abstraction. Now let $(\alpha'_s : \mathcal{A}_s \to \mathcal{D}_s,\ \gamma'_s : \mathcal{D}_s \to \mathcal{A}_s)_{s\in\mathcal{S}}$ be a new connection from $\mathcal{A}$ to $\mathcal{D}$.*

We furthermore assume that there exists a Galois connection from $\mathcal{D}$ to $\mathcal{B}$ with mappings α''_s, γ''_s such that $\alpha_s \sqsupseteq \alpha''_s \circ \alpha'_s$. Then if the refined Petri graph P'^a contains a run $\hat{\mathcal{J}}'_A$, it $\sqsubseteq$-corresponds (with α''_s as $\sqsubseteq$-homomorphisms) to some run $\hat{\mathcal{J}}_A$ in P^a. In particular, if $\hat{\mathcal{J}}'_A$ leads to a marking covering an error edge, then the same is true for $\hat{\mathcal{J}}_A$.

We can iterate abstraction refinement by storing an arbitrary number of spurious counterexamples. Naturally, due to undecidability and the fact that AGTSs are in general Turing-complete, there is no guarantee that this loop will ever terminate.

Example 4. Let us now consider the Petri graph in Fig. 2 using a modulo abstraction with base one (unit abstraction). The edge labelled Error of the Petri graph can be covered by firing transition “Error”. This means that either the property does not hold or the over-approximation is too coarse. In this case one can show that the run is spurious, i.e., it has no counterpart in the original AGTS and the over-approximation is structurally too coarse (case (3a)). Applying abstraction refinement gives us a refined Petri graph (which is not depicted here due to space constraints).

Now an error edge is still in the approximation and a counterexample can be constructed (via rules “Cross Backward”, “Error”). However, this counterexample can not be reproduced without attribute approximation, which means that the abstraction is too coarse and should be refined (case (3b)). By using base two in the modulo abstraction we obtain a Petri graph in which the Error-edge is no longer coverable, which means successful verification.

5 Example: Leader Election

In this section we sketch the modelling and verification of a leader election protocol in a ring architecture with AGTSs. The purpose of the protocol is to elect a unique leader among the stations in a ring-shaped network.

The algorithm uses only local communication and does not depend on the size of the ring. The leader is chosen on the basis of the unique ids of the stations and will eventually be the station with the smallest id (in our case: id 1). Each station sends a message with its id around the ring. Upon reception of the message from its predecessor a station compares the received id with its own id and, if the incoming id is smaller than its own, forwards the message. If the id is larger, then the message is discarded. If a station receives its own id, then it declares itself the leader. The protocol is parametrized in the sense that we can create rings of arbitrary size with a potentially infinite number of stations. We want to show that we never choose the wrong leader, i.e., there is never a situation where we have a station that has a smaller id than the current leader.

Concerning the ids we use interval abstraction with start interval $[0, 1]$. After unfolding the AGTS and analysing it via the coverability technique we obtain a spurious counterexample. Afterwards three iterations of abstraction refinement can be applied: two with structural refinement and one with attribute abstraction in the interval $[0, 2]$. The coverability check then shows that we have successfully verified the protocol. The whole verification procedure for the leader election protocol took 48.15 seconds.[5] More details on this case study will be reported in [17].

6 Conclusion

We have presented a framework for the verification of attributed graph transformation systems, using approximated unfoldings, attribute abstraction and a counterexample-guided abstraction refinement technique.

There are some related approaches to the verification of graph transformation systems in the literature, see for instance [20,21,6,4]. However, there seems to be only a small amount of work on the verification and over-approximation of attributed graph transformation systems. We are currently aware of attributes in the tool Groove [13] for the verification of finite-state graph transformation systems. Furthermore AGTSs could be transformed into the input language of more conventional model checkers that do support attributes.

This combination is clearly of practical interest and also raises interesting methodological questions. As we have shown the combination of structural refinements and refinements of attribute abstractions is non-trivial.

Currently we handle the refinement of attribute abstraction semi-automatically, by leaving the choice mainly to the user. Clearly this is not completely satisfactory. A natural question to ask is whether the counterexample-guided

[5] All experiments were made using Augur 2 [14] written in C++ under Linux and the computer parameters are 2*Genuine Intel(R) 1.66 GHz with 2.0 GB RAM.

abstraction refinement approach based on predicate abstraction and Craig interpolation [10,11] can be adapted to our setting. In this approach the abstraction is refined by generating new predicates over the program variables, based on the counterexample. In our setting the difficulty is not so much how to generate these predicates (after all, we have a specific counterexample) but how to interpret them over the markings of the Petri net. The situation would be easy if all predicates were unary, since in this case we would employ the concept of Galois connections introduced in this article. However generated predicates typically have a higher arity, often predicates are binary predicates of the form $\mathtt{x} < \mathtt{y}$. For the original predicate abstraction approach this is not a problem since there are only finitely many variables and the value of predicates for an abstract state can be described in a finite way. However in our case there can be arbitrarily many tokens and it is not clear to us how to solve the coverability problem for Petri nets with such an abstraction mechanism.

In addition we need more (and larger) case studies in order to test our techniques. Currently we are working on the verification of variants of the Needham-Schroeder protocol a cryptographic protocol used for authentication (see [17]).

References

1. Aziz Abdulla, P., Jonsson, B., Kindahl, M., Peled, D.: A general approach to partial order reductions in symbolic verification. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 379–390. Springer, Heidelberg (1998)
2. Baldan, P., Corradini, A., Esparza, J., Heindel, T., König, B., Kozioura, V.: Verifying red-black trees. In: Proc. of COSMICAH 2005, Proceedings available as report RR-05-04 (Queen Mary, University of London) (2005)
3. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)
4. Bauer, J., Wilhelm, R.: Static analysis of dynamic communication systems by partner abstraction. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007)
5. Cousot, P.: Abstract Interpretation. ACM Computing Surveys (1996)
6. Dotti, F.L., Foss, L., Ribeiro, L., Marchi Santos, O.: Verification of distributed object-based systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 261–275. Springer, Heidelberg (2003)
7. Dotti, F.L., König, B., Marchi Santos, O., Ribeiro, L.: A case study: Verifying a mutual exclusion protocol with process creation using graph transformation systems. Technical Report 08/2004, Universität Stuttgart (2004)
8. Ehrig, H., Padberg, J., Ribeiro, L.: Algebraic high-level nets: Petri nets revisited. In: Ehrig, H., Orejas, F. (eds.) Abstract Data Types 1992 and COMPASS 1992. LNCS, vol. 785, pp. 188–206. Springer, Heidelberg (1994)
9. 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)
10. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. of POPL 2004, pp. 232–244. ACM Press, New York (2004)
12. Jensen, K.: Coloured Petri nets: Status and outlook. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 1–2. Springer, Heidelberg (2003)
13. Kastenberg, H.: Towards attributed graphs in GROOVE. In: Proceedings of Workshop on Graph Transformation for Verification and Concurrency, volume 05-34 of CTIT Technical Report, pp. 91–98 (2005)
14. König, B., Kozioura, V.: Augur 2—a new version of a tool for the analysis of graph transformation systems. In: Proc. of GT-VMT 2006 (Workshop on Graph Transformation and Visual Modeling Techniques). ENTCS, vol. 211, pp. 201–210. Elsevier, Amsterdam (2008)
15. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 197–211. Springer, Heidelberg (2006)
16. Kozioura, V.: Verification of random graph transformation systems. In: Proc. of GT-VC 2006 (Graph Transformation for Verification and Concurrency. ENTCS, vol. 175.4 (2006)
17. Kozyura, V.: Abstraction and Abstraction Refinement in the Verification of Graph Transformation Systems. PhD thesis, Universität Duisburg-Essen, forthcoming
18. Löwe, M., Korff, M., Wagner, A.: An algebraic framework for the transformation of attributed graphs. In: Term graph rewriting: theory and practice, pp. 185–199. John Wiley and Sons Ltd, Chichester (1993)
19. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer ScienceGermany. Springer, Berlin (1985)
20. Rensink, A., Distefano, D.: Abstract graph transformation. In: Proc. of SVV 2005 (3rd International Workshop on Software Verification and Validation). ENTCS, vol. 157.1, pp. 39–59 (2005)
21. Varró, D.: Towards symbolic analysis of visual modeling languages. In: Workshop on Graph Transformation and Visual Modeling Techniques 2002. ENTCS, vol. 72, Elsevier, Amsterdam (2002)

A Modal-Logic Based Graph Abstraction

Jörg Bauer[2], Iovka Boneva[1], Marcos E. Kurbán[3], and Arend Rensink[1]

[1] Institut für Informatik, Technical University of Munich,
85748 Garching bei München, Germany
joba@model.in.tum.de

[2] Formal Methods and Tools Group, EWI-INF, University of Twente
PO Box 217, 7500 AE, Enschede, The Netherlands
{bonevai,rensink}@cs.utwente.nl

[3] Former member of Formal Methods and Tools Group,
EWI-INF, University of Twente

Abstract. Infinite or very large state spaces often prohibit the successful verification of graph transformation systems. Abstract graph transformation is an approach that tackles this problem by abstracting graphs to abstract graphs of bounded size and by lifting application of productions to abstract graphs. In this work, we present a new framework of abstractions unifying and generalising existing takes on abstract graph transformation. The precision of the abstraction can be adjusted according to the properties to be verified facilitating abstraction refinement. We present a modal logic defined on graphs, which is preserved *and* reflected by our abstractions. Finally, we demonstrate the usability of the framework by verifying a graph transformation model of a firewall.

1 Introduction

Formal verification of graph transformation systems aims at statically proving or inferring properties of a graph transformation system, where such properties are typically given in some form of temporal logic. It is crucial to distinguish verification and simulation, the latter being very useful only for debugging, whereas verification establishes a property for *all* computations of a graph transformation system. Problems do arise when approaching this task. One such problem is the possibly infinite behaviour of a system which in most cases makes it impossible to study the whole behaviour of the system. Another problem is space: even for a finite state space, each state can be quite big to represent.

Some approaches to formal verification of graph transformation systems include [1,2,3,4,5,6,7,8].They can be characterised as to which approach to graph transformation is used for modelling, which verification technique is applied, and which applications are tackled. The technique presented in [1] feeds finite-state graph transformation systems, given as a double pushout system, to an off-the-shelf model checker to verify reactive systems. However, we face the more general problem of unbounded systems. The approaches presented in [2,3] both use backwards reachability analysis for hyperedge replacement grammars trying to reach an initial graph by backwards search from a forbidden configuration.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 321–335, 2008.

The technique is applied to mechatronic systems and ad-hoc network routing, respectively, but, unfortunately, is not guaranteed to terminate. An approximation of the behaviour of a graph transformation system in terms of Petri net unfoldings was used in [4] to verify properties of data structures residing in the run-time heap of programs with dynamically allocated heap memory.

In this work, we present a new take on *abstract graph transformation* as introduced independently by [6] and [7]. Abstract graph transformation relies on *abstract interpretation* [9] of graph transformation systems, that is, given some equivalence relation, graphs are quotiented into abstract graphs of bounded, finite size. Application of productions is then lifted to work on abstract graphs. The abstraction first introduced in [10] summarises nodes with similar kind and number of incident edges, while the abstraction of [7] considers similar adjacent nodes. These two abstractions are generalised in this work and put into a unifying framework. To this end, we introduce the notion of *neighbourhood abstraction* as a part of a general abstraction framework. For this abstraction, nodes are summarised if they have similar neighbourhood up to some *radius* i, parameter of the abstraction. This enables abstractions with different precisions. Additionally, the number of possible abstract graphs obtained by neighbourhood abstraction is bounded. We introduce a logic accompanying our abstractions: given a formula our abstraction guarantees that *a*) if the formula holds for the original graph, then it holds for the abstracted graph (preservation); and *b*) if the formula holds for the abstracted graph, then it holds for the original one too (reflection).

Contributions

- Our abstraction framework unifies and generalises previous approaches on abstract graph transformation. For this particular technique, it supposedly establishes the most general treatment of *local* abstractions, that is, abstractions based on equivalence relations, where equivalence is determined by local information on nodes. In contrast, equivalences used in shape analysis [11] of heap programs often consider global properties like reachability.
- Technically, the most surprising result comprises the definition of a modal logic, properties of which are preserved *and* reflected by our abstraction. While preservation is necessary for the soundness of analyses based on our abstraction, reflection is a rather unusual and strong result.
- Our framework allows for automated *abstraction refinement*. If a property cannot be established given a certain neighbourhood size, one may automatically increase this size to obtain more precise results. The only other approach allowing for automated refinement is [5].
- A *canonical representation* of abstract graphs reduces otherwise costly isomorphism checks to simple equality tests.
- While, certainly, our method has its limitations, it works well for an important class of systems, dynamic communication systems. They are characterised by a dynamically changing number of communicating objects and a dynamically changing communication topology. Important examples include ad-hoc network protocols, traffic control- or mechatronic systems. Our

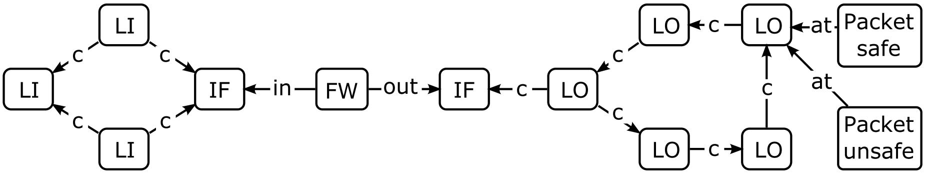


Fig. 1. Two networks delimited by a firewall

method is not suited for the analysis of graphs occurring in runtime heaps. The latter almost always require reachability information to be taken into account, something our approach fails to handle satisfactorily.

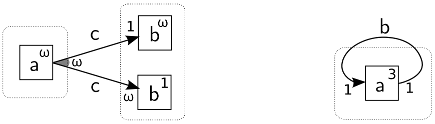
Outline. To start with, we shall present our case study, a firewall system, in Sect. 1.1. Section 2 introduces graphs and the general abstraction mechanism, as well as so called neighbourhood abstraction. In Sect. 3, we present a modal logic that is preserved *and* reflected by neighbourhood abstraction. In Sect. 4, a canonical representation of abstract graphs obtained by neighbourhood abstraction is defined, which is crucial for the representation of graphs in the actual implementation of the transformation. Before we conclude in Sect. 6, we briefly describe, in Sect. 5, how all ingredients can be combined for defining a fully automatic method for system verification.

1.1 Case Study: Firewalls

Figure 1 shows a graph model of two networks delimited by a firewall (FW). It has an internal and an external interface (IF), connected respectively to a network of *in*-locations (LI) to be protected by the firewall, and a network of *out*-locations (LO). Arbitrarily many packets flowing through the network can be created at locations – safe ones at any location and unsafe ones only at out-locations. The flow is bi-directional despite the drawing of directed c-edges. The full set of rules implementing such a firewall are given in [12]. A property we want to verify is that unsafe packets never reach in-locations.

2 Graphs and Graph Abstraction

We consider finite graphs whose edges are labelled from a finite set of labels, Lab. We mimic node labels by labelling special edges whose target is a special node $\perp$. Formally, a *graph* G is a tuple $(N_G, E_G, \mathsf{src}_G, \mathsf{tgt}_G, \mathsf{lab}_G)$ where N_G is a finite set of nodes, E_G is a finite set of edges disjoint from N_G, $\mathsf{src}_G : E_G \to N_G$ and $\mathsf{tgt}_G : E_G \to N_G \cup \{\perp\}$ with $\perp \notin (N_G \cup E_G)$ associate with each edge its source and target nodes, and $\mathsf{lab}_G : E_G \to \mathsf{Lab}$ labels edges. Let G and H be graphs. A *graph morphism* $f : G \to H$ is a function from $N_G \cup E_G \cup \{\perp\}$ to $N_H \cup E_H \cup \{\perp\}$ such that $f(\perp) = \perp$ and $f^{-1}(\perp) = \{\perp\}$; f maps nodes to nodes and edges to edges, *i.e.* $f(N_G) \subseteq N_H$, $f(E_G) \subseteq E_H$; f is compatible with source and target mappings, *i.e.* $\mathsf{src}_H \circ f = f \circ \mathsf{src}_G$, and $\mathsf{tgt}_H \circ f = f \circ \mathsf{tgt}_G$,



(a) $\mu = 1$, $\nu = 1$ (b) $\mu = 1$, $\nu = 3$

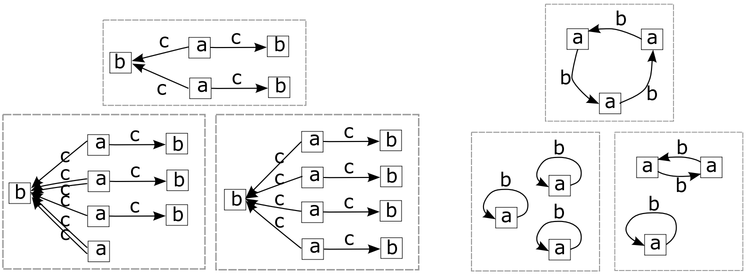
Fig. 2. Examples of abstract graphs

and f preserves labels, $f \circ \mathsf{lab}_G = \mathsf{lab}_H$. A morphism f is called *injective* (resp. *surjective*, resp. *bijective*) if it defines an injective (resp. surjective, resp. bijective) map. A bijective morphism is also called an *isomorphism*. We extend lab_G to a node to determine its set of labels, *i.e.* $\mathsf{lab}_G(v) = \{\mathsf{a} \in \mathsf{Lab} \mid \exists e \in E_G : \mathsf{src}_G(e) = v, \mathsf{tgt}_G(e) = \bot, \mathsf{lab}_G(e) = \mathsf{a}\}$. We write $v \rhd^{\mathsf{a}}_G$ and $v \lhd^{\mathsf{a}}_G$ for the set of a-outgoing edges and a-incoming edges of node v, respectively, *i.e.* $v \rhd^{\mathsf{a}}_G = \{e \in E_G \mid \mathsf{src}_G(e) = v, \mathsf{lab}_G(e) = \mathsf{a}\}$ and symmetrically for $v \lhd^{\mathsf{a}}_G$. For a set of nodes V, $V \rhd^{\mathsf{a}}_G$ (resp. $V \lhd^{\mathsf{a}}_G$) is the extension of $\rhd^{\mathsf{a}}_G$ (resp. $\lhd^{\mathsf{a}}_G$) on sets. Finally, for X, Y nodes or sets of nodes, we denote $X \rhd\!\!\rhd^{\mathsf{a}}_G Y$ the set of a-labelled edges between X and Y, *i.e.* $X \rhd\!\!\rhd^{\mathsf{a}}_G Y = X \rhd^{\mathsf{a}}_G \cap Y \lhd^{\mathsf{a}}_G$. When the graph G is clear from the context, we may omit the subscript G. For brevity, in the sequel of the paper we ignore the node $\bot$ and simply talk about node labels.

A *multiplicity* approximates the cardinality of a finite set. For any natural $\mu > 0$, let $\mathbf{M}_\mu$ be the set $\{0, 1, 2, \ldots, \mu, \omega\}$ where $\omega \notin \mathbb{N}$. The *μ-multiplicity* of a set U is denoted $|U|_\mu$ and defined by: $|U|_\mu = \mathit{Card}(U)$ if $\mathit{Card}(U) \leq \mu$, and $|U|_\mu = \omega$ otherwise. We write $\mathbf{M}^+_\mu$ for the set $\mathbf{M}_\mu \setminus \{0\}$. The usual ordering $\geq$ is extended to elements of $\mathbf{M}_\mu$ by $\omega \geq \lambda$ for all λ in $\mathbf{M}_\mu$. Sums over multiplicities are defined as expected writing $\sum^\mu$ for μ-bounded sums. For the sequel, we fix two naturals, $\nu, \mu > 0$, to denote multiplicities of sets of nodes (ν) and sets of edges (μ), *i.e.* ν and μ are parameters of graph abstractions.

2.1 Abstract Graphs and Abstraction

In this section, we discuss the notion of *abstract graphs*. Abstract graphs, such as the ones of Fig. 2, represent sets of (concrete) graphs called their *concretisations*. Every node of an abstract graph is associated with a *node multiplicity* indicating the number of concrete graph nodes it represents. The dotted rectangles are delimiting groups of nodes induced by an equivalence relation, the *grouping relation*, on them. All edges have associated multiplicity information (from $\mathbf{M}_\mu$) in their end points: *outgoing edges multiplicity*, when associated to the source of the edge, and *incoming edges multiplicity* when associated to the target. Sometimes, this multiplicity is shared by several edges, indicated by the grey arc relating them. Edge multiplicities indicate how many of the depicted edges should be there in a concretisation. Note that edges related in one of their end points all have their other end point in the same group of nodes, and all have the same label. Actually, this is the condition for relating edges. More precisely, edge multiplicities are associated with a triple composed of a node, a label and a group of nodes.



(a)

(b)

Fig. 3. Example concretisations of the abstract graphs on Fig. 2

Consider the abstract graph of Fig. 2(a). It represents a set of bipartite concrete graphs, such as the ones of Fig. 3(a), where a-nodes are connected to b-nodes by c-edges. Each of these graphs has at least two (as $\nu = 1$, ω stands for "two or more") a-nodes and at least three (ω plus one) b-nodes. Moreover, every a-node has at least two (*i.e.* ω) outgoing c-edges going to b-nodes. All b-nodes except one have only one incoming edge; the remaining b-node has at least two incoming edges. The abstract graph of Fig. 2(b) represents a set of concrete graphs having three a-nodes connected to each other forming b cycles, such as in Fig. 3(b).

Let us fix some notations. Let A be a set and $\sim \subseteq A \times A$ be an equivalence relation over A. For $x \in A$, $[x]_\sim$ denotes the equivalence class of x induced by $\sim$, and $A/\!\sim$ is the set of $\sim$-equivalence classes in A. If $\sim$ and $\sim'$ are two equivalence relations over A such that $\sim \subseteq \sim'$, then $\sim$ is called a *refinement* of $\sim'$.

Definition 1 (abstract graph). *An* abstract graph *S is a structure* $(G_S, \sim_S$ $, \mathsf{mult}_{\mathrm{n},S}, \mathsf{mult}_{\mathrm{out},S}, \mathsf{mult}_{\mathrm{in},S})$ *where*

- $G_S = (N_S, E_S, \mathsf{src}_S, \mathsf{tgt}_S, \mathsf{lab}_S)$ *is a graph;*
- $\sim_S \subseteq N_S \times N_S$ *is an equivalence relation on N_S called the grouping relation;*
- $\mathsf{mult}_{\mathrm{n},S} : N_S \rightarrow \mathbf{M}^+_\nu$ *is a node multiplicity function;*
- $\mathsf{mult}_{\mathrm{out},S} : N_S \times \mathsf{Lab} \times N_S/\!\sim_S \rightarrow \mathbf{M}_\mu$ *is an outgoing edges multiplicity function;*
- $\mathsf{mult}_{\mathrm{in},S} : N_S \times \mathsf{Lab} \times N_S/\!\sim_S \rightarrow \mathbf{M}_\mu$ *is an incoming edges multiplicity function.*

Moreover, for any $v \in N_S$, $\mathsf{a} \in \mathsf{Lab}$ *and $C \in N_S/\!\sim_S$, we require* $\mathsf{mult}_{\mathrm{out},S}(v, \mathsf{a}, C)$ $=$ *0 iff* $v \rhd\!\!\rhd^{\mathsf{a}}_{G_S} C = \varnothing$, *and* $\mathsf{mult}_{\mathrm{in},S}(v, \mathsf{a}, C) = 0$ *iff* $C \rhd\!\!\rhd^{\mathsf{a}}_{G_S} v = \varnothing$.

Formally, the relation between concretisations of an abstract graph S and S is captured by *abstraction morphisms* respecting multiplicity.

Definition 2 (abstraction morphism, concretisation). *Let G be a graph and S be an abstract graph. An* abstraction morphism of G into S *is a surjective graph morphism $s : G \rightarrow G_S$ such that the following conditions are met:*

1. *for all* $w \in N_S$*:* $\mathsf{mult}_{\mathrm{n},S}(w) = \left|s^{-1}(w)\right|_{\nu}$*;*
2. *for all* $w \in N_S$*, for all* $\mathsf{a} \in \mathsf{Lab}$*, for all* $C \in N_S/{\sim_S}$*, and for all* $v \in s^{-1}(w)$*:*

$$\mathsf{mult}_{\mathrm{out},S}(w,\mathsf{a},C)=\left|v \rhd\!\!\rhd^{\mathsf{a}}_{G}(s^{-1}(C))\right|_{\mu} \mathit{and}\, \mathsf{mult}_{\mathrm{in},S}(w,\mathsf{a},C)=\left|(s^{-1}(C)) \rhd\!\!\rhd^{\mathsf{a}}_{G} v\right|_{\mu}.$$

If $s : G \to S$ *is an abstraction morphism, then* G *is a* concretisation *of* S*. The set of all concretisations of* S *is written* $Concr(S)$*.*

As another example, Fig. 4, page 329 shows an abstract graph for the firewall example from Fig. 1, with $\nu = \mu = 1$. The corresponding abstraction morphism summarises the two LI-neighbours of the internal interface to the unique LI-neighbour of the internal interface in the abstract graph. The three LO nodes that have only LO neighbours are summarised to a unique node. All other nodes in the abstract graph have multiplicity one and correspond to a unique node in the concrete graph.

Note that the requirements on outgoing (resp. incoming) edge multiplicities guarantee in particular that two different nodes v, v' of graph G can only be summarised by an abstraction morphism, if they have the same outgoing and incoming edges multiplicities with respect to a label and a group of nodes.

Construction of an Abstract Graph. Definitions 1 and 2 are declarative and do not give a hint on the effective construction of an abstract graph. This can be done as follows: Let G be a graph and $\sim, \equiv \subseteq N_G \times N_G$ be two equivalence relations such that $\equiv$ refines $\sim$. Furthermore, assume that for any $v, v' \in N_G$, for any $C \in N_G/{\sim}$, and for any label a: $v \equiv v'$ implies

$$|v \rhd\!\!\rhd^{\mathsf{a}}_{G} C|_{\mu} = |v' \rhd\!\!\rhd^{\mathsf{a}}_{G} C|_{\mu} \quad \text{and} \quad |C \rhd\!\!\rhd^{\mathsf{a}}_{G} v|_{\mu} = |C \rhd\!\!\rhd^{\mathsf{a}}_{G} v'|_{\mu}$$

Then $\equiv$ and $\sim$ induce an *abstract graph*, $S = (G_S, \sim_S, \mathsf{mult}_{\mathrm{n}}, \mathsf{mult}_{\mathrm{out}}, \mathsf{mult}_{\mathrm{in}})$, and an *abstraction morphism*, $s : G \to S$ as follows:

- Extend the equivalence relation $\equiv$ to edges as follows: $e \equiv e'$ if $\mathsf{src}_G(e) \equiv \mathsf{src}_G(e')$, $\mathsf{tgt}_G(e) \equiv \mathsf{tgt}_G(e')$ and $\mathsf{lab}_G(e) = \mathsf{lab}_G(e')$. Then $S_G = (N_S, E_S, \mathsf{src}_S, \mathsf{tgt}_S, \mathsf{lab}_S)$ is the graph quotient of G w.r.t. $\equiv$, *i.e.* $N_S = N_G/{\equiv}$; $E_S = E_G/{\equiv}$; and for any edge $[e]_{\equiv}$ in E_S, $\mathsf{src}_S([e]_{\equiv}) = [\mathsf{src}_G(e)]_{\equiv}$, $\mathsf{tgt}_S([e]_{\equiv}) = [\mathsf{tgt}_G(e)]_{\equiv}$ and $\mathsf{lab}_S([e]_{\equiv}) = \mathsf{lab}_G(e)$. Note that, because of the definition of $\equiv$ on edges $[e]_{\equiv}$ is well-defined.
- Mapping $s : N_G \cup E_G \to N_S \cup E_S$ is defined by $s(v) = [v]_{\equiv}$ and $s(e) = [e]_{\equiv}$ for any $v \in N_G$ and any $e \in E_G$ and extended to preserve $\perp$.
- $\sim_S \subseteq N_S \times N_S$ is the equivalence relation given by $[v]_{\equiv} \sim_S [v']_{\equiv}$ if $v \sim v'$ for all $v, v' \in N_G$.
- $\mathsf{mult}_{\mathrm{n}} : N_S \to \mathbf{M}^{+}_{\nu}$ is defined by $\mathsf{mult}_{\mathrm{n}}(w) = \left|s^{-1}(w)\right|_{\nu}$ for all w in N_S.
- $\mathsf{mult}_{\mathrm{out}}, \mathsf{mult}_{\mathrm{in}} : N_S \times \mathsf{Lab} \times N_S/{\sim_S} \to \mathbf{M}_{\mu}$ are mappings defined by $\mathsf{mult}_{\mathrm{out}}([v]_{\equiv}, \mathsf{a}, C) = |v \rhd\!\!\rhd^{\mathsf{a}}_{G} C|_{\mu}$ $\mathsf{mult}_{\mathrm{in}}([v]_{\equiv}, \mathsf{a}, C) = |C \rhd\!\!\rhd^{\mathsf{a}}_{G} v|_{\mu}$ for all $v \in N_G$, $\mathsf{a} \in \mathsf{Lab}$, and $C \in N_S/{\sim_S}$.

It is obvious that S and s are indeed a well-defined abstract graph and abstraction morphism for two such equivalence relations. The complete formalisation and proof of this construction is in [13]. Note that not all abstract graphs can be thus defined. Such abstract graphs necessarily have concretisations and cannot have parallel edges, which is not the case for general abstract graphs. Still, any abstract graph admitting concretisations and without parallel edges can be defined this way. For a graph G and equivalence relations $\sim$ and $\equiv$ as above we write $\mathit{abstr\_graph}(G, \sim, \equiv)$ and $\mathit{abstr\_morph}(G, \sim, \equiv)$ for the abstract graph and the corresponding abstraction morphism constructed as shown above.

2.2 Isomorphism of Abstract Graphs

Abstract graphs can be abstracted just like graphs, yielding "more abstract" graphs. In this section we describe this abstraction relation, which is composeable. We then use it to define the notion of *isomorphism* between abstract graphs having the interesting property that isomorphic abstract graphs have the same concretisations.

Definition 3 (abstraction morphism between abstract graphs). *An* abstraction morphism *from an abstract graph S to an abstract graph T is a graph morphism $f : G_S \rightarrow G_T$ that complies to the following axioms:*

1. $\forall v, v' \in N_S$: $v \sim_S v'$ implies $f(v) \sim_T f(v')$;
2. $\forall w \in N_T$: $\mathsf{mult}_{\mathrm{n},T}(w) = \left(\sum^{\nu}_{v \in f^{-1}(w)} \mathsf{mult}_{\mathrm{n},S}(v)\right)$;
3. $\forall w \in N_T$, $\forall \mathsf{a} \in \mathsf{Lab}$, $\forall C \in N_T /{\sim_T}$, $\forall v \in f^{-1}(w)$, it holds

$$\mathsf{mult}_{\mathrm{out},T}(w, \mathsf{a}, C) = \sum^{\mu}_{D \,\in\, (f^{-1}(C))/{\sim_S}} \mathsf{mult}_{\mathrm{out},S}(v, \mathsf{a}, D)$$

and similarly for the incoming edges multiplicity.

The axioms in the previous definition are well-defined. In the third axiom we sum up $\mathsf{mult}_{\mathrm{out},S}(v, \mathsf{a}, D)$ and $\mathsf{mult}_{\mathrm{in},S}(v, \mathsf{a}, D)$ for all $D \in (f^{-1}(C)) /{\sim_S}$. It is then necessary that all the triples (v, a, D) belong to the domain of $\mathsf{mult}_{\mathrm{in},S}$, that is, it is necessary that any such D belongs to $N_S /{\sim_S}$. This is indeed the case thanks to the first axiom.

There is an analogy between an abstraction morphism between abstract graphs (Def. 3) and an abstraction morphism from a graph into an abstract graph (Def. 2). Namely, the second axiom in Def. 3 corresponds to the first axiom in Def. 2, but we sum up node multiplicities instead of simply counting nodes. Also the third axiom in Def. 3 and the second axiom in Def. 2 are analogous. Note that the composition of two abstraction morphisms yields another abstraction morphism [13].

We can now define two abstract graphs S and T to be *isomorphic* if there exists an isomorphism $f : G_S \rightarrow G_T$ such that both f and f^{-1} are abstraction morphisms. This leads us to the following interesting statement proven in [13].

Lemma 1. *If two abstract graphs S and T are isomorphic, then they have the same concretisations.*

Note that the inverse is not true. To this end, consider two abstract graphs S and T, where S has a single node of multiplicity two and no edges. T has two nodes, each of multiplicity one, and no edges. S and T both have a unique concretisation (up to graph isomorphism) which is the graph with two nodes and no edges, but S and T are clearly not isomorphic.

2.3 Neighbourhood Abstraction

So far, we have defined the notion of abstract graphs, concretisations, and abstraction morphisms. In our construction of abstract graphs, we assumed the existence of equivalence relations $\equiv$ and $\sim$ having particular properties. We shall now define an interesting choice of such relations inducing the notion of *neighbourhood abstractions*. This conveys the central idea of our work. In such an abstract graph, each node represents concrete graph nodes that have *similar neighbourhood, up to some "radius" i*. This i is a parameter of the precision of the neighbourhood abstraction. By gradually increasing i, we can obtain more precise abstractions, if the current one is too imprecise to verify the desired properties (abstraction refinement). We shall now define equivalence between nodes according to their neighbourhoods and, subsequently, neighbourhood abstraction.

Definition 4 (neighbourhood abstraction). *Let G be a graph. For each natural $i > 0$, we define the i-*neighbourhood equivalence *relation $\equiv_i$ over N_G recursively by:*

- $v \equiv_0 v'$ *if* $\mathsf{lab}_G(v) = \mathsf{lab}_G(v')$;
- $v \equiv_{i+1} v'$ *if* $v \equiv_i v'$, *and* $|v \rhd\!\!\rhd^{\mathsf{a}} C|_\mu = |v' \rhd\!\!\rhd^{\mathsf{a}} C|_\mu$, *and* $|C \rhd\!\!\rhd^{\mathsf{a}} v|_\mu = |C \rhd\!\!\rhd^{\mathsf{a}} v'|_\mu$ *for all label* a *in* Lab *and for all* $C \in N / \equiv_i$.

The level i neighbourhood abstraction of G *is* $abstr\_graph(G, \equiv_{i-1}, \equiv_i)$ *and the corresponding abstraction morphism is* $abstr\_morph(G, \equiv_{i-1}, \equiv_i)$.

Two nodes are mapped to the same node of the abstract graph if they are neighbourhood equivalent up to some radius. The grouping relation is also given by neighbourhood equivalence, but using a smaller radius. Figure 4 shows the level 1 neighbourhood abstraction of the firewall configuration from Fig. 1 for $\mu = 1$ and $\nu = 1$.

It is obvious from the definition that the level $i + 1$ neighbourhood abstraction of a graph refines the level i neighbourhood abstraction. This is the basis of our abstraction refinement mechanism. The neighbourhood abstraction of a graph is defined syntactically, which ties it to its representation. To avoid this inconvenient situation, in the sequel by neighbourhood abstraction of a graph we mean the isomorphism class of the actual abstract graph.

Neighbourhood abstraction behaves well w.r.t. isomorphism (Lemma 2 below). In combination with Lemma 1 this shows that two graphs obtained by neighbourhood abstraction are isomorphic iff they have the same concretisations.

Lemma 2. *Let S and T be two abstract graphs obtained by neighbourhood abstraction. If S and T admit a common concretisation, then they are isomorphic.*

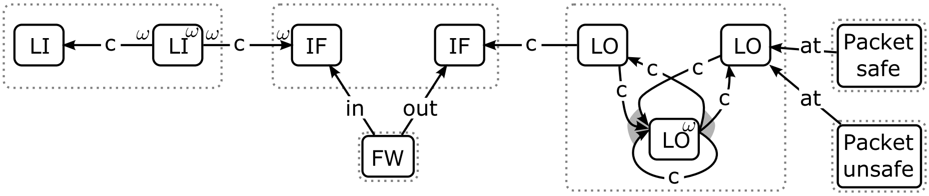


Fig. 4. The level one neighbourhood abstraction of the firewall example. Omitted node and edge multiplicities are equal to one.

3 A Modal Logic for Graphs and Abstract Graphs

So far, we have delineated a novel abstraction to be employed by abstract graph transformation, neighbourhood abstraction. For verification, we also need an accompanying logic to be defined now, which can be evaluated on both concrete and abstract graphs. A central theorem of our work states that this logic is preserved *and* reflected by neighbourhood abstraction.

Before we give the formal definition, let us look at properties for the firewall configuration of Fig. 1: (1) a packet cannot be safe and unsafe simultaneously; (2) an in-location cannot be directly connected to an out-location; (3) an unsafe packet never reaches an in-location; (4) a packet has at most one current position. These can be expressed in our logic as follows, where the $\rangle\mathsf{at}\rangle^{\lambda}$ operator is a forward existential modality, indicating the existence of *at least* λ *outgoing* c-*edge*; $\langle\mathsf{at}\langle^{\lambda}$ is similar, but for incoming edges; $\mathtt{tt}$ stands for the true formula.

(1) $(\mathsf{safe} \rightarrow \neg\mathsf{unsafe}) \wedge (\mathsf{unsafe} \rightarrow \neg\mathsf{safe})$

(2) $\neg\,(\,\mathsf{LI} \wedge \rangle\mathsf{c}\rangle^{1}\!\cdot\mathsf{LO}\,) \;\wedge\; \neg\,(\,\mathsf{LO} \wedge \rangle\mathsf{c}\rangle^{1}\!\cdot\mathsf{LI}\,)$

(3) $\neg\,(\,\mathsf{LI} \wedge \langle\mathsf{at}\langle^{1}\!\cdot\mathsf{unsafe}\,)$

(4) $\mathsf{Packet} \rightarrow \neg\rangle\mathsf{at}\rangle^{2}\!\cdot\mathtt{tt}$

3.1 Syntax and Semantics of the Logic

Logic formulae are defined by the following syntax (for $\mathsf{a} \in \mathsf{Lab}$ and $\lambda \in \mathbf{M}_{\mu}$):

$$\phi ::= \mathsf{a} \;\mid\; \neg\phi \;\mid\; \phi \vee \phi \;\mid\; \rangle\mathsf{a}\rangle^{\lambda}\!\cdot\phi \;\mid\; \langle\mathsf{a}\langle^{\lambda}\!\cdot\phi$$

The *nesting depth* $d(\phi)$ of ϕ measures the maximal number of nested modalities. It is defined recursively as: $d(\mathsf{a}) = 0$, $d(\rangle\mathsf{a}\rangle^{\lambda}\!\cdot\phi) = d(\langle\mathsf{a}\langle^{\lambda}\!\cdot\phi) = 1 + d(\phi)$, $d(\neg\phi) = d(\phi)$, $d(\phi \vee \phi') = d(\phi \wedge \phi') = \mathit{max}(d(\phi), d(\phi'))$ for any a in Lab. We write $\mathcal{L}_i$ for the set of formulae with nesting depth at most i. Logic formulae are interpreted in graph nodes. For a graph G, a node v in N_G, and a formula ϕ, the *satisfaction relation* $G, v \models \phi$ is defined recursively on the structure of ϕ by:

- $G, v \models \mathsf{a}$ if $\mathsf{a} \in \mathsf{lab}(v)$;
- $G, v \models \neg\phi$ if $G, v \not\models \phi$;
- $G, v \models \phi \vee \phi'$ if $G, v \models \phi$ or $G, v \models \phi'$;
- $G, v \models \rangle\mathsf{a}\rangle^{\lambda}\!\cdot\phi$ if $|\{e \in v\triangleright^{\mathsf{a}} \mid G, \mathsf{tgt}(e) \models \phi\}|_{\mu} \geq \lambda$;
- $G, v \models \langle\mathsf{a}\langle^{\lambda}\!\cdot\phi$ if $|\{e \in v\triangleleft^{\mathsf{a}} \mid G, \mathsf{src}(e) \models \phi\}|_{\mu} \geq \lambda$.

If $G, v \models \phi$, we say that ϕ *holds in node* v. Intuitively, a formula of the form $\rangle \mathsf{a} \rangle^{\lambda} \cdot \phi$ holds in a node v if the μ-bounded number of a-labelled edges (e) connecting it to some node v' ($\mathsf{src}_G(e) = v$ and $\mathsf{tgt}_G(e) = v'$) in which ϕ holds is at least λ. Analogously, $\langle \mathsf{a} \langle^{\lambda} \cdot \phi$ holds in v if the number of a-labelled edges connecting some such v' to v is at least λ.

The satisfaction relation is defined for an abstract graph almost in the same way as for a (concrete) graph. The difference is the way it is defined for modalities. There, we no longer count individual edges, but sum up edge multiplicities instead.

- $S, v \models \rangle \mathsf{a} \rangle^{\lambda} \cdot \phi$ if $\lambda \leq \sum_{C \in X}^{\mu} \mathsf{mult}_{\mathrm{out},S}(v, \mathsf{a}, C)$, and
- $S, v \models \langle \mathsf{a} \langle^{\lambda} \cdot \phi$ if $\lambda \leq \sum_{C \in X}^{\mu} \mathsf{mult}_{\mathrm{in},S}(v, \mathsf{a}, C)$,

where $X = \{C \in N_S / {\sim_S} \mid \forall w \in C.\ S, w \models \phi\}$. In the firewall example, $\mathsf{IF} \rightarrow \langle \mathsf{c} \langle^{1} \cdot (\mathsf{LI} \vee \mathsf{LO})$ holds in all nodes of the level 1 neighbourhood abstraction depicted. Note finally that counting makes our logic more expressive than the "usual" modal logic, while still strictly enclosed in first-order logic.

3.2 Preservation and Reflection

Let $s : G \rightarrow S$ be an abstraction morphism from the graph G to the abstract graph S. We say that s *preserves* a property p if whenever p holds in the node v of G, it also holds in the node $s(v)$ of S. Inversely, we say that s *reflects* p if whenever p holds in the node $s(v)$ of S, it also holds in the node v of G. One can also define in a similar manner preservation and reflection by an abstraction morphism between abstract graphs. Preservation and reflection are very important characterisations. If an abstraction preserves a set of safety properties, these properties can be verified on the abstract level. If an abstraction reflects a set of properties, then any characterisation of an abstract graph also holds for its concretisations. If both preservation and reflection hold, verifying a property on a graph is equivalent to verifying it on the abstract level. Neighbourhood abstraction features preservation and reflection of logic formulae with appropriate nesting depth, as stated in the following theorem.

Theorem 1 (Preservation and Reflection). *Let G be a graph and S the level i neighbourhood abstraction of G, for some $i \geq 1$, with corresponding abstraction morphism $s : G \rightarrow S$. Then s preserves and reflects $\mathcal{L}_i(\mathsf{Lab})$.*[1]

An important consequence of it is that neighbourhood abstraction can be parametrised by the properties we want to verify by choosing the level of abstraction

[1] Preservation of formulae with negation may seem in contradiction with the Morphism Preservation Theorem for finite structures [14] stating that a first order formula is preserved by morphism iff it is equivalent to an existential positive formula. Some modal logic formulae cannot be expressed in first-order logic without negation (*e.g.* $\neg \rangle \mathsf{a} \rangle^{\lambda} \cdot \mathtt{tt}$). However, in our case, abstract graphs contain information on the interpretation of negated formulae, by means of the multiplicity functions explaining this apparent contradiction.

that preserves the properties one is interested in. The following lemma formalises the relation between the logic and neighbourhood equivalence:

Lemma 3. *Two nodes v, v' of a graph G are i-neigborhood equivalent if, and only if, the same $\mathcal{L}_i(\mathsf{Lab})$ formulae hold in v and in v'.*

For our running example, let G be the graph of Fig. 1 and S its level 1 neighbourhood abstraction of Fig. 4. Let $s : G \rightarrow S$ be the corresponding abstraction morphism, and let $\phi = \mathsf{LO} \wedge \rangle\mathsf{c}\rangle^1 \cdot \mathsf{IF}$ and $\psi = \mathsf{LO} \wedge \rangle\mathsf{c}\rangle^1 \cdot \rangle\mathsf{c}\rangle^1 \cdot \mathsf{IF}$. In G, ϕ only holds for the LO-neighbour of the out interface, and in S, ϕ only holds for the corresponding abstract node. That is, ϕ of nesting depth 1 is preserved and reflected by s, whereas ψ, a formula of nesting depth 2, is not reflected. Indeed, in S, ψ holds in the LO-node with multiplicity ω but only in one of the pre-images of this node in G.

4 Canonical Representation of the Neighbourhood Abstraction

For abstract graph transformation, it is crucial to determine whether a newly computed abstract graph has been met before. To avoid expensive isomorphism checks on abstract graphs, we can benefit from a *canonical representation* of neighbourhood abstracted graphs. In effect, this reduces isomorphism checks to mere equality tests and is another important contribution we make.

Canonical names. Canonical names occur frequently in literature, e.g., in [11]. Here, a canonical name is a unique representation of an equivalence class w.r.t. a neighbourhood equivalence relation, which is independent of the underlying graph. For instance, each equivalence class for $\equiv_0$ contains only nodes having the same labels and is identified by this set of labels. It becomes the canonical name of this equivalence class. Each relation $\equiv_i$ is equipped with a set NCan^i of canonical names.

Definition 5 (Canonical Name). *The set of* level i node canonical names, NCan^i*, is defined inductively for $i \geq 0$:*

$$\begin{aligned} \mathsf{NCan}^0 &= 2^{\mathsf{Lab}} \\ \mathsf{NCan}^{i+1} &= \mathsf{NCan}^i \times (\mathsf{NCan}^i \times \mathsf{Lab} \rightarrow \mathbf{M}_\mu) \times (\mathsf{NCan}^i \times \mathsf{Lab} \rightarrow \mathbf{M}_\mu). \end{aligned}$$

The set ECan^i *of* level i edge canonical names *is* $\mathsf{ECan}^i = \mathsf{NCan}^i \times \mathsf{Lab} \times \mathsf{NCan}^i$.
Let G be a graph. The mapping name^i_G maps nodes and edges of G to their level i canonical name as follows. For node v of G, $\mathsf{name}^0_G(v) = \mathsf{lab}_G(v)$, and $\mathsf{name}^{i+1}_G(v) = (\mathsf{name}^i_G(v), out, in)$ where for $C \in \mathsf{NCan}^i$ and for each $\mathsf{a} \in \mathsf{Lab}$ (N_C stands for the set of nodes v' such that $\mathsf{name}^i_G(v') = C$),
$out(C, \mathsf{a}) = |v \rhd\!\!\rhd^{\mathsf{a}}_G N_C|_\mu$ $\qquad$ $in(C, \mathsf{a}) = |N_C \rhd\!\!\rhd^{\mathsf{a}}_G v|_\mu$.
For edge e of G, $\mathsf{name}^i_G(e) = (\mathsf{name}^i_G(\mathsf{src}(e)), \mathsf{lab}(e), \mathsf{name}^i_G(\mathsf{tgt}(e)))$.

Note that the number of different level i canonical names is finite. In combination with Lemma 4 below, we conclude that the number of level i neighbourhood abstractions is also finite up to isomorphism facilitating the verification of potentially infinite systems.

In the firewall example (Fig. 4), the different level zero canonical names are $C_1 = \{\mathsf{FW}\}$, $C_2 = \{\mathsf{IF}\}$, $C_3 = \{\mathsf{LI}\}$, $C_4 = \{\mathsf{LO}\}$, $C_5 = \{\mathsf{Packet}, \mathsf{safe}\}$ and $C_6 = \{\mathsf{Packet}, \mathsf{unsafe}\}$. The level one canonical name for the in-interface is $(C_2, \mathbf{0}, \mathit{in})$, where $\mathit{in} = \{(C_1, \mathsf{in}) \mapsto 1, (C_3, \mathsf{c}) \mapsto \omega\}$, and $\mathbf{0}$ is the constant zero function. The canonical name for the bottom-most LO-node is $(C_4, \mathit{out}, \mathit{in})$, where $\mathit{out} = \{(C_4, \mathsf{c}) \mapsto 1\}$ and $\mathit{in} = \{(C_4, \mathsf{c}) \mapsto 1\}$.

There is an obvious relation between canonical names and the neighbourhood equivalence expressed in the following central theorem. As a consequence of it and Lemma 3 we obtain that $v \equiv_i v'$ iff $\mathsf{name}^i_G(v) = \mathsf{name}^i_G(v')$, iff v and v' satisfy the same $\mathcal{L}_i$ logic formulae. This closes the circle between neighbourhood equivalence, canonical names, and logical satisfaction.

Theorem 2. *For any $i \geq 0$, any graph G, any two nodes v, v' of G and any two edges e, e' of G, $v \equiv_i v'$ if, and only if, $\mathsf{name}^i_G(v) = \mathsf{name}^i_G(v')$, and $e \equiv_i e'$ if, and only if, $\mathsf{name}^i_G(e) = \mathsf{name}^i_G(e')$.*

Canonical Representation of the Neighbourhood Abstraction. Let G be a graph. Consider the triple $\mathcal{C}_G = (\mathsf{name}^i(N_G), \mathsf{name}^i(E_G), \mathsf{mult})$, where $\mathsf{mult} : \mathsf{name}^i (N_G) \to \mathbf{M}^+_\nu$ is the function defined by $\mathsf{mult}(C) = \big|\{v \in N_G \mid \mathsf{name}^i_G(v) = C\}\big|_\nu$ for all $C \in \mathsf{name}^i(N_G)$. Then $\mathcal{C}_G$ is a canonical representation of the isomorphism class of the level i neighbourhood abstraction of G, as stated below:

Lemma 4. *Let G, H be graphs, and let $i \geq 1$. The level i neighbourhood abstractions of G and H are isomorphic if, and only if, $\mathcal{C}_G$ and $\mathcal{C}_H$ are equal.*

By $\mathcal{C}_G$ and $\mathcal{C}_H$ are equal, we mean component-wise equality, that is, equality of the sets of node and edge canonical names and equality of the node multiplicity functions that define them. Effectively, this allows us to reduce isomorphism checks on neighbourhood abstracted graphs to mere equality tests.

5 Towards an Automatic Verification Framework

We have defined a framework of neighbourhood abstractions having canonical representations and showed that an accompanying logic is preserved and reflected by it. We have not yet said how the application of a graph production rule is lifted to work on abstract graphs. Unfortunately, for lack of space, we need to refer the reader to [13] for a detailed treatment. In general, a rule application consists of three stages (which is typical of abstract graph transformation in general), where S is an abstract graph and (L, R, p) a production rule.

1. *Materialisation*: Transform the abstract graph S into the less abstract graph S', such that there is an abstraction morphism $S' \to S$ and a matching $m : L \to S'$, the image of which is a concrete sub-graph of S'; *i.e.* a sub-graph

in which all node and edge multiplicities are equal to one. S' is not unique, and the aim of materialisation is to construct the set of abstract graphs $\mathcal{S}$ such that for all concretisations G of S, for all matchings $m : G \to S$, there exists $T \in \mathcal{S}$ and abstraction morphisms $s : G \to T$, $t : T \to S$ such that the image of $s \circ m$ is concrete in T.

2. *Update*: As we are dealing with concrete (sub-)graphs now, rule application is just as usual and applied to each element of materialisation.
3. *Normalisation*: The graphs obtained after updates need not necessarily be in canonical form. Therefore, we need to neighbourhood abstract them again.

We show in [13] that this abstract transformation mechanism is sound, in the sense that it over-approximates the concrete graph abstraction. If a graph G can be transformed by rule (L, R, p) yielding a graph H, then G's abstraction S can be abstractly transformed by the same rule yielding H's abstraction T. Note that general negative application conditions (NACs) cannot be handled, as satisfiability of such conditions is not preserved by the abstraction. However, transformation can be defined for simple NACs which satisfiability can be checked using the canonical name only, as for instance conditions testing for the absence of an adjacent edge or path of bounded length.

The overall verification of a system works as follows. Start with the neighbourhood abstraction of the initial graph of the system to be verified. Given the abstract transformation defined above, successively apply it to construct an abstract version of the graph transformation system which has neighbourhood abstractions as states and which is *guaranteed* to be *finite* regardless the original system. The canonical representation of the neighbourhood abstraction makes it easy to check whether a newly derived state has already been met before, which is a very costly operation in normal graph transformations. Moreover, the abstraction can be parametrised by the property one wants to verify, expressed in the modal logic. As the modal logic is preserved by the abstraction, one may now evaluate formulas on finite, abstract graphs to obtain information about the possibly infinite-state original system. Finally, note that our framework naturally gives rise to abstraction refinement: If the level i neighbourhood abstraction is not conclusive, then one can try level $i + 1$.

Running Example. In the abstract graph transformation system (GTS) induced by the level 1 abstraction of the firewall example, all reachable abstract graphs have one FW-node and two IF-nodes, to which the different possible configurations of the internal and external networks are connected. The number of abstract configurations is bounded, whereas the number of concrete configurations is infinite, because of the possibility of creating packets and connecting new locations. Consider now the four properties listed above. They are all safety properties, defining invariants that should hold in all state of the GTS. These properties indeed hold in all states of the abstract GTS. As the four formulae are of nesting depth one, by reflection of the logic we can deduce that they also hold in all states of the concrete GTS. For this particular example, the abstraction mechanism allows to *verify* the four properties using the level 1 neighbourhood abstraction.

Usability and Limitations. Our verification framework is fully automatic and parametrised by the properties of interest. Due to the local nature of neighbourhood abstractions, it works well on systems where updates are determined locally and where reachability is not important. Typical use cases include the firewall example as given here or wireless traffic control systems as, e.g. the ones investigated in [7]. Also an application to ad-hoc network routing is promising but has not yet been explored.

However, our technique is not so suited for systems, where reachability is of importance, which is often the case for verification of software with dynamically allocated data structures. For instance, in the case of linked lists, our abstraction (regardless of radius) cannot distinguish between circular and non circular lists, which in practice results in lots of spurious states and transitions in the abstract transition system (*i.e.* states and transitions that do not exist in the concrete system). Also, the rather high complexity of our approach might be prohibitive for really large examples. This is yet to be explored by careful experimentation.

6 Conclusion

We presented a framework of graph abstractions, called neighbourhood abstraction, which generalises previous approaches to abstract graph transformation, a method for formal verification of graph transformation systems. The abstraction is based on regrouping nodes with similar neighbourhood, and can be parametrised by the radius of the neighbourhood to be considered. It is guaranteed to yield systems of finite, bounded size facilitating their exploration. We also presented a modal logic that can be interpreted both on graphs and on abstract graphs. The logic and the abstraction are closely related, which makes it possible to parametrise the abstraction so that it preserves and reflects the valuation of formulas. We delineated the implementation of a fully automated verification framework based on our novel abstractions and facilitated by a canonical form of abstract graphs. Interestingly, this framework also allows for automated abstraction refinement. Our proposal is illustrated by an interesting and relevant graph transformation based model of a firewall system. Usability and limitations of our approach were clearly identified. Note that related work was already discussed in the Introduction and that proofs and some other tedious formalities were left out but can be found in [13].

Future Work. We plan to implement our technique within the Groove [15] framework, a standard tool for graph transformations. This will allow for a more thorough exploration of more examples and for a qualified judgement on practical scalability. We believe that our framework caters for all possible *local* abstractions, where locality refers to the portion of the graph used to determine the equivalence of nodes. On the other hand, this complicates the verification of heap-manipulating programs, where reachability, a more global property, is crucial. Therefore, we are working on abstractions taking reachability into account. This is similar to abstractions used in the work of Sagiv *et al.* on shape analysis (see [11] for an overview) of heap-manipulating programs. In this work, the authors use logical structures to represent memory states of programs; abstract structures are 3-valued logical structures. Properties on these structures are defined using first-order logic with transitive closure (FO+TC) enabling the definition of reachability. It seems promising to explore the opportunities offered by FO+TC for abstract graph transformation too.

References

1. Heckel, R.: Compositional verification of reactive systems specified by graph transformation. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 138–153. Springer, Heidelberg (1998)
2. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: ICSE. ACM Press, New York (2006)
3. Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963. Springer, Heidelberg (2008)
4. Baldan, P., Corradini, A., König, B.: Verifying finite-state graph grammars: An unfolding-based approach. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170. Springer, Heidelberg (2004)
5. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920. Springer, Heidelberg (2006)
6. Rensink, A., Distefano, D.: Abstract Graph Transformation. In: International Workshop on Software Verification and Validation (SVV). ENTCS (2005)
7. Bauer, J.: Analysis of Communication Topologies by Partner Abstraction. PhD thesis, Universität des Saarlandes (2006)
8. Bauer, J., Wilhelm, R.: Static analysis of dynamic communication systems by partner abstraction. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634. Springer, Heidelberg (2007)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
10. Rensink, A.: Canonical Graph Shapes. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 401–415. Springer, Heidelberg (2004)
11. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. ACM Transactions on Programming Languages and Systems 24(3) (May 2002)
12. Bauer, J., Boneva, I., Rensink, A., Kurbán, M.E.: A modal-logic based graph abstraction, http://www7.in.tum.de/~joba/icgt08.pdf
13. Boneva, I., Rensink, A., Kurbán, M.E., Bauer, J.: Graph Abstraction and Abstract Graph Transformation. Technical report, University of Twente (2007)
14. Rossman, B.: Existential Positive Types and Preservation under Homomorphisms. In: 20th IEEE Symposium on Logic in Computer Science, pp. 467–476. CSP (2005)
15. 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)

On the Recognizability of Arrow and Graph Languages[*]

H.J. Sander Bruggink and Barbara König

Universität Duisburg-Essen
{sander.bruggink,barbara_koenig}@uni-due.de
www.ti.inf.uni-due.de/people/{bruggink,koenig}

Abstract. In this paper we give a category-based characterization of recognizability. A recognizable subset of arrows is defined via a functor into the category of relations on sets, which can be seen as a straightforward generalization of a finite automaton. In the second part of the paper we apply the theory to graphs, and we show that our approach is a generalization of Courcelle's recognizable graph languages.

1 Introduction

Regular languages have been studied extensively in computer science and they have a large number of applications. For instance, more recent approaches take advantage of regular languages for model-checking [1] and termination analysis [11]. The notion of regularity can be straightforwardly generalized to trees and tree automata. Hence one can talk about regular tree languages and exploit the convenient closure properties that these languages enjoy.

Hence, as a next step, it is natural to ask for a natural notion of regular graph languages. There have been several attempts to answer this question [23,18,22,5], all arriving at slightly different notions of regularity (also called recognizability), of which the notion of Courcelle [5,7] emerged as the one which is widely accepted. To our knowledge these notions of recognizability have not been applied extensively to verification and it is not entirely clear how they relate to graph transformation specified by double-pushout rewriting [4], one of the standard graph transformation approaches.

Courcelle focuses on the notion of recognizability—which is equivalent to regularity in the case of word languages—and which characterizes languages as the inverse image of a monoid morphism from the monoid of words (with concatenation) into a finite monoid. Specifically, he extends the more general notion of Mezei and Wright [17] from recognizability in one-sorted algebras to recognizability in many-sorted algebras by considering a specific algebra of graphs.

Here we compare the algebraic notion of recognizability by Courcelle with a categorical notion of recognizability, strongly related to composition-representative subsets introduced by Griffing [12].

[*] Supported by the DFG project SANDS.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 336–350, 2008.

We show that if we use the notion of Griffing and work in the category of cospans of graphs, we recover exactly the notion of recognizability proposed by Courcelle. Although both approaches rely on the same basic ideas, the proof is non-trivial. Furthermore it gives us a close relation with the double-pushout approach which can be characterized as a reactive system over cospans [21].

In addition we extend the notion of Griffing with a so-called automaton functor that—when instantiated to the word case—specializes to the notion of deterministic or non-deterministic finite automaton. We show that standard constructions on finite automata such as determinization or minimization can also performed on automaton functors.

We will proceed as follows: in Sect. 2 we will briefly introduce the necessary concepts of category theory, graphs and graph morphisms. In Sect. 3, we will introduce our category-theoretic notion of recognizable graph language, and in Sect. 4 we show that this enjoys useful properties, such as closure properties. In Sect. 5 we will apply our notion of recognizable arrow languages to define a notion of recognizable language of graphs, by considering the category of cospans of graphs, and we show that we can restrict ourselves to cospans between discrete graphs, i.e. graphs consisting of nodes only, without affecting the notion of recognizability. Finally, we show that our approach is equivalent to Courcelle's notion of recognizability. All proofs which are not in the paper, are made available in an extended version, which can be obtained from the authors' web pages.

2 Preliminaries

We briefly review the concepts from category theory, graphs and graph morphisms that will play a role in this paper. For more detailed introductions we refer to [16] and [10].

2.1 Category Theory

We presuppose basic knowledge of category theory. The identity arrow of an object G will be denoted by id_G (or just by id if G is clear from the context). Furthermore, arrow composition will be denoted by either ;, where the composed arrow which applies first f and then g is denoted by $f\ ;\ g$. The category $\mathcal{Rel}$ is the category which has sets as objects, relations as arrows and relation composition as composition operator. The category $\mathcal{Set}$ is the subcategory in which all arrows are in fact total functions.

In the rest of this section we define the more advanced concept of a cospan category, which will be used in Sect. 5. The idea of the cospan category is to have a category which has cospans as arrows.

Let $\mathscr{C}$ be a category in which all pushouts exist. A cospan c is a pair of $\mathscr{C}$-arrows $\langle c_\mathrm{L}, c_\mathrm{R}\rangle$ with the same codomain:

$$c\colon J \xrightarrow{c_\mathrm{L}} G \xleftarrow{c_\mathrm{R}} K \ \ .$$

Above, J and K are the domain (or *inner interface*) and codomain (or *outer interface*) of the cospan c, resp., i.e., the cospan can be considered as an arrow from J to K. The identity cospan for an object G is the cospan consisting of twice the identity arrow of G: $\langle \mathsf{id}_G, \mathsf{id}_G \rangle \colon G \to G \leftarrow G$. Let $c\colon J \to G \leftarrow M$ and $d\colon M \to H \leftarrow K$ be cospans (where the codomain of c equals the domain of d): The composition of c and d is defined by the commuting diagram

$$\begin{tikzcd}
 & & M \arrow[dl, "c_{\mathrm{R}}"'] \arrow[dr, "d_{\mathrm{L}}"] & & \\
J \arrow[r, "c_{\mathrm{L}}"] & G \arrow[dr, "f"'] & (\mathrm{PO}) & H \arrow[dl, "g"] & K \arrow[l, "d_{\mathrm{R}}"'] \\
 & & M' & &
\end{tikzcd}$$

where the middle diamond is a pushout, and the resulting composed cospan is defined as:

$$(c\,;d)\colon J \xrightarrow{c_{\mathrm{L}};f} M' \xleftarrow{d_{\mathrm{R}};g} K \ .$$

We want to form a category which has cospans as arrows, but in order to have the new category satisfy all axioms of category theory, we have to do some work. Let two cospans

$$c\colon J \xrightarrow{c_{\mathrm{L}}} G \xleftarrow{c_{\mathrm{R}}} K \quad \text{and} \quad d\colon J \xrightarrow{d_{\mathrm{L}}} H \xleftarrow{d_{\mathrm{R}}} K$$

with the same interfaces be given. We define the equivalence relation $\sim$ as follows: $c \sim d$ if there exists an isomorphism k from G to H, such that $c_{\mathrm{L}}\ ; k = d_{\mathrm{L}}$ and $c_{\mathrm{R}}\ ; k = d_{\mathrm{R}}$. A *semi-abstract cospan* is a $\sim$-equivalence class of cospans. Note that all members of a semi-abstract cospan have the same domain and codomain, and we define the domain and codomain of the semi-abstract cospan to be the domain and codomain of its members.

Now, the category $\mathit{Cospan}(\mathscr{C})$ is defined as the category which has the objects of $\mathscr{C}$ as objects, and semi-abstract cospans as arrows.

2.2 Graphs and Graph Morphisms

Let a set Σ of labels be given. A *hypergraph* G (in the following also called a *graph*) is a four-tuple $\langle V_G, E_G, \mathsf{att}_G, \mathsf{lab}_G \rangle$, where V_G is the set of nodes of G, E_G is the set of edges, $\mathsf{att}_G\colon E_G \to V_G^*$ (where V_G^* is the set of sequences of elements of V_G) is the *attachment function* and $\mathsf{lab}_G\colon E_G \to \Sigma$ is the *labeling function.* By $\emptyset$ we denote the empty graph. A *graph morphism* f between two graphs G and H is a pair $\langle f_{\mathrm{V}}, f_{\mathrm{E}} \rangle$ of functions $f_{\mathrm{V}}\colon V_G \to V_H$ and $f_{\mathrm{E}}\colon E_G \to E_H$ such that the following hold:

$$f_{\mathrm{E}}\ ; \mathsf{lab}_H = \mathsf{lab}_G \quad \text{and} \quad f_{\mathrm{E}}\ ; \mathsf{att}_H = \mathsf{att}_G\ ; f_{\mathrm{V}}^*$$

where f_{V}^* is the natural extension of f_{V} to sequences.

We define the category $\mathcal{HG}\mathit{raph}$ of hypergraphs as the category which has as objects finite (hyper)graphs, and as arrows graph morphisms. Concretely, taking a pushout of morphisms $f\colon U \to G$, $g\colon U \to H$ in the category of graphs means to take the disjoint union of G and H and to factor through the smallest equivalence $\equiv$ (on nodes and edges) that satisfies $f(x) \equiv g(x)$ for all items (i.e., for all nodes and edges) x of U.

In particular, we will work with the category $\mathit{Cospan}(\mathcal{HGraph})$. Cospans of graphs are intimately connected with the double-pushout (DPO) approach to graph rewriting [21]. DPO rewriting rules are spans of graphs of the form

$$p\colon L \xleftarrow{\varphi_{\mathrm{L}}} I \xrightarrow{\varphi_{\mathrm{R}}} R\ .$$

We consider the cospans $\ell\colon \emptyset \to L \overset{\varphi_{\mathrm{L}}}{\leftarrow} I$ and $r\colon \emptyset \to R \overset{\varphi_{\mathrm{R}}}{\leftarrow} I$ as left and right-hand sides. Now, for a graph G let $[G]\colon \emptyset \to G \leftarrow \emptyset$ be the cospan consisting of G with empty source and target. Then DPO rewriting can also be defined as follows: the graph G rewrites to H by applying rule p if and only if $[G] = \ell\ ;c$ and $[H] = r\ ;c$ for some cospan c.

3 Recognizable Languages of Arrows

We consider a fixed category $\mathscr{C}$. In order to be able to talk about sets, we will require that $\mathscr{C}$ is *locally small*, i.e., for every two objects the class of arrows between them is a set, called *homset*. Furthermore in the following a subset of a homset will be called an *(arrow) language*.

Definition 3.1 (Recognizability). *Let $\mathscr{C}$ be a category. We consider a functor $\mathcal{A}\colon \mathscr{C} \to \mathscr{R}el$ where every object X of $\mathscr{C}$ is mapped to a finite set $\mathcal{A}(X)$ (called the set of* states *of X) and every arrow $f\colon X \to Y$ is mapped to a relation $\mathcal{A}(f) \subseteq \mathcal{A}(X) \times \mathcal{A}(Y)$. We assume that every set $\mathcal{A}(X)$ contains a distinguished set $\mathrm{I}_X^{\mathcal{A}}$ of* start states *and a distinguished set $\mathrm{F}_X^{\mathcal{A}}$ of* final states *as subsets.*

The functor $\mathcal{A}$ is also called automaton functor*. It is called* deterministic *whenever every relation $\mathcal{A}(f)$ is a function and every $\mathrm{I}_X^{\mathcal{A}}$ contains exactly one element; this element will be denoted by $\mathrm{i}_X^{\mathcal{A}}$.*

Let J, K be two objects. The (J,K)-language $L_{J,K}(\mathcal{A})$ *(of arrows from J to K) is defined as follows:*

> *$f\colon J \to K$ is contained in $L_{J,K}(\mathcal{A})$ if and only if there exist $s \in \mathrm{I}_J^{\mathcal{A}}$, $t \in \mathrm{F}_K^{\mathcal{A}}$ which are related by $\mathcal{A}(f)$.*

A language $L_{J,K}$ of arrows from J to K is recognizable in $\mathscr{C}$ *if it is the (J,K)-language of a an automaton functor $\mathcal{A}\colon \mathscr{C} \to \mathscr{R}el$.*

The intuition behind the definition is to have a mapping into a finite domain that respects compositionality and identities, that is, which is a functor. The functor property ensures that decomposing the arrow in different ways does not affect acceptance in any way. This is different from the case of words where there is essentially only one way to decompose a word into atomic components.

In a sense $\mathcal{A}(f)$ gives the transition relation of the finite automaton, only that here we do not have a single set of states, but a set of states for every object. This means that we have infinitely many sets (which corresponds to the infinitely many sorts in the case of [5]).

Example 3.2. Let $\mathcal{F}$ be a nondeterministic finite automaton over the alphabet Σ, with state set Q, start states $I \subseteq Q$, final states $F \subseteq Q$ and transition relation $\delta \subseteq Q \times \Sigma \to \mathcal{P}(Q)$. We consider the free monoid of Σ, i.e., the category with one object X and words over Σ as arrows from X to X. We construct the following automaton functor $\mathcal{A}$ which recognizes a language L if and only if L is accepted by $\mathcal{F}$.

- $\mathcal{A}(X) = Q$, with $\mathrm{I}_X^{\mathcal{A}} = I$ and $\mathrm{F}_X^{\mathcal{A}} = F$;
- for $w \in \Sigma^*$, $\mathcal{A}(w) = \{\langle s, t\rangle \mid t \in \hat{\delta}(s, w)\}$, where $\hat{\delta}$ is the extension of δ from letters to words given by
  - $\hat{\delta}(s, \epsilon) = \{s\}$;
  - $\hat{\delta}(s, wa) = \bigcup\{\delta(s', a) \mid s' \in \delta(s, w)\}$.

Example 3.3. Also tree automata can be seen as a special case of our notion of automaton functor: take as the category a Lawvere theory where objects are natural numbers and arrows from m to n are n-tuples of terms with m holes. Arrow composition is the usual term substitution.

As an example take two unary function symbols f, g, a binary function symbol h and a constant a. Now, the 3-tuple $\langle f(x_1), h(x_1, x_2), g(f(a))\rangle$ has two holes, i.e., two variables x_1, x_2. Hence it can be regarded as an arrow from 2 to 3. Then trees can be represented as arrows from 0 to 1, since they are single terms (or 1-tuples of terms) without variables.

Note that this specific instantiation has similarly been considered in [12].

Two automaton functors are equivalent if they recognize the same language. Sometimes we are mainly interested in languages of arrows which start at a fixed object, for instance the initial object; therefore we parametrize the notion of equivalence with a source object.

Definition 3.4. *Let $\mathcal{A}$ and $\mathcal{B}$ be automaton functors. $\mathcal{A}$ and $\mathcal{B}$ are said to be* J-equivalent *if they recognize the same languages $L_{J,K}$ of arrows from J to K, for arbitrary K. They are* equivalent *if they are J-equivalent for all $\mathscr{C}$-objects J.*

It is also possible to find a characterization of recognizable language in terms of congruence classes, similar to Myhill-Nerode equivalences in the case of regular string languages.

Definition 3.5 (Congruence). *Let $\mathscr{C}$ be a category. Let a family of relations*

$$\equiv_{\mathrm{R}} = \{R_{J,K} \mid J, K \textit{ are objects of } \mathscr{C}\}$$

be given, where the components $R_{J,K}$ are equivalence relations on $\mathscr{C}$-arrows from J to K. We call $\equiv_{\mathrm{R}}$ a (right) congruence if the following holds for all arrows $a, a' \colon J \to K$, $b \colon K \to M$:

If $a\ R_{J,K}\ a'$, then $(a\ ;\ b)\ R_{J,M}\ (a'\ ;\ b)$.

A congruence $\equiv_{\mathrm{R}}$ is locally finite *if each $R_{J,K} \in \equiv_{\mathrm{R}}$ is an equivalence relation of finite index (i.e., it has finitely many equivalence classes).*

Let $a\colon J \to K$ be an arrow. In the following we will write $[\![a]\!]_{R_{J,K}}$, or simply $[\![a]\!]_{\mathrm{R}}$ if J and K are clear from the context, to denote the congruence class of which a is a member. We will also usually write $a \equiv_{\mathrm{R}} b$ instead of $a \; R_{J,K} \; b$.

Proposition 3.6. *Let $\mathscr{C}$ be a category, J, K $\mathscr{C}$-objects and $L_{J,K}$ a set of $\mathscr{C}$-arrows from J to K. The language $L_{J,K}$ is recognizable in $\mathscr{C}$, if and only if there exists a locally finite congruence $\equiv_{\mathrm{R}}$ such that $L_{J,K}$ is the union of some equivalence classes of $R_{J,K}$.*

Note that the proof of Prop. 3.6 only works if we fix a specific start object. Dually it would also have been possible to fix a target object and to concentrate on left congruences. But for our examples and for the comparison to the work of Courcelle fixing a start object seems to be more natural.

The paper by Griffing [12] does not introduce the notion of an automaton functor, but it shows that composition-representative subsets (which are our recognizable languages) can be characterized via locally finite congruences. In addition the paper gives another—equivalent—characterization via a functor from $\mathscr{C}$ into a category with finite homsets. The recognizable languages are then the preimages of subsets of a finite homset.

4 Determinism, Closure Properties and Minimization

One of the advantages of our characterization over characterizations utilizing finite homsets or congruences, is that we can talk explicitly about determinization and minimization. In this section we consider these two notions as well as closure properties. These results, different from the results in Sect. 5, are not particularly deep or surprising; usually they can be shown quite straightforwardly. We show them here for completeness and in order to illustrate that our notion of recognizability is reasonable.

Proposition 4.1. *For every automaton functor, there exists an equivalent deterministic automaton functor.*

Proof. (Sketch.) The construction is more or less equivalent to the case of finite automata: we replace every set of states by its powerset. □

Proposition 4.2 (Closure under boolean operators). *Suppose we have two recognizable languages of arrows, $L^1_{J,K}$ and $L^2_{J,K}$. Then also $L^1_{J,K} \cap L^2_{J,K}$, $L^1_{J,K} \cup L^2_{J,K}$ and $(L^1_{J,K})^{\mathrm{C}}$ (the complement of $L^1_{J,K}$) are recognizable.*

Proof. (Sketch.) Again the construction resembles the case of finite automata: we work with deterministic automaton functors, take the cross product of the state sets and define the final states appropriately. □

In the rest of this section we show that each deterministic automaton functor has a unique equivalent minimal one. The construction of a minimal automaton is analogous to the usual construction for finite automata: first we remove unreachable states, and then we fuse indistinguishable states.

The notion of minimality depends on the exact notion of equivalence we use. If we fix the source object in advance, the equivalence classes grow, and therefore smaller minimal automaton functors may be found. The constructions in the general case and the case with fixed source object are exactly the same, except for the fact that with a fixed source more states may be unreachable.

We give here two key definitions, of minimality and reachability, and mention the minimization result.

Definition 4.3. *Let $\mathcal{A}\colon \mathscr{C} \to \mathcal{R}el$ and $\mathcal{B}\colon \mathscr{C} \to \mathcal{R}el$ be automaton functors. We define:*

$$\mathcal{A} \le \mathcal{B} \text{ if for all } G \in \mathcal{O}bj(\mathscr{C}),\ |\mathcal{A}(G)| \le |\mathcal{B}(G)|$$

An automaton functor $\mathcal{A}$ is (J-)minimal *if, for all (J-)equivalent automaton functors $\mathcal{B}$ it holds that $\mathcal{A} \le \mathcal{B}$.*

Definition 4.4. *Let $\mathcal{A}\colon \mathscr{C} \to \mathcal{R}el$ be an automaton functor and K an object of $\mathscr{C}$. A state $s \in \mathcal{A}(K)$ is* J-reachable, *if there exists an arrow $c\colon J \to K$ such that $s \in \mathcal{A}(c)(\mathrm{I}_J^{\mathcal{A}})$. The state s is* reachable, *if it is J-reachable for some J.*

$\mathcal{A}$ is said to be fully (J-)reachable *if for all $\mathscr{C}$-objects K and states $s \in \mathcal{A}(K)$, s is (J-)reachable.*

Proposition 4.5. *For each deterministic automaton functor $\mathcal{A}$, there exists a minimal, (J-)equivalent, deterministic automaton functor $\mathcal{A}^{\min}$ which recognizes the same language and which is unique up to isomorphism.*

Note that the results of this section depend on the specific nature of the categories $\mathcal{R}el$ and $\mathcal{S}et$. It would be interesting to find an abstract characterization which still allows the techniques and constructions of this section.

5 Recognizable Graph Languages

In this section we apply the theory of the previous sections to recognizing languages of graphs. First, we show how graph languages can be recognized by considering the category of (semi-abstract) cospans of graphs. Then we briefly introduce Courcelle's algebraic notion of recognizable graph languages, and finally we show that we can recognize the same graph languages as Courcelle.

5.1 Recognizing Languages of Graphs by Cospans

In the following, the category under consideration will be $\mathcal{C}ospan(\mathcal{HG}raph)$, i.e., the category of cospans of graphs, or put differently, the category of graphs with inner and outer interfaces. If we want to talk only about graphs without interfaces, we can restrict ourselves to languages of cospans with empty interfaces, i.e., cospans where the source and target is the empty graph.

Definition 5.1. *A set L of graphs is recognizable whenever*

$$L_{\emptyset,\emptyset} = \{[G]\colon \emptyset \to G \leftarrow \emptyset \mid G \in L\}$$

is a recognizable language in $\mathcal{C}ospan(\mathcal{HG}raph)$.

Below we give an example for a recognizable graph language. It is not surprising that it is recognizable since it is definable in monadic second-order graph logic and all such languages are recognizable in the sense of Courcelle [5,7]. However, the example provides some intuition into the notion of recognizability.

Example 5.2 (k-Colorability). We set $\mathbb{N}_k = \{0, \dots, k-1\}$. Let G be a graph. A k-coloring of G is a function $f\colon V_G \to \mathbb{N}_k$ such that for all $e \in E_G$ and for all $v_1, v_2 \in \mathsf{att}_G(e)$ it holds that $f(v_1) \neq f(v_2)$ if $v_1 \neq v_2$. We show that the language of all k-colorable graphs is recognizable, by considering the following automaton functor $\mathcal{A}\colon \mathcal{Cospan}(\mathcal{HGraph}) \to \mathcal{Rel}$:

- Every graph J is mapped to $\mathcal{A}(J)$, the set of all valid k-colorings of J:

$$\mathcal{A}(J) = \{f\colon V_J \to \mathbb{N}_k \mid f \text{ is a valid } k\text{-coloring of } J\}\ .$$

- For a cospan $c\colon J \to G \leftarrow K$ the relation $\mathcal{A}(c)$ relates two colorings f_J, f_K, whenever there exists a coloring f for G such that $f(c_{\mathrm{L}}(v)) = f_J(v)$ for every node $v \in V_J$ and $f(c_{\mathrm{R}}(v)) = f_K(v)$ for every node $v \in V_K$.

Specifically we have that $\mathcal{A}(\emptyset) = \{\emptyset\}$ where $\emptyset$ is the empty coloring. Then in order to accept all k-colorable graphs with empty interfaces we take $\mathrm{I}_\emptyset^{\mathcal{A}} = \mathrm{F}_\emptyset^{\mathcal{A}} = \{\emptyset\}$: a graph $\emptyset \to G \leftarrow \emptyset$ is accepted whenever the two empty mappings are related.

Note that it is well known that k-colorability of graphs is an *NP*-complete property. Intuitively this manifests itself in the fact that interfaces may grow unboundedly, leading to an exponential explosion of the size of the state sets. However, if we restrict ourselves to graphs of bounded treewidth, there are efficient algorithms for k-colorability (see also the related discussion in the conclusion).

Example 5.3. Let H be a fixed graph. We consider the language L_H of all graphs G for which a morphism $f\colon G \to H$ exists. The language L_H is recognizable whenever H is finite.

The functor $\mathcal{A}$ associates to every graph J the set of all morphisms $J \to H$. For a cospan $c\colon J \to G \leftarrow K$ it relates a morphism $f_J\colon J \to H$ to a morphism $f_K\colon K \to H$ whenever there exists a morphism $f\colon G \to H$ such that $c_{\mathrm{L}}\ ;f = f_J$ and $c_{\mathrm{R}}\ ;f = f_K$. All states are initial and final.

$$\begin{tikzcd}
J \arrow[r, "c_{\mathrm{L}}"] \arrow[dr, "f_J"'] & G \arrow[d, dotted, "f"] & K \arrow[l, "c_{\mathrm{R}}"'] \arrow[dl, "f_K"] \\
 & H &
\end{tikzcd}$$

This is a weaker notion than recognizability and has been considered before (see for instance [14,3]).

5.2 Robustness

We now show the robustness of recognizability by restricting ourselves to injective, edge-injective and discrete interfaces. These results will also be important for the comparison with Courcelle's notion of recognizability (see Sect. 5.4).

We use proof principles already explored in [9] where robustness proofs are based on the characterization of recognizable languages in terms of locally finite congruences (see Def. 3.5). The results and proofs all follow the same lines: let $\mathscr{D}$ be a subcategory of $\mathscr{C}$ and let J, K be two objects of $\mathscr{D}$. We want to show that every language of arrows from J to K is recognizable in $\mathscr{C}$ if and only if it is recognizable in $\mathscr{D}$. The direction from left to right is obvious, we simply restrict the congruence or the automaton functor accordingly. The real challenge is the direction from right to left. In this case take a congruence $\equiv_{\mathrm{R}}$ on $\mathscr{D}$ and construct a congruence $\equiv'_{\mathrm{R}}$ on $\mathscr{C}$ that is locally finite and refines $\equiv_{\mathrm{R}}$ when restricted to $\mathscr{D}$. Since $\equiv'_{\mathrm{R}}$ refines $\equiv_{\mathrm{R}}$ any union of equivalence classes can still be represented as a union of (possibly more) equivalence classes, and hence recognizability is preserved.

We now show the convenient fact that restricting our attention to cospans with injective interfaces does not limit the descriptive power of the formalism. Hence let $\mathscr{G} = \mathcal{Cospan}(\mathcal{HGraph})$ be a cospan category and let $\mathscr{G}_{\mathrm{inj}}$ be its subcategory that consist only of the cospans with injective interface morphisms.

Proposition 5.4. *Let a class $L_{J,K}$ of graphs with injective interfaces J, K be called* injectively recognizable *whenever $L_{J,K}$ is recognizable in $\mathscr{G}_{\mathrm{inj}}$. Then $L_{J,K}$ is recognizable in $\mathscr{G}$ if and only if it is injectively recognizable.*

We now restrict our attention to the subcategory $\mathscr{G}_{\mathrm{einj}}$ of cospans with interface morphism which are injective on edges. The result is not that interesting in its own right, but it is a necessary auxiliary step for Prop. 5.6.

Corollary 5.5. *Let a class $L_{J,K}$ of graphs with edge-injective interfaces J, K be called* edge-injectively recognizable *whenever $L_{J,K}$ is recognizable in $\mathscr{G}_{\mathrm{einj}}$. Then $L_{J,K}$ is recognizable in $\mathscr{G}$ if and only if it is edge-injectively recognizable.*

Similar to the restriction to injective interfaces we now show the fact that restricting our attention to cospans with discrete interfaces does not limit the descriptive power of the formalism. This allows us to restrict our attention to discrete interfaces in the following. Let $\mathscr{G} = \mathcal{Cospan}(\mathcal{HGraph})$ be a cospan category and let $\mathscr{G}_{\mathrm{dis}}$ be its subcategory that consist only of the cospans with discrete interfaces, i.e., with interface graphs that do not contain edges.

Proposition 5.6. *Let a class $L_{J,K}$ of graphs with discrete interfaces J, K be called* discretely recognizable *whenever $L_{J,K}$ is recognizable in $\mathscr{G}_{\mathrm{dis}}$. Then $L_{J,K}$ is recognizable in $\mathscr{G}$ if and only if it is discretely recognizable.*

In a sense the result above is mirrored in the fact that graph rewriting does not lose any expressive power if we restrict to discrete interfaces.

5.3 Courcelle's Algebra of Graphs

We will now give a short introduction to Courcelle's algebraic notion of recognizable graph languages [5,7]. Courcelle's notion of recognizable graph language is widely accepted as a notion of regularity for graphs. Also, Courcelle showed that if a language is definable in monadic second-order logic then it is recognizable. In [13] Courcelle's notion has been found to be nearly identical to the notions finite graph property and compatible graph property, which were developed in different contexts. For instance compatible properties arise in connection with hyperedge replacement grammars.

In [7] the relevant graph algebra is called HR-algebra (and in addition another algebra, called VR-algebra is investigated). However, here in this paper we use the algebra introduced in [5], since it is closer to our notion of cospan composition and hence yields simpler proofs.

Note that there are two major differences between cospan composition and the algebra of graphs introduced below: first the graph algebra considers only discrete interfaces, and we have already shown how to bridge this gap via Prop. 5.6. Second, cospans have two interfaces whereas graphs in the algebra have only one.

First, we give some preliminary definitions. We set $\mathbb{N}_k = \{0, \ldots, k-1\}$. Let P, Q be arbitrary sets. For functions $f\colon \mathbb{N}_n \to P$ and $g\colon \mathbb{N}_m \to Q$, we define the function $f \odot g\colon \mathbb{N}_{n+m} \to P \cup Q$ as follows:

$$(f \odot g)(i) = \begin{cases} f(i) & \text{if } i < n \\ g(i-n) & \text{otherwise.} \end{cases}$$

An n-ary hypergraph is a pair $\mathbb{G} = \langle \mathsf{base}_{\mathbb{G}}, \zeta_{\mathbb{G}} \rangle$ consisting of a hypergraph $\mathsf{base}_{\mathbb{G}}$ and a mapping $\zeta_{\mathbb{G}}\colon \mathbb{N}_n \to V$, where V is the node set of $\mathsf{base}_{\mathbb{G}}$. The function $\zeta_{\mathbb{G}}$ is called the *interface* of the graph, and its range the *external nodes.* Basically, an n-ary hypergraph corresponds to a graph with an empty internal and discrete external interface.

In [2], the following atomic operations on n-ary graphs are defined:

Redefinition of external nodes. Let an n-ary hypergraph $\mathbb{G}$ be given, and let $\sigma\colon \mathbb{N}_m \to \mathbb{N}_n$ be a function. The redefinition of $\mathbb{G}$ under σ is:

$$\mathsf{redef}_{\sigma}(\mathbb{G}) = \langle \mathsf{base}_{\mathbb{G}}, (\sigma \,;\, \zeta_{\mathbb{G}}) \rangle.$$

Note that this means that $\mathsf{redef}_{\sigma}(\mathbb{G})$ is an m-ary graph.

Fusion of external nodes. Let $\mathbb{G}$ be an n-ary graph, and θ an equivalence relation on $\mathbb{N}_n$. The fusion of $\mathbb{G}$ over θ, denoted $\mathsf{fuse}_{\theta}(\mathbb{G})$, is obtained by fusing the nodes of $\mathbb{G}$ according to θ. The result is again an n-ary graph.

Disjoint union. Let $\mathbb{G}$ be an n-ary graph and $\mathbb{H}$ an m-ary graph. The disjoint union of $\mathbb{G}$ and $\mathbb{H}$ is defined as:

$$\mathbb{G} \oplus \mathbb{H} = \langle \mathsf{base}_{\mathbb{G}} \oplus \mathsf{base}_{\mathbb{H}}, \zeta_{\mathbb{G}} \odot \zeta_{\mathbb{H}} \rangle$$

(we assume here that the node sets of $\mathsf{base}_{\mathbb{G}}$ and $\mathsf{base}_{\mathbb{H}}$ are disjoint and that $\oplus$ denotes the disjoint union on base graphs.)

We will now define recognizability in the sense of Courcelle via congruences. There is an alternative, but equivalent, definition of recognizable subsets as preimages of algebra homomorphisms.

Definition 5.7. *Let* $\equiv_{\mathrm{C}}$ *be an equivalence on* n*-ary hypergraphs that relates only hypergraphs with the same arity. It is called* locally finite *if for each* n *there are only finitely many equivalence classes. It is called a* congruence *if the following conditions hold:*

- *if* $\mathbb{G} \equiv_{\mathrm{C}} \mathbb{H}$*, then* $\mathsf{redef}_{\sigma}(\mathbb{G}) \equiv_{\mathrm{C}} \mathsf{redef}_{\sigma}(\mathbb{H})$*;*

- *if* $\mathbb{G} \equiv_{\mathrm{C}} \mathbb{H}$, *then* $\mathsf{fuse}_\theta(\mathbb{G}) \equiv_{\mathrm{C}} \mathsf{fuse}_\theta(\mathbb{H})$*;*
- *if* $\mathbb{G}_1 \equiv_{\mathrm{C}} \mathbb{H}_1$ *and* $\mathbb{G}_2 \equiv_{\mathrm{C}} \mathbb{H}_2$*, then* $\mathbb{G}_1 \oplus \mathbb{G}_2 \equiv_{\mathrm{C}} \mathbb{H}_1 \equiv_{\mathrm{C}} \mathbb{H}_2$*.*

A set L of n-ary graphs is called Courcelle-recognizable *if it is the union of finitely many equivalence classes of a locally finite congruence.*

We will in the following show that the notion of recognizability of Courcelle coincides with our notion, hence the notion of "Courcelle-recognizability" is redundant. However, we will keep it for the moment in order to properly distinguish the two notions of recognizability.

5.4 Equivalence of the Two Notions of Recognizability

In this subsection we show that our notion of recognizable graph language is equivalent to Courcelle's. In Courcelle's notion there is only one (discrete) interface. The role of cospan composition is played by the operators defined above.

Encouraged by the result of Prop. 5.6 we restrict our attention to cospans with discrete interfaces. The discrete graph with node set $\mathbb{N}_n$ will be called canonical n-graph, and will be represented by Dis_n; we make use of the fact that each discrete graph with n nodes is isomorphic to the canonical n-graph. To formalize the equivalence between both notions of recognizability, we must first associate (sets of) cospans of graphs with n-ary graphs.

Definition 5.8. *For each discrete graph D with n nodes we fix in advance an isomorphism $\mathsf{di}_D \colon \mathsf{Dis}_n \to D$ such that $\mathsf{di}_{D_1 \oplus D_2} = \mathsf{di}_{D_1} \odot \mathsf{di}_{D_2}$.*

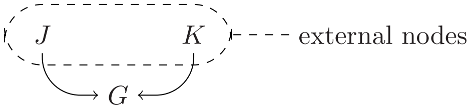
We define the function bend *which maps a cospan*

$$c \colon J \xrightarrow{c_{\mathrm{L}}} G \xleftarrow{c_{\mathrm{R}}} K$$

where J, K are discrete interfaces with n, m nodes, resp., to an $(n+m)$-ary graph as follows:

$$\mathsf{bend}(c) = \langle G, (\mathsf{di}_J \,;\, c_{\mathrm{L}}) \odot (\mathsf{di}_K \,;\, c_{\mathrm{R}}) \rangle \ .$$

The name bend is inspired by the fact that the function basically 'bends' a cospan so that its inner and outer interface are together, and then interprets the resulting figure as a $(m+n)$-ary hypergraph, as illustrated below:



Theorem 5.9. *Let J be a discrete graph. A set of graphs L is the $(\emptyset, J)$-language of some automaton functor $\mathcal{A}$ if and only if* $\mathsf{bend}(L)$ *is Courcelle-recognizable.*

Proof. (Sketch.) We prove the theorem by simulating Courcelle's operations by cospan composition, and cospan composition by Courcelle's operations, so that the congruences can be transferred. Suppose J has n nodes. The simulations of Courcelle's operations work as follows (see Fig. 1):

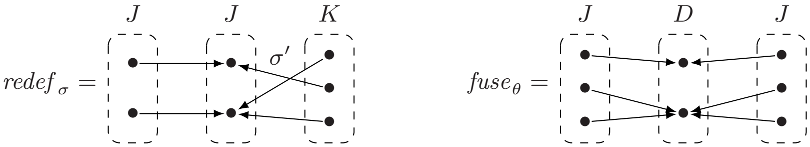


Fig. 1. Simulating Courcelle's graph operation by cospans. On the left: example of cospan simulating redefinition of external nodes; on the right: example of cospan simulating fusion of external nodes.

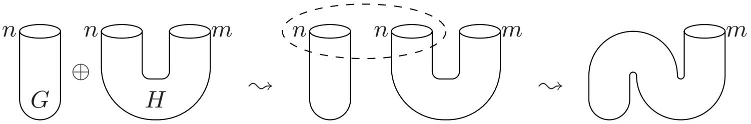


Fig. 2. *Simulating cospan composition with Courcelle's operations.* First, we construct the disjoint union of the graphs In the second step the indicated external nodes are fused and then removed from the external nodes by a redefinition.

- Let $\sigma\colon \mathbb{N}_m \to \mathbb{N}_n$ be a function, and K a discrete graph with m nodes. It can be considered as a graph morphism from Dis_m to Dis_n. Then $\sigma' = \mathsf{di}_K^{-1}$; σ ; di_J is the corresponding graph morphism from K to J. Postcomposing a cospan c with the cospan

$$redef_\sigma\colon J \xrightarrow{\mathsf{id}_J} J \xleftarrow{\sigma'} K$$

  simulates performing the redef_σ operation on c.
- Let θ be an equivalence relation on the elements of $\mathbb{N}_n$ (i.e. an equivalence relation on the nodes of Dis_n). Suppose θ_{map} is the morphism which maps each element of $\mathbb{N}_n$ to its θ-equivalence class and let $\theta' = \mathsf{di}_J^{-1}$; θ_{map}. Then

$$fuse_\theta\colon J \xrightarrow{\theta'} D \xleftarrow{\theta'} J \ ,$$

  where D is the discrete graph with node set $\{[\![v]\!]_\theta \mid v \in \mathbb{N}_n\}$, simulates the fuse_θ operation.
- Disjoint sum is simulated by disjoint sum in the category of graphs and we obtain the congruence property via the congruence property for cospan composition (see the extended proof in the extended version).

The simulation of cospan composition in Courcelle's algebra depends on the fact that $\mathsf{bend}(c\ ;\ d) = \mathsf{redef}_\sigma(\mathsf{fuse}_\theta(\mathsf{bend}(c) \oplus \mathsf{bend}(d)))$ for appropriate σ and θ. In Fig. 2 this is depicted for cospans c, d where c has inner interface $\emptyset$. □

Note that one of the reasons why the proof works is the fact that the category of cospans is compact-closed, which means that certain "bending" laws, similar to the one above, hold.

6 Conclusion

We have shown that a very general categorical notion of recognizability via automaton functors (which is equivalent to a notion suggested by Griffing [12]) is equivalent to a notion of recognizability for graph languages by Courcelle whenever we consider the category of cospans of graphs. The proof of this equivalence is non-trivial.

Furthermore we investigated our notion of automaton functor and showed that it preserves several nice properties which are well-known for finite-state automata. Our main motivation behind this work is to provide automata-based techniques for verification and termination analysis of graph transformation systems. Some preliminary results on termination analysis are reported in [3].

Cospans of graphs were also investigated, in the context of gluing graph structures, by Rosebrugh, Sabadini and Walters [19,20], but in [20] their graph structures represent automata which recognize word languages rather than graph languages. In the future we plan to explore the relations between their and our work in some more detail.

Naturally, efficiency questions arise. The automaton functor we are working with is only locally finite, i.e., the sets of states are finite for every interface, but interfaces might be arbitrarily large. This question has already been addressed by Courcelle, who characterized classes of graphs which can be recognized efficiently. He showed that for the HR-algebra of graphs a graph can be decomposed via interfaces whose size is bounded by $k+1$ if and only if its treewidth is bounded by k [6,7]. Hence a language L can be recognized efficiently (even in linear time!) if there is a bound on the treewidth of the graphs contained in L (see also [8]). This is also known as Courcelle's theorem and applies to properties such as k-colorability that would be NP-complete on graphs of unbounded treewidth.

Since with cospans we have a different notion of interface and different operations, this result by Courcelle does not carry over directly, although we have the same notion of recognizability. This is a point which has to be investigated further, but in order to arrive at a similar result we believe that it is necessary to equip our category with a monoidal operation $\oplus$ (which is the disjoint sum on cospans) and to require that an automaton functor preserves this monoidal operation. We conjecture that, at least in the case of graphs with empty inner interface, adding such a monoidal operation will not affect which sets of (cospans of) graphs are recognizable. Currently it seems that we can only guarantee linear-time algorithms for graphs of bounded pathwidth, since cospans allow only to construct path decompositions of graphs.

Of course, in order to obtain practical algorithms for recognizability, we have to find reasonable ways to represent and handle automaton functors, at least in the case of graphs of bounded treewidth. We have some preliminary ideas how this can be achieved, but it is an interesting problem that has to be studied further.

In this paper we mainly considered cospans of graphs, but there is a more general notion of (DPO) rewriting based on adhesive categories [15,10]. Our notion of recognizability can be easily generalized to this setting, whereas it is not entirely clear how to extend Courcelle's algebra of graphs. One possible application of such a generalization provides us with a method to show that (recognizable) sets of objects in an adhesive category are invariant under DPO rewriting rules. Let $p\colon L \leftarrow I \rightarrow R$ be a DPO rule and let $\equiv_{\mathrm{R}}$ be a congruence[1] characterizing a language L of objects. We observe that whenever an object A is rewritten to B via p, $A \in L$ and $(0 \rightarrow L \leftarrow I) \equiv_{\mathrm{R}} (0 \rightarrow R \leftarrow I)$ (for an initial object 0), then we can conclude that $B \in L$.

Finally, an important result in Courcelle's work is that a language is recognizable whenever it is definable in monadic second-order logic [5,7]. Currently we have no counterpart to this result, but it might be worthwhile to study it in a more categorical setting.

Acknowledgement

The authors would like to thank Tobias Heindel for his valuable comments on the contents of this paper.

References

1. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
2. Bouderon, M., Courcelle, B.: Graph expressions and graph rewritings. Mathematical Systems Theory 20, 81–127 (1987)
3. Bruggink, H.J.S.: Towards a systematic method for proving termination of graph transformation systems (work-in-progress paper). In: Proc. of GT-VC 2007 (Graph Transformation for Verification and Concurrency) (2007)
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, Foundations, ch. 3, vol. 1, World Scientific, Singapore (1997)
5. Courcelle, B.: The monadic second-order logic of graphs I. recognizable sets of finite graphs. Information and Computation 85, 12–75 (1990)
6. Courcelle, B.: Graph grammars, monadic second-order logic and the theory of graph minors. In: Graph Structure Theory, pp. 565–590. American Mathematical Society (1991)
7. Courcelle, B.: The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Foundations, ch. 5, vol. 1. World Scientific, Singapore (1997)
8. Courcelle, B., Lagergren, J.: Equivalent definitions of recognizability for sets of graphs of bounded tree-width. Mathematical Structures in Computer Science 6(2), 141–165 (1996)

[1] One is here actually interested in the weaker notion of a Myhill well-quasi order instead of a congruence, but we omit the details.

9. Courcelle, B., Weil, P.: The recognizability of sets of graphs is a robust property. Theoretical Computer Science 342(2–3), 173–228 (2005)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
11. Geser, A., Hofbauer, D., Waldmann, J.: Match-bounded string rewriting systems. Applicable Algebra in Engineering, Communication and Computing 15(3–4), 149–171 (2004)
12. Griffing, G.: Composition-representative subsets. Theory and Applications of Categories 11(19), 420–437 (2003)
13. Habel, A., Kreowski, H.-J., Lautemann, C.: A comparison of compatible, finite and inductive graph properties. Theoretical Computer Science 110(1), 145–168 (1993)
14. Kuncak, V., Rinard, M.C.: Existential heap abstraction entailment is undecidable. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 418–438. Springer, Heidelberg (2003)
15. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO – Theoretical Informatics and Applications 39(3) (2005)
16. Mac Lane, S.: Categories for the Working Mathematician. Springer, Heidelberg (1971)
17. Mezei, J., Wright, J.B.: Algebraic automata and context-free sets. Information and Control 11, 3–29 (1967)
18. Potthoff, A., Seibert, S., Thomas, W.: Nondeterminism versus determinism of finite automata over directed acyclic graphs. Bulletin of the Belgian Mathematical Society 1, 285–298 (1994)
19. Rosebrugh, R., Sabadini, N., Walters, R.F.C.: Generic commutative separable algebras and cospans of graphs. Theory and applications of categories 15(6), 164–177 (2005)
20. Rosebrugh, R., Sabadini, N., Walters, R.F.C.: Calculating colimits compositionally (2007)
21. Sassone, V., Sobociński, P.: Reactive systems over cospans. In: Proc. of LICS 2005, pp. 311–320. IEEE Computer Society Press, Los Alamitos (2005)
22. Urvoy, T.: Abstract families of graphs. In: Ito, M., Toyama, M. (eds.) DLT 2002. LNCS, vol. 2450, pp. 381–392. Springer, Heidelberg (2003)
23. Witt, K.-U.: Finite graph-acceptors and regular graph-languages. Information and Control 50(3), 242–258 (1981)

Graph Multiset Transformation as a Framework for Massively Parallel Computation*

Hans-Jörg Kreowski and Sabine Kuske

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

Abstract. In this paper, graph multiset transformation is introduced and studied as a novel type of parallel graph transformation. The basic idea is that graph transformation rules may be applied to all or at least some members of a multiset of graphs simultaneously providing a computational step with the possibility of massive parallelism in this way. As a consequence, graph problems in the class *NP* can be solved by a single computation of polynomial length for each input graph.

1 Introduction

In this paper, a new type of graph transformation, called graph multiset transformation, is introduced that is inspired by the concepts of genetic algorithms and DNA computing (see, e.g., [1,2,3,4,5]). Adleman's seminal experiment demonstrates how combinatorial problems may be solved using DNA. Roughly speaking, a tube is filled with certain quantities of properly chosen DNA strands. Then their reactions according to the Watson-Crick complementarity produces DNA molecules, a suitable selection of which represents solutions. Similarly, a genetic algorithm transforms a "population of individuals" step by step into one of "fitter" individuals by means of "mutation," "cross-over," and "selection." If, for example, the individuals are solutions of an optimization problem that differ from the optimum, then the genetic algorithm may yield solutions that are closer to the optimum or even meet it. If one replaces tubes of molecules and populations of individuals by multisets of graphs and chemical reactions and genetic operations by rule applications, one gets the concept of graph multiset transformation.

The basic idea is that graph transformation rules may be applied to all or at least some members of a multiset of graphs simultaneously providing a computational step with the possibility of massive parallelism in this way. As a consequence, graph problems in the class *NP* can be solved by a single computation of polynomial length for each input graph.

* The authors would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 351–365, 2008.

The paper is organized in the following way. The next section provides the prelimininaries concerning graphs and rule-based graph transformation. In Section 3, simple graph transformation units are recalled as devices to model and compute graph algorithms and processes. Section 4 introduces a way to solve decision problems on graphs by means of terminating units. In particular, a graph-transformational variant of the class *NP* is defined. Based on simple and terminating units, graph multiset transformation is proposed as a computational framework with massive parallelism in Section 5 and 6. As a consequence, *NP*-problems can be solved in a polynomial number of computational steps. The Appendix recalls multisets together with some basic definitions used in this paper. Throughout the paper, the well-known *NP*-complete Hamiltonian path problem is discussed as a running example. The proofs are omitted because of the limited space. It may be noted that the basic ideas of graph multiset transformations have been sketched in [6] in a draft way.

2 Graphs and Rule-Based Graph Transformation

In this section, we recall the basic notions and notations of graphs and rule-based graph transformation as far as they are needed in this paper. We use directed and edge-labeled graphs with binary edges.

Let Σ be a set of labels. A *graph* over Σ is a system $G = (V, E, s, t, l)$ where V is a finite set of *nodes*, E is a finite 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 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 $\mathcal{G}_\Sigma$. We reserve a specific label $*$ which is omitted in drawings of graphs. In this way, graphs where all edges are labeled with $*$ may be seen as *unlabeled graphs*. The sum of the number of nodes and the number of edges is the *size* of G, denoted by $size(G)$.

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 $e \in E_G$. If the mappings g_V and g_E are bijective, then g is an *isomorphism*, and G and H are called *isomorphic*. If the mappings g_V and g_E are inclusions, then G is called a *subgraph* of H, denoted by $G \subseteq H$. For a graph morphism $g\colon G \to H$, the image $g(G) \subseteq H$ of G in H is called a *match* of G in H.

Example 1. The graph G_0 in Figure 1 has four Hamiltonian paths which are represented by the graphs H_1, H_2, H_3, and H_4.[1] A box □ represents a node with an unlabeled loop. Therefore, G_0 has four nodes, four loops and five additional unlabeled edges. The other graphs are variants of G_0. We use ⦿ to represent a *begin*-node which is a node with a loop labeled with *begin*. Analogously, ○ represents an *end*-node. If one starts in the *begin*-node and follows the p-labeled

[1] A path is called Hamiltonian if it visits every node exactly once.

edges, one reaches the *end*-node in the graphs H_1, H_2, H_3, and H_4. In each case, the sequence of p-edges defines a Hamiltonian path of G_0, where the intermediate nodes have no loops.

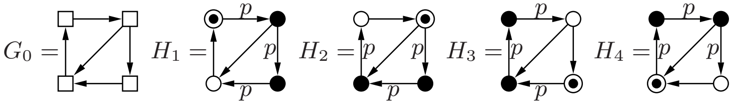


Fig. 1. G_0 with all its Hamiltonian paths

If one removes the right vertical edge and the loops at the source and the target of this edge in the graph G_{12} in Figure 2, then one gets the subgraph Z_0. One may extend the graph Z_0 by a p-edge and an *end*-loop to get G_{123}.

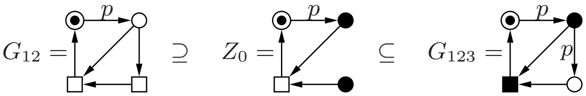


Fig. 2. Two graphs with a common subgraph

There are two graph morphisms from the graph $L_{run} =$ ○⟶□ into G_{12} which map to the subgraphs of the same form.

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 $r = (L \supseteq K \subseteq R)$ to a graph $G = (V, E, s, t, l)$ consists of the following three steps.

1. A match $g(L)$ of L in G is chosen subject to the following conditions.
   – *contact condition*: $v \in g_V(V_L)$ with $s_G(e) = v$ or $t_G(e) = v$ for some $e \in E_G - g_E(E_L)$ implies $v \in g_V(V_K)$.
   – *identification condition*: $g_V(v) = g_V(v')$ for $v, v' \in V_L$ implies $v = v'$ or $v, v' \in V_K$ as well as $g_E(e) = g_E(e')$ for $e, e' \in E_L$ implies $e = e'$ or $e, e' \in E_K$.
2. Now the nodes of $g_V(V_L - V_K)$ and the edges of $g_E(E_L - E_K)$ are removed yielding the *intermediate graph* $Z \subseteq G$.
3. Let $d\colon K \to Z$ be the restriction of g to K and Z, then the pushout of d and the inclusion of K into R yields the resulting graph H and graph morphisms $h\colon R \to H$ and $i\colon Z \to H$. Without loss of generality, one can assume that i is the inclusion of Z into H and that h is the identity on $R - K$. This provides an explicit construction of H because $Z \cup h(R) = H$ and $Z \cap h(R) = d(K) = h(K)$.

The application of a rule r to a graph G is denoted by $G \underset{r}{\Rightarrow} H$, where H is the graph resulting from the application of r to G. A rule application is called a *direct derivation*. The subscript r may be omitted if it is clear from the context.

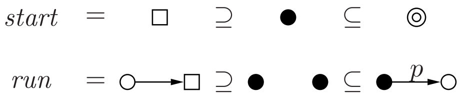
The contact condition guarantees that the removal of $L - K$ from G yields a graph and that the restriction d of g to K and Z is a graph morphism. The identification condition makes sure that G together with g and the inclusion of Z into G is a pushout of d and the inclusion of K into L. Altogether, a direct derivation is given by a double pushout (cf. Figure 3).

$$\begin{tikzcd}
L \arrow[r, phantom, "\supseteq"] \arrow[d, "g"] & K \arrow[r, phantom, "\subseteq"] \arrow[d, "d"] & R \arrow[d, "h"] \\
G \arrow[r, phantom, "\supseteq"] & Z \arrow[r, phantom, "\subseteq"] & H
\end{tikzcd}$$

Fig. 3. Diagram of a double pushout

The sequential composition of direct derivations $d = G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ ($n \in \mathbb{N}$) is called a *derivation* from G_0 to G_n. As usual, the derivation from G_0 to G_n can also be denoted by $G_0 \overset{n}{\underset{P}{\Longrightarrow}} G_n$ where $\{r_1, \ldots, r_n\} \subseteq P$, or just by $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n$. The subscript P may be omitted if it is clear from the context. The string $r_1 \cdots r_n$ is the *application sequence* of the derivation d.

Example 2. Consider the following two rules



The rule *start* describes the removal of an unlabeled loop and the addition of a *begin*-loop and an *end*-loop at the same node, which is depicted by ◎. The rule *run* replaces an unlabeled edge by a *p*-edge removing the two loops of the left-hand side and adding an *end*-loop at the target node of the right-hand side.

Figure 4 shows all derivations that start in the graph G_0 and apply the rule *start* only once in the beginning (while the rule *run* is applied repeatedly afterwards). At first, the rule *start* can be applied to G_0 in four ways deriving the four graphs in the second column from the left of Figure 4. The graphs in the right-most column of Figure 4 are H_1, H_2, H_3, and H_4 representing the Hamiltonian paths of G_0. They are characterized by the property that they do not contain any unlabeled loop.

It is not dificult to prove that the Hamiltonian paths of every unlabeled graph (with a single loop at each node) can be generated in the same way: Apply the rule *start* once and then the rule *run* repeatedly. A derived graph is Hamiltonian if and only if it has no unlabeled loop left.

Given a finite set of rules and a graph G, the number of matches is bounded by a polynomial in the size of G because the sizes of left-hand sides of rules are bounded by a constant. Given a match, the check, whether the contact and the identification condition hold, and the construction of the directly derived graph

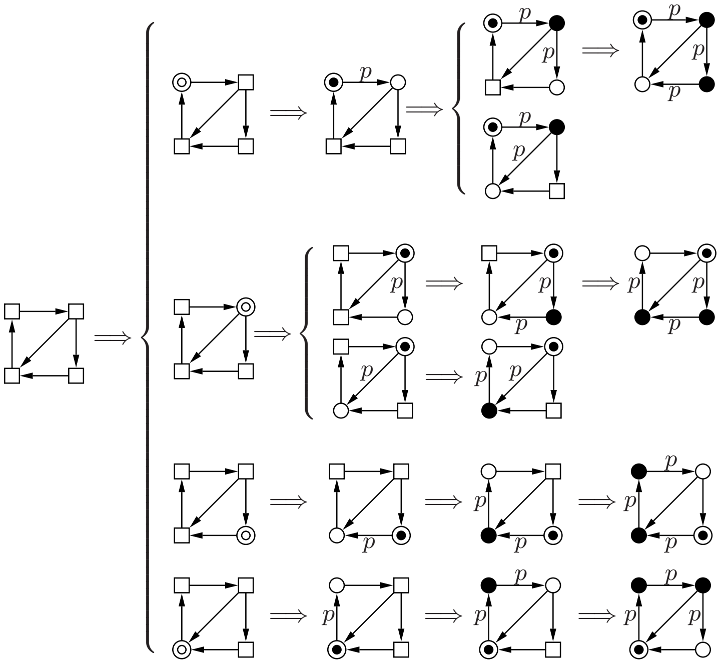


Fig. 4. The derivations starting in G_0 with one application of *start*

is linear in the size of G. Therefore, it needs polynomial time to find a match and to construct a direct derivation, and there is a polynomial number of choices at most. Moreover, the size of the resulting graph differs from the size of the host graph by a constant.

3 Simple Graph Transformation Units

A rule yields a binary relation on graphs and a set of rules a set of derivations. The example of Hamiltonian paths shows (like many other examples would show) that more features are needed to model algorithms and processes on graphs in a proper way. In particular one needs initial graphs to start the derivation process, terminal graphs to stop it, and some control conditions to regulate it. Initial and terminal graphs may be specified by graph class expressions. The notion of simple graph transformation units encompasses all these features to model and compute relations between initial and terminal graphs by means of regulated derivations.

3.1 Graph Class Expressions

A *graph class expression* may be any syntactic entity X that specifies a class of graphs $SEM(X) \subseteq \mathcal{G}_\Sigma$. A typical example is a subset $\Delta \subseteq \Sigma$ with $SEM(\Delta) = \mathcal{G}_\Delta \subseteq \mathcal{G}_\Sigma$. Forbidden and reduced structures are also frequently used. Let F be a graph, then $SEM(forbidden(F))$ contains all graphs G such that there is no graph morphism $f\colon F \to G$. Another useful type of graph class expressions is given by sets of rules P where $SEM(reduced(P))$ contains all P-*reduced* graphs, i.e., graphs to which none of the rules in P can be applied. In the examples, we use the constant expression *unlabeled graphs* denoting the set of unlabeled graphs each node of which is equipped with a single unlabeled loop.

3.2 Control Conditions

A *control condition* is any syntactic entity that cuts down the non-determinism of the derivation process. A typical example is a regular expression over a set of rules (or any other string-language-defining device). Let C be a regular expression specifying the language $L(C)$. Then a derivation with application sequence s is *permitted* by C if $s \in L(C)$.

3.3 Simple Graph Transformation Units

A *simple graph transformation unit* is a system $tu = (I, P, C, T)$, 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.

Each such transformation unit tu specifies a binary relation $SEM(tu) \subseteq SEM(I) \times SEM(T)$ that contains a pair (G, H) of graphs if and only if there is a derivation $G \overset{*}{\underset{P}{\Longrightarrow}} H$ permitted by C.

Example 3. The considerations in Examples 1 and 2 can be summarized by the following simple graph transformation unit:

HP
initial: *unlabeled graphs*
rules: *start, run*
control: *start ; run**
terminal: *forbidden*(□)

The initial graphs are unlabeled graphs with a single unlabeled loop at each node. The rules *start* and *run* are given in Example 2, and the control condition is a regular expression over the set of rules with the sequential composition ; and the Kleene star * (specifying that a single application of *start* can be followed by an arbitrary sequence of applications of *run*). Graphs derived in this way from initial graphs are accepted as terminal if they do not contain any unlabeled loop.

3.4 Computation and Complexity

Using the effective construction of direct derivations, the relation $SEM(tu)$ of a transformation unit $tu = (I, P, C, T)$ is recursively enumerable if $SEM(I)$ is recursively enumerable and $SEM(T)$ and the control condition are decidable. $SEM(tu)$ can be computed as follows:

- Enumerate the graphs of $SEM(I)$.
- For each $G \in SEM(I)$, enumerate all derivations starting in G together with their application sequences.
- For each derived graph $\overline{G}$, check whether $\overline{G} \in SEM(T)$.
- If yes, check whether the respective application sequence belongs to $L(C)$.
- If yes, put $(G, \overline{G})$ into $SEM(tu)$.

The assumptions apply to all graph class expressions and control conditions that are explicitly introduced above.

The time to check whether a graph G belongs to $SEM(\mathit{unlabeled\ graphs})$, $SEM(forbidden(F))$ or $SEM(\mathit{reduced}(P))$ is polynomial in the size of G. If $G \overset{k}{\Rightarrow} H$ and k is bounded by a polynomial in the size of G, then the size of H is also bounded by a polynomial in the size of G. Therefore, to check whether H belongs to $SEM(\mathit{forbidden}(F))$ or $SEM(\mathit{reduced}(P))$ takes also time that is polynomial in the size of G. Finally, the construction of the application sequence can be done together with the derivation without extra effort and its length coincides with the length of the derivation. The membership problem of regular expressions is linear in this length so that it is polynomial in the size of G.

The notion of a transformation unit has been introduced in [7,8,9] as a modeling and structuring concept for graph transformation systems. Here the structuring component is omitted and the computational aspect is emphasized. In addition to the cited papers, one can find more about graph class expressions and control conditions in [10,11]. Habel and Plump [12] have recently shown that a similar kind of graph transformation approach is computationally complete.

4 Solving Decision Problems

A simple graph transformation unit is terminating if, for every initial graph, the number of derivations starting in this graph is finite. In this case, all these derivations can be constructed effectively, and it can be checked whether any of them is permitted by the control condition and derives a terminal graph. This means that a terminating unit can be re-interpreted as a solution of a decision problem on the initial graphs. If the lengths of derivations are bounded by a polynom in addition, one gets a graph-transformational variant of the class *NP* of decision problems with nondeterministic polynomial solutions.

Definition 1. Let $tu = (I, P, C, T)$ be a transformation unit. tu is *terminating* if, for each initial graph $G \in SEM(I)$, there is an upper bound $b(G) \in \mathbb{N}$ such that $n \leq b(G)$ for each derivation $G \underset{P}{\overset{n}{\Rightarrow}} G'$. The function $b\colon SEM(I) \to \mathbb{N}$ given in this way is called *termination bound*.

A well-known sufficient condition for termination can be used in the framework of graph transformation units.

Observation 2. Let $tu = (I, P, C, T)$ be a transformation unit. Let $val\colon \mathcal{G}_\Sigma \to \mathbb{N}$ be a valuation function with $val(G') > val(G'')$ for each direct derivation $G' \underset{P}{\Rightarrow} G''$. Then tu is terminating.

Definition 3. Let $tu = (I, P, C, T)$ be a terminating transformation unit with the termination bound $b\colon SEM(I) \to \mathbb{N}$.

1. A function $D\colon SEM(I) \to \{\text{TRUE}, \text{FALSE}\}$ is called a *decision problem.*
2. *tu solves D* if the following holds for all $G \in SEM(I)$:
$$D(G) = \text{TRUE if and only if } (G, \overline{G}) \in SEM(tu) \text{ for some } \overline{G} \in SEM(T).$$
This is denoted by $COMP(tu) = D$.
3. tu is called *polynomial* if there is a polynom p such that, for all $G \in SEM(I)$, $b(G) \leq p(size(G))$.
4. The class of all decision problems that are solved by polynomial transformation units is denoted by NP_{GT}.

Remarks. 1. If tu is terminating, there is only a finite number of derivations $G \overset{*}{\underset{P}{\Rightarrow}} G'$ for each $G \in SEM(I)$. Hence, it can be checked effectively whether a terminal graph is derived by a permitted derivation or not.
2. The computational framework given by terminating and polynomial transformation units in particular is still nondeterministic because there may be a derivation $G \overset{*}{\underset{P}{\Rightarrow}} G'$ with $G' \in SEM(reduced(P))$, but $G' \notin SEM(T)$, and also a permitted derivation $G \overset{*}{\underset{P}{\Rightarrow}} \overline{G}$ with $\overline{G} \in SEM(T)$. In the polynomial case, it takes polynomial time to build up a single derivation and to check whether its derived graph is terminal or not (cf. [3,4]). Both points together justify the denotation NP_{GT}. The same reasoning shows that a decision problem $D\colon SEM(I) \to \{\text{TRUE}, \text{FALSE}\}$ which is solved by a polynomial transformation unit belongs to the class of NP-problems if one chooses a proper string representation of graphs. Also the converse inclusion holds because one can simulate the computational steps of a Turing machine by the application of graph transformation rules. This consideration yields the following result.

Observation 4. $NP_{GT} = NP$.

Example 4. The rules *start* and *run* (cf. Example 2) decrease the number of unlabeled loops by 1 whenever one of them is applied. Therefore, the unit *HP* (cf. Example 3) is terminating due to Observation 2. Because *HP* finds all existing Hamiltonian paths of every initial graph as terminal graphs, *HP* solves the Hamiltonian path problem. Moreover the termination bound is linear so that the problem is explicitly shown to be a member of NP_{GT}.

Termination has been studied in the context of graph transformation for example by Plump [13], Godard, Métivier, Mosbah, and Sellami [14], and by Ehrig, Ehrig, de Lara, Taentzer, Varró, and Varró-Gyapay [15].

5 Graph Multiset Transformation

In this section, graph multiset transformation is introduced employing ordinary graph transformation as basis. The underlying data structures are finite multisets of graphs. In each derivation step, some of the graphs of a given actual multiset are directly derived into graphs by applying ordinary rules, yielding a new actual multiset where the deriving graphs are replaced by the derived ones. This idea is formalized in Definition 5. A derivation of multisets of graphs corresponds to a set of simultaneous derivations of graphs (cf. Observation 6). In this sense, graph multiset transformation is a framework for massively parallel computation. In particular, one can show that *NP*-problems can be solved by graph multiset transformation in a polynomial number of steps.[2]

Definition 5. Let P be a set of rules. Let $M\colon \mathcal{G}_\Sigma \to \mathbb{N}$ be a finite multiset of graphs and $M' \leq M$ a sub-multiset of M. Let $G_1 \cdots G_n \in \mathit{Perm}(M')$ be one of the sequential representations of M' and $G'_1 \cdots G'_n \in \mathcal{G}^*_\Sigma$ be another sequence of graphs with $G_i \underset{P}{\Rightarrow} G'_i$ for all $i = 1, \ldots, n$. Let $M'' = [G'_1 \cdots G'_n]$ be the multiset of $G'_1 \cdots G'_n$.

Then M *directly derives* the graph multiset $\overline{M} = M - M' + M''$, denoted by $M \underset{P}{\Rightarrow} \overline{M}$.

A sequence $M_0 \underset{P}{\Rightarrow} M_1 \underset{P}{\Rightarrow} \cdots \underset{P}{\Rightarrow} M_n$ of direct derivations of multisets of graphs defines a *(graph multiset) derivation* from $M = M_0$ to $\overline{M} = M_n$ of length n in the usual way. Such derivations are shortly denoted by $M \overset{n}{\underset{P}{\Rightarrow}} \overline{M}$ or $M \overset{*}{\underset{P}{\Rightarrow}} \overline{M}$. The subscript P may be omitted if it is clear from the context.

Remark. It should be noted that the derived multiset does not depend on the choice of the sequential representation of M' because each permutation of the sequence $G_1 \cdots G_n$ corresponds to the respective permutation of $G'_1 \cdots G'_n$ and the multisets of sequences are invariant with respect to permutation.

It is easy to see that graph multiset derivations correspond to derivations of the graphs in the multisets and that the lengths of graph multiset derivations are bounded if and only if the lengths of graph derivations are bounded.

Observation 6. 1. Let $M \overset{k}{\underset{P}{\Rightarrow}} \overline{M}$ be a graph multiset derivation of length k and $G_1 \cdots G_n \in \mathit{Perm}(M)$ a sequential representation of M. Then there is a sequential representation $\overline{G_1} \cdots \overline{G_n} \in \mathit{Perm}(\overline{M})$ such that $G_i \overset{k_i}{\underset{P}{\Rightarrow}} \overline{G_i}$ for all $i = 1, \ldots, n$ with $k_i \leq k$.

2. Let $G_1 \cdots G_n, \overline{G_1} \cdots \overline{G_n} \in \mathcal{G}^*_\Sigma$ be sequences of graphs with $G_i \overset{k_i}{\underset{P}{\Rightarrow}} \overline{G_i}$ for all $i = 1, \ldots, n$. Then there is a graph multiset derivation $[G_1 \cdots G_n] \overset{k}{\underset{P}{\Rightarrow}} [\overline{G_1} \cdots \overline{G_n}]$ with $k = max\{k_i \mid i = 1, \ldots, n\}$.

Observation 6 means, in particular, that graph multiset transformation is a kind of parallel graph transformation that has the same termination properties as ordinary graph transformation discussed above. Therefore, graph multiset transformation can be used as a computational framework similarly to graph transformation. In particular, a terminating transformation unit can solve a decision

[2] The definitions concerning multisets are given in the Appendix.

problem on its initial graphs by means of graph multiset transformation. The computation starts with multiple copies of an initial graph and yields TRUE if some terminal graph occurs in one of the derived multisets. Using polynomial transformation units, the lengths of graph multiset derivations are polynomially bounded and TRUE is computed in a single derivation with high probability if the multiplicity of the initial graph is chosen large enough.

Definition 7. Let $tu = (I, P, C, T)$ be a terminating transformation unit. Let $D\colon SEM(I) \to \{\text{TRUE}, \text{FALSE}\}$ be a decision problem. Then *tu computes D by graph multiset transformation* (GMST) if the following holds.

For each $G \in SEM(I)$, there is a graph multiset derivation $n \cdot [G] \overset{*}{\underset{P}{\Rightarrow}} \overline{M}$ for some $n \in \mathbb{N}$ so that some underlying derivation $G \overset{*}{\underset{P}{\Rightarrow}} \overline{G}$ is permitted by C with $\overline{G} \in car(\overline{M}) \cap SEM(T)$ if and only if $D(G) = \text{TRUE}$.

Remarks. 1. If *tu* computes D by graph multiset transformation, then this may be denoted by $D = COMP_{GMST}(tu)$.

2. P_{GMST} denotes the set of all decision problems that are computed via graph multiset transformation by polynomial transformation units.
3. If *tu* is polynomial and G an initial graph, then the number of derivations starting in G is bounded by a number exponential in the size of G. If the multiplicity n of G is chosen larger than this bound and the derivation $n \cdot [G] \overset{*}{\underset{P}{\Rightarrow}} \overline{M}$ is long, then the probability is high that most permitted derivations starting in G are underlying $n \cdot [G] \overset{*}{\underset{P}{\Rightarrow}} \overline{M}$. Therefore the probability is high to find the proper value of $D(G)$ in a single graph multiset derivation with a polynomial number of steps. This justifies the denotation P_{GMST}.

As a first result on polynomial graph multiset transformation and as the main result of this section, one can show that the classes NP_{GT} and P_{GMST} coincide. Unfortunately, this is not a solution of the P=NP-problem because the class P_{GMST} relies on massive parallelism.

Theorem 8. $NP_{GT} = P_{GMST}$.

Example 5. Based on the unit *HP* in Example 3, Figure 5 shows a graph multiset derivation that starts with two copies of G_0. In the first step, the rule *start* is applied to the left upper node of both copies. There is only one possible match in each case exept for the third step where *run* is applied to the right vertical edge in the upper graph and to the diagonal edge in the lower graph. In the following steps, *run* is applied as long as possible. The horizontal rows of graphs represent the underlying derivations which are both permitted. The derived (multi-)set contains two graphs of which one graph is terminal proving that $COMP_{GMST}(HP)(G_0) = \text{TRUE}$.

The section is closed by a more explicit construction of the computations that solve decision problems by graph multiset transformation. To keep track of underlying derivations that are permitted by the control condition, a finite automaton is used.

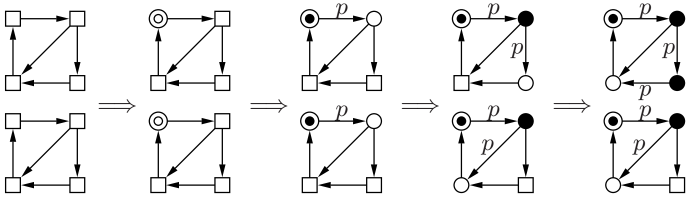


Fig. 5. A graph multiset derivation

Moreover, we assume that terminal graphs are reduced. Therefore, the check for terminality can be postponed until a derivation cannot be prolonged.

Construction 9. Let $tu = (I, P, C, T)$ be a terminating transformation unit with $SEM(T) \subseteq reduced(P)$. Let $A = (S, P, d, s_0, F)$ be a finite automaton with $L(A) = L(C)$ where P is the input alphabet.

As underlying data structures, configurations of the form $(M \overset{*}{\Rightarrow} \overline{M}, W, w)$ with $W \in Perm(\overline{M})$ and $w \in S^*$ are used. In addition, one may assume $length(W) = length(w)$ so that each copy of each graph in $\overline{M}$ is associated with a state of A. Given an initial graph G, a computation can be constructed inductively in the following way.

Induction base: Choose n, and consider $(n \cdot [G] \overset{0}{\Rightarrow} n \cdot [G], G^n, s_0^n)$ as start configuration.

Induction hypothesis: Assume that a configuration

$$(n \cdot [G] \overset{k}{\Rightarrow} \hat{M}, \hat{G}_1 \dots \hat{G}_n, s_1 \dots s_n)$$

is already constructed so that the following holds for $i = 1, \dots, n$:

$$s_i \in d^*(s_0, u_i)$$

where u_i is the application sequence of the underlying derivation $G \overset{k_i}{\Rightarrow} \hat{G}_i$.

Induction step: If possible, then choose for $i = 1, \dots, n$, $\hat{G}_i \underset{r}{\Rightarrow} \overline{G}_i$ with some $\overline{s}_i \in d(s_i, r)$. Otherwise, let $\overline{G}_i = \hat{G}_i$ and $\overline{s}_i = s_i$. Then $[\hat{G}_1 \dots \hat{G}_n] \Rightarrow [\overline{G}_i \dots \overline{G}_n]$ is a direct derivation giving rise to the follow-up configuration

$$(n \cdot [G] \overset{k}{\Rightarrow} \hat{M} \Rightarrow [\overline{G}_1 \dots \overline{G}_n], \overline{G}_1 \dots \overline{G}_n, \overline{s}_1 \dots \overline{s}_n).$$

The construction can be terminated if a configuration

$$(n \cdot [G] \overset{l}{\Rightarrow} \overline{M}, \overline{G}_i \dots \overline{G}_n, \overline{s}_1 \dots \overline{s}_n)$$

is reached such that there is no $\overline{G}_i \underset{r}{\Rightarrow} \overline{\overline{G}}_i$. Consequently, all follow-up configurations remain unchanged. Such a configuration is reached eventually because the transformation unit tu is terminating.

Observation 10. Let $tu = (I, P, C, T)$ be a terminating transformation unit with $SEM(T) \subseteq reduced(P)$. Let $A = (S, P, d, s_0, F)$ be a finite automaton with $L(A) = L(C)$. Let $D = COMP_{GMST}(tu)$. Then the following statements hold.

1. Let $(n \cdot [G] \overset{k}{\Rightarrow} \overline{M}, \overline{G}_1 \ldots \overline{G}_n, s_i \ldots s_n)$ be a configuration constructed above. Let, for $i = 1, \ldots, n, u_i$ be the application sequence of the underlying derivation $G \overset{k_i}{\Rightarrow} \overline{G}_i$. Then $s_i \in d^*(s_0, u_i)$.
2. Let $n \cdot [G] \overset{*}{\Rightarrow} \overline{M}, \overline{G}_1 \ldots \overline{G}_n, s_i \ldots s_n)$ be a terminated configuration for some $G \in \mathit{SEM}(I)$ with $\overline{G}_i \in \mathit{SEM}(T)$ and $s_i \in F$ for some $i = 1, \ldots, n$. Then $D(G) = \text{TRUE}$.
3. If $D(G) = \text{TRUE}$, then there is a terminated configuration of the form $(1 \cdot [G] \overset{*}{\Rightarrow} (1 \cdot [\overline{G}], \overline{G}, s)$ for some $\overline{G} \in \mathcal{G}_\Sigma$ and $s \in F$.

6 Exhaustive Computations

A polynomial graph transformation unit $tu = (I, P, C, T)$ solves a decision problem by means of graph multiset transformation in a polynomial number of steps with a high probability if the multiplicity of the initial graph is large. It does not provide an exact solution because there is no guarantee that a permitted derivation $G \overset{*}{\Rightarrow} \overline{G}$ with $G \in \mathit{SEM}(I)$ and $\overline{G} \in \mathit{SEM}(T)$ belongs to the derivations underlying a computation $M_G \overset{*}{\Rightarrow} \overline{M}$. This may be seen as a drawback. But the problem can be resolved by means of exhaustive computations that cover all derivations and all their prefixes up to a given length.

Definition 11. A computation $n \cdot [G] \overset{k}{\Rightarrow} \overline{M}$ for some $n \in \mathbb{N}$ is *exhaustive* if each derivation $G \overset{l}{\Rightarrow} \hat{G}$ with $l \leq k$ is an initial section of an underlying derivation, meaning that there is a derivation $\hat{G} \overset{*}{\Rightarrow} \overline{G}$ with $G \overset{l}{\Rightarrow} \hat{G} \overset{*}{\Rightarrow} \overline{G} \in \mathit{der}(n \cdot [G] \overset{*}{\Rightarrow} \overline{M})$.

Theorem 12. Let $tu = (I, P, C, T)$ be a transformation unit with $\mathit{SEM}(T) \subseteq \mathit{reduced}(P)$. Let $n \cdot [G] \overset{*}{\Rightarrow} \overline{M}$ for some $n \in \mathbb{N}$ be exhaustive with $\mathit{car}(\overline{M}) \subseteq \mathit{reduced}(P)$. Let $G \overset{*}{\Rightarrow} \overline{G}$ with $\overline{G} \in \mathit{SEM}(T)$ be permitted by C. Then $G \overset{*}{\Rightarrow} \overline{G} \in \mathit{der}(n \cdot [G] \overset{*}{\Rightarrow} \overline{M})$.

Remark. Exhaustive computations can be constructed inductively.

Induction base: $[G] \overset{0}{\Rightarrow} [G]$ is an exhaustive computation of length 0.

Induction step: Let $n \cdot [G] \overset{k}{\Rightarrow} \overline{M}$ be an exhaustive computation of length k which exists by induction hypothesis. Let max be the maximum number of direct derivations starting in some $\overline{G} \in \mathit{car}(\overline{M})$. Let $\mathit{max} \cdot n \cdot [G] \overset{k}{\Rightarrow} \mathit{max} \cdot \overline{M}$ be obtained from $n \cdot [G] \overset{k}{\Rightarrow} \overline{M}$ by copying every rule application max times. Then there are max copies of $\overline{G}$ in $\mathit{max} \cdot \overline{M}$ for each $\overline{G} \in \mathit{car}(\overline{M})$ so that all direct derivations starting in $\overline{G}$ can be constructed. This defines an exhaustive computation $\mathit{max} \cdot n \cdot [G] \overset{k}{\Rightarrow} \mathit{max} \cdot \overline{M} \Rightarrow \hat{M}$ of length $k + 1$.

Example 6. Figure 4 in Example 2 represents the full derivation process starting in G_0 that obeys the control condition $\mathit{start}; \mathit{run}^*$. It can be considered as an exhaustive graph multiset derivation where the columns are the multisets and each column from right to left is filled with enough copies of the present graphs that all alternative rule applications can be applied simultaneously. Altogether, six copies of G_0 are needed to derive all reduced graphs. The derivation length is bounded by the number of nodes of the initial graph, and the reduced graphs contain a terminal graph if and only if the initial graph contains a Hamiltonian path. In other words, the exhaustive derivations of maximum lengths solve the Hamiltonian path problem in a linear number of steps.

7 Conclusion

In this paper, we have proposed graph multiset transformation as a novel framework for the modeling and computation of graph algorithms and decision problems on graphs in particular. The basic idea is to apply rules to various graphs in a multiset simultaneously in a single computational step. In particular, *NP*-problems can be solved polynomially by graph multiset transformation employing exhaustive derivations. A result like this is typical for and should be expected of a computational model with massive parallelism.

We are convinced that future investigations will prove the significance of this approach.

1. Graph multiset transformation may be compared with other types of parallelism within and beyond graph transformation.
2. Graph multiset transformation may be used like genetic algorithms as a heuristic approach. This would mean to start with a comparatively small multiplicity of initial graphs and to employ more sophisticated control conditions to improve the chances of successful computations.
3. The example of the Hamiltonian path problem indicates that simple graph transformation units and their evaluation by graph multiset transformation provides a quite natural way to model graph problems and their solutions. Further case studies can strengthen this view.
4. As pointed out in the Introduction, graph multiset transformation is inspired by Adleman's experiment, in which he solved the Hamiltonian path problem by means of DNA computing in the proper sense using DNA molecules and their reaction with each other. Similarly, it may be possible to translate graph multiset transformation into DNA computing and implement it by a massively parallel machinery in this way.
5. Because of the close relation to genetic algorithms and DNA computing, graph multiset transformation is potentially applicable wherever these both are useful.

Acknowledgement. We are grateful to the anonymous referees for their valuable comments.

References

1. Holland, J.M.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
2. Goldberg, D.E.: The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Addison-Wesley, Reading (2002)

3. Fogel, D.B.: Evolutionary Computation: Toward a New Philosophy of Machine Intelligence, 3rd edn. IEEE Press, Piscataway (2006)
4. Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science 266, 1021–1024 (1994)
5. Păun, G., Rozenberg, G., Salomaa, A.: DNA Computing — New Computing Paradigms. Springer, Heidelberg (1998)
6. Kreowski, H.J.: A sight-seeing tour of the computational landscape of graph transformation. In: Brauer, W., Ehrig, H., Karhumäki, J., Salomaa, A. (eds.) Formal and Natural Computing. LNCS, vol. 2300, pp. 119–137. Springer, Heidelberg (2002)
7. Kreowski, H.J., Kuske, S., Schürr, A.: Nested graph transformation units. International Journal on Software Engineering and Knowledge Engineering 7(4), 479–502 (1997)
8. Kreowski, H.J., Kuske, S.: Graph transformation units and modules. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools, vol. 2, pp. 607–638. World Scientific, Singapore (1999)
9. Kreowski, H.J., Kuske, S.: Graph transformation units with interleaving semantics. Formal Aspects of Computing 11(6), 690–723 (1999)
10. Kuske, S.: More about control conditions for transformation units. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 323–337. Springer, Heidelberg (2000)
11. Kuske, S.: Transformation Units—A Structuring Principle for Graph Transformation Systems. PhD thesis, University of Bremen (2000)
12. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 230–245. Springer, Heidelberg (2001)
13. Plump, D.: Termination of graph rewriting is undecidable. Fundamenta Informaticae 33(2), 201–209 (1998)
14. Godard, E., Métivier, Y., Mosbah, M., Sellami, A.: Termination detection of distributed algorithms by graph relabelling systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 106–119. Springer, Heidelberg (2002)
15. Ehrig, H., Ehrig, K., Taentzer, G., de Lara, J., Varró, D., Varró-Gyapai, S.: Termination criteria for model transformation. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 49–63. Springer, Heidelberg (2005)

Appendix

This appendix recalls the notions and notations of multisets used in the paper.

1. Let X be a set. Then a multiset (over X) is a mapping $M\colon X \to \mathbb{N}$, where $M(x)$ is the *multiplicity* of x in M.
2. The *carrier* of M contains all elements of X with positive multiplicity, i.e.
$$car(M) = \{x \in X \mid M(x) > 0\}.$$
3. A multiset is *finite* if its carrier is a finite set.
4. Let M and M' be multisets. Then M' is a *sub-multiset* of M, denoted by $M' \leq M$, if $M'(x) \leq M(x)$ for all $x \in X$.

5. Let M and M' be multisets. Then the *sum* (*difference*) of M and M' is the multiset defined by

$$(M \pm M')(x) = M(x) \pm M'(x) \text{ for all } x \in X.$$

   Here + and − are the usual sum and difference of non-negative integers with $m - n = 0$ if $m \leq n$ in particular.
6. Using the sum of multisets, the multiplication of multisets with non-negative numbers can be defined inductively for all multisets M by
   (i) $0 \cdot M = \mathbf{0}$ and
   (ii) $(k+1) \cdot M = k \cdot M + M$ for all $k \in \mathbb{N}$

   where the multiset $\mathbf{0}$ is the multiset with the constant multiplicity 0, i.e. $\mathbf{0}(x) = 0$ for all $x \in X$.
7. Each sequence $w \in X^*$ induces a multiset $[w]$ by counting the number of occurrences of each x in w, i.e., for all $x, y \in X$ and $w \in X^*$,
   - $[\lambda](x) = 0$
   - $[yw](x) =$ *if* $x = y$ *then* $[w](x) + 1$ *else* $[w](x)$.
8. Let M be a multiset. Then the set of all sequences w with $[w] = M$ is denoted by $Perm(M)$. An element of $Perm(M)$ is called a *sequential representation* of M. Note that $Perm(M)$ contains all permutations of w if $[w] = M$.
9. The set of multisets over X as well as the set of finite multisets over X give rise to a commutative monoid with the multiset $\mathbf{0}$ as null and the sum as inner composition. Moreover, the set of finite sultisets over X is generated by the sigletons $[x]$ for all $x \in X$ so that the finite multisets are characterized as the free commutative monoid over X.

Pullback Grammars Are Context-Free

Michel Bauderon, Rui Chen, and Olivier Ly

Université de Bordeaux - CNRS
LaBRI - UMR 5800, 351 cours de la Libération Talence 33405 France
Zhongnan University of Economics and Law,
1 Nanhunan Road, Hongshan Wuhan 430073 China
{bauderon,ly}@labri.fr, rui.chen.bordeaux1@gmail.com
http://www.labri.fr/
http://www.znufe.edu.cn/eng/

Abstract. Following earlier work on pullback rewriting, we describe here the notion of graph grammar relevant to our formalism. We then show that pullback grammars are context-free and provide a surprising example, namely the context-free generation of square grids.

Keywords: Pullback, Graph grammar.

1 Introduction

Graph rewriting has been studied at large over the last forty years, giving rise to several distinct formalisms and to a lot of results.

To be short, one may say that two main directions have been explored corresponding to two distinct (somehow dual) approaches to the structure of a graph, either as nodes linked by arrows (vertex rewriting) or as arrows glued by nodes (edge or hyperedge rewriting).

In both directions, three levels of description have been explored: set theoretic, algebraic (namely using universal algebra) or categorical (using category theory as basic tool).

In this last setting, namely using category theory, the main effort has been devoted to edge (and hyperedge) replacement, using pushout as a basic operation and leading to the development of a large theoretical body, via the double and single pushout approach to graph rewriting (the so-called algebraic approach) and their extensions.

In this paper, we focus on the dual approach (in the category theoretical sense!), using pullback as the basic rewriting operation, as introduced by the first author in order to provide vertex replacement with a categorical treatment (an "algebraic approach").

Several papers have been published so far (see [1,2,3]), which describe the basic formalism, and show e.g. how to encode standard node rewriting systems such as NLC or NCE systems by pullback rewriting.

Let us summarize the main peculiarities of this approach (and refer the reader to earlier papers for a more detailed description and further "intuition"):

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 366–378, 2008.

- Node labels describe not only properties of the nodes, but also their relationships. In other words, the alphabet is not just a set as usual, but a graph; and the labeling is not a mere mapping, it is a family of graph morphisms.
- The rewriting mechanism is not the classical notion of substitution (which is a form of union), but a notion of "controlled product", namely categorical pullback.
- Arrows are reversed and so are diagrams (when compared to the push-out approach).

Here we go one step forward, defining the notion of graph grammar (the pullback grammar) relevant to this new rewriting mechanism. We only give the basic definitions and results and limit ourselves to simple graphs. Space limitations do not allow any proof or extended discussions.

Section 2 is devoted to the basic concepts and to the definition of graph rewriting via pullback. Several notions have to be developed or introduced. First, in order to describe several distinct rules and unknowns, we need a slightly more general alphabet than in [1,2]. Then the classical notions of rules, terminal and non terminal items are described.

It must be clear that, as we try to stick to a "good" categorical approach, our interest focuses on arrows. To be more precise, our rules and unknowns are arrows i.e. graph morphisms, even though we may happen to use the words "unknown, neighbor and context" in a loose way when we try to support the intuition of the readers.

In a standard approach to vertex rewriting, the graph has a labeling, namely a mapping to a set of terminal and non terminal labels, describing whether and how a node may be rewritten. A rule associates to a non-terminal label the graph to substitute to the rewritten node ("the right hand side") together with a connection relation.

Our formalism is both simpler and more complex. It is more complex, since our labeling will be given via a family of morphisms which we call unknowns (let say one per non-terminal node although this is not that simple) and which completely describes what will happen to the node. The rule will also be a graph morphism and it will completely describe how the node will be transformed (including the replacement graph and the connection relation). Since the "right hand side" of a classical rule may be a labeled graph, it will also be labeled by a family of morphisms.

It is simpler, because the rewriting is "uniform": application of a rule will be given by a pullback computation whose by-product will be a labeling of the new graph. This is described in section 2.3.

Section 3 describes pullback grammars and one of their main properties: Pullback grammars are context-free in the sense defined by Courcelle in [5] (associative and confluent). And then we give the surprising example of a (context-free) pullback grammar with only one rule which generates square grids and only square grids. This clearly shows the strength of our formalism since square grids are normally taken as a paragon of non context-freeness!

2 Pullback Rewriting

2.1 Graphs

This section is devoted to basic concepts. We consider directed graphs with possibly multiple edges.

Definition 1. *A graph G is a 4-tuple $G = \langle V_G, E_G, s_G, t_G \rangle$ where V_G, E_G are two finite disjoint sets, s_G and t_G are two mappings from E_G to V_G. We call V_G the set of vertices and E_G the set of edges. For every element $e \in E_G$, $s_G(e)$ and $t_G(e)$ are called source vertex and target vertex of the edge e respectively.*

A vertex $v \in V_G$ is *reflexive* if there exists an edge $e \in E_G$ such that $s_G(e) = t_G(e) = v$. A graph G is reflexive if all its vertices are reflexive, it is said to be simple if for any pair x, y of vertices of G, there is at most one edge from x to y.

Definition 2. *Given two graphs G_1, G_2, a graph morphism f from G_1 to G_2 consists of a pair $\langle f_V, f_E \rangle$ such that:*

1. *$f_V : V_{G_1} \to V_{G_2}$ is a mapping from the vertex set of G_1 to the one of G_2;*
2. *$f_E : E_{G_1} \to E_{G_2}$ is a mapping from the edge set of G_1 to the one of G_2;*
3. *$f_V \circ s_{G_1} = s_{G_2} \circ f_E$; $f_V \circ t_{G_1} = t_{G_2} \circ f_E$.*

Having defined graphs and graph morphisms, the set of graphs can be considered within the categorical framework where objects are graphs and arrows are graph morphisms (see [7] for basics about category theory). The central categorical tool in our work is *pullback*, which is defined as follows.

Definition 3. *Given two morphisms $f : B \to A$ and $g : C \to A$, the pullback of f, g is a triple (P, p, q) where p, q are two morphisms $p : P \to B, q : P \to C$ such that:*

- *$f \circ p = g \circ q$;*
- *for every other triple $(H, m : H \to B, n : H \to C)$ such that $f \circ m = g \circ n$, there exists a unique morphism $u : H \to P$ such that $p \circ u = m$ and $q \circ u = n$.*

P is called the pullback object. The condition is illustrated by the diagram in Figure 1.

It is well known that the category of graphs has all products and pullbacks. Moreover, pullbacks of a given pair of morphisms are unique up to isomorphism. Therefore we use in the sequel *the pullback object* for the class of isomorphic objects. It is important to note (since we shall use this basic property extensively) that the category of graphs has a terminal element (the graph with one vertex and one edge, denoted by $\odot$) which is a neutral element for the categorical product.

$$\begin{tikzcd}
H \arrow[dr, "u"] \arrow[drr, bend left, "m"] \arrow[ddr, bend right, "n"'] & & \\
& P \arrow[r, "p"'] \arrow[d, "q"'] & B \arrow[d, "f"'] \\
& C \arrow[r, "g"'] & A
\end{tikzcd}$$

Fig. 1. Definition of pullback

2.2 Alphabet

Our rewriting mechanism needs to distinguish between three kinds of nodes: those which can be transformed, their neighbors and all the remaining nodes. This will of course be done through some form of labeling, e.g. some kind of mapping from the graph to a ”set of letters”. But, since we need to encode the way a node will be rewritten as well as the way relations between nodes will be affected (encoding so-called *connection relations*), we need a notion of *alphabet* which is richer than a simple set: we shall consider a graph, and the labeling (operation to distinguish a vertex from its neighbors and the context) will be done via graph morphisms to this kind of “alphabet”.

Definition 4. *An alphabet graph $A_{m,n}|_{m,n \in \mathbb{Z}_+}$ of type m, n is built by connecting a reflexive vertex to m vertices of a complete reflexive graph K_{m+n}.*

Because an alphabet graph is totally reflexive, there always exists a graph morphism from any graph to a given alphabet graph.

The graph $A_{m,n}$ has an obvious modular decomposition in three modules that we shall call C, N_m and U_n, with respectively $1, m$ and n vertices. To support one's intuition, nodes in C, N_m and U_n, will respectively be called context, neighbors and variables (or non terminals), and so will be any node mapped onto them by any graph morphism.

Example 1. The alphabet graph $A_{3,2}$ is shown in Figure 2. As we shall often do to help intuition, we have labeled the graph vertices using integers. 0 is used to name the unique vertex of C (the context), positive integers to name the vertices of N_3 (the neighbors), negative integers to name those of U_2 (the unknowns or non-terminals). We could as well have used letters, colors or shapes, but integers turn out to be quite convenient. For the sake of simplicity, edges in the alphabet graph are drawn as ↔ to represent the pairwise inverse edges between two vertices.

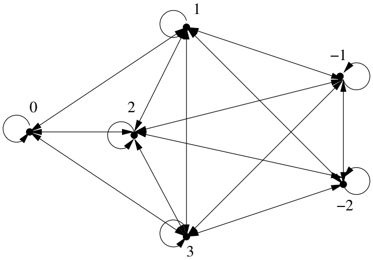


Fig. 2. Alphabet graph $A_{3,2}$

2.3 Unknowns

Definition 5. *Let G be a directed simple graph and $A_{m,n}$ an alphabet graph, an* unknown *l on G is a graph morphism from G to $A_{m,n}$ such that:*

- *there exists at least one vertex v in U_n of $A_{m,n}$ with a non empty pre-image (i.e. such that $l^{-1}(v) \neq \emptyset$)*
- *l is monomorphic on the pre-image of N_m.*

Such an unknown on G distinguishes three kinds of vertices which we shall call context, interface and non-terminal nodes. The context nodes are mapped to the singleton C in the alphabet graph, and the mapping from the interface nodes to N_m is monomorphic. According to this definition, an unknown can indicate more than one non-terminal node and these can be mapped to different nodes of U_n.

Definition 6. *A* non-terminal graph *$G^{\mathcal{L}} = (G, U)$ is a reflexive graph G together with a non empty finite set U of unknowns on G.*

The reflexivity of G is just useful to simplify further definitions. It is easily checked that this is not a restriction since self loop can easily be removed by adding specific terminal rules.

Example 2. In Figure 3 we illustrate a non-terminal graph (G, U). The directed graph G is shown in the left part of the figure. The two unknowns in U are denoted by l_1, l_2. The labels on the nodes of G are pairs consisting of one letter (the name of the node) and one integer showing their image in the alphabet graph.

2.4 Rewriting Rule

Definition 7. *A* rewriting rule *r is a morphism $r : R \to A_{m,n}$ which is isomorphic on the inverse image of the subgraph of $A_{m,n}$ generated by $C \cup N_m$. If the inverse image of U_n is empty, the rule is* erasing. *A rewriting rule r whose graph R is non-terminal is said to be* non-terminal*, otherwise it is* terminal.

We illustrate the notion of rewriting rule in the following simple example.

Example 3. Figure 4 illustrates two rewriting rules. The left hand side shows a terminal rewriting rule $\mathcal{R} = r : R \to A_{m,n}$, while the right hand side shows a non-terminal rule $\mathcal{R}^*$ by introducing an unknown l_R on the graph R. The graph morphisms r and l_R are described below.

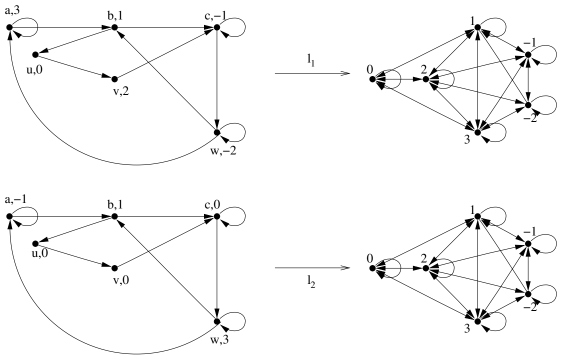
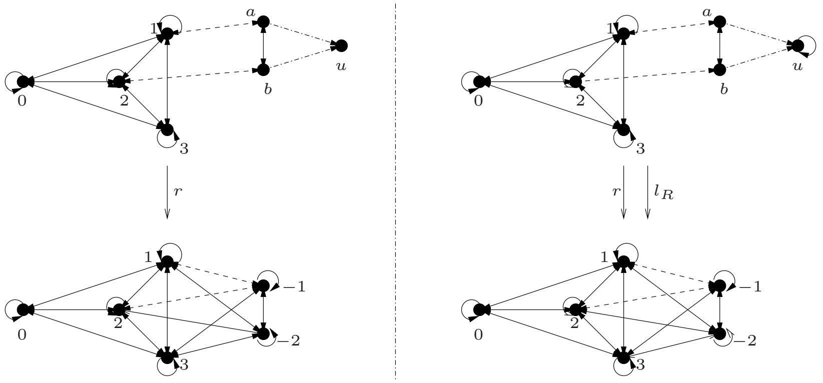


Fig. 3. A graph with two unknowns



A terminal rewriting rule $\mathcal{R}$
$r(0) = 0, r(1) = 1, r(2) = 2, r(3) = 3$
$r(a) = r(b) = r(u) = -1$

A non terminal rewriting rule $\mathcal{R}^*$
$l_R(0) = l_R(1) = l_R(2) = l_R(3) = 0$
$l_R(u) = -2, l_R(a) = 1, l_R(b) = 2$

Fig. 4. Rewriting rules

2.5 Rewriting Graphs by Pullback Computation

As already said in the introduction, the rewriting mechanism will simply be the pullback computation of a pair (l, r) where $l : G \to A_{m,n}$ is an unknown on a graph G and $r : R \to A_{m,n}$ is a rewriting rule.

This is illustrated in an informal way in Figure 5. A non-terminal graph is shown in the bottom left. $[U]$ stands for element to be rewritten. In the right column $[R]$ designates the ingredient to replace for $[U]$. By a single pullback computing step, the $[U]$'s are replaced by $[R]$s. The fact that the initial element $\odot$ is a neutral element for the categorical product plays a key role in the mechanism.

Definition 8. *Let $r : R \to A_{m,n}$ be a rewriting rule. Given a non-terminal graph (G, U) and $l \in U$, the* application *of $\mathcal{R}$ on G at unknown l is the pullback computation of (l, r).*

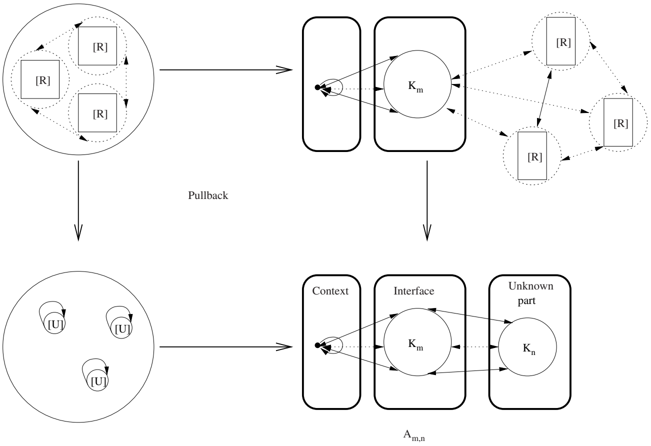


Fig. 5. Pullback rewriting schema

Intuitively the application of a rewriting rule $\mathcal{R}$ to a non-terminal graph at an unknown l consists in replacing the reflexive vertices indicated by the unknown in the original graph with the subgraph provided by the rule graph, and then connecting the added parts to the remaining of the original graph according to the way specified in the rule morphism.

Remark 1. The application of a non-terminal rewriting rule r at unknown $l \in U$ of a non-terminal graph (G, U) produces a new non-terminal graph whose unknowns are well defined. Let us illustrate it by the diagram in Figure 6. Any other unknown in U (here denoted by l_1) gives rise to an unknown $l_1 \circ p$ in the result graph G'. Similarly, for an unknown l_R introduced by the non-terminal rewriting rule, a new unknown is defined by $l_R \circ q$ in the result graph G'.

For the sake of simplicity and if no ambiguity arises, the new unknown $l_1 \circ p$ on G' will still be denoted by l_1. Therefore the rewriting sequence $G \overset{(l,r)}{\Longrightarrow} G_1 \overset{(l_1 \circ p, r_1)}{\Longrightarrow} G_2$ will be simply written as $G \overset{(l,r)}{\Longrightarrow} G_1 \overset{(l_1, r_1)}{\Longrightarrow} G_2$.

Example 4. We first show two rewriting rules r_1 and r_2 in Figure 7.

In Example 2 we have shown a non-terminal graph with two unknowns l_1, l_2 illustrated in Figure 3. By applying r_1, r_2 at unknowns l_1, l_2 respectively, we have two pullback diagrams shown in Figure 8. That is, by computing the pullback of (l_1, r_1) we rewrite c, w in the initial graph. This is illustrated in the left part of the figure. While by computing the pullback of (l_2, r_2) we rewrite a, shown in the right part of Figure 8.

In these and all following examples, a rule application will be depicted as a square diagram, where the alphabet graph will be in the bottom row and the rewriting rule in the right column. The graph computed by the pullback appears in the top left position of the diagram.

$$\begin{tikzcd}
G' \arrow[r, "q"] \arrow[d, "p"'] & R \arrow[d, shift right, "r"'] \arrow[d, shift left, "l_R"] \\
G \arrow[r, shift left, "l"] \arrow[r, shift right, "l_1"'] & A
\end{tikzcd}$$

Fig. 6. An application

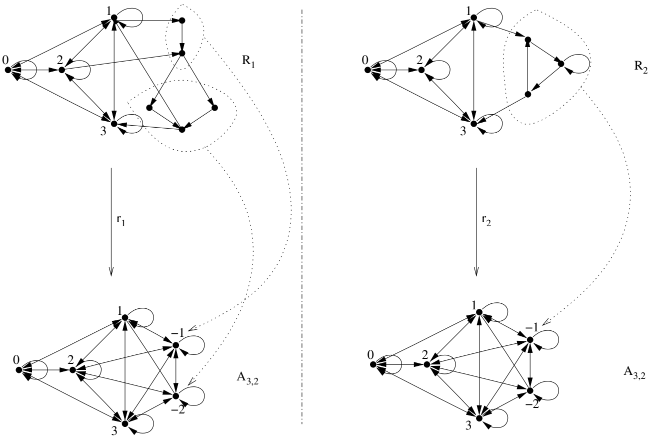


Fig. 7. Rules r_1 and r_2

3 Pullback Grammars

Classically, a formal grammar is a system made of an alphabet, a set of substitution rules, an initial element. Here is our definition of a grammar.

Definition 9. *A pullback grammar is a quadruple* $\mathrm{PGrm} = \langle A, \mathfrak{R}, G_0, \mathfrak{P}\rangle$ *where:*

- *A is the alphabet graph,*
- *$\mathfrak{R}$ is a finite set of rewriting rules. A rewriting rule $\mathcal{R} \in \mathfrak{R}$ can be terminal or non-terminal.*
- *The initial graph G_0 is a non-terminal graph, also called the start symbol.*
- *$\mathfrak{P} \subset \mathfrak{U} \times \mathfrak{R}$ defines the rule application relations where $\mathfrak{U}$ is the set of all the unknowns of G_0 and of every non-terminal rule in $\mathfrak{R}$.*

Rewriting according to the grammar is defined as follows.

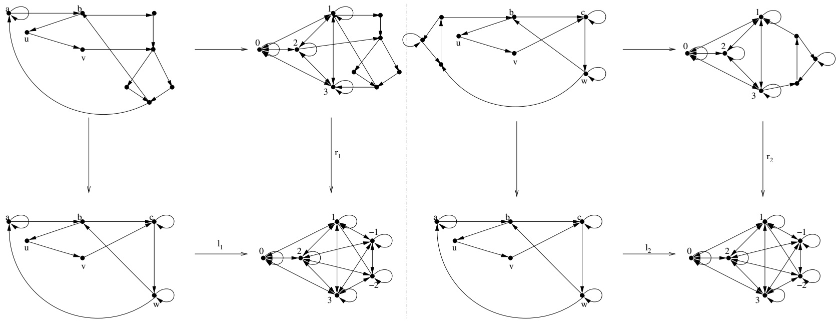


Fig. 8. Examples of two pullback rewriting steps

For any non-terminal graph produced by the grammar, each unknown is associated with a *type*, which is an element of $\mathfrak{U}$. The meaning of the types is explained below. Types are defined inductively as the following:

- for any unknown l associated with G_0 $typ(l) = l$, and for any unknown l_R introduced by a non-terminal rewriting rule r such that $r \in \Re$, $typ(l_R) = l_R$.
- A rewriting is authorized by the grammar if it has the form:

$$(G, U) \overset{(l,r)}{\Longrightarrow} (\bar{G}, \bar{U})$$

  where $l \in U$ and $(typ(l), r) \in \mathfrak{P}$. In this case, if we denote by $G \xleftarrow{\pi} \bar{G} \xrightarrow{\alpha} R$ the pullback of $G \xrightarrow{l} A \xleftarrow{r} R$, one defines the type of any unknown $\bar{l} \in \bar{U}$ of $\bar{G}$ as follows:
  - if $\bar{l} = l' \circ \pi$ for some $l' \in U$ then $typ(\bar{l}) = typ(l')$,
  - if $\bar{l} = l_r \circ \alpha$ for some unknown l_r of introduced by r then $typ(\bar{l}) = typ(l_r)$.

The pairs defined in $\mathfrak{P}$ specify which rules can be applied to the unknowns. However, unknowns appearing in such pairs are only the unknowns of the initial graph and of the non-terminal rules. Let us call these unknowns the initial unknowns. Therefore, one must also specify which rules can be applied to the unknowns of a graph produced by the grammar after several rewriting steps. This is the role of the types. Indeed, the type of such an unknown specifies the original unknown from which it is issued. And thus, together with $\mathfrak{P}$, it defines which rule can be applied to itself.

3.1 Context-Freeness

In [5], the notions of confluence and associativity for a grammar have been defined axiomatically by B. Courcelle. They are used to define in an abstract way what is the context-freeness for grammars (of words, trees or graphs).

Our main result is the following theorem.

Theorem 1. *Pullback grammars are context-free (in the sense of [5]).*

It is an immediate consequence of the following definitions and results. Detailed proofs may be found in [4].

Confluence

Definition 10. *Let G be a non-terminal graph. If for any two applications (l_1, r_1) and (l_2, r_2) where l_1, l_2 are two unknowns on G and r_1, r_2 are two rewriting rules we have*

$$\left.\begin{array}{l} G \overset{(l_1,r_1)}{\Rightarrow} G_1 \overset{(l_2,r_2)}{\Rightarrow} G_{12} \\ G \overset{(l_2,r_2)}{\Rightarrow} G_2 \overset{(l_1,r_1)}{\Rightarrow} G_{21} \end{array}\right\} \Longrightarrow G_{12} \simeq G_{21}$$

where the symbol $\simeq$ means "isomorphic", the rewriting system is said to be confluent*.*

Intuitively when we say a rewriting system is confluent it means that two successive rewriting steps concerning distinct rule applications can be done in any order, giving the same result.

Theorem 2. *Pullback grammars are confluent.*

Associativity

Definition 11. *Let G be a non-terminal graph. If for any two applications $(l, \mathcal{R}_1)$ and $(l_1, \mathcal{R}_2)$ where l is an unknown of G, $\mathcal{R}_1, \mathcal{R}_2$ are two rewriting rules and l_1 is an unknown of R_1, the sequence $G \overset{(l,\mathcal{R}_1)}{\Rightarrow} G_1 \overset{(l_1,\mathcal{R}_2)}{\Rightarrow} G_{12}$ can always be replaced by the following process*

1° $R_1 \overset{(l_1,\mathcal{R}_2)}{\Rightarrow} R_{12}$;
2° $G \overset{(l,\mathcal{R}_{12})}{\Rightarrow} G_{12}$,

then the rewriting system is said to be associative*.*

Theorem 3. *Pullback grammars are associative.*

3.2 Grid-Pattern Generating Pullback Grammar

As an example, we now describe a pullback grammar which generates square grids and only square grids. This shows that the language of square grids is context-free.

Definition 12. *A grid of n-order ($n \in \mathbb{Z}_+$) is a graph Grd_n such that:*

- $V_{Grd_n} = \{(i,j) | i,j \in [1,n] \subset \mathbb{Z}_+\}$
- *The graph is totally reflexive*
- $[(i-1,j),(i,j)] \in E_{Grd_n}$ *for* $i \in [2,n]$ *and* $j \in [1,n]$ *and* $[(i,j-1),(i,j)] \in E_{Grd_n}$ *for* $i \in [1,n]$ *and* $j \in [2,n]$.

There is no other item in the graph.
*The term n-*order *means that an n-order square grid has $n \times n$ vertices.*

From the definition a grid in fact is a reflexive simple graph.

Definition 13. *A* grid-rule $\mathcal{R}_{Grd}$ *is a non-terminal rewriting rule* $((R, l^*), r)$, *where* l^* *is the only unknown brought by the rule.* R *is a graph such that:*

- $V_R = \{(0,0)\} + V_{Grd_3}$
- $E_R = \{[(0,0),(0,0)], [(0,0),(1,1)], [(1,2),(2,1)], [(1,3),(3,1)]\} + E_{Grd_3}$

The graph morphism r *from* R *to the alphabet graph* $A_{1,3}$ *is defined as follows:*

- $r_V((0,0)) = 0$
- $r_V((1,1)) = 1$
- $r_V((1,2)) = r_V((1,3)) = -2$, $r_V((2,1)) = r_V((3,1)) = -1$, $r_V((2,2)) = r_V((2,3)) = r_V((3,3)) = r_V((3,2)) = -3$
- *The mapping of edges is consistent with the node mapping.*

The unknown $l^* : R \to A_{1,3}$ *is defined as follows:*

- $l^*{}_V((0,0)) = l^*{}_V((1,1)) = 0$
- $l^*{}_V((1,2)) = l^*{}_V((2,2)) = l^*{}_V((2,1)) = 1$
- $l^*{}_V((1,3)) = l^*{}_V((3,2)) = -1$
- $l^*{}_V((2,3)) = l^*{}_V((3,1)) = -2$
- $l^*{}_V((3,3)) = -3$
- *The mapping of edges corresponds just with the nodes mapping.*

Figure 9 visualizes the *grid-rule*. We use different shapes for nodes to illustrate the mapping relations between rule graph and the alphabet graph $A_{1,3}$, that is, the vertices in the rule graph are mapped to the vertices having the same shape. To simplify the drawing, the graphs in the illustrations are not explicitly shown as directed.

With these elements, we can now define our pullback grammar.

Definition 14. *A grid-generating pullback graph grammar* GGrid *is a triple* $\langle A_{1,3}, \{\mathcal{R}_{Grd}\}, \odot\rangle$ *where* $A_{1,3}$ *is the alphabet graph,* $\mathcal{R}_{Grd}$ *is the grid-rule, and* $\odot$ *is the initial symbol which is the terminal object in the category of graphs.*

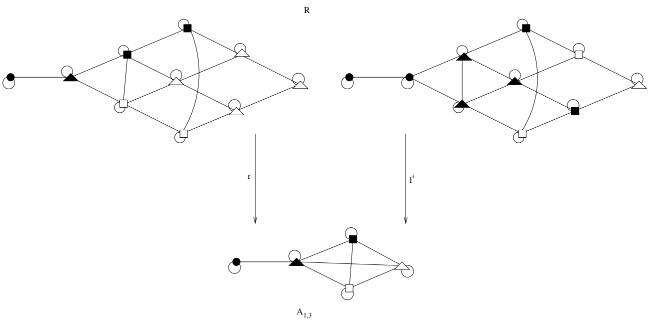


Fig. 9. Grid-generating rule

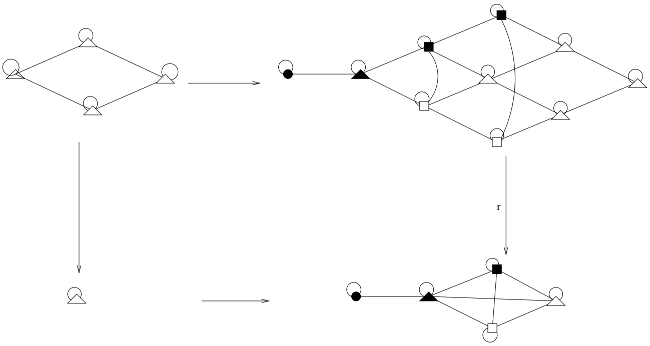


Fig. 10. $\odot \Rightarrow Grd_2$

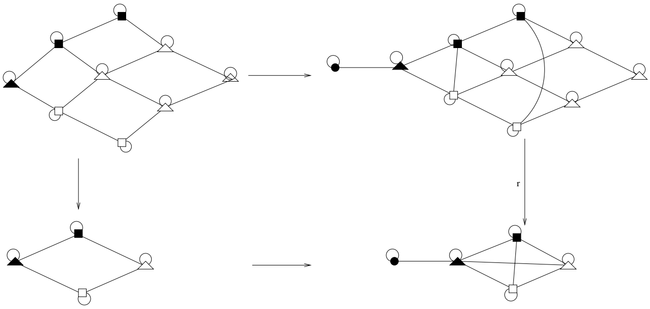


Fig. 11. $Grd_2 \Rightarrow Grd_3$

We illustrate the two first rewriting steps beginning from $\odot$ by Figures 10 and 11.

Theorem 4. GGrid *generates the set of all grids.*

We omit here the formal proof and refer the reader to [4] for details. □

4 Conclusion

This short paper recalls the basics of pullback rewriting in graphs and defines the corresponding pullback grammars.

Our main result is the fact that pullback rewriting is intrinsically confluent and associative, hence that all pullback grammars are context-free according to the definitions of [5].

As a consequence we provide an example of a context-free pullback grammar which generates the square grids and only square grids, a well known obstruction to context-freeness in classical frameworks.

Obviously, a lot remains to be done, characterizing the expressive power of pullback grammars (at least VR hence HR), finding a parsing method for pullback grammars, etc. Some other results are available in [4] and will soon be submitted for publications.

It must also be clear that a significant part of this approach can be carried to more general categories (for instance, [2] deals with node rewriting in hypergraphs) in order to define a kind of High Level Pullback systems. This will be the object of later work.

Acknowledgment

It is a pleasure for us to thank the anonymous referees whose comments have greatly helped us improve our text.

References

1. Bauderon, M.: A uniform approach to graph rewriting: the pullback approach. In: Nagl, M. (ed.) WG 1995. LNCS, vol. 1017, pp. 101–115. Springer, Heidelberg (1995)
2. Bauderon, M., Jacquet, H.: Node rewriting in graphs and hypergraphs: a categorical framework. Theoretical Computer Science 266(1-2), 463–487 (2001)
3. Bauderon, M., Jacquet, H., Klempien-Hinrichs, R.: Pullback rewriting and application. Electr. Notes Theor. Comput. Sci. 51, 83–92 (2001)
4. Chen, R.: Graph Transformation and Graph Grammar Based on Pullback Operation, PhD thesis, Université Bordeaux 1 (2007)
5. Courcelle, B.: An axiomatic approach to context-free rewriting and its application to NLC graph grammars. Theoretical Computer Science 55, 141–181 (1987)
6. Engelfriet, J., Rozenberg, G.: Node Replacement Graph Grammars. In: [8], pp. 1–94.
7. Pierce, B.C.: Basic Category Theory for Computer Scientists. MIT Press, Cambridge (1991)
8. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific Publishing, Singapore (1997)

On Switching to H-Free Graphs

Eva Jelínková[1] and Jan Kratochvíl[1,2]

[1] Department of Applied Mathematics*
[2] Institute for Theoretical Computer Science
Charles University
Malostranské nám. 25, 118 00 Praha, Czech Republic**
{eva,honza}@kam.mff.cuni.cz

Abstract. In this paper we study the problem of deciding if, for a fixed graph H, a given graph is switching-equivalent to an H-free graph. Polynomial-time algorithms are known for H having at most three vertices or isomorphic to P_4. We show that for H isomorphic to a claw, the problem is polynomial, too. Further, we give a characterization of graphs switching-equivalent to a $K_{1,2}$-free graph by ten forbidden induced subgraphs, each having five vertices. We also give the forbidden induced subgraphs for graphs switching-equivalent to a forest of bounded vertex degrees.

1 Introduction

Seidel's switching is a graph operation which makes a given vertex adjacent to precisely those vertices to which it was non-adjacent before, while keeping the rest of the graph unchanged. Two graphs are called switching-equivalent if one can be made isomorphic to the other by a sequence of switches. The class of graphs that are pairwise switching-equivalent is called a switching class.

The concept of Seidel's switching was introduced by the Dutch mathematician J. J. Seidel in connection with algebraic structures, such as systems of equiangular lines, strongly regular graphs, or the so-called two-graphs. For structural properties of two-graphs, cf. [12]–[14].

In this paper, we study the computational complexity of problems related to Seidel's switching. The problem of deciding if a given graph is switching-equivalent to a graph having a certain desired property has already been addressed by several authors.

As observed by Kratochvíl et al. [9] and also by Ehrenfeucht at al. [2], there is no correlation between the complexity of the problem and the complexity of the property itself. For example, the problems of deciding if a graph contains a Hamiltonian path or cycle are well known to be NP-complete [3]. However, Kratochvíl et al. [9] proved that any graph is switching-equivalent to a graph containing a Hamiltonian path, and it is polynomial to decide if a graph is

* Supported by project MSM0021620838 of the Czech Ministry of Education.
** Supported by grant 1M0545 of the Czech Ministry of Education.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 379–395, 2008.

switching-equivalent to a graph containing a Hamiltonian cycle. These results have been extended to graph pancyclicity by Ehrenfeucht et al. [1].

On the other hand, the problem of deciding switching-equivalence to a regular graph was proven NP-complete by Kratochvíl [11], and switching-equivalence to a k-regular graph for a fixed k is polynomial, while both the regularity and k-regularity of a graph can be tested in polynomial time. Three-colorability and switching-equivalence to a three-colorable graph are both NP-complete [2].

In this paper, we focus on hereditary properties, i.e., properties closed on induced subgraphs. In particular, we consider the property of being H-free for fixed graphs H. Given a graph G, we want to decide if G is switching-equivalent to an H-free graph.

Let K_n denote the complete graph on n vertices, let P_n be the path with n vertices and let I_n be the discrete graph with n vertices and no edges. Polynomial-time decision algorithms are known if H has at most three vertices or is isomorphic to a P_4. The algorithm for K_2 or I_2 is simple (see [5]), the one for $K_{1,2}$ or $K_2 + K_1$ is due to Kratochvíl et al. [9]. Hayward [6] and independently Hage at al. [5] found an algorithm for K_3 or I_3; the result is a core of the polynomial-time algorithm for recognizing P_3-structures of graphs. The case of P_4 has been solved by Hertz [8] in connection to perfect switching classes.

It can be observed that an algorithm for H, when run on a complement of the input graph, gives an algorithm for $\overline{H}$. But the switching-equivalence of H and H' does not yield any obvious relation of algorithms for H and H'.

All the known algorithms mentioned above are either direct or based on a reduction to 2-SAT. We use a different method—a reduction to a system of linear equations modulo 2. With this method, we prove in Section 3 that a polynomial-time algorithm exists even for $K_{1,3}$. We thus extend the solved graphs H by another graph on four vertices, which is the maximum number of vertices so far.

1.1 Characterizations by Forbidden Induced Subgraphs

Let P be a hereditary property. Then there exists a class $\mathcal{F}(P)$ of minimal forbidden induced subgraphs such that a graph G is switching-equivalent to a graph with P if and only if G does not contain any element of $\mathcal{F}(P)$ as an induced subgraph. If $\mathcal{F}(P)$ is finite, this can be tested in polynomial time.

Hence, when examining the computational complexity of recognizing switching classes with P, the characterization by $\mathcal{F}$ is valuable. However, a polynomial-time algorithm may exist even when $\mathcal{F}$ is infinite. And, in some cases, more efficient algorithms are already known.

A finite characterization by $\mathcal{F}$ is known, for example, for switching classes whose all graphs are perfect (due to Hertz [8]), and for switching classes whose all graphs contain a 1-perfect code (due to Kratochvíl [10]). Hage and Harju [4] give the class $\mathcal{F}$ for switching classes containing an acyclic graph; apart from the cycles C_n for $n \geq 7$, there are 905 graphs in $\mathcal{F}$, each having at most nine vertices. A computer program was employed to obtain this result.

In some cases, characterizations by finite $\mathcal{F}$ are known for switching-equivalence to H-free graphs. Namely, for H isomorphic to K_2 or I_2 the class $\mathcal{F}$

consists of two graphs on three vertices [5]. For P_4 it consists of four graphs on five vertices, as shown by Hertz [8].

In Section 4 we describe the class $\mathcal{F}$ for $H = K_{1,2}$ by ten forbidden induced subgraphs, each having five vertices. This characterization does not bring any algorithmic improvement for deciding switching-equivalence to a $K_{1,2}$-free graph; checking all induced subgraphs on five vertices requires time $\Omega(n^5)$, whereas the already known algorithm of Kratochvíl et al. [9] runs in time $\mathcal{O}(n^3)$. However, this characterization is interesting, because the minimal forbidden induced subgraphs are few and small; for $H = K_3$, there are at least hundreds of graphs in $\mathcal{F}$. We discuss this in Section 6.

More generally, we want to recognize switching-equivalence to $\mathcal{H}$-free graphs, i.e., graphs that are H-free for every $H \in \mathcal{H}$, where $\mathcal{H}$ is a fixed graph class. If we consider the class $\mathcal{C}$ of all cycles, then $\mathcal{C}$-freeness is the same as acyclicity. For switching-equivalence to acyclic graphs, the set $\mathcal{F}$ was described by Hage and Harju [4] in a result mentioned above.

Note that any subclass of acyclic graphs is 1-degenerated; hence it follows from a result of Kratochvíl [11] that switching-equivalence to it can be decided in polynomial time $\mathcal{O}(n^5)$.

When $\mathcal{H} = \mathcal{C} \cup \{K_{1,d+1}\}$, the $\mathcal{H}$-free graphs are forests of maximum degree at most d; in case that $d = 1$, they are called partial matchings. For switching-equivalence to partial matchings, the set $\mathcal{F}$ was described by Herman [7]. In Section 5 we extend this result to any integer $d \geq 5$, and give a superset of $\mathcal{F}$ for $d = 2, 3, 4$.

2 Preliminaries

2.1 Basic Definitions

Throughout this paper, the symmetric difference of two sets A and B is denoted by $A \triangle B$. All graphs considered are finite, undirected, and without loops or multiple edges. Unless defined otherwise, by n we denote the number of vertices of the currently discussed graph.

We say that the graph H is an *induced subgraph* of G, written $H \leq G$, if $V_H \subseteq V_G$ and $E_H = \binom{V_H}{2} \cap E_G$. For a set $A \subseteq V_G$ we call the graph $(A, \binom{A}{2} \cap E_G)$ the *subgraph of* G *induced by* A and denote it by $G[A]$. If an isomorphic copy of H is an induced subgraph of G, we shall for simplicity say that G *contains* H *as an induced subgraph* or just that G *contains* H. We say that a graph G is H*-free* if it does not contain H. For a vertex $v \in V_G$, the subgraph of G induced by $V \setminus \{v\}$ is denoted by $G - v$. The disjoint union of two graphs G and G' is denoted by $G + G'$. The symbol $\overline{G}$ stands for the edge-complement of the graph G.

2.2 Seidel's Switching

Definition 1. *Let* G *be a graph.* Seidel's switch of a vertex $v \in V_G$ *results in a graph called* $S(G, v)$ *whose vertex set is the same as of* G *and the edge set is the symmetric difference of* E_G *and the full star centered in* v*, i.e.,*

$$V_{S(G,v)} = V_G$$
$$E_{S(G,v)} = E_G \setminus \{xv : x \in V_G,\ xv \in E_G\}) \cup \{xv : x \in V_G,\ x \neq v,\ xv \notin E_G\}.$$

It is easy to observe that the result of a sequence of vertex switches in G depends only on the parity of the number of times each vertex is switched. This allows generalizing switching to vertex subsets of G.

Definition 2. *Let G be a graph. Then the* Seidel's switch of a vertex subset *$A \subseteq V_G$ is called $S(G, A)$ and*

$$S(G, A) = (V_G, E_G \bigtriangleup \{xy : x \in A,\ y \in V_G \setminus A\}).$$

Definition 3. *We say that two graphs G and H are* switching equivalent *(denoted by $G \sim H$) if there is a set $A \subseteq V_G$ such that $S(G, A)$ is isomorphic to H. The set*

$$[G] = \{S(G, A) : A \subseteq V_G\}$$

is called the switching class *of G.*

Proposition 1. *Let G be a graph and let A be a vertex subset of G. Then $S(\overline{G}, A) = \overline{S(G, A)}$.*

Proof. A straightforward case analysis shows that the edges in $S(\overline{G}, A)$ and $\overline{S(G, A)}$ are the same.

It follows that the graph $S(G, A)$ is H-free if and only if the graph $S(\overline{G}, A)$ is $\overline{H}$-free. Hence, G is switching-equivalent to an H-free graph if and only if $\overline{G}$ is switching-equivalent to an $\overline{H}$-free graph. Thus, the algorithm for $H = K_{1,3}$ described in Section 3 also gives one for $H = K_3 + K_1$.

2.3 Hereditary Properties and Forbidden Induced Subgraphs

We say that P is a *graph property* if P is isomorphism-closed class of graphs; a class of graphs is *hereditary* if it is closed on induced subgraphs. For a hereditary graph property P we define the class $\mathcal{F}(P)$ of minimal forbidden induced subgraphs for P in the following way:

$$\mathcal{F}(P) = \{F : F \notin P, \forall F' \leq F : F' \in P\}.$$

Then a graph G has the property P if and only if it does not contain any element of $\mathcal{F}$.

Lemma 1. *Let P be a hereditary graph property. Then the property of being switching-equivalent to a graph with P is hereditary as well.*

Proof. Let G be a graph, let $A \subseteq V_G$ such that $S(G, A)$ has P. Clearly, for any $G' \leq G$, it holds that $S(G', A \cap V_{G'}) \leq S(G, A)$. Hence, by hereditariness, $S(G', A \cap V_{G'})$ has P, so G' is also switching-equivalent to a graph with P.

In view of Lemma 1 we may further define the class $\mathcal{F}(S(P))$ as the class of minimal forbidden induced subgraphs for switching-equivalence to P.

We study the property of being H-free for fixed graphs H. It can easily be observed that this property is hereditary; we want to find the class $\mathcal{F}(S(\text{“being } H\text{-free”}))$. Similarly as for algorithms, there is a relation between results for H and for $\overline{H}$. From Proposition 1 we get the following corollary.

Proposition 2. *For every graph F it holds that $F \in \mathcal{F}(S(\text{“being } H\text{-free”}))$ if and only if $\overline{F} \in \mathcal{F}(S(\text{“being } \overline{H}\text{-free”}))$.*

Proof. Assume that a graph F is an element of $\mathcal{F}(S(\text{“being } H\text{-free”}))$, but $\overline{F}$ is not in $\mathcal{F}(S(\text{“being } \overline{H}\text{-free”}))$. Then $\overline{F}$ is either not forbidden or not minimal.

In the former case, $\overline{F}$ is switching-equivalent to an $\overline{H}$-free graph. But then, by Proposition 1, F is switching-equivalent to an H-free graph, which is a contradiction. In the latter case, we get in the same way a contradiction with the minimality of F. The other implication follows from the fact that $\overline{\overline{F}} = F$.

Hence a finite class of forbidden induced subgraphs for H also yields one of the same cardinality for $\overline{H}$; and the result of Section 4 for $K_{1,2}$ also gives the class for $K_2 + K_1$. In case of the triangle K_3 and the claw $K_{1,3}$, the corresponding polynomial-time algorithms exist (due to Hayward [6] and Hage et al. [5], and the result of Section 3), but the classes of forbidden induced subgraphs are not known to be finite. We discuss this in Section 6.

3 Switching to a Claw-Free Graph

In this section, we show that it is polynomial to decide if a given graph can be switched to a $K_{1,3}$-free graph. A graph isomorphic to $K_{1,3}$ is called a *claw*.

Definition 4. *We call a graph on four vertices a* dangerous graph *if it is switching-equivalent to a claw. We say that the vertex of degree three in a claw is the* root *of the claw; vertices of degree one are the* nails *of the claw.*

Our aim is to find a switch of a graph so that it destroys all claws and creates no new one. Clearly, the dangerous subgraphs are the only subgraphs that we need to care about. The following crucial lemma indicates which ways of switching dangerous graphs are good and which are not.

Lemma 2. *All the dangerous graphs (up to isomorphism) are the claw $K_{1,3}$ itself, the four-cycle C_4 and four isolated vertices I_4. By switching an even number of vertices in a dangerous graph, we obtain*

- *a claw from a claw*
- *a non-claw from a non-claw.*

By switching an odd number of vertices in a dangerous graph, we obtain

- *a claw from a non-claw*
- *a non-claw from a claw.*

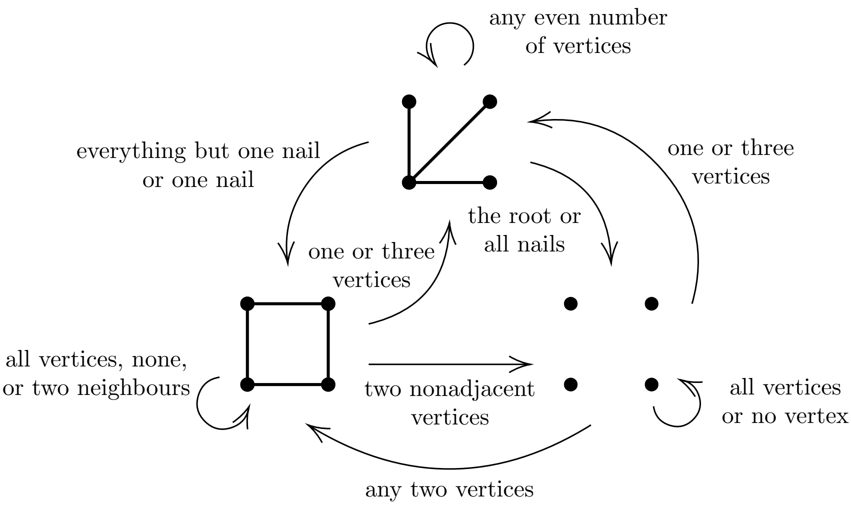

any even number of vertices
everything but one nail or one nail
one or three vertices
the root or all nails
one or three vertices
all vertices, none, or two neighbours
two nonadjacent vertices
all vertices or no vertex
any two vertices

Fig. 1. Dangerous graphs and switching between them

Proof. All possible switches between the graphs $K_{1,3}$, C_4, and I_4 are demonstrated in Fig. 1. A case analysis of the switches follows.

Switching four or zero vertices does not modify the graph, therefore it does not change the property of being a claw; it remains to consider switching one, two or three vertices. Switching any two vertices of a claw creates a claw. Switching the root or all nails of a claw yields an I_4, and switching one nail or everything but one nail yields a C_4. Thus switching any vertex subset of a claw results in one of $K_{1,3}$, C_4, and I_4, and those are really all the dangerous graphs (up to isomorphism). Moreover, we get a claw from a claw if and only if we switch an even number of its vertices.

Switching one or three vertices in a C_4 gives us a claw. By switching two adjacent vertices we get a C_4, and by switching two non-adjacent vertices we obtain an I_4. Thus we get a claw from a C_4 if and only if we switch an odd number of vertices.

Switching one or three vertices in an I_4 yields a claw, whereas switching two yields a C_4. Again, a claw arises from an I_4 if and only if an odd number of vertices are switched.

Corollary 1. *Let G be a graph and $A \subseteq V_G$. Then $S(G, A)$ is claw-free if and only if for every dangerous induced subgraph H of G the following is true:*

- $|V(H) \cap A|$ *is odd if H is a claw,*
- $|V(H) \cap A|$ *is even if H is not a claw.*

Proof. By Lemma 2, switching such an A creates a non-claw from every (dangerous) claw and a non-claw from every dangerous non-claw. But a claw can arise only from a dangerous graph; therefore switching A destroys all claws and creates no new one.

Conversely, if a set A yields a claw-free switch, then by Lemma 2 it contains an odd number of vertices out of every dangerous claw, and an even number of vertices out of every dangerous non-claw.

Theorem 1. *Given a graph G, we can find a set $A \subseteq V_G$ such that $S(G, A)$ is claw-free, or find out that no such set A exists in polynomial time.*

Proof. We show that vertex subsets with the desired properties can be described by a system of linear equations over $GF(2)$ with $\mathcal{O}(n)$ variables and $\mathcal{O}(n^4)$ equations. Solutions of such an equation system can be computed, for example, by the Gaussian elimination in time $\mathcal{O}(n^6)$. In particular, it can be decided in time $\mathcal{O}(n^6)$ whether a solution of such a system exists. The equations, too, can be constructed in polynomial time.

We compute in $GF(2)$. To every vertex $v \in V_G$ we assign a variable x_v, and for every dangerous subgraph H on vertices v_i, v_j, v_k, v_l, we form an equation

$$x_{v_i} + x_{v_j} + x_{v_k} + x_{v_l} = 1$$

if H is a claw or

$$x_{v_i} + x_{v_j} + x_{v_k} + x_{v_l} = 0$$

if H is not a claw. Clearly, we get at most $\binom{n}{4} = \mathcal{O}(n^4)$ equations in this way.

Every assignment of values to the variables yields a vertex subset

$$A = \{v \in V_G : x_v = 1\}.$$

The equations express parity requirements for the size of the intersection of A and dangerous subgraphs. Then, according to Corollary 1, the solutions of this system correspond to all vertex subsets A such that $S(G, A)$ is claw-free.

4 Switching Classes Containing a $K_{1,2}$-Free Graph

In this section, we prove that the class $\mathcal{F}(S(\text{“being } K_{1,2}\text{-free”}))$ is finite and contains (up to isomorphism) exactly the graphs listed in Fig. 2 and Fig. 3.

The lemma below states that the graphs in Fig. 2 and Fig. 3 are forbidden and minimal. It can be proved by a straightforward case analysis; due to space limitations, the proof is placed in the Appendix.

Lemma 3. *The graphs G_1, G_2, G_3, G_4 are (up to isomorphism) all the graphs switching equivalent to G_1. The graphs G_5, …, G_{10} are (up to isomorphism) all the graphs switching equivalent to G_5.*

None of them can be switched to a $K_{1,2}$-free graph, while all their induced subgraphs can.

Lemma 4. *Let G' be a graph not containing any of G_1, …, G_{10} as an induced subgraph. Then G' is switching-equivalent to a $K_{1,2}$-free graph.*

Proof. If G' does not contain any $K_{1,2}$, we are done. Otherwise we seek a switch of G' that is $K_{1,2}$-free. Throughout the proof, we use the following observation.

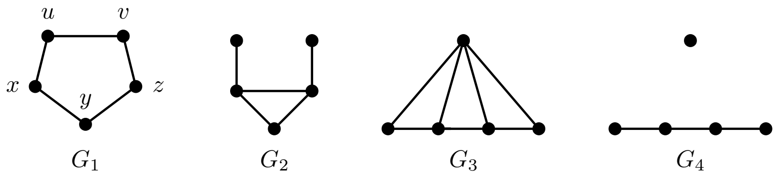


Fig. 2. Forbidden graphs switching-equivalent to G_1

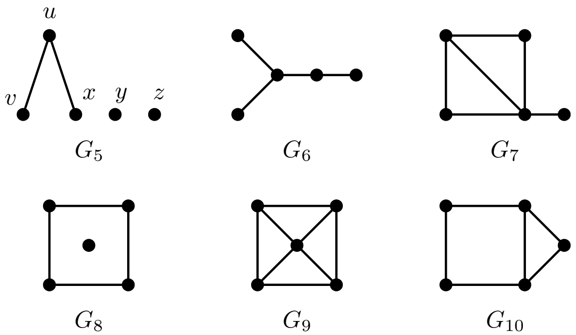


Fig. 3. Forbidden graphs switching-equivalent to G_5

Observation 1 *A graph G is $K_{1,2}$-free if and only if G is a disjoint union of cliques.*

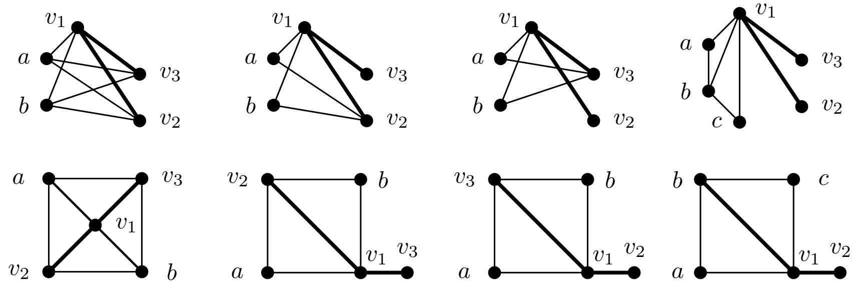
We choose a $K_{1,2}$ in G', denote its vertex of degree two by v_1 and the other ones v_2 and v_3. Without loss of generality we may assume that in G', the vertex v_1 is adjacent to all other vertices. If it is not, we switch G' appropriately and consider the switched graph instead of G'. Obviously, the new graph does not contain any of $G_1, \ldots, G_{10}$, either, and the $K_{1,2}$ on v_1, v_2, v_3 is preserved.

We divide the vertices of G' into four groups according to their neighborhood in v_1, v_2, v_3 in the following way: A is the set of vertices adjacent to v_1, v_2, v_3, B is the set of vertices adjacent to v_1 and v_2, C is the set of vertices adjacent to v_1 and v_3, and D is the set of vertices adjacent to v_1 only. Our aim now is to show that the groups A, B, C, are cliques, whereas D is a disjoint union of cliques.

In Fig. 4, the upper left graph represents one of the forbidden situations: it contains the fixed $K_{1,2}$ with its vertices denoted by v_1, v_2, v_3, and two vertices $a, b \in A$ along with all the edges connecting any vertex of A to v_1, v_2, v_3.

We want A to be a clique, so we want to prevent the case that a and b are not adjacent. But the non-adjacency of a and b yields a forbidden graph G_9, as shown in the leftmost column of Fig. 4 – the lower left graph is the same as the upper left one, it is only redrawn to make its isomorphism to G_9 more obvious. This proves that A induces a clique in G'.

The two cases in the middle of Fig. 4 show the forbidden subgraphs created by a missing edge in B, C, with each forbidden situation (or just a part of it)



A missing edge $\{a, b\}$ in A creates a G_9

A missing edge $\{a, b\}$ in B creates a G_7

A missing edge $\{a, b\}$ in C creates a G_7

A $K_{1,2}$ on $\{a, b, c\}$ in D creates a G_7

Fig. 4. Forbidden graphs created by missing edges in A, B, or C and by a $K_{1,2}$ in D

redrawn in the way described above. By the assumptions, no forbidden subgraph is contained in G', so we conclude that all those groups induce cliques in G'.

The group D need not be a clique; we just want it to be a disjoint union of cliques. We prove this by showing that if D contains a $K_{1,2}$ (and thus is not a disjoint union of cliques), it creates a forbidden subgraph. This situation is the last case of Fig. 4. So D is a disjoint union of cliques.

We proceed by showing that the edges between vertices of different groups are determined already. We shall for simplicity speak about adjacency of whole cliques instead of single vertices.

Definition 5. *We say that two cliques K_1 and K_2 are* adjacent *if all vertices of K_1 are adjacent to all vertices of K_2. We say that two cliques K_1 and K_2 are* non-adjacent *if no vertex of K_1 is adjacent to any vertex of K_2.*

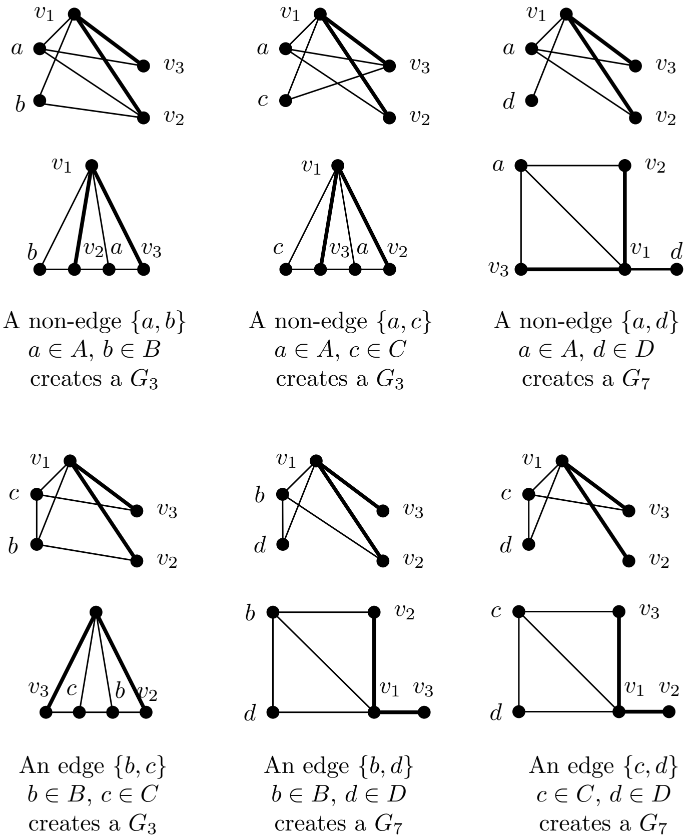
The desired situation is as follows:

- The group A is adjacent to B, C and D.
- The groups B, C, and D are pairwise non-adjacent.
- All the cliques in D are pairwise non-adjacent as well.

Moreover, the vertex v_1 has the same neighborhood (in the rest of the graph) as vertices of A, and is adjacent to all of them; so we can add v_1 to the clique A while keeping all the desired adjacency properties of A. Analogously, we add v_2 to C and v_3 to B.

By examining all the possible cases, we show that an unwanted, or wanted but missing edge would create a forbidden subgraph. The upper part of Fig. 5 shows bad edges between A and the other groups, and the forbidden graphs caused. The lower part shows bad edges among B, C, and D. That finishes the case analysis of edges in G'.

It remains to prove that the graph $G^* = S(G', A)$ is a disjoint union of cliques. That is true, because G^* and G' differ only in edges between A and some other



A non-edge $\{a, b\}$
$a \in A$, $b \in B$
creates a G_3

A non-edge $\{a, c\}$
$a \in A$, $c \in C$
creates a G_3

A non-edge $\{a, d\}$
$a \in A$, $d \in D$
creates a G_7

An edge $\{b, c\}$
$b \in B$, $c \in C$
creates a G_3

An edge $\{b, d\}$
$b \in B$, $d \in D$
creates a G_7

An edge $\{c, d\}$
$c \in C$, $d \in D$
creates a G_7

Fig. 5. Bad edges between A and B, C, D; bad edges among B, C, and D

group, so the cliques of G' are preserved. The groups B, C, D are pairwise non-adjacent, and A becomes non-adjacent to the other cliques as well. Therefore $G^* = S(G', A)$ is the desired $K_{1,2}$-free switch of G'.

Theorem 2. *$\mathcal{F}(S($"being $K_{1,2}$-free"$)) = \{G_1, \ldots, G_{10}\}$.*

Proof. The theorem follows immediately from Lemmas 3 and 4.

5 Switching to *d*-Forests

In this section we describe the class $\mathcal{F}(S($"being acyclic and $K_{1,d+1}$-free"$))$. An acyclic $K_{1,d+1}$-free graph is a forest with all degrees at most d; we call it a *d-forest.*

For a graph G, we say that a set $A \subseteq V_G$ is *feasible* if $S(G, A)$ is a d-forest. Fig. 6 contains examples of graphs with the feasible sets marked in black. These graphs will be important in the further proofs.

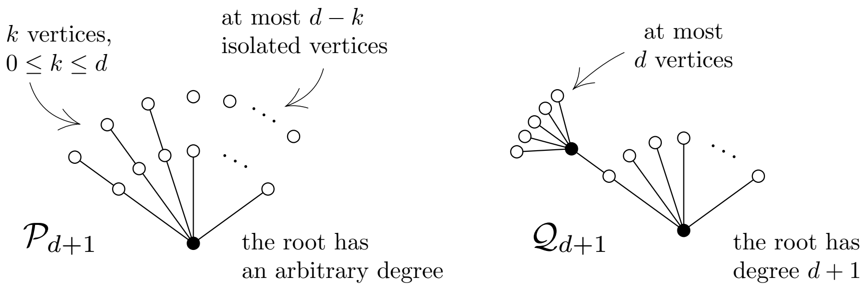


Fig. 6. Graph classes $\mathcal{P}_{d+1}$ and $\mathcal{Q}_{d+1}$

The graphs $A^0_{d+1}, \ldots, A^{d+1}_{d+1}$, B_{d+1}, C_{d+1}, D_{d+1}, E_{d+1} in Fig. 7 are called the *fan graphs.*

Lemma 5. *Let $d \geq 2$. The fan graphs $A^0_{d+1}, \ldots, A^{d+1}_{d+1}$, B_{d+1}, C_{d+1}, D_{d+1}, E_{d+1} are minimal forbidden induced subgraphs for switching to d-forests.*

Proof. To prove the minimality, it suffices to show that by removing any vertex from any fan graph we obtain a graph that is switching-equivalent to a d-forest.

All fan graphs are acyclic. The reader may verify that removing a vertex from any fan graph either destroys all occurrences of induced $K_{1,d+1}$ in it (and thus makes it a d-forest), or makes it an element of $\mathcal{P}_{d+1}$ or $\mathcal{Q}_{d+1}$. The elements of $\mathcal{P}_{d+1}$ and $\mathcal{Q}_{d+1}$ can be switched to a d-forest in the way indicated in Fig. 6.

To prove that the graphs are forbidden, we show that none of them can be switched to a d-forest.

A graph isomorphic to $K_{1,d+1}$ is called a *multiclaw*. We say that vertices of degree one in a multiclaw are *nails*, and the vertex of degree $d+1$ is the *spur*. A vertex non-adjacent to any vertex of a multiclaw M is called a *crumb* of M.

The following lemma will be useful for examining the switches of fan graphs.

Lemma 6. *Let G contain a multiclaw M, and let A be a feasible set. Then A contains either*

- *all nails and not the spur, or*
- *the spur and no nails.*

Furthermore, any crumb of M is an element of A if and only if the nails of M are.

Lemma 6 may be proved by a straightforward case analysis; due to space constraints, it is proved in the Appendix.

We continue the proof of Lemma 5 by showing that no fan graph can be switched to a d-forest. We say that a vertex set A is *feasible* if switching A makes the considered graph a d-forest. For contradiction, assume that a fan graph has a feasible set A. We consider only feasible sets that satisfy the conditions of Lemma 6. Moreover, we assume that A contains the bottom spur of the considered graph, and it does not contain any of its nails nor any crumbs.

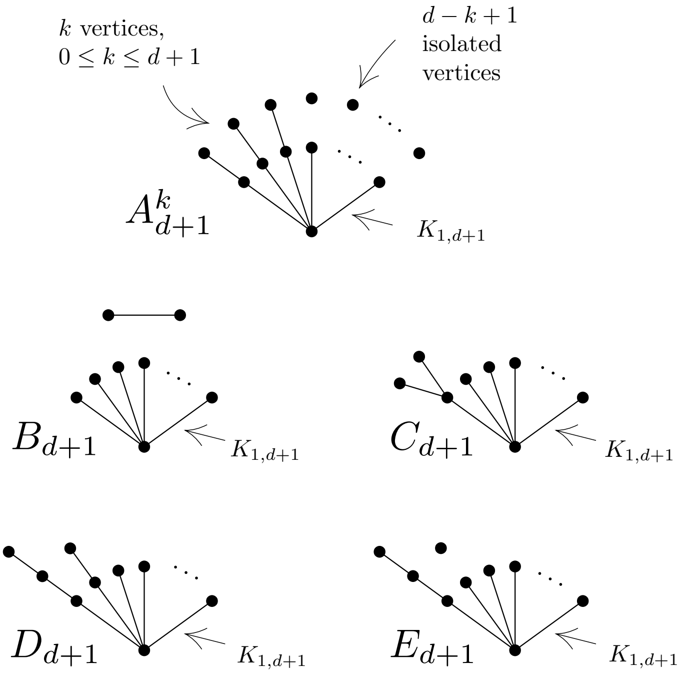


Fig. 7. The fan graphs

- Any graph A^k_{d+1}, where $0 \le k < d+1$, contains at least one crumb c. Then if any other non-crumb vertex is in A, it yields a multiclaw together with its non-adjacent nails and with c. Hence A contains the spur only, but that also yields a multiclaw in $S(A^k_{d+1}, A)$ (with the same spur).
- The graph A^{d+1}_{d+1} contains no crumbs; any other vertex is adjacent to a nail. If A contains the spur only, then there is a multiclaw in $S(A^{d+1}_{d+1}, A)$. Hence A contains at least one other vertex x. But then all the remaining vertices are in A, because the converse would yield a triangle (as indicated in Fig. 9). Since $d \ge 2$ there are at least three other vertices in A, and those vertices together with their adjacent nails yield a cycle of length six.
- In B_{d+1}, the above assumptions determine the set A entirely. However, $S(B_{d+1}, A)$ then contains a triangle, and is not a d-forest.

The analysis of the remaining graphs is fairly similar; due to space limitations, it is placed in the Appendix.

We conclude that, indeed, no fan graph can be switched to a d-forest, and that the fan graphs are minimal forbidden subgraphs for switching to a d-forest.

The following result was proved by Hage and Harju [4].

Theorem 3. *There are 27 switching classes of graphs having at most* 9 *vertices in* $\mathcal{F}(S($*"being acyclic"*$))$. *The switching classes* $[C_n]$ *of cycles are the only graphs having at least* 10 *vertices in* $\mathcal{F}(S($*"being acyclic"*$))$.

Let $\mathcal{A}$ denote the class $\mathcal{F}(S(\text{"being acyclic"}))$, and let $\mathcal{F}_{d+1}$ denote the union of switching classes of the fan graphs $A^0_{d+1}, \ldots, A^{d+1}_{d+1}$, B_{d+1}, C_{d+1}, D_{d+1}, E_{d+1}.

Theorem 4. *Let $d \geq 2$. Then $\mathcal{F}(S(\text{"being a d-forest"}))$ is a subset of $\mathcal{A} \cup \mathcal{F}_{d+1}$. For $d \geq 5$, the classes $\mathcal{F}(S(\text{"being a d-forest"}))$ and $\mathcal{A} \cup \mathcal{F}_{d+1}$ are equal.*

Proof. By Lemma 5 and by the definition of $\mathcal{A}$, no graph on the right-hand side can be switched to a d-forest. The graphs in $\mathcal{F}_{d+1}$ are minimal with such property; the graphs in $\mathcal{A}$ are minimal for switching to acyclicity.

If $d \geq 5$, then all graphs in $\mathcal{F}_{d+1}$ have at least 9 vertices, hence none of them is an induced subgraph of a graph in $\mathcal{A}$ except for, possibly, a cycle. However, it is easy to see that no induced subgraph of a cycle is forbidden. Therefore, the graphs in $\mathcal{A}$ are minimal for switching to a d-forest as well.

It remains to prove that if a graph G contains no element of $\mathcal{A}$ nor $\mathcal{F}_{d+1}$, then it can be switched to a d-forest. By the definition of $\mathcal{A}$ again, G has an acyclic switch G^a.

If G^a contains no $K_{1,d+1}$, then we are done. Otherwise, we root the component of G^a that contains a $K_{1,d+1}$ so that the spur becomes the root x. Let neighbors of x be called the *first level*; vertices adjacent to the first level (excluding the root) the *second level* etc. We distinguish the following two cases.

- There is a vertex v in the third level; let w be its ancestor. If the degree of the root was at least $d+2$, then the vertex v with w, the root and the first-level vertices non-adjacent to w form a B_{d+1}. So the degree of the root is exactly $d+1$.

  There is no other second-level vertex than w, otherwise G^a contains a D_{d+1}. Furthermore, G^a is connected, otherwise G^a contains a E_{d+1}. And there are at most d vertices in the third level, because they are all non-adjacent to the first level, and there may be no A^0_{d+1} in G^a. Altogether, we have that G^a is an element of $\mathcal{Q}_{d+1}$.
- There is no vertex in the third level. Since G^a does not contain a A^0_{d+1}, there are at most d connected components in it. And since G^a does not contain a B_{d+1}, all components except for the rooted one are just isolated vertices (crumbs). Moreover, any branching in the first level would imply a C_{d+1}. Thus, each first-level vertex has at most one descendant. And finally, there is no A^0_{d+1}, so the number of second-level vertices plus the number of crumbs is at most d. Therefore, G^a is an element of $\mathcal{P}_{d+1}$.

In both cases, G^a can be switched to a d-forest in the way depicted in Fig. 6. Thus, by transitivity, G is switching-equivalent to a d-forest as well, which finishes the proof.

6 Concluding Remarks

We have examined the problem of switching-equivalence to an H-free graph for fixed graphs H. Polynomial-time decision algorithms are known for this problem if H has at most three vertices or is isomorphic to a P_4. In Section 3 we have shown that the problem is polynomial even when H is isomorphic to a claw. Thus all known results for particular graphs H yield polynomial-time algorithms, and the following problem (first examined in [9]) remains open.

Problem 1. Is there a graph H such that the problem of deciding switching-equivalence to an H-free graph is NP-complete?

We have described the class $\mathcal{F}(S(\text{“being } K_{1,2}\text{-free”}))$. In case of the triangle K_3 and the claw $K_{1,3}$, polynomial-time decision algorithms exist (due to Hayward [6] and Hage et al. [5], and the result of Section 3), but the classes $\mathcal{F}$ are not known to be finite. Moreover, it seems that neither of the classes is reasonably small. According to a computer search, both classes soon grow to enormous size, which we find interesting in view of the fact that the class for $K_{1,2}$ has ten graphs only, and the graphs K_3 and $K_{1,2}$ differ just in one edge. Also, the minimal forbidden induced subgraphs for $K_{1,2}$ all have five vertices; but for K_3 we have found hundreds of minimal forbidden induced subgraphs on nine or more vertices, and even one on fifteen vertices (shown in Fig. 8). For $K_{1,3}$ there are also hundreds of minimal forbidden induced subgraphs on nine vertices; the maximum number of vertices for which we have found one is twelve.

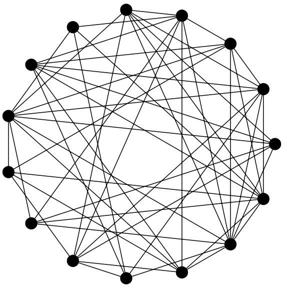


Fig. 8. One of hundreds of minimal forbidden induced subgraphs for $H = K_3$

This indicates that even if an upper bound M for the number of vertices of a graph in $\mathcal{F}$ was found, a naive search through all graphs on at most M vertices would be nearly impossible. However, the classes may still be finite, which yields the following question.

Problem 2. Is there a graph H such that $\mathcal{F}(S(\text{“being } H\text{-free”}))$ is infinite?

Of course, if there is a graph H such that deciding switching-equivalence to an H-free graph is NP-complete, then the class $\mathcal{F}$ is surely infinite (assuming that P is not NP). So a positive answer to Problem 1 is also a positive answer to Problem 2, but not conversely.

The proof in Section 3 uses a reduction to a linear equation system over $GF(2)$. A similar reduction works for H isomorphic to a $K_{1,2}$; however, for $K_{1,2}$ there is already a more efficient algorithm due to Kratochvíl et al. [9]. It might be interesting to find out if a similar approach – possibly using a larger finite field instead of $GF(2)$ – works for some other graphs as well; we have not found any.

Acknowledgement

We would like to thank Jiří Sgall for his useful suggestions.

References

1. Ehrenfeucht, A., Hage, J., Harju, T., Rozenberg, G.: Pancyclicity in switching classes. Inform. Process. Letters 73, 153–156 (2000)
2. Ehrenfeucht, A., Hage, J., Harju, T., Rozenberg, G.: Complexity issues in switching classes. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 59–70. Springer, Heidelberg (2000)
3. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)
4. Hage, J., Harju, T.: A characterization of acyclic switching classes of graphs using forbidden subgraphs. SIAM J. Discrete Math. 18, 159–176 (2004)
5. Hage, J., Harju, T., Welzl, E.: Euler graphs, triangle-free graphs and bipartite graphs in switching classes. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 148–160. Springer, Heidelberg (2002)
6. Hayward, R.B.: Recognizing P_3-structure: A switching approach. J. Combin. Th. Ser. B 66, 247–262 (1996)
7. Herman, J.: Computational Complexity in Graph Theory, master thesis, Charles University in Prague (2006)
8. Hertz, A.: On perfect switching classes. Discrete Appl. Math. 94, 3–7 (1999)
9. Kratochvíl, J., Nešetřil, J., Zýka, O.: On the computational complexity of Seidel's switching. In: Proc. 4th Czech. Symp., Prachatice 1990, Ann. Discrete Math., vol. 51, pp. 161–166 (1992)
10. Kratochvíl, J.: Perfect codes and two-graphs. Comment. Math. Univ. Carolin. 30, 755–760 (1989)
11. Kratochvíl, J.: Complexity of hypergraph coloring and Seidel's switching. In: Bodlaender, H.L. (ed.) WG 2003. LNCS, vol. 2880, pp. 297–308. Springer, Heidelberg (2003)
12. Seidel, J.J.: Graphs and two-graphs. In: Proc. 5th. Southeastern Conf. on Combinatorics, Graph Theory, and Computing, Winnipeg, Canada (1974)
13. Seidel, J.J.: A survey of two-graphs. In: Teorie combinatorie, Atti Conv. Lincei, vol. 17, pp. 481–511. Academia Nazionale dei Lincei, Rome (1973)
14. Seidel, J.J., Taylor, D.E.: Two-graphs, a second survey. In: Algebraic methods in graph theory, vol. II, Conf. Szeged 1978, Colloq. Math. Janos Bolyai, vol. 25, pp. 689–711 (1981)

Appendix

Proof (of Lemma 3). Since being switching-equivalent is an equivalence relation, it suffices to examine all the possible switches of G_1 and G_5. Without loss of generality the set A of switched vertices is of size at most two (otherwise we switch $A' = V_G \setminus A$ instead of A, which gives us the same resulting graph). By switching one vertex in G_1 we obtain a G_2. By switching two adjacent vertices in G_1 we get a G_3, and switching two non-adjacent ones gives us a G_4.

By switching u we get a G_5. By switching one of v, x we get a G_6. Switching one of y, z produces a G_7. Switching u and one of v, x yields a G_7. By switching u and one of y, z we get a G_6. By switching v and x we get a G_8. Switching y and z produces a G_9 and, finally, switching one of v, x and one of y and z creates a G_{10}.

Further, we want to show that none of the graphs $G_1, \ldots, G_{10}$ can be switched to a $K_{1,2}$-free graph; in other words, the switching class of G_1 does not contain a $K_{1,2}$-free graph. Indeed, none of $G_1, \ldots, G_{10}$ is $K_{1,2}$-free, as can be seen in Fig. 2 and Fig. 3.

As for the minimality, we note that any graph on four (or less) vertices can be switched to a $K_{1,2}$-free graph. Assume that it contains a $K_{1,2}$ on vertices a, b, c, where b is the vertex of degree two, and let d be the fourth vertex. If d is adjacent to at least two of a, b, c, then we switch the set $\{b, d\}$. Otherwise, we switch $\{b\}$. The resulting graph contains at most one edge, hence it is $K_{1,2}$-free.

Proof (of Lemma 6). Since $S(G, A)$ is a d-forest, $S(M, A)$ is not a multiclaw. Thus, $A \cap V_M$ is a proper nonempty subset of V_M.

Assume that A violates the first condition above. Then, without loss of generality, A contains the spur s, a nail x, and it does not contain a nail y. There are at least three nails in a multiclaw, because $d \geq 2$. If A contains no more nails, then $S(M, A)$ is again a multiclaw (whose spur is now x). Otherwise, A contains another nail y; then the vertices s, x, y, z induce a cycle in $S(M, A)$, which is a contradiction.

It remains to prove that crumbs are in A if and only if nails are. Suppose the contrary: without loss of generality, A contains no nails and it does contain a crumb c. Then the vertex c together with the nails forms another multiclaw in $S(G, A)$, which is not possible.

Proof (of Lemma 5, continued)

- C_{d+1} contains two non-multiclaw vertices, x and y. If none of them is in A, then they form a cycle in $S(C_{d+1}, A)$ with the spur and their adjacent nail n. If both are in A, then they yield a four-cycle with any two nails non-adjacent to them. Hence one of them, say, x, is in A, and y is not. But then x forms a multiclaw in $S(C_{d+1}, A)$ together with y and all the formerly non-adjacent nails.
- In D_{d+1}, the neighbor x of the crumb c must be in A, otherwise x, c and the spur form a triangle in $S(D_{d+1}, A)$ (see Fig. 9). Then y is also in A,

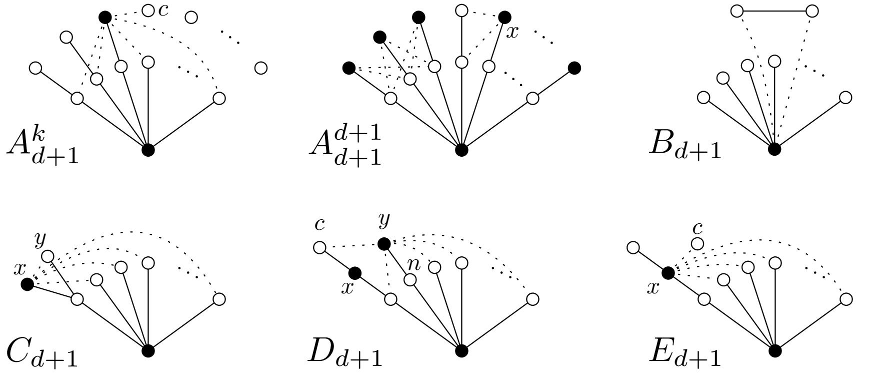


Fig. 9. Illustration of the proof of Lemma 5. Vertices of A are marked black, the others white.

otherwise there is a triangle x, y, n. But in that case there is a multiclaw on y, c and all the nails except for n.

- In E_{d+1}, the vertex x is in A for the same reasons as in D_{d+1}. Then x, the crumb c and the nails formerly non-adjacent to x yield a multiclaw in $S(E_{d+1}, A)$.

A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation*

Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar Tudósok krt. 2
bergmann.gabor@gmail.com,
{rath,ahorvath,varro}@mit.bme.hu

Abstract. In graph transformation, the most cost-intensive phase of a transformation execution is pattern matching, where those subgraphs of a model graph are identified and matched which satisfy constraints prescribed by graph patterns. Incremental pattern matching aims to improve the efficiency of this critical step by storing the set of matches of a graph transformation rule and incrementally maintaining it as the model changes, thus eliminating the need of recalculating existing matches of a pattern. In this paper, we propose benchmark examples where incremental pattern matching is expected to have advantageous effect in the application domain of model simulation and model synchronization. Moreover, we compare the incremental graph pattern matching approach of VIATRA2 with advanced non-incremental local-search based graph pattern matching approaches (as available in VIATRA2 and GrGen).

Keywords: incremental graph pattern matching, RETE, benchmarking.

1 Introduction

Incremental graph pattern matching approaches [18,1,2,3] have recently become a hot topic in the graph transformation community. The core guideline is to improve the execution time of the time-consuming pattern matching phase by additional memory consumption. Essentially, the (partial) matches of the left-hand side (LHS) of graph transformation rules are stored explicitly, and these match sets are updated incrementally in accordance with elementary model changes. While model manipulation becomes slightly more complex, all matches of a graph pattern can be retrieved in constant time in exchange by eliminating the need for recomputing existing matches.

Up to now, the performance evaluation of such incremental graph pattern matching approaches have been limited to dedicated benchmark examples, all of which were characterized by traditional, batch-like execution strategy.

In the current paper, we first propose two benchmark examples where an incremental pattern matching strategy appears to be very beneficial: (i) in the simulation example, enabled transitions of a Petri net are maintained in an incremental way, while (ii) in

* This work was partially supported by the SENSORIA European project (IST-3-016004). The fourth author was also supported by the János Bolyai Scholarship.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 396–410, 2008.

the model synchronization example, an object-relational mapping is carried out where changes in the source UML model are propagated incrementally to the corresponding relational database model.

In addition, we evaluate the performance of the incremental graph transformation engine of the VIATRA2[1] framework on various benchmark examples. A full-fledged comparison is provided with respect to the non-incremental version [4] of the VIATRA2 engine, furthermore, an initial comparison is provided with GrGEN.NET [5], which is currently considered to be the fastest graph transformation engine.

The rest of the paper is structured as follows. Section 2 briefly introduces model simulation captured by graph transformation rules, which serves as one of the benchmarks presented in the paper. In Sect. 3, an incremental graph pattern matching approach is overviewed, which was implemented in the VIATRA2 framework. As the main contribution, novel benchmark examples are presented in Sect. 4 for model simulation and model synchronization with performance evaluation discussed in Sect. 5. Finally, Sect. 6 summarizes the related work and Sect. 7 concludes the paper.

2 Foundations of Model Simulation

This section overviews the foundations of modeling language specification and simulation. In order to specify the abstract syntax of most modeling language, the concept of metamodeling is used. For simulating the behaviour of models, the paradigm of graph transformation [6] is applied.

2.1 Running Example: Simulation of Petri Nets

In the current paper, we will use the simulation of Petri nets as one of our performance benchmarks for incremental pattern matching. We will use the same example to demonstrate the technicalities of incremental pattern matching in graph transformation tools.

Petri nets (Fig. 1) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. Petri nets are bipartite graphs, with two disjoint sets of nodes: *Places* and *Transitions*. Places may contain an arbitrary number of Tokens. A token distribution (marking) defines the state of the modelled system. The state of the net can be changed by firing enabled transitions. A transition is enabled if each of its input places contains at least one token and no place connected with an inhibitor arc contains a token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by Input Arcs) and add a token to all output places (as defined by Output Arcs).

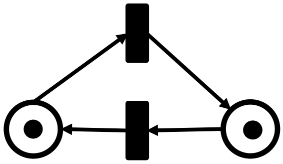


Fig. 1. A sample Petri net

2.2 Foundations of Metamodeling

A *metamodel* describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called *classes*. A class may have attributes that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra *attributes*. *Associations* define connections between classes. Both ends of an association may have a *multiplicity* constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. The most typical multiplicity constraints are i) the at-most-one (0..1), and (ii) the arbitrary (denoted by *). A simple Petri net metamodel is shown in Fig. 2.

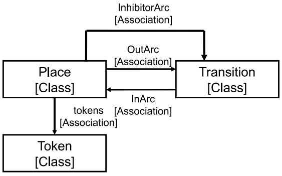


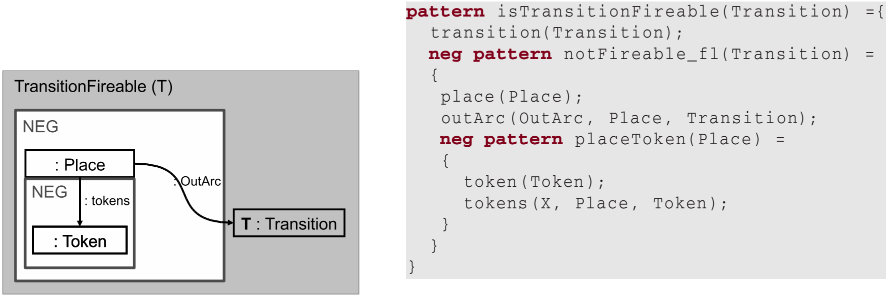
Fig. 2. Petri net metamodel

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required.

2.3 Graph Patterns and Graph Transformation

Graph patterns are frequently considered as the atomic units of model transformations [7]. They represent conditions (or constraints) that have to be fulfilled by a part of the instance model in order to execute some manipulation steps on the model. A basic graph pattern consists of graph elements corresponding to the metamodel. A *negative application condition* (NAC), defined by a negative subpattern, prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [8].

As an example, the firing enabledness condition for a Petri net transition may be expressed using a graph pattern as shown in Fig. 3 using the VIATRA2 notation. This pattern uses nested negative application conditions to express that a Transition is



```
pattern isTransitionFireable(Transition) ={
  transition(Transition);
  neg pattern notFireable_fl(Transition) =
  {
   place(Place);
   outArc(OutArc, Place, Transition);
   neg pattern placeToken(Place) =
   {
     token(Token);
     tokens(X, Place, Token);
   }
  }
}
```

Fig. 3. Petri-net firing condition

enabled if every input Place instance connected to the Transition instance has at least one Token instance associated and no inhibitor input Place instance contains tokens. In this example, embedded NACs are used to express universal quantification with double negation of existence.

Graph transformation (GT) [9] provides a high-level rule and pattern-based manipulation language for graph models. Graph transformation rules can be specified by using a left-hand side – LHS (or precondition) pattern determining the applicability of the rule, and a right-hand side – RHS (postcondition) pattern which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in the RHS are created, and other model elements remain unchanged. For instance, a GT rule may specify how to remove (or add) a token from a place, as shown in Fig. 4.

```
// Removes a token from the place 'Place'.
gtrule removeToken(in Place, in Transition) = {
 precondition find sourcePlaceWithToken
     (Transition, Place, Token);
 postcondition find sourcePlaceWithoutToken
     (Transition, Place, Token);
}
// Adds a token from the place 'Place'.
gtrule addToken(in Place, in Transition) = {
 precondition find targetPlaceWithoutToken
     (Transition, Place, Token);
 postcondition find targetPlaceWithToken
     (Transition, Place, Token);
}
```

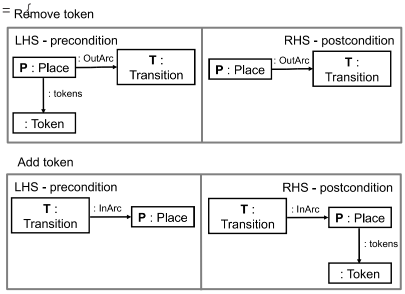


Fig. 4. Graph transformation rules for firing a transition

Complex model transformation can be assembled from elementary graph patterns and graph transformation rules using some kind of control language. In our examples, we use an abstract state machine (ASM) [10] for this purpose as available in the VIATRA2 framework. The following transformation (which will be used as a benchmark example in Sect. 4.1) simulates the firing of a transition, i.e. the removal of tokens from input places and the addition of tokens to output places (see Fig. 5).

```
rule fireTransition(in T) = seq {
 /* perform a check to confirm that the transition is fireable */
 if (find isTransitionFireable(T))
 seq
 {/* remove tokens from all input places */
  forall Place with find inputPlace(T, Place)
   do apply removeToken(T, Place); // GT rule invocation
  /* add tokens to all output places */
  forall Place with find outputPlace(T, Place)
   do apply addToken(T, Place);
 }
}
```

Fig. 5. Transformation program for firing a transition

3 RETE-Based Incremental Graph Pattern Matching

The incremental graph pattern matcher of the VIATRA2 framework [1] adapts the RETE algorithm, which is a well-known technique in the field of rule-based systems.

RETE network for graph pattern matching. RETE-based pattern matching relies on a network of nodes storing *partial matches* of a graph pattern. A partial match enumerates those model elements which satisfy a subset of the constraints described by the graph pattern. In a relational database analogy, each node stores a *view*. Matches of a pattern are readily available at any time, and they will be incrementally updated whenever model changes occur.

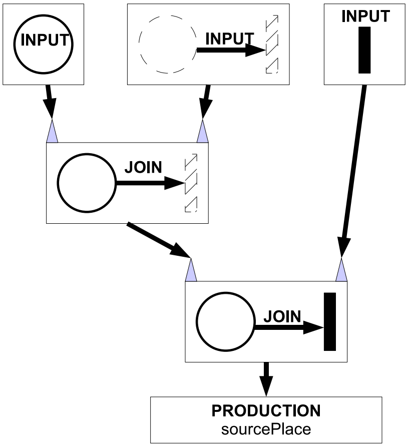


Fig. 6. Simple RETE matcher

Input nodes serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing a view representing all the instances that conform to the type. Similarly, there is an input node for each relation type, containing a view consisting of tuples with source and target in addition to the identifier of the edge instance.

At each *intermediate node*, *set operations* (e.g. filtering, projection, join, etc.) can be executed on the match sets stored at input nodes to compute the match set which is stored at the intermediate node. The match set for the entire pattern can be retrieved from the output *production node*. An intermediate node of a RETE is the join node, which performs a natural join on its input nodes in terms of relational algebra. A *negative node* contains the set of tuples stored at the primary input which do *not* match any tuple from the secondary input (which corresponds to anti-joins in relational databases).

Updates after model changes. Input nodes receive notifications about each elementary model change (i.e. when a new model element is created or deleted) and release an update token on each of their outgoing edges. Such an update token represents changes in the partial matches stored by the RETE node. Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set.

Upon receiving an update token, a RETE node determines how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes. An example RETE network is depicted in Fig. 6.

The match set can be retrieved from the network instantly without re-computation, which makes pattern matching very efficient. As a trade-off, there is increased memory consumption, and update operations become more complex.

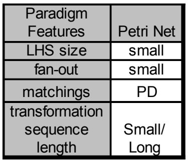
4 Benchmarks for Incremental Graph Transformation

In the paper, we propose two new benchmark problems as an extension to the Varro benchmarks [11]. From a problem-specific viewpoint, they address two important application scenarios, namely, model simulation and model synchronization, which were only partially covered in [11]. Moreover, from a tool-oriented viewpoint, they provide the first test sets for the "as-long-as-possible" optimization strategy, which was not measured up to now. Finally, the change propagation scenario in model synchronization is a highly realistic challenge for model transformation tools.

4.1 Simulation Scenario Based on Petri Net Firing

Description. We selected the Petri net benchmark for the scenario of simulation of visual languages with dynamic operational semantics. This scenario summarizes typical domain specific language simulation with the following characteristics: (i) mostly static graph structure, (ii) relatively small and local model manipulations, and (iii) typical *as-long-as-possible* (ALAP) execution mode. This benchmark focuses on the effective reusability of already matched elements as typical firing of a transition only involves a small part of the net. While an incremental pattern matcher can track the changes of the Petri net and updates only the involved sub-matchings, non-incremental local search based approaches will have to restart the matching from scratch after the net changed.

Test case generation. In the Petri net test set, we selected "regular" Petri nets as *test cases*, which are generated automatically. Here regular means that the number of *places* and *transitions* are approximately equal (where their exact ratio is around 1.1). Furthermore, the net has only a low number of tokens, and thus, there are few fireable transitions in each marking.



Paradigm Features	Petri Net
LHS size	small
fan-out	small
matchings	PD
transformation sequence length	Small/ Long

Fig. 7. Feature matrix of Petri Net benchmark

To generate the elements of the test set we used six reduction operations (in the inverse direction to increase the size of the net) which are described in [12] as means to preserve safety and liveness properties of the net. These operations are combined with a weighted random operation selection. This allows fine parametrization of the number of transitions and places with an average fan-out of 3-5 incoming and outgoing edges. In all test cases, the generation started from the Petri net depicted in Fig. 1 (which is trivially a live net) and the final test graphs are available in PNML [13] format at [14]. As the size of a Petri net cannot be described by only a single parameter we used the number of property preserving we applied to indicate the relative ”size” of test cases.

Execution phases. A step in the iterative execution sequence contains two phases: (i) a fireable transition is non-deterministically selected by pattern *isTransitionFireable* (Fig. 3) and then (ii) the GT rules *addToken* and *removeToken* are applied to simulate the token flow (Fig. 4).

Despite its simple execution semantics, it is easy to derive additional Petri nets as new benchmark scenarios with significantly different run-time characteristics for the different graph transformation tools. For example, a Petri net with an equal number of transitions, places and tokens but with few fireable transitions can be used as a benchmark where type-based optimization strategies of pattern matcher algorithms are neutralized, which forces the pattern matchers to use other heuristics.

Note that the only assumption we made on our Petri net test cases is to use *live* and *bounded* nets to have a potentially unbounded execution sequence. We selected 1000 consecutive transition firings as *Short* execution sequences and 1000000 transition firings as *Long* execution sequences.

For this benchmark, we compared the total execution time of the simulation sequences. As the actual firing transitions are non-deterministically selected by the tools, we allowed the pattern matchers to select their own execution paths, but this turned out to have only insignificant effects on execution times.

Characteristics. In order to give a comparable description of our proposed benchmarks with the ones defined in [11] we also use *feature matrices* to describe the characteristics of the new test sets. The definition of the features are the following:

- *Pattern size*, or the number of nodes and edges in the LHS graph, is a critical factor in the runtime phase of pattern matching.
- The maximum degree of nodes (*fan-out*) in the model is the number of edges that are adjacent to a certain node.
- The third feature is the *number of matches* during the test case execution.
- The *length of the transformation sequence* also affects the overall execution time. For example, with a large number of rule applications, the relative cost of one-time overhead of the pattern matcher is decreased.

Fig. 7 presents the feature matrix describing the Petri net test case. Note that if the characteristics of a feature depends on the concrete parameter settings of the test case, then it is called parameter dependent (marked PD).

4.2 Model Synchronization Scenario by Object-Relational Mapping

Description. The Object-to-Relational schema mapping (ORM) benchmark, as presented in the current paper, is an extension of the original benchmark proposed in Sect.4 of [11]. The original transformation processed UML class diagrams to produce corresponding relational database schemas, according to the known mapping rules. Since a straightforward application of the incremental pattern matching approach is the synchronization between source and target models, we extended the benchmark by two additional sequences: (i) after the initial mappings are created, the *source models are modified*, and, in an additional pass, (ii) the system has to *synchronize the changes to the target model* (i.e. find the changes in the source and alter the target accordingly). A local seach-based algorithm has to search for the changes first, while an incremental pattern matcher can track changes in the source model so that the model parts affected are instantly available for the synchronization sequence.

Test case generation. In order to produce sufficiently large model graphs for the measurements, we implemented a simple generator as described in [11]. By this approach, a fully connected graph is created, i.e. for N UML classes, $N(N-1)$ directed associations are defined (with each association represented as three nodes – an association node and two endpoints). Additionally, each UML class can reference K attributes, thus, for a given N and K, $N+3N(N-1)+NK$ nodes and $4N(N-1)+NK$ edges are created (Fig. 8). Although the model produced is not "realistic" in the sense that very few practical UML class diagrams are fully connected, the method is quite efficient in creating large graphs quickly.

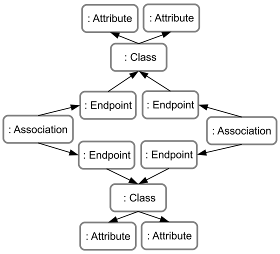


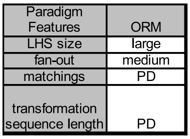
Fig. 8. Generated UML class diagram for N=K=2

Execution phases. The transformation sequence is comprised of four main phases:

1. The *generation* phase creates the model graph.
2. The *build* phase creates the initial mapping of the UML model into the relational schema domain, with reference models connecting mapped model objects.
3. The *modification* phase modifies the UML models programmatically to emulate user editing actions.
4. Finally, the *synchronization* phase locates the affected model elements and makes changes in the schema model accordingly.

Characteristics. For this benchmark, we compare the execution times for the last (synchronization) phase. In order to scale the synchronization sequence as the model size grows, we designed the modification sequence to extend roughly linearly with the model. Thus, in the default case, it is composed of the following operations: (i) first, one third of generated classes, along with their attributes and referenced associations are deleted; (ii) then, one fifth of remaining associations are deleted; (iii) next, every second attribute is renamed; (iv) finally, a new class is added and a new fully connected graph is created (with the remaining UML classes and the newly added class as nodes, ignoring existing associations). The feature matrix based on the notation in Sect. 4.1 is shown in Fig. 9.

Transformation rules for synchronization. In incremental synchronization, to avoid rebuilding target models in each pass, a *reference model* is used to establish a mapping relationship between source and corresponding target model elements (Fig. 10). With correspondence edges, it is possible to track changes in both the source and target models: for instance, the graph pattern on Fig. 11 matches *tables* in the schema model which are no longer referenced by classes or associations in the UML models (*orphan tables*).



Paradigm Features	ORM
LHS size	large
fan-out	medium
matchings	PD
transformation sequence length	PD

Fig. 9. Feature matrix for the ORM benchmark

Similarly, a newly created class may be matched by a negative condition forbidding the existence of a mapped table. Renames (value changes) may be expressed e.g. by

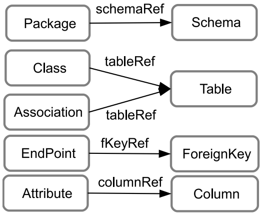


Fig. 10. Reference metamodel

```
pattern orphanTable(T) =
{
  table(T);
  neg pattern mapped(T) =
  {
    class(C);
    table(T);
    class.tableRef(_REFN, C, T);
  } or {
    association(A);
    table(T);
    association.tableRef(_REFN, A, T);
  }
}
```

Fig. 11. Graph pattern for orphan tables

matching for both the attribute and the mapped column, and looking for pairs where the name (attribute value) is different. The modification sequence results in the following synchronization sequence: (i) all orphan tables belonging to the deleted classes and their associations are deleted; (ii) all orphan tables belonging to the deleted associations are deleted; (iii) column names mapped to the renamed attributes are changed; (iv) new tables are added for the newly created class and the new associations.

5 Measurement Results

The measurements reported in this paper have been carried out on a standard desktop computer with a 2 GHz Intel Core2 processor with 2 gigabytes of system RAM available, running version 1.6.0_05 of the 32-bit Sun Java SE Runtime (for VIATRA2) and version 3.0 of the .NET Framework on Windows Vista (for GrGEN.NET). In general, ten test runs were executed, and the results were calculated by averaging the values excluding the highest and lowest number. The transformation sequences were coded so that little or no output was generated; in the case of VIATRA2, we refrained from disabling the GUI. Execution times were measured with millisecond precision as allowed by the operating system calls.

5.1 Distributed Mutual Exclusion Algorithm

In order to compare the pattern matcher algorithms used in this paper with an already available benchmark, we evaluated the performance of the VIATRA2 local search based (VIATRA/LS) and incremental (VIATRA/RETE) pattern matchers along with the GrGEN.NET with the *distributed mutual exclusion algorithm* test set defined in [11] which is not a primary application filed for incremental pattern matching.

The results are shown in Fig. 12 with logarithmically scaled axes, where the *size of the process ring* represents the number of processes in the run, which is, in turn, the runtime parameter for the test case. We can make the following observations: (i) the scaling complexity is a high order polynomial for VIATRA/LS and close to linear for VIATRA/RETE and linear for GrGEN.NET; (ii) this test set seems to fit better for optimized local search based approaches as incremental caching of non-reuseable model

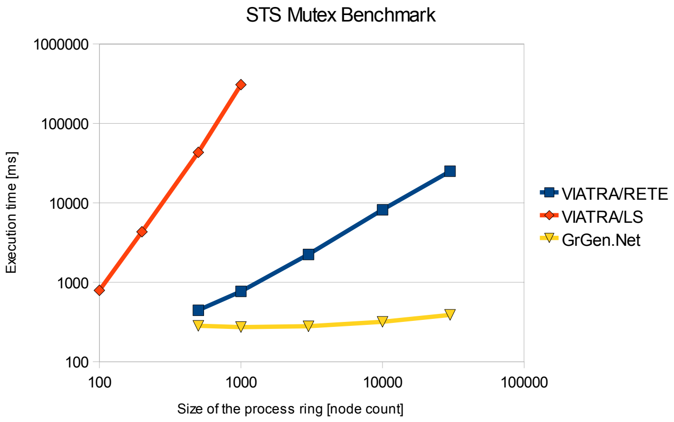


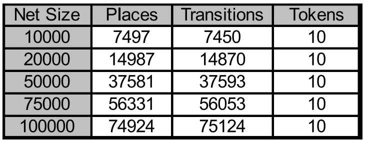
Fig. 12. Results for the Short Transformation Sequence mutex benchmark

elements produced in the second phase increases the overhead of the cache synchronization. Additionally, by looking at memory consumption figures, it can be seen that the static graph structure limits the memory overhead to the same order of magnitude for VIATRA/RETE and GrGEN.NET.

5.2 Simulation of Petri Nets

The Petri net synchronization benchmark was executed with *short* (1000) and *long* (1000000) execution sequences.

The size parameters of the nets used as test cases are depicted in Fig. 13. *Net size* represents the number of randomly applied inverse property preserving operations used during their generation, while *Places*, *Transitions* and *Tokens* represent their actual number. The results are shown in Fig. 14 with logarithmically scaled axes, where model size indicates the *net size* of the test case.



Net Size	Places	Transitions	Tokens
10000	7497	7450	10
20000	14987	14870	10
50000	37581	37593	10
75000	56331	56053	10
100000	74924	75124	10

Fig. 13. Size of test cases

As it can be seen from the graph, VIATRA/RETE has a predictable linear scaling up to model size of 10^5 with a speed of at least two orders of magnitude faster than VIATRA/LS. As expected, the incremental approach works well for large model sizes as long as there is enough memory (the spike in case of long transformation sequences occured because of garbage collection as the heap was exceeded).

VIATRA/RETE matches and outperforms the GrGEN.NET tool for very large models in case of both short and long execution sequences. Moreover, with additional memory provided, the characteristics of VIATRA2 are expected to be better for even larger models with predictable execution time.

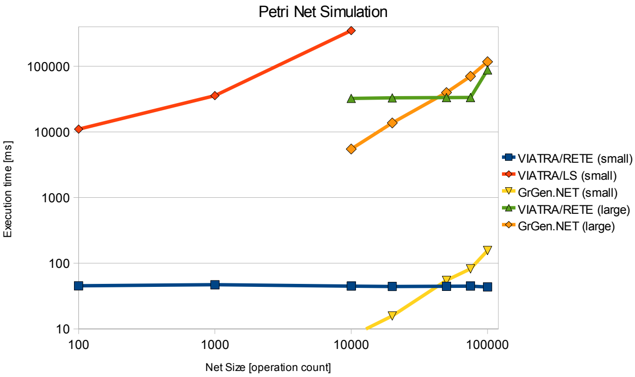
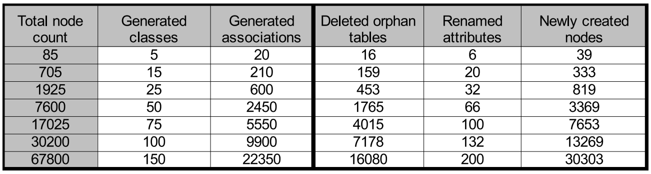


Fig. 14. Results for the Petri net firing benchmark

This result is a significant achievement considering the architectural and run-time differences between VIATRA2 and GrGEN.NET. Most notably, GrGEN.NET uses compile-time optimizations and an entirely different model persistence approach based on compile-time generated type information, whereas VIATRA2 uses a generic model storage supporting dynamic typing and support for interactive applications such as a notification and transaction management mechanism (note that the VIATRA2 GUI was not disabled for the measurement, while GrGEN.NET was used without GUI through GrShell). However, for fairness, it should be pointed out that (unlike the mutual exclusion case) this benchmark was prepared by ourselves (i.e. by GrGEN non-experts), thus additional language or tool-specific optimizations might be available.

5.3 Object-Relational Mapping Synchronization

The ORM synchronization benchmark was executed with the VIATRA2 tool (due to time constraints, measurements with GrGEN.NET and others are left as future work). Models up to 67800 nodes (with edges, the total model size is 157800 model elements) were generated (Fig. 15) and the execution time for the *build* and *synchronization* phases was measured.



Total node count	Generated classes	Generated associations	Deleted orphan tables	Renamed attributes	Newly created nodes
85	5	20	16	6	39
705	15	210	159	20	333
1925	25	600	453	32	819
7600	50	2450	1765	66	3369
17025	75	5550	4015	100	7653
30200	100	9900	7178	132	13269
67800	150	22350	16080	200	30303

Fig. 15. Model and synchronization sequence sizes for the ORM benchmark

The results are shown in Fig. 16 (model size is the total number of nodes). It is again revealed that the scaling characteristic of both phases is exponential for VIATRA/LS and linear for VIATRA/RETE. With respect to synchronization, the constant

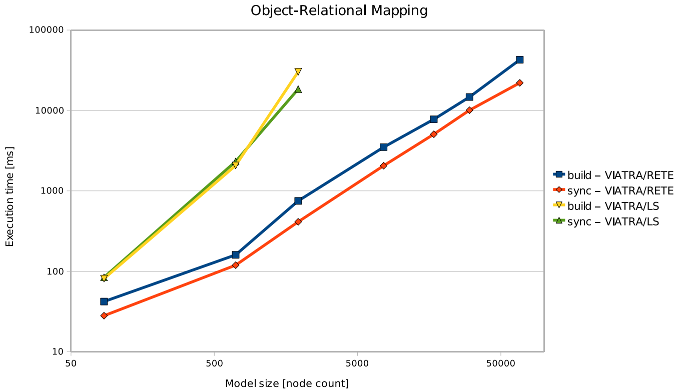


Fig. 16. Results for the ORM synchronization benchmark

difference between the *build* and *sync* phases for VIATRA/RETE means a constant multiplier; thus, since the model elements affected by the modification sequence are a linear fraction of the whole model, it can be concluded that the execution time for the synchronization process is a linear function of the model elements affected (as expected), and independent of the size of the rest of the model. VIATRA/LS, on the other hand, exhibits an ever increasing time difference between *build* and *sync*, thus, the time taken for the synchronization process increases exponentially with the number of affected model elements (again, as expected, since in case of local search, the system has to locate the changed elements first which is an additional graph traversal). It is important to note that for “practical” model sizes (e.g. below the 5000 node count range), VIATRA/RETE can perform a synchronization affecting a considerable portion of the model in the 10-500 msec range which makes the approach very suitable for interactive applications.

In addition to execution times, the memory consumed by the Java Virtual Machine was also recorded. The sequence for the RETE matcher (75, 100, 114, 245, 490, 750, 1000 megabytes respectively for model sizes from 85 to 67800 nodes) shows a linearly expanding RETE network as the node count grows, which is in-line with our expectations based on the nature of the RETE building algorithm (note that the above figures include the whole user interface with a complete Eclipse instance).

5.4 Summary

Analyzing the results obtained in our test cases, the following conclusions can be drawn:

(i) A major concern of any incremental pattern matching implementation is the increased memory consumption. While our implementation does indeed consume more memory than the standard local search-based VIATRA engine, this overhead, even for the extreme model sizes in the benchmark problems, is still within the bounds of RAM available in modern desktop computers making the approach feasible for a wide range of applications.

(ii) Within the memory boundaries, our new RETE-based pattern matcher provides a *predictable, linear scaling* up to the 10^5 model size range in all three scenarios. While even generic transformations experience a speed-up, the real potential of the implementation is revealed in the scenarios especially suited for incremental pattern matching where the execution speed matches, or even surpasses the speed of the fastest conventional graph transformation tool employing compile-time optimization.

(iii) By comparing the run-time characteristics of our multiple test cases, it seems evident that the best results could be achieved by employing different pattern matching strategies for different execution phases, or, even for different patterns in a model transformation program.

6 Related Work

Incremental pattern matching. Incremental updating techniques have been widely used in different fields of computer science. Now we give a brief overview on incremental techniques that are used in the context of graph transformation. The transformation engine of TefKat [16] performs an SLD resolution based interpretation during which a search space tree is constructed to represent the trace of transformation execution. This tree is maintained incrementally in consecutive steps of transformations as described in [17]. The uniform, incremental handling of model elements and patterns can be considered a unique, advanced feature of the approach. [18] proposes a graph pattern matching technique, which constructs and stores a tree for partial matchings of a pattern, and incrementally updates it, when the model changes. The main advantage of this solution is that only matchings, which appear as leaves of the tree, have to be physically stored, which possibly saves a significant amount of memory. The memory saving technique of [18] is orthogonal to the structure of the underlying RETE network, and, thus, it can expectedly be used for our approach as well, but the exact integration requires further research and implementation tasks.

RETE networks. RETE networks [19], which stem from rule-based expert systems, have already been used as an incremental graph pattern matching technique in several application scenarios including the recognition of structures in images [20], and the co-operative guidance of multiple uninhabited aerial vehicles in assistant systems as suggested by [21]. Our contribution extends this approach by supporting a more expressive and complex pattern language.

Graph transformation benchmarking. Some of the measurements in the current paper are conceptual continuations of the comprehensive graph transformation benchmark proposed in [11], which gave an overview on typical application scenarios of graph transformation together with their characteristic features. [15] suggested some improvements to the benchmarks described in [11] and reported measurement results for many graph transformation tools including AGG [22], PROGRES [23], Fujaba [24], and GrGEN.NET [5]. A similar approach to graph transformation benchmarking was used for the AGTIVE Tool Contest [25], including a simulation problem for the Ludo table game. Our Petri net firing test case is better suited for benchmarking performance since it can be parameterized to scale up to large model sizes and long transformation sequences.

7 Conclusion and Future Work

In the current paper, we have have proposed two new test cases as performance benchmarks of graph transformation which are suitable for assessing incremental graph transformation strategies. For this purpose, we focused on two scenarios: (i) The Petri net *model simulation* benchmark was designed to provide a parameterizable and scalable test case for analyzing the impact of incremental pattern matching on a typical simulation scenario; (ii) the Object-Relational Mapping scenario was adapted to *model synchronization* which is a prime target for an event-driven application of graph transformation, where models have to be mapped on-the-fly as the user is editing the model.

We carried out various measurements to assess the performance of the incremental pattern matcher of the VIATRA2framework [1], which clearly demonstrate the viability of the approach: very fast execution with predictable, linear scaling up to memory limitations.

By analyzing the test runs with a Java code profiler, we have identified some key areas where the performance of the VIATRA/RETE tool could be further improved in the future, such as (i) optimizing VIATRA model persistence, especially with regard to type information storage and attribute handling; (ii) employing more efficient search plan generation for the construction of the RETE network; (iii) reducing code interpretation overhead by precompiling model manipulation sequences into native Java calls.

In additon to improving performance, we plan to provide support for mixing different pattern matching strategies to allow the transformation designer to specify which pattern matcher implementation should be used on a per-pattern basis. Additionally, we plan to investigate the possibilities of adaptive pattern matching strategy change based on automatic profiling.

References

1. Bergmann, G., et al.: Incremental pattern matching in the VIATRA transformation system. In: GRaMoT 2008, 3rd International Workshop on Graph and Model Transformation, 30th International Conference on Software Engineering (accepted, 2008)
2. Matzner, A., Minas, M., Schulte, A.: Efficient graph matching with application to cognitive automation. In: Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007). Springer, Heidelberg (2007)
3. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
4. Varró, G., Horváth, Á., Varró, D.: Recursive graph pattern matching with magic sets and global search plans. In: Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007). Springer, Heidelberg (2007)
5. Geiss, R., et al.: 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)
6. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations: Foundations. World Scientific, Singapore (1997)
7. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. 68(3), 214–234 (2007)

8. 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)
9. Ehrig, H., et al.: Handbook on Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
10. Börger, E., Särk, R.: Abstract State Machines. A method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
11. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Dallas, Texas, USA, pp. 79–88. IEEE Press, Los Alamitos (2005)
12. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE, April 1989, published as Proceedings of the IEEE, vol. 77(4), pp. 541–580 (1989)
13. Jungel, M., Kindler, E., Weber, M.: The Petri Net Markup Language. In: Algorithmen und Werkzeuge fur Petrinetze (AWPN), Koblenz (June 2002)
14. The VIATRA2 Framework: official website (2008), http://viatra.inf.mit.bme.hu
15. Geiss, R., Kroll, M.: On improvements of the Varro benchmark for graph transformation tools. Technical Report 2007-7, Universität Karlsruhe, IPD Goos 12 (2007)
16. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: Proc. International Workshop on Model Transformation in Practice (MTiP 2005) (October 2005)
17. Hearnden, D., et al.: Incremental model transformation for the evolution of model-driven systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
18. Varró, G., Varró, D., Schürr, A.: Incremental graph pattern matching: Data structures and initial experiments. In: Proc. Graph and Model Transformation (GraMoT 2006). Electronic Communications of the EASST, vol. 4 (2006)
19. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1), 17–37 (1982)
20. Bunke, H., Glauser, T., Tran, T.H.: An efficient implementation of graph grammar based on the RETE-matching algorithm. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 174–189. Springer, Heidelberg (1991)
21. Matzner, A., Minas, M., Schulte, A.: Efficient graph matching with application to cognitive automation. In: Proc. 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance, Kassel, Germany, October 2007, pp. 293–308 (2007)
22. Ermel, C., Rudolf, M., Taentzer, G.: The AGG-Approach: Language and Tool Environment. In: [9], pp. 551–603. World Scientific, Singapore (1999)
23. Schürr, A.: Introduction to PROGRES, an attributed graph grammar based specification language. In: Nagl, M. (ed.) WG 1989. LNCS, vol. 411, pp. 151–165. Springer, Heidelberg (1990)
24. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: The 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland. ACM Press, New York (2000)
25. The AGTIVE Tool Contest: official website (2007), http://www.informatik.uni-marburg.de/~swt/agtive-contest

15 Years of Triple Graph Grammars

Research Challenges, New Contributions, Open Problems

Andy Schürr and Felix Klar

Technische Universität Darmstadt, Real-Time Systems Lab,
Merckstr. 25, 64283 Darmstadt, Germany
{schuerr,klar}@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de

Abstract. Triple graph grammars (TGGs) have been invented 15 years ago as a formalism for the declarative specification of bidirectional graph-to-graph translations. In this paper we present a list of still open problems concerning the interpretation and the expressiveness of TGGs. We will comment on extensions proposed to improve the original approach and the drawbacks that arise thereof. Consequently a more precise formalization of compulsory properties of the translation of triple graph grammars into forward and backward graph translation functions is given. Regarding these properties an interpretation and implementation of negative application conditions is derived that does not destroy the benefits of the original approach. Additionally a new demand-driven forward/backward translation rule application strategy is proposed. It guarantees for the first time automatically a correct ordering of rule applications without imposing any additional requirements on the structure of the regarded graphs.

Keywords: triple graph grammars, graph translation, model transformation, rule application.

1 Introduction

The concept of triple graph grammars (TGGs) has been invented in 1993 as a generalization of Pratt's pair grammars [1]. The first publication [2] appeared in 1994; it introduced TGGs as a declarative formalism for the specification of bidirectional translations between different graph languages, sometimes also called exogenous graph or model transformations. TGGs as defined in [2] are grammars that generate languages of graph triples which consist of two related graphs—often called source and target graphs—plus a correspondence graph "between" them. The correspondence graph realizes an explicit representation of correspondence relationships between the nodes of source and target graphs. Any TGG can be compiled into a pair of so-called forward and backward graph translations (FGTs/BGTs) that take either the source or the target graph as input and produce an appropriate target or source graph (plus the needed correspondence relationship graph between them) as output.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 411–425, 2008.

From the very beginning TGGs have been used to specify and implement mappings between software engineering documents with a main focus on model-to-model translations [3,4]. In this application scenario the correspondence relationships play the role of typed traceability links between software engineering artifacts. Since 1994 quite a number of papers have been published that introduced extensions of the original TGG definitions and presented various kinds of applications [5,6,7,8,9]. Furthermore, at least 6 different implementations of TGGs do exist with different properties concerning the functionality and the realization of derived FGTs and BGTs [10,11,9,7,12]. Rather recently, the fundamental ideas of TGGs have even been adopted by the OMG in their model transformation language standard QVT [13]. For a more or less complete survey of all existing TGG publications the reader is referred to [14].

Despite of the fact that quite a number of research groups around the world are nowadays actively developing and using TGGs, some fundamental problems are still unsolved until today:

1. TGGs have been invented to specify mappings between two languages of graphs, but most published approaches either use inefficient graph grammar parsing and/or backtracking algorithms or rely on not very well-defined constraints of processed TGGs such that they are not able to guarantee important properties of derived FGTs/BGTs with their TGG.
2. In practice urgently needed negative application conditions (NACs) of TGG productions are either simply excluded or handled in a way that destroys the fundamental properties of TGGs, i.e. derived FGTs/BGTs may generate graph triples that can't be generated by the original TGG.
3. Finally, appropriate means for modularization, refinement, and reuse of TGGs have not been studied until recently despite of the fact that quite large TGG specifications have already been created and used in industrial case studies.

In this submission we study problems 1 and 2 listed above. For first ideas how to address problem 3 the reader is referred to [15].

The next section introduces a running example of a TGG together with its derived FGT/BGT rules. Furthermore, we use the running example to argue that NACs are needed in practice and to discuss the difficulties how to translate them into appropriate NACs of FGT/BGT rules. In Sect. 3, the basic formal definitions of TGGs are repeated (for reasons of completeness). Furthermore, this section lists some fundamental properties that all TGG approaches developed in the past and the future should respect: consistency, completeness, expressiveness, and efficiency. Section 4 is a survey of related publications that evaluates existing TGG approaches w.r.t. the fundamental TGG properties introduced in Sect. 3. Section 5 then introduces a new control algorithm that automatically determines a proper ordering of forward/backward rule applications even in the presence of NACs using a very efficient eager, demand-driven approach. Section 6 finally summarizes the achievements of this paper and lists a number of still open problems that have to be addressed by future research activities.

2 Informal Introduction of TGGs

This section sketches TGGs and describes the challenges that result from using NACs in TGGs. As a running example we will use a cut-out of the well-known example of the translation of class diagrams (with single inheritance) into relational database schemata. Figure 1 shows a TGG schema that defines the correspondence structure between elements of class diagrams and relational database schemata. Each class together with all its subclasses corresponds to one table. Sequentially ordered class attributes are mapped onto sequentially ordered table columns.

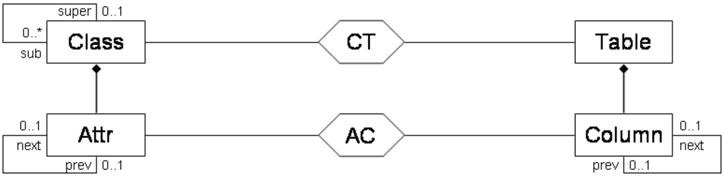


Fig. 1. TGG schema that defines a correspondence structure (center) between elements of class diagrams (left) and relational database schemata (right)

The bidirectional mapping is specified in Fig. 2 by means of five TGG productions.[1] These productions define the simultaneous creation of elements of both languages and the correspondence graph that relates these elements. Correspondence links are denoted as hexagonal nodes that connect elements of both languages. In the following we will sketch the behavior of the TGG productions.

The first production creates the root of a new class hierarchy together with a new table. The second production associates a new subclass of an already existing class with its corresponding already existing table. Productions 3 to 5 deal with attributes and columns such that their sequential order is preserved. Production 3 creates the first attribute of a class and the first column of the related table. Production 4 creates the first attribute of a (sub-)class and a new last column of the related table, whereas production 5 creates another new last attribute and a new last related column. Note that each production of the TGG creates at most one node of the involved graph languages. These nodes labeled with ++ are the *primary nodes* of their productions. Later on we will use the restriction that each TGG production creates only at most one primary node of each graph language to present an efficiently working graph translation algorithm.[2]

The presented TGG clearly shows that NACs are urgently needed in practice. Without the NACs listed in Fig. 2 invalid graphs, as shown in Fig. 3, would be processed as input or created as output of an FGT or BGT. Our running example relates languages of graphs with rather similar constraints such that

[1] TGG productions are also sometimes called TGG rules in other publications.

[2] TGG productions may create additional secondary graph elements if connected with the selected primary node.

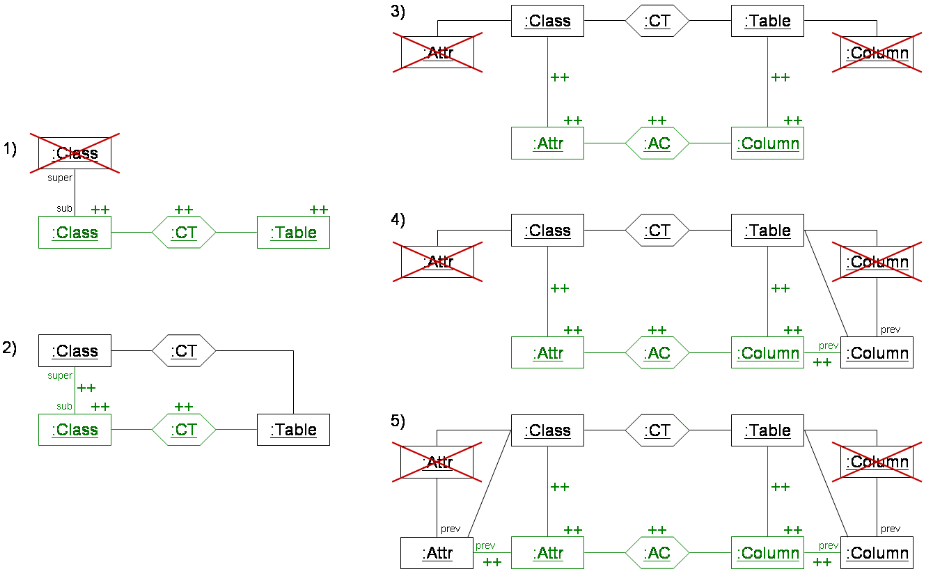


Fig. 2. TGG productions for translating class diagrams and relational database schemata (including negative application conditions)

the introduced NACs are mainly needed to prohibit processing of invalid input graphs that would produce invalid output graphs—with one exception: if we drop the NAC in production 4 that deals with columns then we are able to translate correct class diagrams into incorrect relational database schemata, where one column may have more than one predecessor column. To summarize, TGGs without NACs often do not characterize precisely the language of consistent pairs of connected graphs we are interested in. We might get rid of the NACs of productions 1 through 4 if we introduce production priorities such that production 2 has higher priority than production 1 and production 5 has higher priority than production 4 which has higher priority than production 3. But there is no way to get rid of the NACs of production 5 by means of production priorities. The NAC of this production prohibits the creation of (schema incompatible) graphs, where one attribute or column may have multiple successors.

Our running example also shows that there is no obvious way how to handle NACs of TGG productions, when we derive the corresponding FGT/BGT rules. Let us assume that we are interested in the forward translation of a class diagram into a relational database schema. As discussed in [16] we have two obvious options how to handle the NACs for class diagram elements. The first option is that we simply ignore the negative nodes of the productions, when we generate the related FGT rule. In this case generated FGT rules are able to translate class diagrams that may or may not be derivable by the given TGG into relational database schemata that cannot be derived by the TGG, too. Another option is to simply preserve the NACs of TGG productions, when the corresponding FGT rules are generated. This results in a useless set of FGT rules that even are not

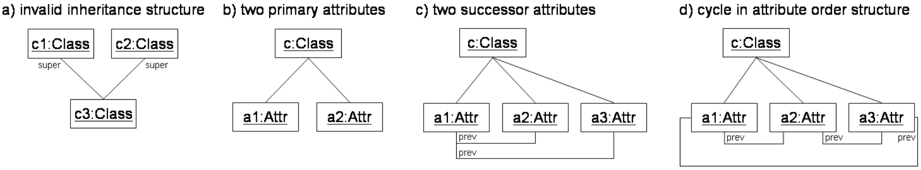


Fig. 3. Examples of invalid input graphs

able to handle attribute and column lists with more than one element. This is due to the fact that the NAC of the generated FGT rule 3 blocks the application of this rule for classes with more than one attribute. As a consequence generated FGT rules have to interpret NACs as follows: a FGT/BGT rule with a NAC is applicable if we either find no match for this NAC in the input graph or if all potential matches of the NAC are not yet translated; therefore, we will present a rule application algorithm later on that keeps track of all already translated elements of an input graph and handles not yet translated elements like non-existing elements. This algorithm thus simulates the creation of the input graph while actually creating the related output graph.

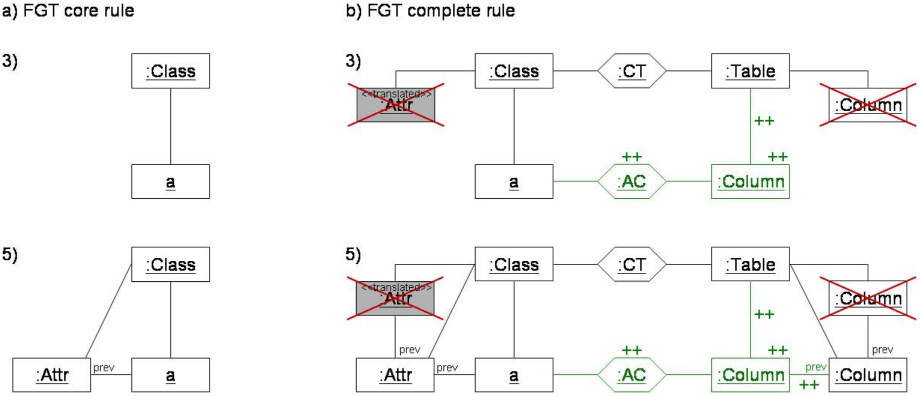


Fig. 4. Examples of core rules and complete rules required for FGT

In order to be able to simulate the creation of the input graph faithfully we have to introduce so-called *core rules* that are used by the algorithm presented in Sect. 5. FGT core rules as presented in Fig. 4 for TGG productions 3 and 5 check whether a regarded fragment of the input graph potentially matches the input part of their related complete FGT rules. As a side effect they identify all elements of the input graph that must have been already translated by other FGT rules before the selected FGT rule can be applied. We call these input graph elements *context elements*. The context element required by rule 3 is a class that is the owner of attribute a. Rule 5 has an additional context element—an attribute of the class that is the owner of attribute a and the direct predecessor of attribute a.

Figure 4 also shows the complete FGT rules derived from TGG productions 3 and 5. The FGT rules consist of a matching part (normal nodes and NACs) and a translating part (nodes marked with ++). The matching part of FGT rule 5 starts at the given attribute `a`. The owning class is retrieved and it is checked whether it has an attribute that is the direct predecessor of attribute `a`. Additionally the owning class must have been already translated into a table. The NAC at the left-hand side ensures that the matched attribute is not the predecessor of an attribute that has already been translated. The input graph might contain a predecessor-relationship nevertheless, but the NAC (with stereotype <<translated>>) ensures that it has not already been translated by a previous call to an FGT rule. Thus the just regarded NAC guarantees that class attributes are processed in the given order. The table must have a last column, i.e. it is not the previous column of any other column of the table. If such a match is found, a new column is added by the translating part as the last column of the table and associated with the given attribute a.

The presented example finally shows that TGGs usually define m-to-n relations and not functions between two graph languages. In our case, e.g. an infinite number of different class hierarchies (without attributes) is mapped onto a single relational database table (without columns). Furthermore, the attributes of an unordered set of n subclasses may be mapped in $n!$ different ways onto a sequentially ordered list of columns of the related table. Nevertheless, FGTs and BGTs derived from a TGG are usually implemented as partial functions that take one graph as input and produce either no result or another graph as output. This is due to the fact that most of the time when TGGs are used in practice we are not interested in the sometimes infinite set of all possible outputs graphs that are related to a given input graph. On the contrary it is usually sufficient if FGTs and BGTs compute an appropriate representative of the set of all output graphs only. In the case of the running example we would expect, e.g., that a BGT computes the smallest class diagram that corresponds to a given database schema; such a BGT maps a given table with columns onto a single class with attributes without using rule 2 shown in Fig. 2.

3 Fundamental TGG Properties and Research Challenges

Motivated by the observations of the last section, we will introduce FGTs/BGTs derived from a given TGG as pairs of partial functions that have to preserve well-defined compatibility properties. Furthermore, we present a compact characterization of TGG-related research activities in the form of a "grand TGG research challenge". We will use these definitions in the following section to evaluate the usefulness of various TGG approaches that have been published in the past. Let us start with the basic definition of TGGs as introduced in [2]:

Definition 1. *Graphs, Graph Morphisms, and Graph Operators.*
A quadruple $G := (V, E, s, t)$ *is a graph with* $elements(G) := V \cup E$*, where*
(1) V *is a finite set of vertices,* E *is a finite set of edges, and*
(2) $s, t : E \rightarrow V$ *are functions assigning sources and targets to edges.*

Let $G := (V, E, s, t), G' := (V', E', s', t')$ be two graphs. A pair of functions $h := (h_V, h_E)$ with $h_V : V \to V'$ and $h_E : E \to E'$ is a graph morphism from G to G', i.e. $h : G \to G'$, iff
(3) $\forall\, e \in E : h_V(s(e)) = s(h_E(e)) \wedge h_V(t(e)) = t(h_E(e))$.

Furthermore, the operators $\subseteq$ for subgraph, $\cup$ for union of graphs with gluing of nodes and edges (with same identifiers), and $\setminus$ for the deletion of the removal of graph elements, are defined as usual, and with $h : G \to G'$ being a morphism, $h(G) \subseteq G'$ denotes that subgraph in G' which is the image of h.

Definition 2. *Monotonic Productions and Graph Rewriting.*

Any tuple of graphs $p := (L, R)$ with $L \subseteq R$ is a monotonic production and p applied to a given graph G produces another graph $G' \supseteq G$, denoted by: $G \overset{p}{\rightsquigarrow} G'$, with respect to redex[3] selecting morphisms $g : L \to G$ and $g' : R \to G'$, iff:

(1) $g' \mid L = g$, i.e. g and g' are identical mappings w.r.t. L.
(2) g' maps new elements of $R \setminus L$ onto unique new elements of $G' \setminus G$.

Definition 3. *Graph Triples.*

Let LG, RG, and CG be three graphs, and $lr : CG \to LG$, $rr : CG \to RG$ are morphisms which represent m-to-n relationships between the left-hand side graph LG and the right-hand side graph RG via the correspondence graph CG in the following way:

$x \in LG$ is related to $y \in RG :\Leftrightarrow \exists\, z \in CG : x = lr(z) \wedge rr(z) = y$.
The resulting graph triple is denoted as follows: $GT := (LG \overset{lr}{\leftarrow} CG \overset{rr}{\rightarrow} RG)$.

Definition 4. *Production Triples and Graph Triple Rewriting.*

Let $lp := (LL, LR)$, $rp := (RL, RR)$, and $cp := (CL, CR)$ be monotonic productions. Furthermore, $lh : CR \to LR$ and $rh : CR \to RR$ are graph morphisms such that their restrictions $lh\mid_{CL}: CL \to LL$ and $rh\mid_{CL}: CL \to RL$ are morphisms, too, which relate the left- and right-hand sides of productions lp and rp via cp to each other. The resulting production triple is denoted as follows:
$p := (lp \overset{lh}{\leftarrow} cp \overset{rh}{\rightarrow} rp)$.
And the application of such a production triple to a graph triple
$GT := (LG \overset{lr}{\leftarrow} CG \overset{rr}{\rightarrow} RG)$
produces another graph triple
$GT' := (LG' \overset{lr'}{\leftarrow} CG' \overset{rr'}{\rightarrow} RG')$,
i.e.: $GT \overset{p}{\rightsquigarrow} GT'$, which is uniquely defined (up to isomorphism) as shown in [2].

Definition 5. *Triple Graph Grammar and Triple Graph Grammar Language.*

A triple graph grammar TGG is a tuple (P, AT), where P is the set of its TGG productions and AT is its axiom graph triple. The language $\mathcal{L}(TGG)$ is the set of all graph triples that can be derived from $AT = (LA \overset{lr}{\leftarrow} CA \overset{rr}{\rightarrow} RA)$ using a finite number of TGG production rewriting steps.

[3] A redex is a subgraph within a host graph matching a production's left-hand side.

The definition of a TGG presented above introduces an axiom graph triple that has been omitted when we presented our running example in Sect. 2. In that case the axiom graph is a triple of three empty graphs. On the other hand, our running example is accompanied by a graph schema definition that is not captured by the formal definitions listed above due to lack of space. The same is true for node attributes, attribute expressions, and NACs used in Sect. 2. These missing parts of the formalization of TGGs can be added easily. Furthermore, they do not have any impact on the following explanations of desirable properties of TGGs and derived FGTs and BGTs in this section.

In Sect. 2 we did already explain how FGT and BGT rules can be derived from the set of productions of a given TGG. Furthermore, the initial TGG publication [2] did already prove that for any graph triple $(LG \overset{lh}{\leftarrow} CG \overset{rh}{\rightarrow} RG)$ that belongs to $\mathcal{L}(TGG)$ we can find a sequence of forward translation rule applications that translates $(LG \overset{la}{\leftarrow} CA \overset{ra}{\rightarrow} RA)$ into $(LG \overset{lr}{\leftarrow} CG \overset{rr}{\rightarrow} RG)$ and a sequence of backward translation rule applications that translates $(LA \overset{la}{\leftarrow} CA \overset{ra}{\rightarrow} RG)$ into $(LG \overset{lr}{\leftarrow} CG \overset{rr}{\rightarrow} RG)$, too. The problem with this proof is that it does not include the presentation of an efficient algorithm that computes the needed sequence of FGT/BGT rule applications for obvious reasons: the graph grammar membership problem is NP-complete even for rather restricted classes of graph grammars. Nevertheless, TGGs have been used in the past rather successfully for solving practical problems by usually using rather simple and efficient FGT/BGT rule application control algorithms. In all cases we are aware of these algorithms are realized in such a way that applied to an input graph they either fail or compute a single graph as output. As a consequence a set of FGT/BGT rules together with an appropriate control algorithm implement a pair of partial FGT/BGT functions on graphs that are compatible with the corresponding TGG in a certain sense. It was and still is always a matter of debate how to define the desired level of compatibility of FGT/BGT functions with their TGGs as well as how to increase the expressiveness of TGG productions without sacrificing needed efficiency and compatibility properties. Based on the considerations listed above we can now formulate the

"*Grand Research Challenge of the Triple Graph Grammar Community*":

A "useful" TGG together with a FGT/BGT rule derivation strategy and the related rule application control strategy should not violate the following four design principles:

1. *Consistency*: whenever an FGT function maps a graph LG onto a graph RG together with a correspondence graph CG, then $(LG \overset{lr}{\leftarrow} CG \overset{rr}{\rightarrow} RG)$ must be an element of $\mathcal{L}(TGG)$. An analogous property holds for the BGT function.
2. *Completeness*: whenever a triple graph $(LG \overset{lr}{\leftarrow} CG \overset{rr}{\rightarrow} RG)$ is an element of $\mathcal{L}(TGG)$ then the FGT function maps LG onto an RG' that is "somehow" equivalent to RG. An analogous property holds for the BGT function.

3. *Efficiency*: derived FGT/BGT functions are efficiently executable in the sense that they have a polynomial space and time complexity $O\left(m \times n^k\right)$ with m = number of rules, n = size of input graph, and k = maximum number of nodes of a rule.
4. *Expressiveness*: TGGs have to support modeling constructs that are urgently needed for solving practical problems and that have been added to regular graph transformations rules decades ago. This includes the support for NACs as discussed earlier on as well as complex attribute conditions etc.

The original TGG approach as introduced in [2] combined with a naive backtracking parsing algorithm fulfills the consistency and completeness properties listed above, but violates the efficiency and expressiveness requirements. As a consequence it cannot and has not been used in practice without any modifications. All TGG approaches that have been implemented in the meantime for solving practical problems circumvented the efficiency problem by using a one or two pass translation algorithm that visits all elements of a regarded input graph in a predefined separately specified order. As a consequence these approaches violate the completeness criteria listed above in the general case. Furthermore, TGG approaches that are more expressive than the original definition in addition tend to violate the above mentioned consistency criteria (cf. Sect. 4).

It is the main contribution of this paper to introduce a new TGG approach that increases the expressiveness of the original approach by adding NACs to TGG productions without violating consistency. Furthermore, the rule application control algorithm introduced in Sect. 5 uses for the first time a demand-driven rule ordering approach. In contrast to all other algorithms we are aware of it does not rely on a predefined order of the elements of an input graph or TGG productions. The presented algorithm translates a given input graph in a single pass into an output graph without any book keeping overhead rather efficiently. The main drawback of the algorithm lies in the fact that we cannot guarantee its completeness for arbitrary TGGs. It is up to future work to first of all define the above introduced completeness criteria more precisely and to characterize then families of TGGs for which we can guarantee the completeness of the new FGT/BGT rule application algorithm.

But please note that the algorithm is complete for the running example used in this paper for the following two reasons: (1) due to heavy usage of NACs there is no pair of FGT/BGT rules that can be applied using redexes that overlap in elements created/translated by the corresponding TGG productions; (2) the presented algorithm implicitly and eagerly tries to maximize the positive context of an applicable rule, i.e. gives rules without NACs a higher priority than rules with NACs. As a consequence we have called the new algorithm an *eager demand-driven rule application control strategy*. Thus, the algorithm always produces an output graph for a given input graph when the presented TGG is able to generate a graph triple that contains these two graphs.

4 Related Work

Based on the characterization of useful TGG approaches in the preceding section we are now prepared to evaluate and assess various forms of TGGs that have been published in the past.

As already mentioned the first TGG publication [2] introduced a rather straight-forward translation of TGG productions into FGT and BGT rules. It relied on the existence of graph grammar parsing algorithms with exponential worst-case space and time complexity. As a consequence a first generation of follow-up publications [3,5] all made the assumption that the regarded graphs have a dominant tree structure and that the three subcomponents of a TGG production possess one and only one primary node. Based on these assumptions a control algorithm is used that simply traverses the tree skeleton of an input graph node by node and selects an arbitrary matching FGT/BGT rule for a regarded node that has a node of this type as its primary node. This algorithm defines FGT/BGT functions that are neither consistent nor complete in the general case. Both properties are endangered by the fact that the selected tree traversal order does not guarantee that rules are applied in the appropriate order. It may happen that the application of a rule fails because one of its context nodes has not yet been processed or that a rule is applied despite of the fact that one of its context nodes has not yet been matched by another rule beforehand.

As a consequence, [16] introduces an algorithm that still relies on a tree traversal, but keeps track of the set of already processed nodes and uses a waiting queue to delay the application of rules if needed. This algorithm defines consistent FGT/BGT functions, but has an exponential worst-case behavior concerning the number of re-applications of delayed rule instances.

Another class of TGG approaches attacked the rule ordering problem in a rather different way [17]. It introduced a kind of controlled TGGs, where each rule explicitly creates a number of child rule instances that must be processed afterwards. As a consequence it is the responsibility of the specifier that the thus explicitly encoded rule application strategy guarantees that rule applications never fail because of the fact that required context nodes have not yet been processed. Thus, one of the main advantages of a rule-based approach is destroyed that basic rules can be added and removed independently of each other and that it is not necessary to encode a proper graph traversal algorithm explicitly.

All publications mentioned so far refrain from the usage of NACs. Some of them even argue that NACs cannot be added to TGGs without destroying their fundamental properties! But, rather recently a whole bunch of application-oriented TGG publications simply introduced NACs without explaining how derived FGT and BGT rules and their rule application strategies look like precisely. The publications even give the reader the impression that NACs can be evaluated faithfully on a given input graph without regarding the derivation history of this graph w.r.t. its related TGG. [9], e.g., explicitly makes the proposal to handle complex graph constraints in this way, whereas [18] and [7] ignore the problems associated with the usage of NACs completely.

Finally, we have to reference [19] as the first publication that studied useful properties of FGTs and BGTs including "invertibility" from a formal point of view. The authors of this paper are interested in pairs of translation relations that are inverse to each other. As a consequence they have to impose hard restrictions on TGGs in order to be able to construct their proofs. Furthermore, the paper has a main focus on consistency, whereas efficiency, expressiveness, and completeness are out-of-scope.

To summarize, after 15 years of TGG research activities we are still not able to handle TGGs with NACs appropriately, i.e. we did not yet find the right compromise between expressiveness of TGG productions and the here defined consistency and completeness properties of derived forward and backward translations. In the following section we will, therefore, propose a new approach how to derive FGT or BGT rules from TGG productions with primary nodes and NACs (as mentioned above) and how to control the order of the application of these rules. This algorithm neither relies on the existence of dominant tree structures nor does it impose the burden on the specifier of a TGG to define the rule application order explicitly. Furthermore, we will show that the resulting TGG approach is more expressive than its predecessors, but nevertheless consistent and efficient. Unfortunately, we are not yet able to guarantee completeness for a precisely defined subset of TGGs with NACs.

5 The Algorithm

In this section we sketch our new batch algorithm that controls the application of forward or backward translation rules derived from a TGG. It is able to handle NACs and does not rely on a user-defined traversal algorithm of the input graph. Please note that we have to omit quite a number of details including the computation of rule matches, and the execution of matching rules.

Procedure `translate` processes every node of the input graph together with its adjacent edges by invoking procedure `translateNode`. It may use any strategy to traverse the input graph to increase the efficiency of the translation process, but its consistency does not rely on a specific strategy. As we will see the algorithm uses recursive calls of the procedure `translateNode` to reorder node visits on demand. Two global sets guarantee that cycles are broken (as present in graph d of Fig. 3) and that nodes of the input graph are only translated once. Additionally `translatedElements`, which contains all nodes and edges that have been translated so far, is important for the handling of NACs.

It is the job of the recursively defined procedure `translateNode` to compute the appropriate order in which rules are applied to their primary node matches in the input graph. Whenever a node of the input graph shall be translated into a new subgraph of the output graph, the procedure in a first step guarantees the following property: all context nodes in the input graph that are potentially needed by any rule that may translate the input node are determined by a core rule match and translated recursively beforehand (if possible).

The algorithm dealing with NACs (in pseudo code)

```
procedure Graph translate(inputGraph: Graph) {
  global outputGraph: Graph;
  global translatedElements: ElementSet = nullSet;
  global justRegardedNodes: NodeSet = nullSet;

  for all nodes in inputGraph {
    translateNode(node);
  }
  if (translatedElements contains all elements of inputGraph)
    print "Valid output graph has been constructed";
  else
    print "Could not construct complete valid output graph";
  return outputGraph;
}

prodecure translateNode(n: Node) {
  if (n in translatedElements or in justRegardedNodes)
    return;
  else {
    add n to justRegardedNodes;
    select all rules that have node of n.class as primary node;
    for all rule in rules {
      compute rule.coreRule match in inputGraph with n as primary node;
      if (at least one coreRule match found) {
        for all contextNode in context elements of coreRule match {
          translateNode(contextNode);
          if (contextNode not in translatedElements)
            break;
        }
        if (translatedElements contains all context elements of coreRule match)
          add rule to appropriateRules;
      }
    }
    for all rule in appropriateRules {
      execute rule for some match and extend outputGraph
        (regarding NACs that must not find a match with translatedElements);
      if (successfully executed) {
        determine all elements of inputGraph that have been translated by the rule;
        add all these elements (including n) to translatedElements;
        break;
      }
    }
    remove n from justRegardedNodes;
  }
}
```

We will now apply our running example to the algorithm and have a look at its interesting parts in detail. In the following we let the algorithm translate a valid input graph that contains a class `c` and three attributes `a1, a2, a3`, i.e. four nodes and five edges (cf. input graph d shown in Fig. 3, but without the edge between `a1` and `a3`). `a1` is the first attribute and `a3` the last one. The visiting order of nodes in procedure `translate` is arbitrary, so let's assume the order is `a2, a3, c, a1`. The first interesting situation is the selection of rules for `a2`. These rules have attributes as primary nodes, so rules 3, 4, and 5 derived from the TGG productions shown in Fig. 2 are selected. The context is now calculated by the according FGT core rules. Every context node is then translated by a recursive call to `translateNode`. In our example FGT core rules 3 and 5 deliver `c` respectively `c` and `a1` as context of `a2`. As the order of context translation is arbitrary, too, we assume that first `c` as delivered by rule 3 is translated. The translation of c is straightforward, as it has no context that must be translated. The only appropriate rule is rule 1, as FGT core rule 2 won't find a superclass of c in the input graph.[4] After the successful execution of the complete FGT rule 1 and translation of c, a1 is translated. Context node c has been translated already and the appropriate rules are 3 and 4. Rule 5 won't find a match as a1 is the first attribute and so has no preceding attribute. Imagine the complete FGT rule 4 is now selected. It cannot translate a1, too—due to the fact that its right-hand side does not match. At least the complete FGT rule 3 is executed successfully. Note that for every translated attribute the translated elements are the attribute itself, the edge to the class and (if not the first attribute) the edge to the preceding attribute. Now the recursive translation of a2's context is complete. The determined appropriate rules are 3, 4, and 5 which are tested in this order in the worst case. Rule 3 won't succeed, as the NAC finds a1 that is attached to c and has been translated already. Rule 4 won't succeed either, as the right-hand side does not match and additionally the NAC finds a1. Finally, the complete FGT rule 5 is executed. It will succeed, because it finds a match in the input graph and the NACs also hold. Now procedure translateNode will return and a3 will be translated. Its context nodes c and a2 have been translated already. The appropriate rules are 3, 4, and 5 again. Rules 3 and 4 won't succeed for the same reasons as mentioned for the translation of a2, but rule 5 will succeed. Finally, all nodes have been translated and translatedElements contains all nodes and edges of the input graph. So the algorithm has successfully constructed a valid output graph.

It is obvious that the above introduced algorithm is rather efficiently (w.r.t. to the definition of efficiency in Sect. 3). Each node of the input graph is processed only once as a primary node of its related set of FGT or BGT rules with one exception: it may happen that the demand-driven translation of some context nodes fails again and again due to the fact that they either cannot be translated at all or due to the fact that some nodes of the output graph are still missing. Assuming that rule applications never fail if they have found a match in the input graph we can simply add another global set of nodes to the algorithm which keeps track of not translatable nodes in the input graph. With the help of this set the algorithm can be modified such that not translatable nodes are only handled once (or we could even abort the execution of the algorithm if we are not interested in partial translation results).

Remains the question whether the presented algorithm implements FGT and BGT functions that are consistent and complete w.r.t. to their TGG. Due to lack of space a precise formal characterization of classes of TGGs that guarantee *consistency and completeness* of implemented FGT and BGT functions is out-of-scope of this paper and we can only sketch arguments and ideas how to answer this question. It is rather obvious that the algorithm implements *consistent* functions due to the fact that we simulate the TGG derivation process of the input graph using the global set translatedElements and directly perform the TGG's derivation process on the output graph. It is also quite obvious that

[4] Rules without context (like rule 1) are automatically added to appropriateRules.

the presented algorithm does not implement *complete* FGT/BGT functions in the general case for the following reasons: The procedure *translateNode* executes an arbitrarily selected rule with an arbitrarily selected match if more than one rule with more than one match can be used to translate a regarded primary node of the input graph. Furthermore, the procedure *translateNode* uses an eager approach that translates context nodes of a just regarded input node as early as possible. As a consequence it may happen that the check of a NAC fails due to the fact that some elements in the input graph have been translated too early. Thus we either have to modify the algorithm presented above such that it is able to explore all derivation alternatives, which would have significant impact on its efficiency, or to find useful classes of TGGs for which we can guarantee the completeness of the developed algorithm that are nevertheless useful in practice.

6 Conclusion

In this paper we gave a retrospect on 15 years of Triple Graph Grammar research activities. Furthermore, we introduced four fundamental properties that all TGGs should fulfill that are used for solving practical problems: consistency, completeness, efficiency, and expressiveness. As usual some of these properties tend to exclude each other, like completeness and efficiency of specified translation processes. The presented new TGG-based graph translation algorithm guarantees consistency and increases the expressiveness of TGGs by handling negative application conditions properly. Furthermore, it reduces the amount of work for the TGG developer by, for the first time, not relying on a predefined ordering of nodes of the input graph or the TGG rules. Unfortunately, using this algorithm we cannot guarantee completeness in the general case. It is our experience from industrial case studies that we can neither afford an exponential worst-case behavior to guarantee completeness of translation functions nor reduce the expressiveness of the regarded classes of TGGs such that we are able to design an efficiently working complete translation algorithm. Nevertheless, it is our plan for future research activities to identify new classes of TGGs with NACs for which we can prove consistency and completeness of efficiently executable forward/backward graph translations. Furthermore, we will study the problem how to add support for negative application conditions to incrementally working graph translation algorithms as presented in [17].

References

1. Pratt, T.W.: Pair Grammars, Graph Languages and String-to-Graph Translations. Journal of Computer and System Sciences 5, 560–595 (1971)
2. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
3. Lefering, M.: Software document integration using graph grammar specifications. In: 6th International Conference on Computing and Information. Journal of Computing and Information, vol. 1, pp. 1222–1243 (1994)

4. Lefering, M., Schürr, A.: Specification of integration tools. In: Nagl, M. (ed.) IPSEN 1996. LNCS, vol. 1170, pp. 324–334. Springer, Heidelberg (1996)
5. Jahnke, J., Schäfer, W., Zündorf, A.: A design environment for migrating relational to object oriented database systems. In: 12th International Conference on Software Maintenance (ICSM 1996), vol. 163 (1996)
6. Becker, S., Haase, T., Westfechtel, B., Wilhelms, J.: Integration tools supporting cooperative development processes in chemical engineering. In: 6th Biennial World Conf. on Integrated Design and Process Technology (IDPT 2002), Pasadena, California, USA, Society for Design and Process Science (2002)
7. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varro, D., Varro-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: MTiP workshop at MODELS 2005 (2005)
8. Guerra, E., de Lara, J.: Attributed typed triple graph transformation with inheritance in the double pushout approach. Technical Report UC3M-TR-CS-2006-00, Universidad Carlos III, Madrid (Spain) (2006)
9. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn, Germany (June 2007)
10. Team MOFLON: MOFLON, http://www.moflon.org/
11. Becker, S.M., Herold, S., Lohmann, S., Westfechtel, B.: A Graph-Based Algorithm for Consistency Maintenance in Incremental and Interactive Integration Tools. Journal of Software and Systems Modeling 6(3), 287–315 (2007)
12. Guerra, E., de Lara, J.: Model view management with triple graph grammars. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 351–366. Springer, Heidelberg (2006)
13. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2005), http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01
14. Real-Time Systems Lab: TGG publication survey. Internet, http://www.es.tu-darmstadt.de/english/research/projects/tgg/publications/
15. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: The 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, pp. 285–294. ACM, New York (2007)
16. Königs, A.: Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation. PhD thesis, Technische Universität Darmstadt (2008)
17. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
18. Grunske, L., Geiger, L., Lawley, M.: A graphical specification of model transformations with triple graph grammars. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 284–298. Springer, Heidelberg (2005)
19. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422. Springer, Heidelberg (2007)

Pattern-Based Model-to-Model Transformation

Juan de Lara[1] and Esther Guerra[2]

[1] Universidad Autónoma de Madrid, Spain
jdelara@uam.es
[2] Universidad Carlos III de Madrid, Spain
eguerra@inf.uc3m.es

Abstract. We present a new, high-level approach for the specification of model-to-model transformations based on declarative patterns. These are (atomic or composite) constraints on triple graphs declaring the allowed or forbidden relationships between source and target models. In this way, a transformation is defined by specifying a set of triple graph constraints that should be satisfied by the result of the transformation.

The description of the transformation is then compiled into lower-level operational mechanisms to perform forward or backward transformations, as well as to establish mappings between two existent models. In this paper we study one of such mechanisms based on the generation of operational triple graph grammar rules. Moreover, we exploit deduction techniques at the specification level in order to generate more specialized constraints (preserving the specification semantics) reflecting pattern dependencies, from which additional rules can be derived.

1 Introduction

Model-Driven Development (MDD) is a software engineering paradigm where models are the core asset. They are used to specify, simulate, test, verify and generate code for the application to be built. Most of these activities include the specification and execution of model transformations, some of them between different languages. The transformation of a model conformant to a meta-model into another one conformant to a different meta-model is called model-to-model (M2M) transformation, and is the topic of this paper.

There are two main approaches to M2M transformation: *operational* and *declarative*. The first one is based on rules or instructions that explicitly state how and when the elements of the target model should be created starting from the elements of the source one. In declarative approaches, a description of the mappings between the source and target models is provided. This description states the relation that should hold between two models rather than how to create and link their elements. Declarative approaches are higher-level than operational ones since they form a compact description of a set of (operational) rules. In addition, they are inherently bidirectional because they do not specify any causality. Thus, they bring together in a single specification forward (i.e. source-to-target) and backward (i.e. target-to-source) transformations.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 426–441, 2008.

The state-of-the-art on declarative M2M transformation notations includes a handful of languages (see Section 5). However, sometimes they lack a formal foundation and analysis techniques able to prove properties of the transformation [1]. In other cases, specifications are not fully declarative and may require a control mechanism or defining a causality between existing elements and those to be created in a given relation [2,3], introducing some degree of operationality.

In this paper, we propose a purely declarative, formal approach to M2M transformation based on *triple patterns* to express the relations between source and target models. These are similar to graph constraints [4] but for triple graphs, made of two graphs related through an intermediate one. Patterns can specify positive (the relation they declare must hold) or negative information (the relation must not hold) and can be constrained by positive and negative restrictions. This high-level specification is compiled into lower-level mechanisms based on triple graph grammar operational rules [3] to achieve forward and backward transformations, as well as to relate two existing models. The compilation is performed in two steps. First, we use deduction rules to derive additional patterns that reflect dependencies and refine existing patterns with negative restrictions. Then, a rule for the chosen transformation direction is derived from each pattern.

The advantages of our technique are the following. First, it is purely declarative, based on patterns and constraints. This contrasts with other declarative approaches (such as Triple Graph Grammars (TGGs) [2,3,5]) where a causality has to be given between the existing elements and the ones that have to be created. As we exploit interactions between patterns, these dependencies are automatically derived. Second, it has a formal foundation that allows the study of the M2M transformation specification, in both declarative (i.e. patterns) and operational (i.e. derived rules) formats. Finally, we have devised deduction techniques, able to derive semantic information from the very patterns. For example, having a positive pattern demanding a certain structure and a negative one forbidding its duplication allows generating two rules: one creating the structure if it is not present, and another one reusing it if it already exists.

Paper Organization. Section 2 introduces triple patterns and Section 3 presents their deduction rules. Then, Section 4 shows how to derive the operational rules. Section 5 presents related work. Finally, Section 6 ends with the conclusions.

2 Specifying Transformations: Triple Patterns

This section introduces the different kinds of triple patterns, their satisfiability and the characteristics of the underlying operational mechanisms. These concepts rely on the notion of triple graph, which we introduce first.

Triple graphs are made of two graphs related through an intermediate one. We can use any graph model for these three graphs, from standard unattributed graphs $(V; E; s, t\colon E \to V)$ to more complex attributed graphs (e.g. E-graphs [4]).

Definition 1 (Triple Graph). *A triple graph* $TrG = (G_s, G_c, G_t, cs\colon V_{G_c} \to V_{G_s}, ct\colon V_{G_c} \to V_{G_t})$ *is made of two graphs* G_s *and* G_t *called source and target, related through the nodes of the correspondence graph* G_c*.*

Nodes in the correspondence graph G_c have morphisms to nodes of the source and target graphs. If $\exists m \in V_{G_c}$ s.t. $x \xleftarrow{cs} m \xrightarrow{ct} y$ we write $x\, rel\, y$. Other kinds of mappings could be used as well, for example the simpler one in [2], where the correspondence functions are graph morphisms or the more complex one in [6] where the correspondence functions can relate edges or be undefined. We use the notation $TrG|_x$ (for $x \in \{s, t, c\}$) to refer to the G_x component of TrG, and write $\langle G_s, G_t\rangle$ for a triple graph with source and target graphs G_s and G_t, and $\langle G_s, \emptyset, G_t\rangle$ for a triple graph with empty correspondence.

Next, we define triple graph morphisms.

Definition 2 (Triple Graph Morphism). *A triple graph morphism* $f = (f_s, f_c, f_t) : TrG^1 \to TrG^2$ *is made of three graph morphisms* $f_x : TrG^1|_x \to TrG^2|_x$ *(with* $x = \{s, c, t\}$*), where* $f_s|_V \circ cs^1 = cs^2 \circ f_c|_V$ *and* $f_t|_V \circ ct^1 = ct^2 \circ f_c|_V$.

Source and target graphs can be typed by a type graph, or more in general by a *meta-model*, which includes inheritance [7]. In the latter case, we use the term *model* instead of graph. Given meta-model MM, $L(MM)$ refers to the set of all valid models conformant to (typed by) it. Similarly, we use the notion of meta-model triple [6] for the typing of triple graphs.

Triple patterns are similar to graph constraints [4,8], but defined on triple graphs. We use them to describe the allowed and forbidden relationships between source and target models. We consider both simple and composite patterns.

Definition 3 (Pattern). *Given triple injective morphism* $C \xrightarrow{q} Q$ *and sets* $N_{Pre} = \{Q \xrightarrow{c_i} C_i\}_{i \in Pre}$, $N_{Post} = \{Q \xrightarrow{c_j} C_j\}_{j \in Post}$ *of negative pre- and post-conditions:*

- $\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \Rightarrow P(Q) \bigwedge_{j \in Post} \overrightarrow{N}(C_j)$ *is a simple pattern (S-Pattern).*
- $\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} \overrightarrow{N}(C_j)$ *is a composite pattern (C-Pattern).*
- $\overrightarrow{N}(C_j)$ *is a negative pattern (N-Pattern).*

Remark. The notation $\overleftarrow{P}(\cdot)$, $\overleftarrow{N}(\cdot)$ and $\overrightarrow{N}(\cdot)$ is just syntactic sugar to indicate a positive pre-condition, a negative pre-condition or a negative post-condition.

An S-Pattern is made of a positive graph Q restricted by negative pre- and post-conditions (Pre and $Post$ sets). Intuitively, Q should be present in triple graph TrG whenever no negative pre-condition C_i is found; and if Q is found, then no occurrence of the negative post-conditions should be found. That is, while pre-conditions express restrictions for the pattern Q to occur, post-conditions describe forbidden graphs. A C-Pattern is an S-Pattern with an additional positive pre-condition graph C. Thus an S-Pattern is a C-Pattern with C and q empty. Finally, an N-Pattern is a C-Pattern where C and Q are empty and there is only one negative post-condition, forbidden to occur.

Definition 4 (M2M Specification). *A M2M specification* $S = \bigwedge_{i \in I} P_i$ *is a conjunction of patterns, where each* P_i *can be simple, composite or negative.*

Remark. For technical reasons, we assume that initially in a specification only N-patterns have negative post-conditions. This is not a restriction, as any post-condition can be expressed as an N-pattern. In fact, a M2M specification is usually made of just N- and S-patterns, from which we automatically derive C-patterns with positive pre-conditions encoding pattern dependencies, and transform N-patterns into post-conditions for the other patterns (see Section 3).

Example. Fig. 1 shows some patterns in an example M2M specification, inspired by the class to relational database transformation [1]. S-Pattern `C-T` states that a C node (a class) that is not connected to another one (i.e. it does not have a parent) should be related to a T (table). S-Pattern `A-Co` states that a C node connected to an A (an attribute), should be related to a T with a Co (column). Differently from TGGs, we don't need to specify here a positive pre-condition stating that a relation between a C and a T should already exist. This dependency is detected by the deduction rules we present in Section 3. S-Pattern `A-Co2` specifies that in the case of two C nodes connected through an R (a directed relation), the associated T node of the source C should have as foreign key (F node) an attribute of the target class. Finally, N-Pattern `notDupF` forbids two Fs between two Ts.

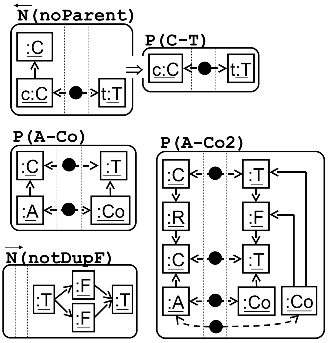


Fig. 1. M2M Specification

Next we define the satisfaction of a pattern. As S- and N-Patterns are special cases of C-Patterns, it is enough to formulate C-Pattern satisfaction.

Definition 5 (Pattern Satisfaction). *Triple graph* TrG *satisfies* $CP = [\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j \in Post} \overrightarrow{N}(C_j)]$, *written* $TrG \models CP$, *iff:*

- CP *is* forward satisfiable, $TrG \models_F CP$*:* $[\forall m^s \colon P_s \to TrG$ *s.t.* $(\forall i \in Pre$ *s.t.* $N_i^s \ncong P_s$, $\nexists n_i^s \colon N_i^s \to TrG$ *with* $m^s = n_i^s \circ a_i^s)$, $\exists m \colon Q \to TrG$ *with* $m \circ q^s = m^s$, *s.t.* $\forall j \in Post \; \nexists n_j \colon C_j \to TrG$ *with* $m = n_j \circ c_j]$,
- *and* CP *is* backwards satisfiable, $TrG \models_B CP$*:* $[\forall m^t \colon P_t \to TrG$ *s.t.* $(\forall i \in Pre$ *s.t.* $N_i^t \ncong P_t$, $\nexists n_i^t \colon N_i^t \to TrG$ *with* $m^t = n_i^t \circ a_i^t)$, $\exists m \colon Q \to TrG$ *with* $m \circ q^t = m^t$, *s.t.* $\forall j \in Post \; \nexists n_j \colon C_j \to TrG$ *with* $m = n_j \circ c_j]$,

with $P_x = C +_{C|_x} Q|_x$, $N_i^x = C +_{C|_x} C_i|_x$ *and* $N_i^x \xleftarrow{a_i^x} P_x \xrightarrow{q^x} Q$ $(x \in \{s,t\})$, *see the left of Fig. 2.* $C +_{C|_x} Q|_x$ *is the pushout object of* C *and* $Q|_x$ *through* $C|_x$*.*

Remark. Morphisms $q^x \colon P_x \to Q$ $(x = \{s,t\})$ uniquely exist due to the universal pushout property (as $C|_x \hookrightarrow C \xrightarrow{q} Q = C|_x \xrightarrow{q_x} Q|_x \hookrightarrow Q$). For the same reason, $a_i^x \colon P_x \to N_i^x$ uniquely exist (as $C|_x \hookrightarrow C \xrightarrow{e_i^s} N_i^x = C|_x \xrightarrow{q_x} Q|_x \xrightarrow{c_i|_x} C_i|_x \xrightarrow{d_i} N_i^x$). Moreover, $b_x^i = c_i \circ q_x$. ■

C-Patterns have a universal quantification, therefore we split them into two directed constraints. For this purpose we demand that in forward satisfaction,

$$\begin{tikzcd}
C_i|_s \arrow[d, "d_i"'] \arrow[dr, phantom, "P.O."] & & C|_s \arrow[ll, "b_s^i"'] \arrow[dl, hook] \arrow[dr, "q_s"] & & \\
N_i^s \arrow[ddrr, bend right, "n_i^s"', "/" marking] \arrow[dr, phantom, "="] & C \arrow[l, "e_i^s"'] \arrow[dr, "c^s"] \arrow[rr, phantom, "P.O."] & & Q|_s \arrow[dl, "p^s"'] & \\
& \arrow[dr, phantom, "="] & P_s \arrow[ull, bend left, "a_i^s"] \arrow[r, "q^s"] \arrow[d, "m^s"] \arrow[dr, phantom, "="] & Q \arrow[r, "c_j"] \arrow[dl, "m"] \arrow[d, phantom, "="] & C_j \arrow[dll, bend left, "n_j", "/" marking] \\
& & TrG & &
\end{tikzcd}$$

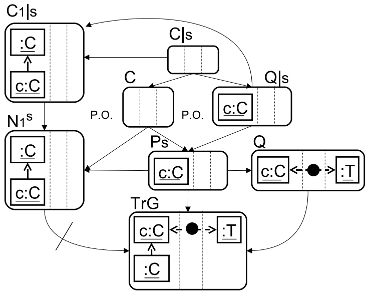


Fig. 2. Forward Satisfaction of Pattern (left). Forward Satisfaction Example (right).

for each occurrence of $P_s = Q|_s +_{C|_s} C = \langle Q|_s, C|_c, C|_t \rangle$ satisfying the negative pre-conditions, an occurrence of Q must be found satisfying the negative post-conditions, see the left of Fig. 2. A positive pattern graph Q is satisfied either because no m^s is found (*vacuous satisfaction*), because m^s and some negative pre-conditions are found (*negative satisfaction*), or because m^s and m are found and the negative pre- and post-conditions are not found (*positive satisfaction*). Note that if the resulting directed negative pre-condition N_i^x is isomorphic to P_x, then it is not taken into account. This is needed as many pre-conditions express a restriction in either source or target but not on both. In addition to forward satisfaction, similar conditions are demanded for the target graph (backwards satisfiability). A graph satisfies specification S if it satisfies all its patterns.

Example. The right of Fig. 2 shows the forward satisfaction of S-Pattern C-T by a triple graph. We have that $TrG \models_F C - T$ as there are two occurrences of m^s, the first one is shown in the figure (upper node C in TrG) and is positively satisfied, while the second (lower C) is negatively satisfied. We also have $TrG \models_B C - T$, as there is just one m^t, positively satisfied. Thus, $TrG \models C - T$.

Please note that the specification does not explicitly state if a class with a parent should be connected with a table or not. An additional pattern could describe such situation. The forward operational mechanism presented in Section 4 does not add such table, as it minimally enforces the specification.

Starting from a specification S, lower level operational mechanisms are derived to perform forward ($\overrightarrow{S}$) and backward transformations ($\overleftarrow{S}$), as well as to relate two existing models ($\overleftrightarrow{S}$, omitted for space constraints, see [9]).

Definition 6 (Operational Mechanisms). *Specification S has the following associated operational transformations:*

- **Forward:** *A function $\overrightarrow{S} : V_S(MM_S) \rightarrow TrG$ with domain $V_S(MM_S) = \{M_s \in L(MM_s) | \exists \langle M_s, X \rangle \models S\}$ s.t. $\forall M_s \in V_S(MM_S)$ $[\overrightarrow{S}(M_s) \models S] \wedge [\overrightarrow{S}(M_s)|_s \cong M_s]$.*
- **Backwards:** *A function $\overleftarrow{S} : V_T(MM_T) \rightarrow TrG$ with domain $V_T(MM_T) = \{M_t \in L(MM_t) | \exists \langle X, M_t \rangle \models S\}$ s.t. $\forall M_t \in V_T(MM_T)$ $[\overleftarrow{S}(M_t) \models S] \wedge [\overleftarrow{S}(M_t)|_t \cong M_t]$.*

The previous definitions are similar to the concept of *correct transformation* given in [10], but in addition we forbid modifying the source (resp. target) model in forward (resp. backwards) transformations.

3 Deduction and Annotation Mechanisms for Patterns

Next we present the deduction rules we use to: (i) generate new patterns that take dependencies into account, which guide the order of pattern enforcement by the operational mechanism; (ii) enrich S- and C-Patterns with pre- and post-conditions derived from other patterns; and (iii) deduce positive information from N-Patterns. We use two operations: *deduction*, which infers new patterns, and *annotation*, which makes dependencies among patterns explicit.

For example, from the specification in Fig. 1, the deduction rules generate a new pattern to reflect the dependency between `C-T` and `A-Co` (to take into account whether a pair (C, T) is already related, before relating a pair (A, Co)). The deduction rules also add negative post-conditions derived from the `notDupF` N-pattern to the rest of patterns, and produce new patterns that reuse part of `notDupF` so that duplication of F objects is not possible.

Most deduction rules are based on the maximal intersection of two triple graphs, called *maximal intersection object* (MIO), which is defined next.

Definition 7 (MIO). *Given triple graphs TrG_1 and TrG_2, a maximal intersection (MI) is given by a span of injective morphisms $(TrG_1 \xleftarrow{m_1} M \xrightarrow{m_2} TrG_2)$, s.t. $M \ncong \emptyset \wedge \nexists\ M' \ncong M$ with $(TrG_1 \xleftarrow{m_1'} M' \xrightarrow{m_2'} TrG_2)$ and $m_{12}\colon M \to M'$ injective s.t. the diagram to the left of Fig. 3 commutes. Object M is called MIO.*

MIOs are not unique, as the example to the right of Fig. 3 shows: M_1 and M_2 are both MIOs, but not M_3 as M_1 is bigger. The set of all MIs (resp. MIOs) of TrG_1 and TrG_2 is denoted by $MI(TrG_1, TrG_2)$ (resp. $MIO(TrG_1, TrG_2)$).

$$\begin{tikzcd}
 & & M' \arrow[dll, "m_1'"'] \arrow[drr, "m_2'"] & & \\
 & \arrow[r, phantom, "="] & M \arrow[u, hook, dashed, "m_{12}"'] \arrow[dl, "m_1"'] \arrow[dr, "m_2"] \arrow[r, phantom, "="] & & \\
m_1'(M') \arrow[dr, hook, bend right] \arrow[r, phantom, "="', near start, shift right=4] & m_1(M) \arrow[l, hook'] \arrow[d, hook] & & m_2(M) \arrow[r, hook] \arrow[d, hook] \arrow[r, phantom, "="', near end, shift right=4] & m_2'(M') \arrow[dl, hook', bend left] \\
 & TrG_1 & & TrG_2 &
\end{tikzcd}$$

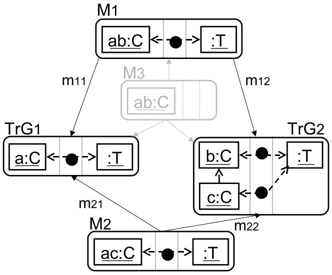


Fig. 3. Conditions for MIO (left). Example (right).

Patterns in a specification may have dependencies inducing a certain order of enforcement by the operational mechanism. We make such dependencies explicit by annotating patterns with additional graphs, related to the positive graph Q.

Dependencies are calculated by the intersection of two patterns, and can be interpreted as restrictions that must not hold when the pattern is enforced.

Definition 8 (Annotated Pattern). *An annotated pattern* $(P, \{n_k \colon D_k \to Q\}_{k \in K})$ *contains a pattern* P *and a set of dependencies* D_k *to* P*'s positive graph* Q*.*

Before presenting the deduction rules, we define an operation called *pre-condition weakening* (PW), which tests whether the positive graph of a C-Pattern is included in another one, and then adds the negative pre-conditions from the former to the latter. We show it here in its simplest form for S-Patterns, see [9] for the complete definition.

Definition 9 (PW). *PW on* $[\bigwedge_{i \in Pre^1} \overleftarrow{N}(C_i^1) \Rightarrow P(Q^1)] \wedge [P(Q^2)]$ *with* $Q^1 \hookrightarrow Q^2$ *results in* $[\bigwedge_{i \in Pre^1} \overleftarrow{N}(C_i^1) \Rightarrow P(Q^1)] \wedge [\bigwedge_{i \in Pre^1} \overleftarrow{N}(C_i^1 +_{Q^1} Q^2) \Rightarrow P(Q^2)]$

Remark. The specification resulting from PW is not equivalent to the original one. The second pattern is added negative pre-conditions, so that it is satisfiable by more graphs, namely by those in which $\exists n^i \colon C_i^1 +_{Q^1} Q^2 \to TrG$ (injective), as then Q^2 is not forced to occur. However, we use this operation to make coherent a specification: as an occurrence of the second pattern implies an occurrence of the first, by adding the negative pre-conditions we ensure that a positive satisfaction of the second implies a positive satisfaction of the first.

Example. As S-Pattern `C-T` is included in `A-Co`, PW adds the negative restriction $\overleftarrow{N}(noParent)$ from the former to the latter. The resulting pattern is shown to the right of Fig. 4 (second row, to the left).

Next, we show some deduction rules that preserve the specification semantics. We only show the simplest forms of them (for S-Patterns), see [9] for additional definitions and proofs. We first present the deduction rule for two S-Patterns called *S-Deduction* and its annotation mechanism $SA(\_,\_)$. S-Deduction creates a new pattern handling an intersection of two S-Patterns, while the annotation mechanism adds such intersections as dependencies to the two original patterns.

Proposition 1 (S-Deduction). *From* $\bigwedge_{k \in \{1,2\}} [\bigwedge_{i \in Pre^k} \overleftarrow{N}(C_i^k) \Rightarrow P(Q^k)]$*, we deduce the new patterns*$\bigwedge_{M \in MIO(Q^1,Q^2)} [\bigwedge_{i \in Pre^1 \cup Pre^2} \overleftarrow{N}(C_i') \wedge \overleftarrow{P}(M) \Rightarrow P(Q^1 +_M Q^2)]$*, where the* C_i' *are calculated as shown to the left of Fig. 4.*

Definition 10 (S-Annotation). *Given two annotated S-Patterns* (P_i, D_i) *with positive graphs* Q^i*:* $SA((P_1, D_1), (P_2, D_2)) = \{(P_i, D_i \bigcup_{M \in MIO(Q^1,Q^2)} \{M \to Q^i\})\}_{i=1,2} \bigcup_{M \in MIO(Q^1,Q^2)} \{(SD(P_1, P_2, M), \emptyset)\}$*, with* $SD(P_1, P_2, M)$ *the resulting pattern from applying S-Deduction using* M*.*

Example. The right of Fig. 4 shows an example of S-Annotation, where the newly generated pattern (bottom right) considers the fact that the relation demanded by pattern `C-T` may already exist. The added dependencies (D_1) ensure

$$\begin{tikzcd}
 & & M \arrow[dll, "m_1"'] \arrow[drr, "m_2"] & & \\
Q^1 \arrow[drr] \arrow[dd, "c_i^1"'] & & P.O. & & Q^2 \arrow[dll] \arrow[dd, "c_j^2"] \\
 & & Q^1 +_M Q^2 \arrow[dl, "P.O."'] \arrow[dr, "P.O."] & & \\
C_i^1 \arrow[r] & C_i' & & C_j' & C_j^2 \arrow[l]
\end{tikzcd}$$

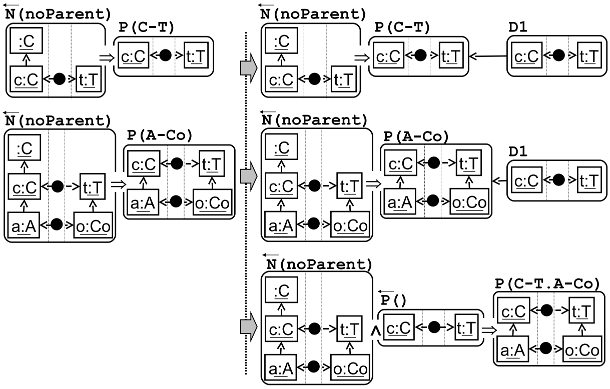


Fig. 4. Negative Pre-Conditions in S-Deduction (left). S-Annotation Example (right).

that the first and second patterns will only be enforced by the operational mechanisms when no occurrence of D_1 is found. As we will see later, this makes the TGG operational rules generated for the first two patterns mutually exclusive with the one of the third, as well as confluent. Moreover, the rule for the third pattern will be able to reuse the structure created by the rule of the first.

Next deduction rule is used to take into consideration the interaction of N-patterns, which express global negative constraints, with other patterns.

Proposition 2 (N-Deduction). $[\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \;\Rightarrow\; P(Q) \bigwedge_{j \in Post} \overrightarrow{N}(C_j)] \wedge [\overrightarrow{N}(C_N)]$ *is equivalent to* $[\bigwedge_{i \in Pre} \overleftarrow{N}(C_i) \Rightarrow P(Q) \bigwedge_{j \in Post} \overrightarrow{N}(C_j) \bigwedge_{C_r \in RS} \overrightarrow{N}(C_r)] \wedge [\overrightarrow{N}(C_N)]$ *with* $RS = \{r^n : Q \to C_r\}_{C_r \in PO(MI(Q,C_N))}$ *and* $PO(MI(Q, C_N))$ *is the set of pushout objects of all spans in* $MI(Q, C_N)$.

Proof(Sketch). We have related C_N in all possible (maximal) ways with Q, which is given by the pushout of each span in $MI(Q, C_N)$. This is similar to the procedure to convert a graph constraint into a post-condition [4,8]. ■

Remark. Removing $\overrightarrow{N}(C_N)$ does not yield an equivalent specification, as e.g. a graph with no occurrence of Q is allowed to have an occurrence of C_N. Note however that we will delete N-Patterns when generating the TGG operational rules, as these by construction cannot generate any forbidden pattern.

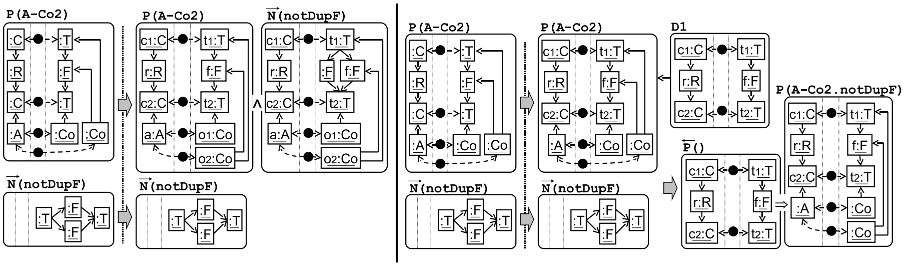


Fig. 5. N-Deduction Example (left). NP-Deduction and Annotation Example (right).

Example. The left of Fig. 5 shows how N-Pattern `notDupF` induces a negative constraint on S-Pattern `A-Co2`, resulting in the S-Pattern to its right. There are two isomorphic MIOs (both made of two Ts and one F) resulting in two isomorphic negative constraints, so that one is eliminated.

The following deduction rule detects N-Patterns that forbid a repetition of structures and generates a positive pattern that reuses such structure. First, we define the completion of a triple graph M with respect to a graph T such that $M \hookrightarrow T$. The completion adds to $M|_t$ all elements that are related to elements of $M|_s$ and belong to $T - M$, and similar for source elements. In addition, completion includes all unrelated elements of T.

Definition 11 (Completion). *$C(M,T) = G$ iff G is the smallest graph s.t. $M \hookrightarrow G \hookrightarrow T \wedge (\forall n \in V_{G|_s}, \nexists m \in V_{T|_t} - V_{G|_t} s.t.\ n\ rel\ m) \wedge (\forall x \in V_{G|_t}, \nexists y \in V_{T|_s} - V_{G|_s} s.t.\ y\ rel\ x) \wedge (\nexists z \in (V_{T|_s} \cup V_{T|_t}) - (V_{G|_s} \cup V_{G|_t})$ s.t. z is unrelated). G also contains all edges of T with source and target in nodes of G.*

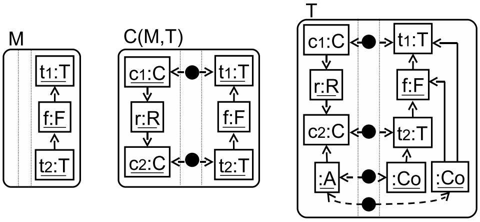


Fig. 6. Example of Completion

Example. Fig. 6 shows an example of completion, where graph `M` is completed with respect to graph `T`, yielding graph `C(M, T)`. Note that $M \hookrightarrow C(M,T) \hookrightarrow T$.

Proposition 3 (NP-Deduction). $[\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \Rightarrow P(Q)] \wedge [\overrightarrow{N}(S)]$, *with S the*

pushout of two isomorphic graphs $S_1 \cong S_2$ and $S_1 \in MIO(Q,S)$, is equivalent to $[\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \Rightarrow P(Q)] \wedge [\overrightarrow{N}(S)] \wedge [\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C(S_1,Q)) \Rightarrow P(Q)]$.

Proof. $C(S_1,Q) \hookrightarrow Q$, thus $[\overleftarrow{P}(C(S_1,Q)) \Rightarrow P(Q)]$ is subsumed by $P(Q)$. ■

The NP-Deduction rule has an associated annotation rule $NP(\_,\_)$, which adds a dependency to the S-Pattern equal to the positive pre-condition of the newly generated pattern (see details in [9]).

Example. The right of Fig. 5 shows the derivation of C-Pattern `A-Co2.notDupF` from `A-Co2` and `notDupF`. The latter is made of the pushout of two isomorphic graphs made of two Ts and one F, which belongs to `MIO(A-Co2, NotDupF)`. The completion of one of the isomorphic graphs with respect to `A-Co2` is the pre-condition graph $\overleftarrow{P}()$ of `A-Co2.notDupF`. The newly generated pattern reuses two Ts and one F so that the rule to be generated from it will not produce the situation forbidden by `notDupF`. The annotation procedure adds a dependency to `A-Co2` so that the generated rule will be mutually exclusive with the one for the deduced pattern.

4 Generating the Operational Rules

Next we show the generation of operational TGG rules from a specification. Compilation into other formalisms is also possible, e.g. to a constraint satisfaction problem [11]. We first present the structure of non-deleting TGG rules.

Definition 12 (Non-Deleting Oper. TGG Rule). *A TGG rule* $r = (L \xrightarrow{l} R, pre = \{n_i\colon L \to N^i_L\}_{i\in I}, post = \{n_j\colon R \to N^j_R\}_{j\in J})$ *is made of an injective morphism* l *of triple graphs, and sets* pre *and* $post$ *of negative pre- and post-conditions.*

Next we show how to generate a TGG rule given an annotated C-Pattern. The main idea is to use $P_s = C +_{C|_s} Q|_s = \langle Q|_s, C|_c, C|_t\rangle$ as the LHS (for the forward rule) and Q as the RHS. The negative pre- and post-conditions of the C-Pattern are used as negative pre- and post-conditions of the rule. Note the similarities with the satisfiability of patterns (Def. 5 and Fig. 2). The rule's RHS is used as a negative pre-condition so that satisfiability is enforced only once. Finally, dependencies are converted into negative pre-conditions.

Definition 13 (Derived TGG Rule). *Given annotated pattern* $T = (\bigwedge_{i\in Pre} \overleftarrow{N}(C_i) \wedge \overleftarrow{P}(C) \Rightarrow P(Q) \bigwedge_{j\in Post} \overrightarrow{N}(C_j), D = \{n^k\colon D_k \to Q\}_{k\in K})$*, the following TGG operational rules are derived:*

- **Forward.** $\overrightarrow{r_T}$: $(L = \langle Q|_s, C|_c, C|_t\rangle \xrightarrow{(id,q_c,q_t)} R = Q, pre = \{L \xrightarrow{n} R\} \cup \{a^s_i\colon L \to N^s_i | L \ncong N^s_i\}_{i\in Pre} \cup \{s^k\colon L \to S^k\}_{k\in K}, post = \{n_j\colon R \to C_j\}_{j\in Post})$.
- **Backwards.** $\overleftarrow{r_T}$: $(L = \langle C|_s, C|_c, Q|_t\rangle \xrightarrow{(q|_s,q|_c,id)} R = Q, pre = \{L \xrightarrow{n} R\} \cup \{a^t_i\colon L \to N^t_i | L \ncong N^t_i\}_{i\in Pre} \cup \{s^k\colon L \to S^k\}_{k\in K}, post = \{n_j\colon R \to C_j\}_{j\in Post})$.

where $N^x_i \cong C_i|_x +_{C|_x} C$*, and* $a^x_i\colon L \to N^x_i$ *is uniquely determined (see Fig. 2, where* $P_x = L$*).* S^k *is the left-extension of* D_k*, see left of Fig. 7, where* $n^k \circ b^k = r \circ l^k$ *and* $d^k \circ b^k = s^k \circ l^k$ *are pullback and pushout squares respectively.*

$$\begin{tikzcd}
B_k \arrow[r, "b^k"] \arrow[d, "l^k"'] \arrow[dr, phantom, "P.B."] & D_k \arrow[d, "n^k"] \arrow[ddr, bend left, "d^k"] & \\
L \arrow[r, "r"] \arrow[drr, bend right, "s^k"'] & R = Q \arrow[dr, phantom, "P.O."] & \\
 & & S^k
\end{tikzcd}$$

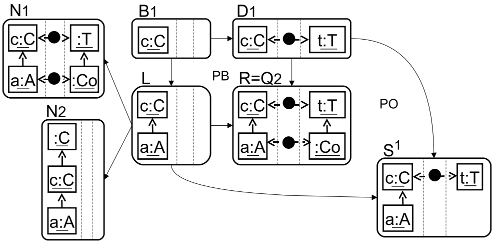


Fig. 7. Left Extension of $D_k \to Q$ (left). Generated Forward Rule A-Co (right).

Example. The right of Fig. 7 shows the generated forward rule from the annotated pattern A-Co of Fig. 4. Note how the NAC S^1 forbids applying the rule if the node C has an associated T. In this case, the rule generated from the derived pattern C-T.A-Co in Fig. 4 would be applicable (see rule C-T.A-Co in Fig. 8).

Before generating the rules we use the deduction and annotation mechanisms on the initial M2M pattern specification in order to transform N-patterns into negative post-conditions of the other patterns, generate patterns that take into consideration the satisfaction of other patterns, and identify dependencies between them. As stated before, we assume that the initial specification does not include patterns with both a positive graph and a negative post-condition.

Definition 14. *[Generation of Operational TGG Rules] Given specification S:*

1. *Use PW (Def. 9) on all possible patterns.*
2. *Use S-Annotation (Def. 10) for each pair of S-Patterns.*
3. *Use NP-Annotation on all possible patterns, initial and derived.*
4. *Use N-Deduction (Prop. 2) on all possible patterns and eliminate N-Patterns.*
5. *Take each derived pattern, and add to it all dependencies of the patterns it was derived from. Do not add such dependencies if they are included in the positive pre-condition of the derived pattern, as the pattern would be useless.*
6. *Generate an operational TGG rule for each causal pattern (Def. 12).*

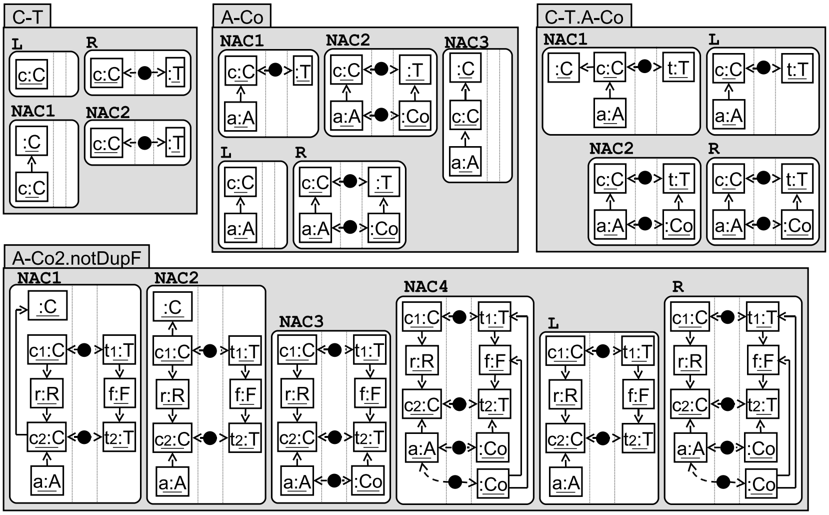


Fig. 8. Some of the Generated Forward Operational Rules

Example. Fig. 8 shows some of the generated forward rules. Rule `C-T` is generated from pattern `C-T`. NAC1 results from a pre-condition, while NAC2 is equal to the RHS. Rule `A-Co` results from pattern `A-Co`. NAC3 comes from the PW operation with pattern `C-T`, NAC2 is equal to RHS, and NAC1 is derived from a dependency when making S-Deduction with `C-T`. Rule `C-T.A-Co` is generated from a pattern derived from `C-T` and `A-Co` through S-Deduction. Its first NAC comes from a dependency induced by their source patterns. Finally, rule `A-Co.notDupF` results from NP-Deduction (see the right of Fig. 5), where NAC1 and NAC2 come from pre-conditions of the patterns from which it is derived, and NAC3 comes from a dependency. These two last rules have some additional NACs (not shown), stemming from N-Deduction with pattern `notDupF`. The procedure generates a total of 10 rules (see [9]).

4.1 Correctness of the Operational Mechanisms

Now we show the correctness of the generated rules, focussing on forward rules as a similar reasoning holds for the backwards case. The generated rules: (i) produce models satisfying the specification, (ii) are confluent, (iii) terminate, and (iv) transform each source model for which there is a correct target model. We give an intuition, see [9] for more details.

(i) follows from the construction of the TGG rules. Their LHS is $\langle Q|_s, C|_c, C|_t\rangle = C +_{C|_s} Q|_s = P_s$, which is the *base* graph from which forward satisfaction is checked (see Fig. 2). As $R = Q$, morphism $m\colon Q \rightarrow TrG$ exists after the application of the rule. The rule negative pre- and post-conditions are derived from the negative pre- and post-conditions of the pattern. Thus, the rule can be applied iff the base morphism m^s exists and the negative pre- and post-conditions of the pattern are satisfied. The additional NAC $\cong$ R makes the rule enforce the pattern once. As initially all forbidden graphs are expressed as N-Patterns, and we have performed N-Deduction, no rule can produce a forbidden result. Since we start with an empty target graph, backwards satisfaction is also obtained.

(ii) follows because S-annotation adds dependencies (which are transformed into NACs) to the initial patterns, and these are appropriately propagated to their derived patterns in step 5 of the rule generation process. Note, however, that initially we may have patterns included in others: $[P(Q_1)] \wedge [P(Q_2)]$ with $Q_1 \hookrightarrow Q_2$. In this case S-Deduction generates $[P(Q_1)] \wedge [P(Q_2)] \wedge [P(M) \Rightarrow P(Q_1 +_M Q_2)]$ (assuming just one MIO), from which we generate three rules. There is a conflict between the first two rules (i.e. a critical pair). However in a situation where both the first and the second are applicable (e.g. if we have $Q_2|_s \hookrightarrow TrG$), applying the first and the third is equivalent to applying the second. Besides, we cannot apply the first and the second due to the generated NACs. Thus, even in this case, rules are confluent (see [9]).

Example. Consider the rules for the patterns `C-T` and `A-Co` and the and their derived pattern (`C-T.A-Co` see Fig. 8). Assume a situation where both `C-T` and `A-Co` are applicable. If `C-T` is applied first, then `A-Co` is disabled, but `C-T.A-Co` can be applied. If `A-Co` is applied first, then no other rule is applicable. However, in both cases we reach the same result.

(iii) follows from the fact that (a) each rule has its RHS as a NAC, therefore it can only be applied once for each initial match in the source model; and (b) a forward rule only changes the target model.

(iv) cannot be achieved for arbitrary M2M specifications. We restrict to what we call *Injective Positive Specifications*, which contain enough positive patterns to produce the operational TGG rules. Next definition introduces the forward case (FIP), the backwards one is similar.

Definition 15 (FIP Spec.). *Specification* $S = \bigwedge_{i=1..n} T_i$ *is FIP, iff* $\forall M_s \in L(MM_s)$ *s.t.* $\exists TrG = \langle M_s, X\rangle \models S, \exists k_i \in \mathbb{N}, \exists P_s^i \leftarrow S_{uv}^{ij} \rightarrow P_s^j$ *with* $P_s^m = C^m +_{C^m|_s} Q^m|_s$, $u = \{1..k_i\}$, $v = \{1..k_j\}$ *and* $S_{uv}^{ij} \not\cong P_s^i$ *if* $i = j$, *s.t.* G *is the colimit of the diagram to the left of Fig. 9 (with all arrows injective) with* $G \hookrightarrow TrG$ *and* $G|_t \cong TrG|_t$.

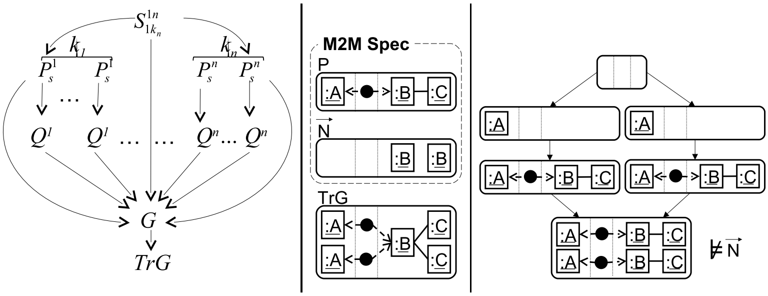


Fig. 9. Condition for FIP (left). Non-FIP Specification (center). Invalid Graph (right).

Remark. The definition considers k_i occurrences of each pattern T_i. Two occurrences of patterns T_i and T_j can overlap, and this is modelled by S_{uv}^{ij}. We forbid P_s^i be the overlap of two occurrences of the same pattern Q^i, as the operational mechanism minimally enforces each pattern (i.e. rules have a NAC equal to the RHS). We have made a simplification in the diagram, but each occurrence of T_i should satisfy its negative pre- and post-conditions.

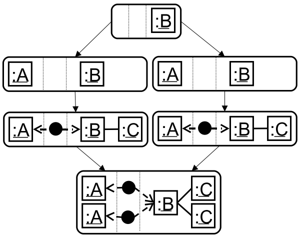


Fig. 10. FIP Specification

Example. Consider the specification in the center of Fig. 9, and assume no deduction is performed. There is a valid triple graph TrG with two As in its source, but the rules generated without deduction cannot create such graph, as they would produce two Bs.

NP-Deduction can turn some non-FIP specifications into FIP, because it creates a new pattern that reuses an already created structure. The right of Fig. 9 shows that if NP-Deduction is not applied, we cannot handle a graph with two As. Fig. 10 shows that after NP-Deduction the resulting pattern can handle such graph as it reuses a B and is applied twice. It is up to future work to determine further deduction rules to cover additional non FIP-specifications.

5 Related Work

Some declarative approaches to M2M transformation use a textual syntax, e.g. PMT [12] or Tefkat [13]. These two particular notations are uni-directional, whereas we generate forward and backward transformations.

Among the visual declarative approaches, a prominent example is the QVT-relational language [1]. The relations may include *when* and *where* clauses that identify pre- and post-conditions and can refer to other relations. From this specification, executable QVT-core is generated that performs forward/backward transformations. This approach is similar to ours, but we compile our patterns to TGG rules, allowing the analysis of the generated transformation [4]. Besides, we can analyse at a higher-level (i.e. pattern level) as our patterns have a formal foundation [9]. Moreover, we automatically detect pattern dependencies and perform pattern inference. In QVT-relations dependencies must be made explicit in the *when* and *where* clauses, and there is no equivalent to our N-Patterns. An attempt to formalize QVT-core is found in [14].

In [15], transformations are expressed through positive patterns that rely on OCL constraints, but no operational mechanism is given. In BOTL [16], the mapping rules use a UML-based notation that allows reasoning about applicability or meta-model conformance. We can reason both at the specification and operational levels.

TGGs [3] formalize the synchronized evolution of two graphs through declarative rules. From this specification, low-level operational TGG rules are derived to perform forward and backward transformations, as well as to relate two existing graphs. We also generate these operational rules from our patterns. However, whereas in declarative TGG rules dependencies must be made explicit (i.e. we must say which elements should exist and which ones are created), in our patterns this information is derived. For instance, in TGGs, a rule like pattern `C-T.A-Co` has to be specified, it is not enough to give `C-T` and `A-Co`.

Although inspired by TGGs, our patterns are a different approach: patterns specify relations, not rules. Similar to graph constraints [8,17], a M2M specification by patterns describes a language of valid triple graphs. Moreover, TGGs have some limitations. First, they allow neither specifying negative information, nor deriving positive information from negative one (like NP-Deduction). In [5], the lack of negation is alleviated by assigning execution priorities to rules. However, this is insufficient to simulate general application conditions, it has an operational nature, and implies knowing the rule generation mechanism and execution engine. Second, a control mechanism is needed to guide the execution of the operational rules, such as priorities [5] or their coupling to editing rules [2]. One can see TGGs as a subset of our approach, where a TGG rule is a pattern of the form $\overleftarrow{P}(L) \Rightarrow P(R)$ without negative conditions or deduction techniques.

In [18], an algorithm is given for the derivation of declarative TGGs from example pairs of models. Interestingly, the user does not have to specify the correspondence nodes in these pairs. The employed techniques resemble our use of MIOs, but our patterns are richer, allowing negative pre- and post-conditions, and our theory supports further derivation techniques (e.g. NP-Deduction).

With respect to graphical patterns, in [17] a logic of constraints is proposed, in which constraints are existentially satisfied, while ours are universal. Moreover, we provide deduction techniques specially tailored for M2M specifications and triple patterns. In [19] we presented a simpler notion of pattern and used it to extend normal rules to synchronous TGGs. We applied it to the synchronization of the concrete and abstract syntax of visual models. The patterns were restricted to work with positive information, and the execution of the derived rules was associated to editing rules (like traditional TGGs). Here we present a new concept of pattern, which allows expressing negative conditions, introduce deduction rules and present a new algorithm for TGG rule derivation that is suitable for M2M transformation and does not need a normal rule to start with.

6 Conclusions and Future Work

In this paper we have presented a new formal approach to declarative M2M transformation. Relations between source and target models are expressed as different kinds of patterns, from which operational TGG rules are derived implementing forward/backwards transformations and taking into account pattern interactions. This is done by deduction mechanisms that detect interdependencies and produce new patterns that reuse structures created by other patterns. This is one of the strengths of the present work: pattern dependencies are automatically calculated and not explicitly given by the designer such as with QVT and TGGs.

We have already identified analysis properties, both at the specification (e.g. language covering, pattern conflicts) and operational levels (e.g. hippocratic transformations [10]). We have omitted them by lack of space (see [9]).

Although we generate operational TGG rules from a pattern specification, other target formalisms could be used as well (e.g. OCL, Alloy). In fact, one of our next goals is expressing a specification in terms of a constraint satisfaction problem, in the lines of [11]. This would eliminate some problems of the compilation into rules, such as the restriction to handle FIP specifications only. Note that with the theory presented so far we can handle attributes, but not attribute conditions or computations. Our aim is to use OCL and the analysis techniques we proposed in [11].

It would be interesting to extend the set of derived operational rules to handle incremental synchronization and change propagation. More complex patterns able to deal with recursion or having parameters are also under consideration. Finally, we aim to formalize a part of QVT using this technique.

Acknowledgements. We thank the reviewers for their useful comments. Work supported by the Spanish Ministry of Education and Science, projects TSI2005-08225-C07-06 and TIN2006-09678.

References

1. QVT (2005), http://www.omg.org/docs/ptc/05-11-01.pdf
2. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)

3. 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)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Springer, Heidelberg (2006)
5. Königs, A.: Model transformation with Triple Graph Grammars. In: MTiP 2005 (2005)
6. Guerra, E., de Lara, J.: Event-driven grammars: Relating abstract and concrete levels of visual languages. In: ICGT 2004, pp. 317–347 (2007)
7. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. TCS 376(3), 139–163 (2007)
8. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph rewriting - a constructive approach. ENTCS 2 (1995)
9. de Lara, J., Guerra, E.: Pattern-based model-to-model transformation: Long version. arXiv:0804.4745v1 [cs.SE] (2008), http://arxiv.org/abs/0805.4745v1
10. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
11. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Analysing graph transformation rules through OCL. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063. Springer, Heidelberg (2008)
12. Tratt, L.: A change propagating model transformation language. JOT 7(3), 107–126 (2008)
13. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
14. Greenyer, J.: A study of model transformation technologies: Reconciling TGGs with QVT. Master's thesis, University of Paderborn (2006)
15. Akehurst, D.H., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
16. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. ENTCS 72(3) (2003)
17. Orejas, F., Ehrig, H., Prange, U.: A logic of graph constraints. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 179–198. Springer, Heidelberg (2008)
18. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations and application scenarios. Tech. Rep. TR-RI-07-284, U. Paderborn (2007)
19. de Lara, J., Guerra, E., Bottoni, P.: Triple patterns: Compact specifications for the generation of operational triple graph grammar rules. In: GT-VMT 2007. Electronic Communications of the EASST, vol. 6 (2007)

Adaptive Star Grammars for Graph Models

Frank Drewes[1], Berthold Hoffmann[2], and Mark Minas[3]

[1] Umeå universitet, Sweden
[2] Universität Bremen, Germany
[3] Universität der Bundeswehr München, Germany

Abstract. Adaptive star grammars generalize well-known graph grammar formalisms based on hyperedge and node replacement while retaining, e.g., parseability and the commutativity and associativity of rule application. In this paper, we study how these grammars can be put to practical use for the definition of graph models. We show how to use adaptive star grammars to specify *program graphs*, models of object-oriented programs that have been devised for investigating refactoring operations. For this, we introduce notational enhancements and one proper extension (application conditions). The program graphs generated by the grammar comprise not only the nested composition of entities, but also scope rules for their declarations. Such properties cannot easily be defined by meta-models like Uml class diagrams. In contrast, adaptive star grammars cover several aspects of class diagrams.

1 Introduction

Graphs have always been used for visual communication, in science and beyond. In computer science, graph models gained a new quality since the Unified Modeling Language Uml emerged: when models drive the development of object-oriented software, they must be processed by computers, and need to be defined precisely. In this paper we investigate how graph models can be specified by graph grammars. As a case study, we choose program graphs, a language-independent model for object-oriented programs that has been devised for defining and analyzing refactoring operations [15,20]. Program graphs have a particular structure: They are composed in a nested fashion according to a context-free syntax, and contain references from entities to their declarations that respect specific scope rules. We define program graphs by adaptive star grammars [4], an extension of the well-known hyperedge and node replacement grammars [11]. We further extend these grammars to enhance the definition and understanding of rules. One of these extensions, application conditions, increases the generative power of the formalism. Without it, the syntax and scope rules of program graphs could probably not have been defined completely. When we compare the grammar to common model definitions by meta-models, it turns out that the grammar allows to infer the structural information about incidences and multiplicities of edges that is specified, e.g., in Uml class diagrams. The syntax and scope rules for program graphs, however, could only be specified by logical predicates, e.g., in the object constraint language Ocl of Uml.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 442–457, 2008.

In Section 2, we recall the definition of adaptive star grammars and add some extensions that enhance their practical use as a specification language. We introduce program graphs in Section 3, explain their adaptive star grammar, and describe their general properties and parsing. In Section 4, we discuss the relation of our grammatical formalism to object-oriented meta-models. We conclude with a discussion of related and future work in Section 5.

2 Adaptive Star Grammars and Their Extensions

Graph grammars generalize the idea of Chomsky grammars to graphs: A set of graph transformation rules defines how the graphs of the language can be derived by applying them successively to a given start symbol.

Node replacement and hyperedge replacement [11] have been studied most thoroughly as grammars for deriving graph languages. Their rules remove a nonterminal node, and attach a replacement graph to its neighbor nodes. The sort (i.e., the label) and the direction of the edge connecting a neighbor to the nonterminal determine completely how the neighbor is attached to the replacement graph; in hyperedge replacement, the number of neighbors is even fixed for every nonterminal. Both formalisms can specify context-free compositions of graphs in the sense of [3], but fail to define simple languages such as the language of all graphs. Adaptive star grammars [4] overcome these limitations by means of a cloning mechanism that makes its rules more powerful.

2.1 Adaptive Star Grammars

Let us briefly define the concepts needed; for more detailed definitions, see [4].

Graphs. Let $\mathbf{S}$, the set of *sorts*, be the disjoint union of countably infinite disjoint sets $\dot{\mathbf{S}}$ and $\bar{\mathbf{S}}$, which will be used as node and edge labels, resp. For the sake of clarity, subsets S of $\mathbf{S}$ may be specified as pairs $(\dot{S}, \bar{S}) = (S \cap \dot{\mathbf{S}}, S \cap \bar{\mathbf{S}})$.

A *graph* $G = \langle \dot{G}, \bar{G}, s_G, t_G, \dot{\ell}_G, \bar{\ell}_G \rangle$ consists of

- finite sets $\dot{G}$ and $\bar{G}$ of *nodes* and *edges*, resp.,
- *source* and *target functions* $s_G, t_G \colon \bar{G} \to \dot{G}$, and
- functions $\dot{\ell}_G \colon \dot{G} \to \dot{\mathbf{S}}$ and $\bar{\ell}_G \colon \bar{G} \to \bar{\mathbf{S}}$ assigning a sort to each node and edge, resp.

For a node x in G, the subgraph consisting of x and its adjacent nodes and incident edges is denoted by $G(x)$. A *border node of* x is a node in $\dot{G}(x) \setminus \{x\}$. In addition, we use common terminology regarding graphs, such as subgraph, disjoint union and isomorphism, assuming that the reader is familiar with them.

Rules and Replacement. Adaptive star grammars are based upon a simple kind of graph transformation that replaces a subgraph of the form $G(x)$ by another graph. For this, define a *rule* $r = \langle y, R \rangle$ to be a pair consisting of a graph R and a distinguished node $y \in \dot{R}$. We call $R(y)$ and $R \setminus \{y\}$ the left- and right-hand side of r, resp.

Let G be a graph and $x \in \dot{G}$ such that $G(x) \cong_g R(y)$ for some isomorphism g. Then the graph $H = G[x \,/_g\, r]$ is obtained from the disjoint union of G and R by identifying $R(y)$ with $G(x)$ according to the isomorphism g and removing x and its incident edges. In the following, we simply write $H = G[x/r]$ instead of $G[x \,/_g\, r]$. (But note that x and r do not necessarily determine g uniquely, and we may have $G[x \,/_g\, r] \ncong G[x \,/_h\, r]$ for $h \neq g$.)

In the following, we assume that $\dot{\mathbf{S}}$ is divided into two infinite disjoint sets $\dot{\mathbf{S}}_{\mathrm{N}}$ and $\dot{\mathbf{S}} \setminus \dot{\mathbf{S}}_{\mathrm{N}}$ of *nonterminals* and *terminals*, resp. For each rule r as above, y will be a nonterminal node, i.e., $\ell_R(y) \in \dot{\mathbf{S}}_{\mathrm{N}}$, and its border nodes will be terminal.

Multiple Nodes. To make rules adaptive, we invent *multiple nodes*, similar to the set nodes of [19]. A multiple node x represents any number of ordinary nodes, the *clones of* x. Formally, multiple nodes are designated by special sorts. For this, we assume that $\dot{\mathbf{S}} \setminus \dot{\mathbf{S}}_{\mathrm{N}}$ is partitioned into a set of *singular* node sorts and a set $\ddot{\mathbf{S}}$ of *multiple* node sorts, where the latter is made up of pairwise distinct copies of the singular node sorts. For a singular node sort a, its copy in $\ddot{\mathbf{S}}$ is denoted by $\ddot{a}$. Likewise, the set of multiple nodes of a graph G is denoted by $\ddot{G}$. In figures, a multiple node of sort $\ddot{a}$ is distinguished by drawing it with a dashed line and a "shade" (see Figure 2 on page 450), but labeled with the singular sort a. A graph that does not contain any multiple node is said to be a singular graph. Note that there are no multiple nonterminal nodes.

Cloning. Let G be a graph. A function $\mu\colon \ddot{G} \to \mathbb{N}$ is a *multiplicity for* G. The graph G^{μ} is obtained from G by *cloning* each node $x \in \ddot{G}$ according to μ, as follows. If $\dot{\ell}_G(x) = \ddot{a}$, then x and its incident edges are replicated $\mu(x)$ times, where the sort of the copies is changed into a, i.e., clones are always singular. If $\mu(x) = 0$, then x and its incident edges are simply deleted.

Adaptive Star Grammars. Call a graph *well formed* if it contains neither adjacent nonterminal nodes nor indistinguishable edges, i.e., parallel edges with identical sorts. A *star* is a graph of the form $G(x)$ that contains neither loops nor parallel edges, such that x is nonterminal and its border nodes are terminal. An *adaptive star rule over* $S \subseteq \mathbf{S}$ is a rule $r = \langle y, R\rangle$, where R is a well-formed graph with sorts in S, and $R(y)$ is a star. A *clone of* r is a rule $\langle y, R'\rangle$ such that

- R' is well formed, and
- there is a multiplicity μ for R such that R' can be obtained from R^{μ} by identifying some of the border nodes of y with each other (where, of course, only nodes of the same sort can be identified).

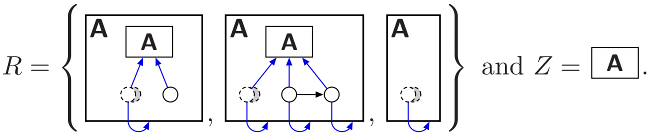
An *adaptive star grammar* is a system $\Gamma = \langle S, \mathcal{P}, Z\rangle$, where $S \subseteq \mathbf{S}$ is a finite set of sorts, $\mathcal{P}$ is a finite set of adaptive star rules, and Z is the initial star (all sorts being taken from S). Given a graph G, we write $G \Longrightarrow_{\mathcal{P}} H$ if $H = G[x\,/r]$ for some node $x \in G$ and a clone r of an adaptive star rule in $\mathcal{P}$. The *adaptive star language* generated by Γ is the set of all terminal graphs G that can be derived from Z:

$$\mathcal{L}(\Gamma) = \{G \mid Z \Longrightarrow_{\mathcal{P}}^{+} G \text{ and } \dot{\ell}_G(x) \in S \setminus \dot{\mathbf{S}}_{\mathrm{N}} \text{ for all } x \in \dot{G}\},$$

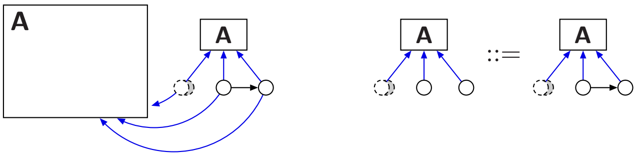
where $\Longrightarrow_{\mathcal{P}}^{+}$ denotes the transitive closure of $\Longrightarrow_{\mathcal{P}}$.

Example 1 (The Language of Unlabeled Graphs). To depict a rule $\langle y, R\rangle$ in a readable and space-efficient manner, we use the following drawing conventions inspired by [13]; see also [11]. The node y designating the left-hand side of the rule is drawn as a large rectangle covering all other nodes of R as well as the edges between them. Edges connecting y with its border nodes attach to the rectangle from the outside. The label of y is written in the upper left corner of the rectangle.

As an example, consider the adaptive star grammar Γ which is given by $\Gamma = ((\{\circ, \ddot{\circ}, \mathbf{A}\}, \{\text{-}\}), R, Z)$, where



It derives arbitrary graphs without loops over the "invisible" sorts ○ and -, that is, the class of finite unlabeled graphs. Starting with Z, the first rule makes it possible to add an arbitrary number of border nodes. The second rule adds edges between border nodes, and the third removes the nonterminal node. A more conventional way to draw the second rule can be obtained by "unfolding" it in two steps, like this:



Moving out the right-hand side Separating the sides

Let us briefly discuss the two main differences between the definition of adaptive star grammars used here and the one in [4]. Firstly, using the terminology of [4], we have restricted ourselves to *early cloning*, i.e., cloning produces only singular nodes. As shown in [4], this does not restrict the generative power of adaptive star grammars. Secondly, rules can be applied to subgraphs $G(x)$ that are not stars, but contain parallel edges. Note that, although the left-hand side $R(y)$ of an adaptive star rule $r = \langle y, R\rangle$ is a star, cloning it prior to application may involve taking a quotient that identifies border nodes of y in R^{μ} with each other. This may sound alarming, as it was shown for quite a similar type of adaptive star grammars in [5] that all recursively enumerable languages can be generated. However, note that we restrict this ability to the case where the resulting right-hand side R' is well formed. In particular, R' must not contain indistinguishable edges. As a consequence, adaptive star grammars of the sort defined above can be simulated by ordinary ones (i.e., those in [4]) by using subsets of $\bar{S}$ to label edges in stars and turning every rule into a finite number of rules (corresponding to the allowed quotients).

2.2 Extensions of Adaptive Star Grammars

We now introduce and discuss three extensions of adaptive star grammars that turn out to be useful when writing complex grammars. The first two extensions do not increase the generative power, but are nevertheless important in order to keep the grammars readable, because they lead to a considerable reduction of the number of rules.

Extension 1 (Subsorting). In order to avoid the necessity to draw lots of similar rules, we enrich the set S with additional *abstract sorts* and equip the so-enlarged sets $\dot{S}$ and $\bar{S}$ with *subsort relations*, partial orders denoted by "$\rightarrowtriangle$". For $\ddot{a}, \ddot{b} \in \ddot{\mathbf{S}}$, we require that $\ddot{a} \rightarrowtriangle \ddot{b}$ if and only if $a \rightarrowtriangle b$.

Sorts that are not abstract are *concrete*. For graphs G, H, we write $G \rightarrowtriangle H$ if H and G are equal up to their node and edge sorts and, for all $x \in \dot{G}, e \in \bar{G}$, we have $\dot{\ell}_G(x) \rightarrowtriangle \dot{\ell}_H(x)$ and $\bar{\ell}_G(e) \rightarrowtriangle \bar{\ell}_H(e)$. A graph is concrete if it contains only concrete sorts as labels, and abstract otherwise.

Now, every adaptive star rule $\langle y, R \rangle$ stands for the set of all $\langle y, R' \rangle$ such that R' is a concrete graph with $R \rightarrowtriangle R'$. Clearly, this extension does not make adaptive star grammars more powerful, since the set of all such graphs R' is finite for any given R (as S is finite). However, it can help to avoid a combinatorial explosion of the number of rules that have to be drawn.

Extension 2 (Multiple Subgraphs, Options, and Repetitions). We extend the notion of multiple nodes, as follows. In a rule $r = \langle y, R \rangle$, an induced subgraph M with $y \notin \dot{M}$ may be designated as a *multiple subgraph*. Thus, the multiple subgraph consists of the nodes in $\dot{M}$ and all edges $e \in \bar{R}$ with $s_R(e), t_R(e) \in \dot{M}$. Distinct multiple subgraphs must either be disjoint or properly nested. Graphically, we indicate a multiple subgraph by enclosing it in a dashed box with a dashed shade (see Figure 2). A multiple subgraph may be cloned any number of times prior to rule application. Here, cloning means to choose a multiple subgraph M not contained inside another multiple subgraph, and replacing it with $n \geqslant 0$ isomorphic copies. Each of the copies is connected to the border nodes of M (i.e., the nodes in $\dot{R} \setminus \dot{M}$ which are adjacent to nodes in $\dot{M}$) by copies of the edges that connect M with its border nodes. The cloning procedure is repeated until there are no multiple subgraphs left.

In addition to multiple subgraphs, it is useful to introduce *optional subgraphs*, drawn with dashed borders without a dashed shade, which may have $n \in \{0, 1\}$ clones, and *repeated subgraphs*, drawn with solid borders and a dashed shade, which may have $n \geqslant 1$ clones. If an optional or repeated subgraph consists of a single node, we apply these drawing conventions to the node itself rather than adding a box around it. (Thus, for multiple, optional, and repeated nodes, this is the notation known from PROGRES [19].)

The reader may have noticed that multiple, optional, and repeated nodes and subgraphs lift the use of regular expressions "E^*", "$E^?$", and "E^+" in the extended Backus-Naur form of context-free rules to the graph case, thus avoiding the necessity to use clumsy auxiliary rules. Similar to the case of context-free grammars, the expressive power of adaptive star grammars is not affected. To see this, note that a multiple subgraph M in a rule r as above may be implemented as follows. We remove M from R, except for the border nodes of y that are contained in M. Singular border nodes of y in M become multiple. Now, we add a new nonterminal x (labeled with a new sort), which is connected to all border nodes of M as well as to the border nodes of y contained in M, by edges labeled with pairwise different sorts. This yields the modified rule $\langle y, R' \rangle$. Finally, two new rules are added to the grammar, which generate any number of copies of the multiple subgraph by iteration. The terminating rule is $\langle x, R'(x) - \dot{M} \rangle$. We leave the straightforward definition of the iteration rule to the reader.

Extension 3 (Application Conditions). The example presented in the next section shows that the power of adaptive star rules suffices to express almost all the important structural properties of program graphs. Nevertheless, in some cases, the subgraph derived by a rule is subject to a condition which we do not know how to capture by adaptive star rules. Therefore, we extend them by allowing required or forbidden terminal subgraphs to be attached to the border nodes in the left-hand side of a rule. In fact, we only need required and forbidden edges between border nodes. We emphasize these edges by underlaying them in grey, and cross them out if they are forbidden.

Note that we cannot offer a formal proof for the necessity of application conditions. In particular, the three uses of application conditions in Fig. 2 can perhaps be avoided by a more ingenious grammar. However, we have not been able to come up with such a solution. Finding it or disproving its existence remains a subject for future work.

3 A Grammar for Program Graphs

Program graphs have been devised in [15], as a language-independent representation of object-oriented code for studying refactoring operations. They capture concepts that are common to many object-oriented languages, like single inheritance and method overriding. The simplified graphs defined here represent only the data flow of programs. See [20] for a specification of program graphs that covers their control flow as well.

A Sample Program Graph. Figure 1 shows a program graph for the following rudimentary program:

```
class Cell is                          subclass ReCell of Cell is
  var cts: Any;                          var backup: Any;
  method get() Any is                    method restore() is
    return cts;                            cts := backup;
  method set(const n: Any) is            override set(const n: Any) is
    cts := n                               backup := cts;
                                           super.set(n)
```

The node sorts C, M, K, V, B, E distinguish program entities: *classes*, *method signatures*, *constants*, *variables*, *bodies* (of methods), and *expressions*. In the figure, some nodes are annotated with the names of the entities they represent.

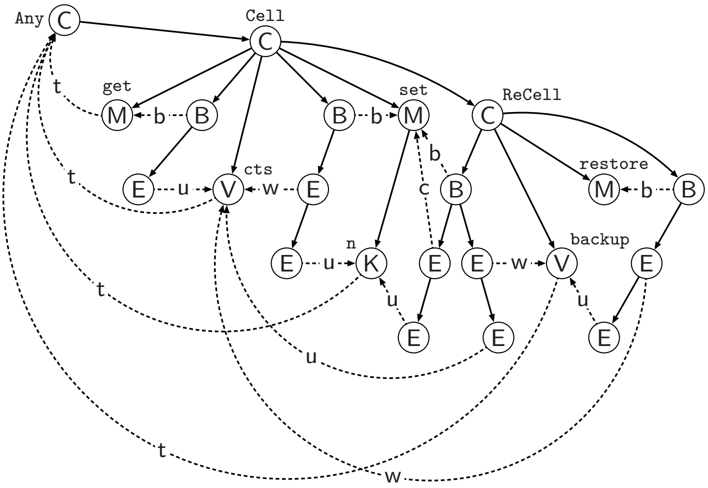


Fig. 1. A program graph

(Note that there are three nodes of sort C, rather than two, the leftmost one representing the predefined root class Any.) Edge sorts distinguish relations between program entities. Arrows drawn as “$\longrightarrow$” represent the *composition* of entities, and arrows of the form “$\cdots x \dashrightarrow$”, with $x \in \{\mathsf{t}, \mathsf{b}, \mathsf{c}, \mathsf{u}, \mathsf{w}\}$, represent *references* of entities to declarations: the *typing* of features by classes, the *binding* of bodies to the signatures of methods, the *call* of methods within expressions, the *use* of constant and variable values in expressions, and the *writing* of values of expressions to variables in assignments.

For instance, the node of sort C representing the class ReCell is composed of four entities. The leftmost node among these four, of sort B, represents the body overriding the body of set in Cell. In turn, the nodes beneath this node represent the expressions (and constants) occurring in the body, as well as the ways in which they call, use, or write to other visible entities.

Program graphs have a particular structure: Their composition follows the syntax of the underlying program texts, whereas their references obey the scope rules for the visibility of declarations in programs. Due to this structural complexity, the generative power of ordinary context-free graph grammars does not suffice to specify the set of all program graphs. Instead, this is now done using an adaptive star grammar, with the extensions introduced in Subsection 2.2.

The Concrete Sorts. The nonterminal node sorts **Prg**, **Hy**, **Cls**, **Fea**, **Sig**, **Bdy**, **Exp**, and **Act** label placeholders for the *program*, a class *hierarchy*, a *class* or *feature* definition, the *signature* or *body* of a method, an *expression*, or *actual parameters*, respectively. In rules, nonterminal nodes are connected to their border nodes by edges of different sort and direction. Every nonterminal node x is incident with one or two *anchor edges*, drawn as “$\longrightarrow$”, and may be incident with *scope edges* of the form “$-v\rightarrow$”, with $v \in \{\mathsf{g}, \mathsf{h}, \mathsf{i}\}$. Sort and direction of such an edge indicate the *role* of the node at the other end in the derivations of x:

- An anchor edge points to the program entity, represented by a singular node, the components of which are being defined by x. (The only situation in which a nonterminal node has two anchor edges is when a method consisting of a signature and a body is being defined.)
- Ingoing scope edges connect nodes representing declarations within the scope of x; these nodes represent the declarations of constants, variables, methods, and classes that may be referred to in the program entity defined by x.
- Outgoing scope edges connect the nonterminals **Hy**, **Cls**, and **Sig** to the set of declarations that are made visible by the program entities defined by these nonterminals.

The edge sorts g, h, and i classify the visibility of declarations as *global*, *hereditary*, and *internal*, respectively. (In our program graphs, classes are always global, and for the example program above, we assume that all method declarations are global, whereas attribute declarations are internal to a class.) Further edges of sort s connect "selected" nodes of type M to nonterminals **Bdy**, **Exp**, and **Act**, and nodes of type K to nonterminals **Act**. The major purpose of these edges is to keep track of a method signature when deriving a body for it.

The Abstract Sorts and Subsort Relation. In addition to the sorts mentioned above, we use abstract node sorts D, F, and A, together with the following subsort hierarchy on the declarations: A *declaration* D is either a class C, or a *feature* F, which in turn is either a method M, or an *attribute* A, which in turn is either a constant K, or a variable V. Moreover, the scope sorts g, h, and i are subsorts of an abstract scope sort that is just drawn as "⟶". See the sort graphs in Figure 3(a) and (b).

The Rules. The adaptive star rules defining program graphs are shown in Figure 2, and will be explained in the following paragraphs.

Start Rule. The rule start initiates the derivation of a class hierarchy. It applies to the initial star of the grammar, which consists of the nonterminal **Prg** with an anchor node representing the root Any of the class hierarchy. Together with the root node, the clones of the multiple D-node represent all globally visible declarations of the program. All of them are connected to the **Hy**-labeled nonterminal, say x, by in- and outgoing g-edges. Intuitively, this means that they are passed to the rule hy as declarations that are both visible and going to be defined by the derivation of x. Thus, these nodes play two roles in the derivations of x. Note that, in hy, these roles are split as its left-hand side does not contain parallel edges (as required by the definition of adaptive star rules). Thus, quotients will have to be taken in order to apply this rule.

Class Hierarchy. The rule hy defines a class hierarchy, consisting of a root class, plus $n \geqslant 0$ sub-hierarchies. The root classes of the sub-hierarchies are direct subclasses of the root class. The root class defines sets of features of different visibilities: Its internal features are only visible within itself, whereas its hereditary features are also visible to its sub-hierarchies. Its global declarations, together with those of its sub-hierarchies, are the global declarations of the whole hierarchy. The global method declarations are furthermore passed as "hereditary"

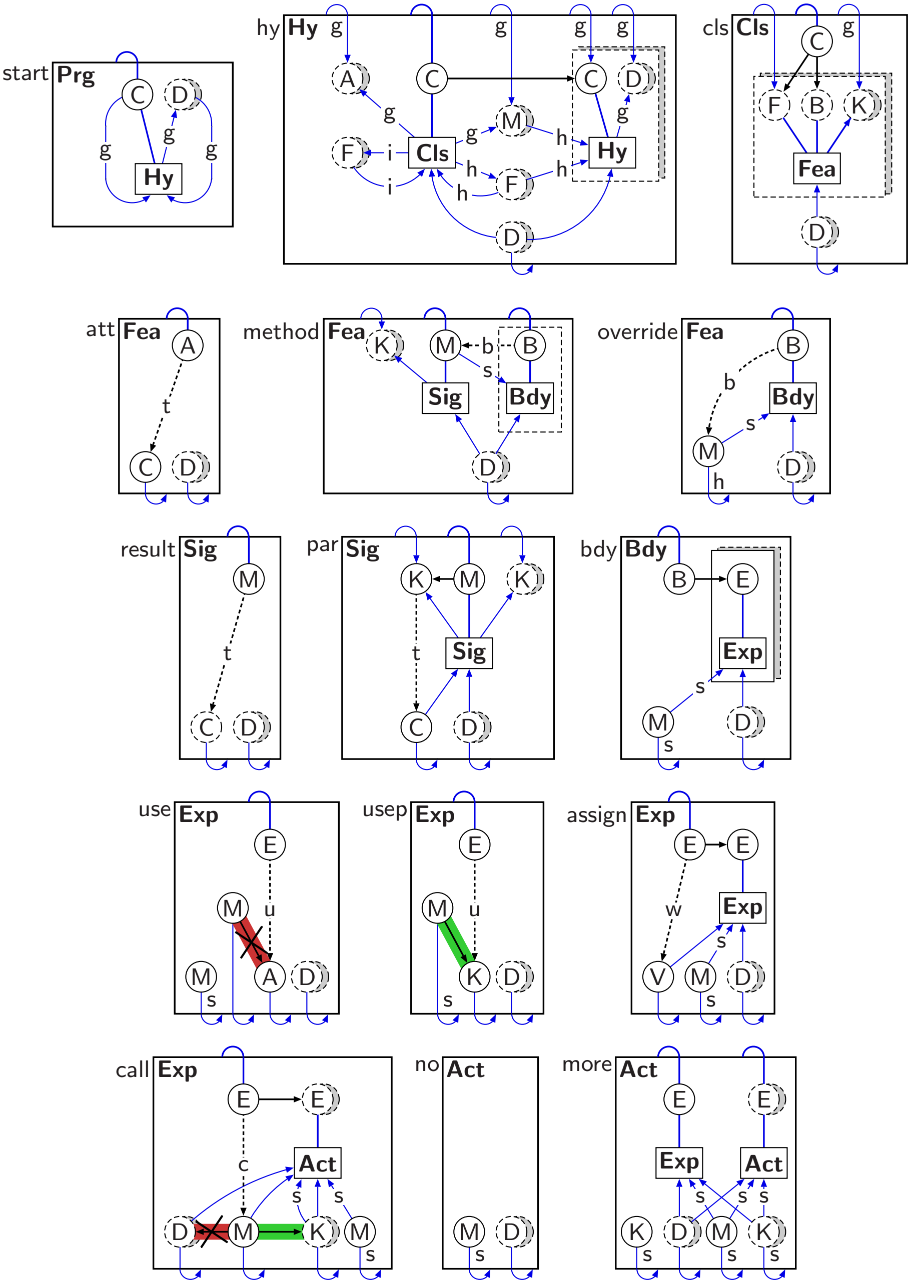


Fig. 2. The rules deriving program graphs

to the sub-hierarchies, because only such methods may be overridden. These methods are thus visible to the sub-hierarchies in two roles, since they are also contained, as "global", in the visible declarations.

Class Definitions. The rule cls derives a set of features as members of a class. Note that the derived features that may have global, hereditary, or internal visibility, depending on which subsort of the abstract derived scope sort is used. The multiple K-node represents the parameters of all methods in the class; they are passed around as global, so that they can be accessed when their method is used. (See the rules for expressions below, which contain application conditions to make sure that the visibility rules are obeyed.)

Feature Definitions. The rules for **Fea** derive the features that may be defined in a class: either a (constant or variable) attribute (rule att), a new method (rule method), or a method overriding a hereditary method (rule override). The new method is abstract if its optional body is missing. In rules method and override, the M-node being defined or overridden is passed as a "selected" entity to **Bdy** (and further on to **Exp** and **Act**) in order to access the parameters of the signature in expressions.

Method Signatures and Bodies. The rules result and par derive signatures of methods. Methods have an optional result type, and any number of formal parameters. The type of a result or a parameter is the class represented by the target of a t-edge. The rule bdy derives method bodies, each consisting of a set of expressions.

Expressions. The rules use, usep, assign, and call specify four kinds of expressions: the use of a visible attribute value, the use of a parameter of a method within its body, an assignment writing the value of an expression to a variable, and the call of a method with a list of actual parameters. Each of these rules inserts a reference to a declaration that is visible in the expression. The application conditions in use and usep make sure that an attribute is visible only if it is not the (constant) parameter of a method, and that a parameter is visible only within the method it belongs to, i.e., the selected one. In rule call, the application conditions make sure that all parameters of the method to be called are subsumed under the multiple K-node – there are no edges to other nodes in the set of visible entities.

Actual Parameters. The rules no and more generate as many expressions (actual parameters) as there are formal parameters of the method being called. To see this, not that each application of more "releases" one K-node.

Properties of Program Graphs. The rules given in Figure 2 have been designed so that the program graph grammar and the language it derives have the following properties:

- The composition edges induce a syntax tree. This is a spanning tree of the program graph. (Thus, program graphs are connected.)

- The composition and anchor edges of the rules induce *syntactic sub-rules* of the grammar, yielding a hyperedge replacement grammar that generates the syntax trees in a purely context-free manner.[1]
- The outgoing scope edges of a nonterminal n point to declaration nodes that are contained in the syntax tree derived by the syntactic sub-rule of n. These border nodes are uniquely determined by the underlying syntax tree.
- The ingoing scope edges of a nonterminal n connect it to the subset of declaration nodes that are method parameters, or visible in the syntax tree generated by n.
- The rules for a nonterminal n may insert reference edges to the ingoing declaration nodes that are visible. Thus terminal program graphs are syntax trees with references from entities to declarations that are visible to these entities.

The reader may have noticed that the program graph grammar has been designed with attribute grammars [14] in mind, a formalism that has been devised for specifying the contextual analysis and translation of (textual) programming languages. More precisely, the grammar corresponds to an attribute grammar with a semantic basis that has sets of syntax tree nodes as values, and set operations as semantic functions (like set union, and member selection). The anchor edges indicate what would be the underlying syntax of the attribute grammar, and the scope edges specify the values of its (inherited and synthesized) attributes.

Parsing. A parsing algorithm for adaptive star grammars has been developed in [17,4]. However, the rules given above must be transformed in order to make them acceptable for the algorithm:

- Optional, multiple, and repeated nodes and subgraphs can be defined as outlined in Extension 2.
- Empty rules, i.e., rules where the replacement just consists of the border nodes of their distinguished node, are also forbidden. In the program graph grammar, this occurs when the multiple subgraph in rule cls has no clone, or when the optional class node in rule result is missing. Empty rules can also be removed, by a transformation similar to the removal of empty rules from context-free word grammars [4].

As mentioned above, application conditions are a real extension of the formalism. Consequently, it is not clear whether adaptive star grammars with application conditions have a decidable membership problem in general. However, for “attribute-grammar-like” rules like the one presented here, a parsing algorithm can take advantage of the structure of such rules. First, the syntax tree can be parsed. Then the outgoing declaration nodes of the nonterminal stars can be determined, thus defining the ingoing declaration nodes. This, finally, allows to check whether the references are in place. During this process, it is easy to check whether the application conditions are fulfilled.

[1] The auxiliary rules defining the multiple subgraphs in the sub-rules become tree-generating hyperedge replacement rules as well.

4 Adaptive Star Grammars and Meta-models

The static structure of object-oriented software is commonly described by meta-models like UML class diagrams. A class diagram contains specifications of three kinds of properties:

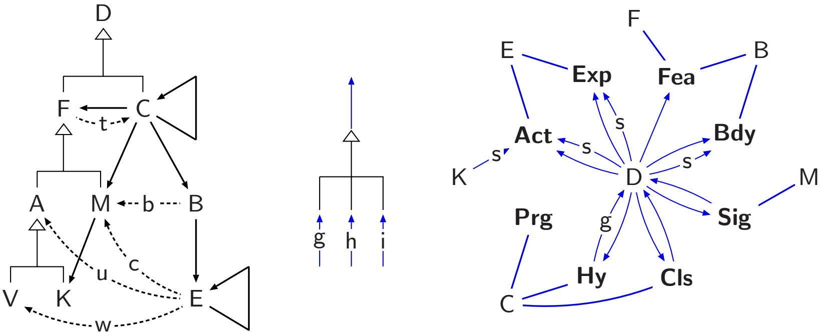
- *Inheritances* describe "is-a" relationships between classes.
- *Incidences* specify which associations may exist between which classes.
- *Multiplicities* define how many associations of some kind may leave and enter some class, thus distinguishing "1:1", "1:n", "n:1", and "n:m" relationships.

For an adaptive star grammar like that of program graphs, label subsorting does already specify the inheritances of a meta-model. Incidences can then be determined by constructing a *sort graph* as follows:

- Form the disjoint union of the start graph with all graphs of all rules, where all multiple node labels are turned into singular ones.
- Identify all equally labeled nodes with each other.
- Insert all inheritance edges in that graph. (This requires "meta-edges" between edges.)
- For some edge e labeled ℓ, remove all edges that are parallel to e and labeled by ℓ or a subsort of ℓ.
- If a node may have equally labeled edges to (or from) every direct sub-node of some node n, replace these edges by a single edge to n.

In Figure 3, the sort graph for program graphs is shown in three pieces: the left-hand side shows the subgraph induced by its composition and reference edge sorts, whereas the subgraph on the right-hand side is induced by its anchor and visibility edge sorts. In the middle, we depict the subsort relation on edge sorts. Altogether, the graphs in Figure 3 correspond to *schema graphs* in PROGRES [19], and to *type graphs with inheritance* in algebraic graph transformation [8].

The multiplicities known from UML class diagrams could be inferred by a similar, yet slightly more complicated procedure. For instance, rules start and hy



(a) Terminal sorts (b) Scope edges (c) Nonterminal sorts

Fig. 3. Node and edge sorts in program graphs

imply that composition edges incident with C-nodes have the source multiplicity $0 \leqslant n$ and the target multiplicity $0 \leqslant m \leqslant 1$, i.e., a C-node has at most one ingoing composition edge indicating its supertype, and may have any number of outgoing composition edges that indicate its subtypes. In the notation of UML class diagrams, this could be specified by annotating the loop edge at the node C in Figure 3 with "$\xrightarrow{*\quad ?}$". Most of the composition edges have the multiplicity "$\xrightarrow{*\quad\quad 1}$", saying that $n \geqslant 0$ edges may leave, and one edge enters a node.[2] Reference edges have the multiplicity "$\xrightarrow{?\quad\quad *}$", saying that $n \in \{0, 1\}$ of these edges may leave, and $n \geqslant 0$ of these edges may enter a node.

However, even a class diagram with multiplicities is not as precise as the program graph grammar. The scope rules for programs specified in the rules cannot be described by incidences and multiplicities alone. In an object-oriented meta model, they could be specified by logical formulas written in, e.g., OCL, the logical language of UML.

Vice versa, adaptive star grammars do not specify attributes for nodes and edges. However, this could be added in a straightforward way; if needed, their values could be determined by attribute functions. For instance, an attribute arity of B-nodes could be initialized to 0 in rules method and override, and increased in rule par.

In summary, one may say that adaptive star grammars provide a generative alternative to the use of meta models combined with logics. It is certainly a matter of debate which one appears more natural and is easier to use. However, as argued in [6], general graph transformation rules can conveniently describe model transformations, such as refactoring operations. This means that software refactoring systems based on graph transformation could make use of adaptive star grammars for specifying the models to be transformed, and of general graph transformation rules to specify the actual transformations. This avoids the need to mix different paradigms, one of them being used for the specification of the models and the other one for the description and implementation of valid transformations.

5 Conclusions

Adaptive star grammars are rather straightforward extensions of hyperedge and node replacement, and still have some properties of context-free grammars, e.g., nondeterministic versions of associativity and confluence. Adaptive star grammars allow to derive graph languages that define rather advanced software models, specifying properties as they are typically needed to represent programming languages.

The correctness of the grammar presented in Section 3 has not been proved formally. Nevertheless, such a proof should be rather straightforward (though technically tedious), exploiting the similarity of this particular example with attribute grammars pointed out earlier. This would formally support our claim that the power of adaptive star grammars is close to what is necessary for being

[2] Composition edges leaving nodes of sort B have the multiplicity "$\xrightarrow{+\quad 1}$" .

able to specify program graphs and similar structures. However, without application conditions, adaptive star grammars seem to be unable to cope with complex constraints like the correspondence of actual to formal parameters of methods. It is also open whether other constraints, like the type rules for expressions, can be expressed with adaptive star grammars.

In this paper, we have omitted certain features of the program graphs described in [20]. There, method bodies may contain local declarations and control flow. These features can be captured by refining and extending the rule set presented here. The nodes in [20] do also have attributes, for instance strings representing the identifier of a declaration, or numbers representing the arity of a method. Attributes are convenient to have; they cannot be handled by the grammatical formalism itself, but have to be evaluated while applying a rule, somewhat similar to the way in which application conditions are handled.

Certainly, some properties of models should not be described by the grammatical formalism. For instance, type rules could probably be specified by adaptive rules (with application conditions) as well, but only in a rather complicated way. Generally, attribute values and their dependencies, and negative conditions on the structure should rather be defined by predicates, e.g., in Ocl.

So far, grammars, in particular graph grammars, have hardly been considered for defining software models. Graph grammars have been primarily used to define semantics of models or model transformation. Hölscher et al. [12] specify the semantics of some UML diagram types by graph grammars. They specify, among others, state charts and interaction diagrams, i.e., the behavior of object-oriented software. However, graph grammars are used to specify the behavior in terms of the modeling diagrams, and not the syntactic structure of a program as described here. Syntax, in particular abstract syntax, is now usually represented by meta-models together with additional constraints. Typical examples of this approach are based on meta-modeling frameworks MOF [18] or EMF [10]. Moflon [1] allows to specify the abstract syntax of domain specific languages and their implementation using MOF and OCL. Graph transformation systems are used for model transformation and model integration. Tiger [9] uses EMF for specifying the abstract syntax of visual languages by a meta-model. Graph transformations are used to specify editing operations, i.e., behavior of a generated visual editor. While (graph) grammars had been the main approach to specify syntax of (visual) languages (e.g., [16]) in the past, they are now mostly used for behavior specification. Syntax specification by meta-modeling has become the more popular approach, probably since meta-models appear to be easier to build and understand than graph grammars. However, we think that more complex languages, like software models as described in this paper, can be described by graph grammars in a better and more concise manner.

The survey paper [7] has discussed the general scenario of transforming between different graph models. This allows to prove a kind of commutativity between the model transformations and the rules of the graph grammars defining the models concerned. However, it gives no means to check the integrity of the models themselves, as there is no parsing algorithm for the general graph grammar rules considered there.

In the graph reduction specifications proposed by Bakewell et al. [2], the inverses of the rules have convergent reductions that supply a parsing algorithm. However, they have only be used to specify the shape of data structures involving pointers so far.

Adaptive star grammars are used for shaped generic graph transformation [6]. The rules proposed there may contain variable nodes as placeholders for graphs (and also multiple nodes). Adaptive star grammars are used to specify the legal substitutions for variables. We plan to refine generic transformation so that it its is "shape-preserving": the transformed graphs shall be shaped according to an adaptive star grammar, and rules replace one well-shaped subgraph by another well-shaped graph. This will then allow to define model transformations that may not only be shown to preserve incidence constraints, but the shape of models as well.

In order to use adaptive star grammars in practice, we need a graph systems wherein they can be edited, transformed, and parsers for them can be generated. Such a system is under way. Furthermore, we are interested in the following methodical question: Can grammars be systematically developed from a context-free "kernel" defining spanning trees, as this has been done for program graphs?

Acknowledgments. We want to thank our former colleague Niels Van Eetvelde. Without his work on the first adaptive star grammar for program graphs [20], this paper could not have been written. Furthermore, we thank the anonymous referees for their constructive comments.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A standard-compliant metamodeling framework with graph transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer, Heidelberg (2006)
2. Bakewell, A., Plump, D., Runciman, C.: Specifying pointer structures by graph reduction. Mathematical Structures in Computer Science (to appear, 2008)
3. Courcelle, B.: An axiomatic definition of context-free rewriting and its application to NLC rewriting. Theoretical Computer Science 55, 141–181 (1987)
4. Drewes, F., Hoffmann, B., Janssens, D., Minas, M.: Adaptive star grammars and their languages. Technical Report 2008-01, Departement Wiskunde-Informatica, Universiteit Antwerpen (2008)
5. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Van Eetvelde, N.: Adaptive star grammars. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 77–91. Springer, Heidelberg (2006)
6. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Van Eetvelde, N.: Shaped generic graph transformation. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007). LNCS. Springer, Heidelberg (to appear, 2008)

7. Ehrig, H., Ehrig, K.: An overview of formal concepts for model transformations based on typed attributes graph transformation. In: Proc. Graph and Model Transformation Workshop (GraMoT 2005). Electronic Notes in Theoretical Computer Science, vol. 152(4) (2006)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (2006)
9. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In: ASE 2005: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 134–143. ACM Press, New York (2005)
10. EMF, Eclipse Modeling Framework web page (2006), http://www.eclipse.org/emf/
11. Engelfriet, J.: Context-free graph grammars. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, Beyond Words, vol. 3, pp. 125–213. Springer, Heidelberg (1999)
12. Hölscher, K., Ziemann, P., Gogolla, M.: On translating uml models into graph transformation systems. J. Vis. Lang. Comput. 17(1), 78–105 (2006)
13. Kaul, M.: Syntaxanalyse von Graphen bei Präzedenz–Graph–Grammatiken. Dissertation, Univ. Passau (1985)
14. Knuth, D.E.: Semantics of context-free languages. Math. Sys. Theory 2(2), 127–145 (1968); . Correction: Math. Sys. Theory 5(1), 95-96 (1971)
15. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour-preserving transformation. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002)
16. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (2002)
17. Minas, M.: Parsing of adaptive star grammars. Electronic Communications of the European Association of Software Science and Technology 4 (2006), www.easst.org/eceasst
18. Object Management Group. Meta Object Facility (MOF) Core Specification, version 2.0 edn., Document - formal/06-01-01 (January 2006)
19. Schürr, A., Winter, A., Zündorf, A.: The Progres approach: Language and environment. In: Engels, G., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages, and Tools, ch. 13, vol. II, pp. 487–550. World Scientific, Singapore (1999)
20. Van Eetvelde, N.: A Graph Transformation Approach to Refactoring. Doctoral thesis, Universiteit Antwerpen (May 2007)

Tutorial Introduction to Graph Transformation

Reiko Heckel

University of Leicester, UK
reiko@mcs.le.ac.uk

Abstract. This tutorial is intended as a general introduction to graph transformation for participants to the conference or its satellite events who are not familiar with the mainstream approaches and concepts of the area. The tutorial will start with an informal introduction to the basic concepts of graph transformation, such as graphs, rules, transformations, discussing semantic choices such as the handling of dangling edges during rewriting, and extensions such as attributes, types, or inheritance.

In the second part, the tutorial will give a survey of typical applications of graph transformation, for example as a specification language and semantic model for concurrent and distributed systems, as a model transformation language for defining syntax, semantics, and manipulation of visual models, etc.

Finally, the tutorial will go into some details about the theory of (in particular) the algebraic approach to graph transformation, its formal foundations and relevant theory and tools. This shall enable the participants to better appreciate the conference and its satellite events.

1 Motivation

Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science, and many other areas of science and engineering. For example, for most of the activities in the software development process, visual notations are used including state diagrams, control flow graphs, architectural diagrams, and the UML family of languages. Models based on these notations can be seen as graphs and thus graph transformations are involved, either explicitly or behind the scenes, when specifying how these models should be built and interpreted, and how they evolve over time and are mapped to implementations. At the same time, graphs provide a universally adopted data structure, as well as a model for the topology of object-oriented, component-based and distributed systems. Computations in such systems are therefore naturally modelled as graph transformations, too.

2 Content

In this tutorial, we will introduce the basic concepts and approaches to graph transformation, demonstrate their application to, in particular, software engineering problems, and provide a high-level survey of graph transformation theory and tools.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 458–459, 2008.

We start by introducing a simple set-theoretic presentation of the double-pushout approach [3] whose features are common to most graph transformation approaches and which provides the foundation for further elaboration. Then, we discuss alternatives and extensions, like *multi objects*, *programmed* transformations concerned with controlling the (otherwise non-deterministic) rewriting process, as well as *application conditions*, restricting the applicability of individual rules.

Typical applications of graph transformation to software engineering problems are presented in terms of examples. They include

- Model and program transformation;
- Syntax and semantics of visual languages;
- Visual behaviour modelling and programming.

In particular, we distinguish between the use of graph transformation as a *modelling notation* (and semantic model) to reason on particular problems, like functional requirements or architectural reconfigurations of individual applications, and its use as a *meta language* to specify the syntax, semantics, and manipulation of visual modelling languages, like the UML.

The last part of the tutorial is dedicated to a survey on the algebraic approach to graph transformation, its formal foundations and relevant theory [2]. This shall enable attendees to work their way through the relevant literature and to benefit more from the presentations at the conference.

Previous versions of this tutorial together with accompanying papers have been presented in [1,4].

References

1. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 402–429. Springer, Heidelberg (2002)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. In: EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
3. Ehrig, H., Pfender, M., Schneider, H.J.: Graph grammars: an algebraic approach. In: 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 167–180. IEEE Computer Society Press, Los Alamitos (1973)
4. Heckel, R.: Graph transformation in a nutshell. In: Heckel, R. (ed.) Proceedings of the School on Foundations of Visual Modelling Techniques (FoVMT 2004) of the SegraVis Research Training Network. Electronic Notes in TCS, vol. 148, pp. 187–198. Elsevier, Amsterdam (2006)

Workshop on Graph Computation Models

Mohamed Mosbah[1] and Annegret Habel[2]

[1] LaBRI - ENSEIRB - University of Bordeaux 1
351 Cours de la Libération, 33405 Talence, France
mosbah@labri.fr
[2] Fachbereich Informatik, Universität Oldenburg
Postfach 2503, D-26111 Oldenburg, Germany
habel@informatik.uni-oldenburg.de

Abstract. A variety of computation models have been developed using graphs and graph transformations. These include models for sequential, distributed, parallel or mobile computation. A graph may represent, in an abstract way, the underlying structure of a computer system, or it may stand for the computation steps running on such a system. In the former, the computation can be carried on the corresponding graph, implying a simplification of the complexity of the system. The aim of the workshop is to bring together researchers interested in all aspects of computation models based on graphs, and their applications. A particular emphasis will be made for models and tools describing general solutions.

1 Graph Computation Models

There are a lot computation models based on graphs, and their applications. The computation models include mobile computing, programming, data transformations, concurrent and distributed computing. In the following, we will sketch some of these graph computation models.

1.1 Graph Relabeling Systems and Distributed Computing

Graph relabeling systems have been successfully used as a suitable tool for encoding distributed algorithms, for proving their correctness and for understanding their power. In this model, a network is represented by a graph whose vertices denote processors, and edges denote communication links. The local state of a processor (resp. link) is encoded by the label attached to the corresponding vertex (resp. edge). A rule is a local transformation of labels. A relabeling system is defined by a finite set of such rules. The application of the rules are asynchronous: there is no global clock available, and two conflict-free applications of rewriting rules may occur simultaneously, provided they do not attempt to modify the same local context in the host graph. Thus, the behaviour of the network is defined by its initial labeling and the rule base of the associated local rewriting calculus. Problems of interest in distributed computing include node election, node enumeration, spanning tree construction, termination detection, synchronisation, inter-node agreements, or local recognition of global properties.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 460–462, 2008.

These studies rely on rule-based local computations on network graphs on the one hand, and the recognition and classification of certain initial network configurations on the other hand. The non-existence of deterministic distributed solutions to certain problems leads to propose also the investigation of probabilistic distributed algorithms, the formulation of which seems rather simple, but their analysis is difficult. An important aspect is the relationship between the three principal paradigms of distributed computing – local computations, message passing, shared memory – and the comparison of their expressive powers. Similar questions arise where those three paradigms are compared with mobile agent systems. See e.g. [1,2,3,4].

1.2 Graph Reduction

Pointer manipulation is notoriously dangerous in languages like C where there is nothing to prevent: the creation and dereferencing of dangling pointers; the dereferencing of nil pointers or structural changes that break the assumptions of a program, such as turning a list into a cycle. The goal is to improve the safety of pointer programs by providing (1) means for programmers to specify pointer data structure shapes, and (2) algorithms to check statically whether programs preserve the specified shapes. In [5], these aims are approached as follows. 1. Develop a formal notation for specifying shapes (languages of pointer data structures); that is the main cfoncern of this paper. We show how shapes can be defined by graph reduction specifications, which are the dual of graph grammars in that graphs in a language are reduced to an accepting graph rather than generated from a start graph. Polynomially terminating graph reduction specifications whose languages are closed under reduction allow a simple and efficient membership test for individual structures, yet seem powerful enough to specify all common data structures. 2. The effect of a pointer algorithm on the shape of a data structure is captured by abstracting the algorithm to a graph rewrite system annotated with the intended structure shape at the start, end and intermediate points if needed. A static verifier then checks the shape annotations.

1.3 Term Graph Rewriting

The theory of term graph rewriting allows to reason about computations on expressions with shared subexpressions. Sharing improves the efficiency of computations in space and time, and is ubiquitous in implementations of functional and logic programming languages, systems for automated deduction, and computer algebra systems. Term graph rewriting provides a model to reason about the correctness, completeness and efficiency of term rewriting with shared subexpressions. This model reflects the properties of real implementations more adequately than the conventional, tree-based model of term rewriting. Sharing makes term graph rewriting different from term rewriting with respect to both efficiency and properties such as termination and confluence, thus requiring a theory different from established term rewriting theory. See, e.g., [6,7].

2 Organization

The workshop will include tutorials, contributed papers, and system demonstrations. The tutorials will introduce many types of graph transformations and their use to study computation models. The system demonstrations based on graph computation models.will range from alpha-versions to fully developed products that are used in education, research or being prepared for commercialisation.

Workshop Chairs and Organizers

Mohamed Mosbah	University of Bordeaux 1	France
Annegret Habel	University of Oldenburg	Germany

Program Commitee

Frank Drewes	Umea University	Sweden
Rachid Echahed	IMAG, Grenoble	France
Emmanuel Godard	University of Marseille	France
Stefan Gruner	University of Pretoria	South Africa
Annegret Habel	University of Oldenburg	Germany
Dirk Janssens	University of Antwerp	Belgium
Hans-Jörg Kreowski	University of Bremen	Germany
Mohamed Mosbah	University of Bordeaux 1	France
Detlef Plump	University of York	United Kingdom

References

1. Godard, E., Métivier, Y., Muscholl, A.: Characterizations of classes of graphs recognizable by local computations. Theory of Computing Systems 37, 249–293 (2004)
2. Derbel, B., Mosbah, M.: Distributed graph traversals by relabelling systems with applications. ENTCS 154(2), 79–94 (2006)
3. Bauderon, M., Métivier, Y., Mosbah, M., Sellami, A.: From local computations to asynchronous message passing systems. Research Report RP 1271-02, Université Bordeaux I (2002)
4. Derbel, B., Mosbah, M., Gruner, S.: Mobile agents for implementing local computations in graphs. In: Ehrig, H., et al. (eds.) ICGT 2008. LNCS, vol. 5214. Springer, Heidelberg (2008)
5. Bakewell, A., Plump, D., Runciman, C.: Specifying pointer structures by graph reduction. Mathematical Structures in Computer Science (to appear, 2008)
6. Plump, D.: Term graph rewriting. In: Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools, vol. 2, pp. 3–61. World Scientific, Singapore (1999)
7. Echahed, R.: On term-graph rewrite strategies. ENTCS 204, 99–110 (2008)

Graph-Based Tools: The Contest

Arend Rensink[1] and Pieter Van Gorp[2]

[1] Universiteit Twente
rensink@cs.utwente.nl
[2] Universiteit Antwerpen
pieter@pietervangorp.com

Abstract. This event is the second instance of the contest for graph transformation-based tools, which was first held as part of the AGTIVE workshop. The aim is to stimulate tool development by providing a sense of competitiveness, as well as the chance to get to know and learn from the features of other, related tools.

1 Aims of the Workshop

Tools are crucial for the promotion of graph transformation in industry. It is only with the ready availability of reliable, easy-to-use tools that the attractions and benefits of graph transformation can ever become clear to anyone not having a prior education in this field. Furthermore, given the inherent complexities of the method, tool performance is an important issue. As a community we should be constantly working to improve tool support in all these aspects.

A variety of tool environments exists, supporting different graph transformation approaches and to some degree serving different purposes. There are some examples of tool comparisons, e.g., ([2], [3], [6]); furthermore, Varro et al. ([9]) propose some benchmarks to be used for such purposes. Nevertheless, having a certain application in mind, it is difficult for newcomers to decide the right graph transformation tool to use. Moreover, even for most of the tool experts it is true that they know much about one or two tools but little about the others.

To stimulate both the continued improvement of tools and the wider dissemination of knowledge about existing tools, GraBaTs 2008 comprises a *tool contest*, building upon the contest held as part of the AGTIVE 2007 symposium (see ([7])). The aim is to compare the expressiveness and the performance of graph transformation tools, along a number of selected case studies. This year, we have extended the contest with a live session to also measure the usability of tools in a controlled environment. The desired outcome is threefold:

- To learn about the pros and cons of each tool for different applications. A deeper understanding of the relative merits of different tool features will help to improve graph transformation tools and to indicate open problems.
- To identify common functionality across tools. By identifying features that are becoming mainstream, developers may decide to reuse functionality from other tools and focus more on unique added value.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 463–466, 2008.

- To instill a sense of challenge and competition that will motivate tool developers to continue their efforts. There is nothing like seeing, and being inspired by, the features of other tools to stimulate progress in one's own.

The case studies, outlined in Sect. 2, were selected from the entries received after an open call for cases. The second phase of the contest consists of the development of solutions to these case studies. At the time of writing, we cannot yet estimate the response to the call for solutions.

In addition, with respect to the first tools contest we have added the concept of a *live session*. The idea of this session is that the participants (limited to those who have explicitly registered to this session) will receive a case description on the spot, and will be asked to provide a solution, using the tool of their choice, within a fixed time frame of half a day. This experiment complements the solutions submitted to the pre-defined cases; it will provide valuable data about the flexibility and ease of use of the various tools.

2 The Cases

A Case Study for Program Refactoring. This case study, described in ([4]), concerns the implementation of three non-trivial source code (viz., Java) refactorings: *Encapsulate Field*, *Move Method*, and *Pull-up Method*. Input and output are to be given in a GXL-formatted encoding of Java programs. The case aims to enable comparison of various features of graph transformation tools, such as their expressiveness and their ability to interact with the user.

The AntWorld Simulation Tool Case. This case study, described in ([10]), complements the Sierpinski triangles benchmark studied in AGTIVE 2007 (cf. ([8])). The case has been designed such that the tools will most likely not run into memory problems. Over time, the number of Ants grows to the square of executed rounds. Thus, the focus of the benchmark is the movement of Ants. This can again be done with reasonably simple rules that mainly employ local search. Thus, the AntWorld simulation stresses local rule application. Another aspect of this case is the visualisation of the ants.

Transforming BPMN process models to BPEL process definitions. This case study, described in ([1]), considers the definition of model transformations between two languages for business process modelling, namely BPMN and BPEL. The model transformations should achieve four evaluation criteria: completeness, correctness, readability and reversibility.

3 Conclusion

The previous instance of the tool contest, summarised in ([7]), identified strong and weak points and made a number of recommendations. In this second instance we have taken measures to resolve the weaknesses and take the recommendations into account.

Explicit challenges. It was recommended to include explicit challenges in the case descriptions. This has been implemented by requiring case submissions to include a description of the challenges involved.

Ranking. It was recommended to take measures to enable the ranking of submissions. We have responded to this by requiring that all case studies provided a set of variation points. This enables a comparison of the solutions on a common basis while leaving enough room for differentiation. Moreover, a reference server is set up so that solutions can be run and compared on an equal basis, making a more objective ranking possible.

Case categories. It was recommended to include more case categories, such as, for instance, "NP-complete problems" and verification issues. Unfortunately, neither of these topics is featured among the selected cases (see above).

Separate workshop. It was recommended to organise the next instance of the tool contest as a separate workshop, so that there would be more time for all submissions to be demonstrated (this being an important incitement for further improvements). Clearly, this recommendation has been implemented.

At the time of writing, we cannot yet make a statement about the success of these measures. In any event, we hope to have a lively workshop, with a healthy mixture of competitiveness and cooperation.

References

1. Dumas, M.: Transforming BPMN process models to BPEL process definitions (2008), http://www.fots.ua.ac.be/events/grabats2008/cases/grabats2008translationcase.pdf
2. Fuß, C., Mosler, C., Ranger, U., Schultchen, E.: The jury is still out: A comparison of AGG, Fujaba, and PROGRES. In: Graph Transformation and Visual Modeling Techniques (GT-VMT). Electronic Comm. of the EASST, vol. 6 (2007)
3. 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)
4. Hoffmann, B., Perez, J., Mens, T.: A case study for program refactoring (2008), http://www.fots.ua.ac.be/events/grabats2008/cases/grabats2008refactoringcase.pdf
5. Nagl, M., Schürr, A., Zündorf, A. (eds.): Applications of Graph Transformation with Industrial Relevance. LNCS. Springer, Heidelberg (2008)
6. Rensink, A., Schmidt, Varró, D.: Model checking graph transformations: A comparison of two approaches. In: Ehrig, H., Engels, G., Parise-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 226–241. Springer, Heidelberg (2004)
7. Rensink, A., Taentzer, G.: AGTIVE 2007 graph transformation tool contest. In: [5]

8. Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T.: Generation of Sierpinski triangles: A case study for graph transformation tools. In: [5]
9. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 79–88. IEEE Computer Society, Los Alamitos (2005)
10. Zündorf, A.: The antworld simulation tool case (2008), http://www.fots.ua.ac.be/events/grabats2008/cases/grabats2008performancecase.pdf

Workshop on Petri Nets and Graph Transformations

Paolo Baldan[1] and Barbara König[2]

[1] Dipartimento di Matematica Pura e Applicata,
Università di Padova, Italy
baldan@math.unipd.it
[2] Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany
barbara_koenig@uni-due.de

The *Workshop on Petri Nets and Graph Transformations*, which is currently at its third edition, is focused on the mutual relationship between two prominent specification formalisms for concurrency and distribution, namely Petri nets and graph transformation systems. It belongs to folklore that Petri nets can be seen as rewriting systems over (multi)sets, the rewriting rules being the transitions, and, as such, they can be seen as special graph transformation systems, acting over labelled discrete graphs. The basic notions of Petri nets such as marking, enabling, firing, steps and step sequences can be naturally "translated" to corresponding notions of graph transformation systems. Due to this close correspondence there has been a mutual influence between the two fields, which has lead to a fruitful cross-fertilisation.

Several approaches to the concurrent semantics of graph transformation systems as well as techniques for their analysis and verification have been strongly influenced by the corresponding theories and constructions for Petri nets (see, e.g., [11]). For instance, the truly concurrent semantics of algebraic graph transformations presented in [3,2] can be seen as a generalisation of the corresponding semantic constructions developed for Petri nets in [23,15]. Similarly, the concurrent semantics for EMS systems in [13] is partly inspired by the Goltz-Reisig process semantics for Petri nets. More recently, several approaches to the analysis and verification of graph transformation systems properties have been proposed (see, e.g., [19,5,22,7,18]) and also in this case the relation with Petri nets has been often a source of inspiration. In particular, some approaches are inspired by analogous techniques previously developed in the domain of Petri nets, e.g., based on invariants or on finite prefixes of the unfolding, and some others reduce the verification of a graph transformation systems to the analysis of a suitable abstraction expressed in the form of a Petri net.

Classical Petri net models have been integrated with graph transformation systems in order to define rule-based changes in the Petri net structure. This can be used for a stepwise refinement of Petri net models, which leads from an abstract description of the system to the desired model, or to formalise model transformation over Petri net models. Alternatively, transformations over Petri nets can be used to define dynamically reconfiguring Petri nets, i.e., extended

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 467–469, 2008.

Petri net models where the standard behaviour, expressed by the token game over a fixed structure, is enriched with the possibility of altering the net structure (see, e.g., reconfigurable nets of [1] and high-level replacement systems applied to Petri nets in [17,8])

As mentioned above, the theory of rewriting over categories of Petri falls into the realm of high-level replacement systems, an extension of graph transformation systems to general categories, the so-called called HLR categories [9], including, e.g., algebraic specifications. The HLR approach has been generalised with the introduction of adhesive categories [14] and adhesive HLR systems [10], which provide a quite elegant and general framework where (double-pushout) rewriting can be developed. The view of Petri nets as rewriting systems over adhesive categories [20] or as bigraphical reactive systems [16] has been recently used to automatically derive compositional behavioural equivalences for Petri nets. More generally, adhesive categories appear as a promising framework where notions, constructions and results arising in the areas of Petri nets and graph transformation can be given a unified, abstract presentation (see, e.g., [21,4]).

As a further link between the two models, recall that graph transformation systems are also used for the development, the simulation, or animation of various types of Petri nets, e.g., via the the definition of visual languages and environments [6,12].

The workshop is aimed at favouring the cross-fertilisation and the exchange between the areas of Petri nets and of graph transformation, by gathering researchers working in the field of low- and high-level Petri nets, and researchers working in the field of rewriting, including graph transformation, high-level replacement systems and rewriting systems over adhesive categories.

References

1. Badouel, E., Llorens, M., Oliver, J.: Modeling concurrent systems: Reconfigurable nets. In: Arabnia, H.R., Mun, Y. (eds.) Proceedings of PDPTA 2003, vol. 4, pp. 1568–1574. CSREA Press (2003)
2. Baldan, P.: Modelling concurrent computations: from contextual Petri nets to graph grammars. PhD thesis, Department of Computer Science, University of Pisa, Available as technical report n. TD-1/00 (2000)
3. Baldan, P., Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Rossi, F.: Concurrent Semantics of Algebraic Graph Transformation Systems. In: Ehrig, et al. (eds.) [11]
4. Baldan, P., Corradini, A., Heindel, T., König, B., Sobociński, P.: Processes for adhesive rewriting systems. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 202–216. Springer, Heidelberg (2006)
5. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)
6. Bardohl, R., Ermel, C.: Scenario animation for visual behavior models: A generic approach. Software and System Modeling 3(2), 164–177 (2004)
7. Dottí, F.L., Foss, L., Ribeiro, L., Santos, O.M.: Verification of distributed object-based systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 261–275. Springer, Heidelberg (2003)

8. Ehrig, H., Gajewsky, M., Parisi-Presicce, F.: Replacement systems with applications to algebraic specifications and petri nets. In: Ehrig, et al. (eds.) [11]
9. Ehrig, H., Habel, A., Kreowski, H.-J., Parisi-Presicce, F.: Parallelism and concurrency in High-Level Replacement Systems. Mathematical Structures in Computer Science 1, 361–404 (1991)
10. 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)
11. Ehrig, H., Kreowski, J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Concurrency, Parallelism and Distribution, vol. Volume III. World Scientific, Singapore (1999)
12. Ermel, C., Ehrig, K.: View transformation in visual environments applied to petri nets. In: Ehrig, H., Padberg, J., Rozenberg, G. (eds.) Proceedings of PNGT'04. ENTCS, vol. 127, pp. 61–86. Elsevier, Amsterdam (2005)
13. Janssens, D.: ESM systems and the composition of their computations. In: Ehrig, H., Schneider, H.-J. (eds.) Dagstuhl Seminar 1993. LNCS, vol. 776, pp. 203–217. Springer, Heidelberg (1994)
14. Lack, S., Sobociński, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
15. Meseguer, J., Montanari, U., Sassone, V.: On the semantics of Place/Transition Petri nets. Mathematical Structures in Computer Science 7, 359–397 (1997)
16. Milner, R.: Bigraphs for petri nets. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 686–701. Springer, Heidelberg (2004)
17. Padberg, J., Ehrig, H., Ribeiro, L.: High level replacement systems applied to algebraic high level net transformation systems. Mathematical Structures in Computer Science 5(2), 217–256 (1995)
18. Padberg, J., Enders, B.: Rule invariants in graph transformation systems for analyzing safety-critical systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 334–350. Springer, Heidelberg (2002)
19. Rensink, A.: Explicit state model checking for graph grammars. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 114–132. Springer, Heidelberg (2008)
20. Sassone, V., Sobocinski, P.: A congruence for Petri nets. In: Proceedings of PNGT 2004. ENTCS, vol. 127, pp. 107–120. Elsevier, Amsterdam (2005)
21. Sobociński, P.: Reversing graph transformations. In: Proceedings of PNGT 2006. Electronic Communications of the EASST, vol. 2 (2006)
22. Varró, D., Varró-Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination analysis of model transformations by Petri nets. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 260–274. Springer, Heidelberg (2006)
23. Winskel, G.: Event Structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)

Summary of the Workshop on Natural Computing and Graph Transformations

Ion Petre[1,2] and Grzegorz Rozenberg[3,4]

[1] Academy of Finland
[2] Computational Biomodeling Laboratory, Åbo Akademi University
Turku 20520, Finland
ipetre@abo.fi
[3] Leiden Institute for Advanced Computer Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
[4] Department of Computer Science, University of Colorado at Boulder
Boulder, Co 80309-0347, USA
rozenber@liacs.nl

Natural Computing is a research area concerned with computing taking place in nature and with human-designed computing inspired by nature. It is a fast growing, genuinely interdisciplinary field involving, among others, biology, mathematics and computer science. Graphs and graph transformations are of great interest in this field in several respects. On the one hand, graphs are often used in the modeling of natural processes either as a representation of the hierarchical structures involved in the process or as a way to formalize the features of reality on several levels of abstraction. Several graph related formalisms such as Petri nets, abstract state machines, automata, membrane systems, mobile ambients, etc., are already used as modeling tools for natural processes. On the other hand, in human-designed computing inspired by nature, graph theoretical formulations and problems are often used as benchmarks for the investigation of the potential of the proposed computational paradigms.

The topics of interest of this workshop include, but are not limited to:

- Applications of membrane systems in modeling natural processes
- Petri nets in systems biology
- Abstract state machines in systems biology
- Visual languages in systems biology
- Modeling with finite and hybrid automata
- Unconventional solutions to graph theoretical problems
- Hierarchies of models and their refinement

We would like to thank all members of the program committee for helping selecting a high level scientific program for our workshop. They were: Tero Harju (Turku), Reiko Heckel (Leicester), Natasha Jonoska (Tampa), George Paun (Bucharest), Ion Petre (Turku), Leila Ribeiro (Rio Grande do Sul) and Grzegorz Rozenberg (Leiden).

We would also like to thank the four invited speakers and all the authors of the submitted papers for their talks at the workshop. Here is the list of the invited speakers and the abstracts of their talks:

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 470–472, 2008.

Robert Brijder (Leiden, the Netherlands): Self-assembly of graphs using multisets.

Self-assembly is the ubiquitous process in which individual components autonomously assemble into intricate complexes. Here we consider a model having multisets of "attach points" assigned to nodes of a graph. Using pre-defined rules, these attach points can "grab" other nodes present in the environment. In this way graphs self-assemble into larger structures. By restricting the general model in different ways, one can distinguish different levels of generative power. In particular a pumping lemma analog for these self-assembling graphs is obtained.

Tero Harju (Turku, Finland): Graph transformations related to gene assembly.

Stichotrich ciliates have a heavily fragmented storage genome which is defragmented and reorganized in gene assembly to a fully functional genome. This process of gene assembly has been modeled by three types of operations that have been presented by three rather natural operations on undirected graphs. We survey some of the known results and discuss some open problem concerning these graph operations.

Ina Koch (Berlin, Germany): Petri nets and systems biology.

The huge amount of experimental data in molecular biology enables researchers to derive larger and more complex models of biochemical processes. Because of experimental limitations or ethical reasons, often, more qualitative than quantitative (kinetic) data is known. Petri net theory represents one of the concepts enabling discrete qualitative and quantitative modeling to estimate the network behavior. Moreover, modeling at different abstraction levels is supported. The first application of Petri nets to biochemical networks has been published in 1993. Meanwhile, many Petri net models of different biochemical subjects have been published.

The aim of the talk is to demonstrate which Petri net properties and analysis techniques are suitable in particular for modelling biochemical systems and why. Several biological examples will be used to show different modeling types. Further, new developed analysis techniques, which are mainly based on invariant analysis, will be introduced and discussed such as MCT-sets and T-clusters for network reduction and Mauritius maps as new data structure and visualization technique to support knockout analysis. In this context, the special biological tasks will be explained. The talk will finish with open problems in the field of computational systems biology.

Andrei Păun (Bucharest, Romania): Discrete nondeterministic modeling of cellular pathways.

Computer modeling of molecular signal cascades can provide useful insight into the underlying complexities of biological systems. We provide a refined approach of the discrete modeling of protein interactions within the environment of a single cell. The technique we offer utilizes the Membrane Systems paradigm which, due to its hierarchical structure, lends itself readily to mimic the behavior of cells. Since our approach is non-deterministic and discrete, it provides interesting contrast to the standard deterministic ordinary differential equations techniques. We argue that our approach may outperform ordinary differential equations when modeling systems with relatively low numbers of molecules - a frequent occurrence in cellular signal cascades. Refinements over our previous modeling efforts include the addition of nondeterminism for handling reaction competition over limited reactants, increased efficiency in the storing and sorting of reaction waiting times, and modifications of the model reactions. Results of our discrete simulation of the type I and type II Fas-mediated apoptotic signal cascade are illustrated and compared with two approaches: one based on ordinary differential equations and another based on the well-known Gillespie's algorithm.

ICGT 2008 Doctoral Symposium

Andrea Corradini[1] and Emilio Tuosto[2]

[1] Dipartimento di Informatica, Pisa
[2] Department of Computer Science, Leicester

For the first time in the history of the ICGT conference series, this year a specific event, the *ICGT 2008 Doctoral Symposium*, is explicitly dedicated to Ph.D. students and young researchers who completed their doctoral studies within the past two years. In fact, the Doctoral Symposium consists of some technical sessions, held during the main conference, dedicated to presentations by doctoral students and young researchers, and giving them a unique opportunity to present their work and to interact with established researchers of the graph transformation community and with other students.

We received many submissions of very high quality among which the following sixteen three-pages abstracts are included in the ICGT 2008 proceedings and will be presented at the conference:

1. Dénes Bisztray, *Verification of Architectural Refactorings: Rule Extraction and Tool Support.*
2. Filippo Bonchi, *Abstract Semantics by Observable Contexts.*
3. Duc-Hanh Dang, *Triple Graph Grammars and OCL for Validating System Behavior.*
4. Mike Dodds, *From Separation Logic to Hyperedge Replacement and Back.*
5. Davide Grohmann, *Security, cryptography and directed bigraphs.*
6. Mohammad Hammoudeh, *Modelling Clustering of Sensor Networks with Synchronised Hyperedge Replacement.*
7. Tobias Heindel, *Grammars Morphisms and Weakly Adhesive Categories.*
8. Frank Hermann, *Process Construction and Analysis for Workflows modelled by Adhesive HLR Systems with Application Conditions.*
9. Ákos Horváth, *Towards a Two Layered Verification Approach for Compiled Graph Transformation.*
10. Ajab Khan, *Model-based Analysis of Network Reconfigurations using Graph Transformation Systems.*
11. Carlos Matos, *Service Extraction from Legacy Systems.*
12. Karl-Heinz Pennemann, *Development of Correct Graph Transformation Systems.*
13. Michael Striewe, *Using a Triple Graph Grammar for State Machine implementations.*
14. Pieter Van Gorp, *Model-Driven Development of Model Transformations.*

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 473–474, 2008.

15. Hong Qing Yu and Yi Hong, *Graph Transformation for the Semantic Web: Queries and Inference rules.*
16. Erhard Weinell, *Transformation-based operationalization of Graph Languages.*

Those contributions have been selected according to their originality, significance, and general interest by a few members of the ICGT 2008 Program Committee on the basis of the submitted abstracts. We would like to explicitly thank

- Paolo Baldan
- Luciano Baresi
- Hartmut Ehrig
- Annegret Habel
- Reiko Heckel
- Barbara König
- Hans-Jörg Kreowski
- Dirk Janssens
- Juan de Lara
- Tom Mens
- Mauro Pezzé
- Detlef Plump
- Gabriele Taentzer

for having participated actively to the review process of the abstracts submitted for the Doctoral Symposium.

After the conference, authors of selected contributions will be invited to submit a full paper for the refereed post-proceedings of the Doctoral Symposium, which is expected to be published as a volume of the Electronic Notes in Theoretical Computer Science series.

Verification of Architectural Refactorings: Rule Extraction and Tool Support

Dénes Bisztray

Department of Computer Science, University of Leicester
dab24@mcs.le.ac.uk

1 Introduction

With the recent success of the component-based and service-oriented paradigm, the complexity of software also increased. To tackle complexity, architectural models aid the developers. However, a software in constant use, must continually evolve, otherwise it becomes progressively less satisfactory [1]. During the adaptation to changed requirements and improvement of internal structure, changes may be required to preserve the observable behaviour of the systems. In OO programming, such behaviour-preserving transformations are called refactorings [2].

For distributed and service-oriented applications, the important changes take place at the architectural models. These changes have to be checked for behaviour preservation. To avoid the costly verification of refactoring steps on large systems we extract a (usually much smaller) rule from the transformation performed and verify this rule instead. However, the notion of observable behaviour has to be established, formal requirements for extracting the refactoring rule and methods that can verify its behaviour preservation.

2 Method

The method of verifying architectural level refactorings consists of three ingredients: the modelling language used, its semantics, and the relation capturing our idea of behaviour preservation. For representing the type and instance-level architecture of our system, we use UML *component* and *composite structure diagrams* in conjunction with *activity diagrams* specifying the workflows executed by component instances as described in [3].

To verify the semantic relation between source and target models, the *semantics* of the combined structure diagram is expressed in a denotational style, using Communicating Sequential Processes (CSP) [4] as semantic domain. CSP is a process algebra for concurrent systems.

A mapping *sem* has been defined from the UML diagrams to CSP processes by means of graph transformation rules. The *semantic relation* of behaviour preservation can conveniently be expressed using one of the refinement and equivalence relations on CSP processes.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 475–477, 2008.

3 Rule Verification and Extraction

The verification a refactoring over sufficiently large systems can be very costly, while affecting only a relatively small fragment of the overall model. It would be advantageous if we could focus our verification on those parts of the model that have been changed, that is, verify the refactoring *rules* rather than the actual steps. This is indeed possible, as we shown in [3]: assuming a refactoring $G \Longrightarrow H$, via the graph transformation rule $p : L \rightarrow R$, if the relation $\mathcal{R}$ holds for $sem(L)\ \mathcal{R}\ sem(R)$ then $sem(G)\ \mathcal{R}\ sem(H)$ also holds. This is possible due to the compositional property [3] of the semantic mapping.

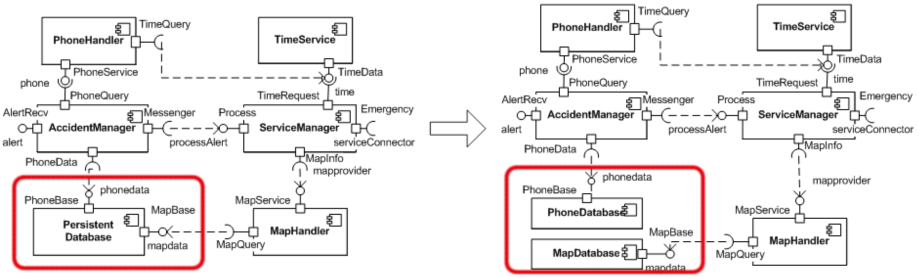


Fig. 1. Example Refactoring

An example refactoring is depicted in Figure 1. As the interfaces are unchanged and the behaviour is encapsulated inside the components, the marked parts can be extracted. Then, this transformation rule can be verified instead of dealing with the complete system. Although this extraction was obvious, there can be complicated refactorings that span change on multiple component behaviour. To determine the mechanics of producing a rule, we perform extractions on proven and successful refactorings. In general for a transformation $G \Longrightarrow H$ with $sem(G)\ \mathcal{R}\ sem(H)$, we want to extract the smallest rule such that:

1. when applying it on G at the appropriate match, the transformation step produces H
2. $sem(L)\ \mathcal{R}\ sem(R)$

As *requirement 1* basically needs an initial pushout [5], extracting a rule that fulfills it is solved. However, determining the necessary context for *requirement 2* is an open question. Tool support is needed not only for verifying complicated system refactorings but also for supporting rule extraction mechanisms.

4 Implementation

A visual editor has been implemented using the Eclipse Graphical Modelling Framework, to support the creation and editing of all three aspects of the software architecture model. As in UML the instantiation is an association between type and instance objects, the components and their instances are situated in the same diagram. The activity diagram of the owned behaviour is embedded inside the components, similarly to *activities* as shown in Figure 1. This combined diagram is called *combined structure diagram*. The metamodel of the combined structure is represented as an Eclipse Modeling Framework (EMF) model which is essentially an attributed typed graphs.

The transformation is implemented using the Tiger EMF Transformer [6] tool. It consists of 45 rules organised in 4 major groups (type-level, owned behaviour, instance-level, renaming) as detailed in [7]. The production rules are defined by rule graphs, namely a left-hand side (LHS), a right-hand side (RHS) and possible negative application conditions (NACs). The rules were designed using the EMT Visual Editor.

FDR2 is a refinement checker for establishing properties of models expressed in CSP [8]. We use FDR2 to check trace refinement in the CSP expressions generated by the EMT transformation.

Currently the architecture and rule design along with the semantic mapping and verification are supported. Automated mechanisms for rule extraction is future work.

References

1. Lehman, M.M.: Laws of software evolution revisited. In: European Workshop on Software Process Technology, pp. 108–124 (1996)
2. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code, 1st edn. Addison-Wesley Professional, Reading (1999)
3. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 347–361. Springer, Heidelberg (2008)
4. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs (1985)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science). An EATCS Series. Springer, New York (2006)
6. Tiger Developer Team: Tiger EMF Transformer (2007), http://www.tfs.cs.tu-berlin.de/emftrans
7. Bisztray, D.: Verification of architectural refactoring rules. Technical Report CS-08-001, Department of Computer Science, University of Leicester (2008), http://www.cs.le.ac.uk/people/dab24/refactoring-techrep.pdf
8. Formal Systems Europe Ltd: FDR2 User Manual (2005), http://www.fsel.com/documentation/fdr2/html/index.html

Abstract Semantics by Observable Contexts

Filippo Bonchi

Dipartimento di Informatica, Università di Pisa
fibonchi@di.unipi.it

The operational behavior of interactive systems is usually given in terms of transition systems labeled with actions, which, when visible, represent both observations and interactions with the external world. The abstract semantics is given in terms of behavioral equivalences, which depend on the action labels and on the amount of branching structure considered. Behavioural equivalences are often congruences with respect to the operations of the language, and this property expresses the compositionality of the abstract semantics.

A simpler approach, inspired by classical formalisms like λ-calculus, Petri nets, term and graph rewriting, and pioneered by the Chemical Abstract Machine [1], defines operational semantics by means of *structural axioms* and *reaction rules.* Process calculi representing complex systems, in particular those able to generate and communicate names, are often defined in this way, since structural axioms give a clear idea of the intended structure of the states while reaction rules, which are often non-conditional, give a direct account of the possible steps. Transitions caused by reaction rules, however, are not labeled, since they represent evolutions of the system without interactions with the external world. Thus reduction semantics in itself is neither abstract nor compositional.

One standard solution, pioneered in [2], is that of defining a *saturated transition system* as follows:

a process p can do a move with label $C[-]$ and become p' iff $C[p] \rightsquigarrow p'$.

Saturated semantics, i.e., the abstract semantics defined over the saturated transition system, are always congruences, but they are usually untractable since they have to tackle all possible contexts of which there are usually an infinite number. Moreover, in several paradigmatic cases, saturated semantics are too coarse. For example, in Milner's Calculus of Communicating Systems (CCS, [3]), saturated bisimilarity cannot distinguish "always divergent processes" and for this reason Milner and Sangiorgi introduced *barbs* [4]. These are observations on the states representing the ability to interact over some channels.

In [5], Sewell introduced a different approach that consists in deriving a transition system where labels are not all contexts but just the minimal ones allowing a system to reach a rule. In such a way, one obtains two advantages: firstly one avoids considering all contexts, and secondly, labels precisely represent *interactions*, i.e., the portion of environment that is really needed to react. This idea was then refined by Leifer and Milner in the *theory of reactive systems* [6], where the categorical notion of *idem relative pushout* precisely captures this idea of minimal context.

The main theorem of this theory guarantees that if relative pushouts (RPOs) exist in the category representing the syntax of the formalism, then the abstract

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 478–480, 2008.

semantics defined over such a derived LTS are congruences with respect to the operators of the language. Since Lawvere-like categories usually do not have RPOs, Milner introduced *bigraphs* [7] with the aim of enconding process calculi whose operational behaviour is expressed by reaction rules and then derive a labeled transition semantics for them.

In this thesis, we try to use *borrowed contexts* rewriting [8] and some encodings of process calculi into graphs. In this perspective, spans of graphs represent reaction rules, double pushout rewrites mimic reduction of processes (represented as graphs) and borrowed contexts rewrites mimic labeled transitions of processes where, again, the labels represents the minimal contexts that is needed to perform a reaction. In [9], we have shown that in the case of CCS, this approach works well, i.e., the derived LTS is very close to the standard one, and the resulting bisimilarity coincides with the standard bisimilarity. Moreover, this approach has some interesting advantages with respect to bigraphs that we cannot detail in this abstract. However for several other interesting formalism the abstract semantics resulting from such approach are too strict. This is not due to our graphical encodings or to the borrowed contexts technique, but it is a bug of the general idea of considering the minimal contexts as labels.

In our opinion, considering as labels the minimal contexts that allow a certain reduction, allows the observer to observes too much, and thus the resulting semantics are usually too fine. One result of the thesis (presented in [10]) is that of providing evidence of this through several interesting formalisms modeled as reactive systems (without using borrowed contexts): Logic Programming, a fragment of open π-calculus, and an interactive version of Petri nets.

Moreover, we introduce two alternative definitions of bisimilarity that *efficiently* characterize saturated bisimilarity, namely *semi-saturated bisimilarity* and *symbolic bisimilarity* [10]. These allow us to reason about saturated semantics without considering all contexts, but saturated semantics are in several cases too coarse. In order to have a framework that is suitable for many formalisms, we add to the above approach *observations*. Indeed, in our opinion, labels cannot represent both interactions and observations, because these two concepts are in general different, like for example, in the asynchronous calculi where receiving is not observable. Thus, we believe that some notion of observation, either on transitions or on states (e.g. barbs [4]), is necessary.

A further result of the thesis (presented in [11]) is that of providing a generalization of the above theory starting not just from purely reaction rules, but from transition systems labeled with observations. Here we can easily reuse saturated transition systems by defining them as follows:

$$\text{a process } p \text{ can do a move with context } C[-] \text{ and observation } o \text{ and become } p' \text{ iff } C[p] \xrightarrow{o} p'.$$

Again, saturated semantics, i.e. abstract semantics defined over the above transition systems, are congruences. Analogously to the case of reactive systems, we can define semi-saturated bisimilarity and symbolic bisimilarity as efficient characterizations of saturated semantics. The definition of symbolic bisimilarity which arises from this generalization is similar to the abstract semantics of several works [12,13,14,15]. Here (and in [11]) we consider open [14] and asynchronous [12,16] π-calculus, by showing that their abstract semantics are instances of our general concepts of saturated and symbolic semantics. We also apply our approach to open Petri nets [17] (that are an interactive version of P/T Petri nets) obtaining a new symbolic semantics for them, that efficiently characterizes their abstract semantics.

References

1. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science 96, 217–248 (1992)
2. Montanari, U., Sassone, V.: Dynamic congruence vs. progressing bisimulation for ccs. Fundamenta Informaticae 16(1), 171–199 (1992)
3. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
4. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
5. Sewell, P.: From rewrite to bisimulation congruences. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 269–284. Springer, Heidelberg (1998)
6. 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)
7. Milner, R.: Bigraphical reactive systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 16–35. Springer, Heidelberg (2001)
8. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 151–166. Springer, Heidelberg (2004)
9. Bonchi, F., Gadducci, F., König, B.: Process bisimulation via a graphical encoding. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 168–183. Springer, Heidelberg (2006)
10. Bonchi, F., König, B., Montanari, U.: Saturated semantics for reactive systems. In: Logic in Computer Science, pp. 69–80. IEEE, Los Alamitos (2006)
11. Bonchi, F., Montanari, U.: Symbolic semantics revisited. In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 395–412. Springer, Heidelberg (2008)
12. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 147–162. Springer, Heidelberg (1996)
13. Buscemi, M., 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)
14. Sangiorgi, D.: A theory of bisimulation for the pi-calculus. Acta Informatica 33(1), 69–97 (1996)
15. Wischik, L., Gardner, P.: Explicit fusions. Theoretical Computer Science 340(3), 606–630 (2005)
16. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
17. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open petri nets based on deterministic processes. Mathematical Structures in Computer Science 15(1), 1–35 (2005)

Triple Graph Grammars and OCL for Validating System Behavior*

Duc-Hanh Dang

University of Bremen, Computer Science Department, Database Systems Group
hanhdd@informatik.uni-bremen.de

Abstract. We propose an approach based on the integration of Triple Graph Grammars (TGGs) and the Object Constraint Language (OCL) for checking the conformance between use case and design models.

1 Introduction

Triple graph grammars (TGGs) have been first formally proposed in [1]. They seem to be a promising approach for explaining relationships between models within model-driven development. A need and a challenge within software engineering is how to explain the relationship between behavioral models, a use case model as a functional requirement model and a design model as a realization of use cases. The major difficulty in this relationship lies in the informality of use cases, and the loose relationship between the use case model and the design model. We propose a novel approach employing TGGs for checking the conformance between a use case model and a design model. Within our approach, an integration of TGGs and OCL is proposed and implemented in our tool UML-based Specification Environment (USE).

A use case is a textual description of the system's behavior as the system responds to a request from actors (user types). A use case model may be realized by a design model. The relationship between the use case model and the design model should be a behavioral refinement. Explaining the relationship as well as concepts of refinement is a topic of ongoing research activities.

In [2], we propose an approach for explaining this relationship. The heart of the approach is to check the synchronization between system scenarios at the use case level and the design level. A step in a system scenario at the use case level may define the next step in the corresponding scenario at the design level, and vice versa, a step at the design level may define the next step at the use case level. In order to realize the approach, we define metamodels of system models at the use case level and the design level. The metamodels are extended by graph transformation rules, which allow defining a dynamic model evolution as a simulation of system evolution. We employ TGGs in order to relate transformation rules of the graph transformation systems towards synchronization between system scenarios.

[*] I would like to thank my supervisor Martin Gogolla for contributions to this work.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 481–483, 2008.

Within our approach, a mechanism for checking the conformance between a use case model and a design model is established. Test cases, which are generated from the use case model, are employed for defining scenarios. The mechanism represents system states in an easy-to-understand and visual form. It is significant for the early design phase. Within our approach, we do not address structural refinement: use case and design models share a domain class diagram. However, our approach has the possibility to support the structural refinement.

Some technical challenges arise within our approach. Our approach requires the integration of TGGs and OCL. In the recent literature, a need for the integration has been expressed. TGGs and QVT (Queries, Views, Transformations) share many building blocks. But in contrast to TGGs, QVT includes the declarative language OCL, which allows to express properties and to navigate in complex models. Attaching OCL constraints to TGG rules seems not difficult, but a challenge arises when inducing operations from the integration. To the best of our knowledge, an approach for the integration has not been proposed. Another challenge within our approach is that use cases and a design model must be specified as graph transformation systems. Some papers have specified a design model, e.g., UML interaction diagrams, as a graph transformation system. However, a metamodel for representing scenarios has not been proposed.

The rest of this paper is structured as follows. Section 2 discusses our approach for the integration of TGGs and OCL. Section 3 describes how TGGs can be employed for synchronizing scenarios. The paper is closed with a summary of our expected results. Due to the paper format, full references are not included. The final work will include a detailed discussion of related work.

2 Integrating TGGs and OCL

This section discusses our approach for the integration of TGGs and OCL. First of all, we define requirements for the integration of TGGs and OCL as follows.

- Supporting OCL formulas for attribute expressions.
- Supporting OCL conditions as pre- and posconditions of TGG rules (corresponding to negative application conditions and the right-hand side of TGG rules respectively), which cannot be represented by graphs.
- Supporting constraints on the metamodel of the middle domain of TGG rules. (Supposing that all of the constraints must be taken from TGG rules)
- Supporting bi-directionally executable TGG rules in a synchronous way.

We propose the language USE4TGG for the integration of TGGs and OCL: One part of USE4TGG has a one-one mapping to TGGs and the remaining part covers OCL concepts. The approach is similar to our previous work [3,4]. A TGG rule in USE4TGG is transformed into an operation, which is realized in USE by taking two views on it: OCL pre- and postconditions are employed as operation contracts, and command sequences are taken as an operation realization.

USE4TGG fulfills all above requirements except the final requirement. Our restriction is that OCL formulas must be presented in an assignment-like style if we want to obtain a corresponding operation realization. We plan to develop an algorithm for translating TGG rules into operations for synchronizing models.

3 TGGs and OCL for Synchronizing Scenarios

This section describes our approach employing TGGs and OCL for synchronizing scenarios of a use case model and a design model.

In order to define graph transformation systems controlling scenarios of a use case model and a design model, we have already developed the language UCL for specifying use cases and the language DML for specifying scenarios for executing design models. A set of rules for selecting scenarios of the use case model and the design model has been defined. Due to the paper format, we do not describe these languages in detail.

Effects of TGG rules on scenarios at the use case level include: (1) To assign input values to variables; (2) to select the next activity; (3) to select the next activity from effects of the design model. Effects of TGG rules on scenarios at the design level include: (1) To assign input values to variables and to select the next operation; (2) to select the next action of the current operation; (3) to select the next operation corresponding to the current activity of use cases.

Our next task is to define TGGs rules for relating the graph transformation systems, and to implement a TGG engine for realizing the synchronization.

4 Expected Contributions

The followings are expected results of our work.

- Discovering a new application scenario of TGGs for synchronizing graph transformation systems. The result is significant for software engineering.
- Proposing an approach for the integration of TGG and OCL and implementing a corresponding TGG engine in the USE tool.
- Proposing and realizing an approach based on TGGs for checking the conformance between a use case model and a design model.

References

1. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, S. (ed.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
2. Dang, D.H.: Validation of System Behavior Utilizing an Integrated Semantics of Use Case and Design Models. In: Pons, C. (ed.) Proc. Doctoral Symposium ACM/IEEE 10th Int. Conf. MoDELS 2007, CEUR Workshop Proceedings, vol. 262 (2007)
3. Gogolla, M., Büttner, F., Dang, D.H.: From Graph Transformation to OCL using USE. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2008. Springer, Berlin (2008)
4. Dang, D.H., Gogolla, M., Büttner, F.: From UML Activity Diagrams to CSP Processes: Realizing Graph Transformations in the UML and OCL Tool USE. In: AGTIVE 2007 Tool Contest, 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance (2007), http://gtcases.cs.utwente.nl/wiki/UMLToCSP/USE

From Separation Logic to Hyperedge Replacement and Back

(Extended Abstract)

Mike Dodds

University of Cambridge, UK[*]
md466@cl.cam.ac.uk

1 Introduction

Hyperedge replacement grammars and separation-logic formulas both define classes of graph-like structures. In this paper, we describe two effective translations between restricted hyperedge replacement grammars and formulas in a fragment of separation logic. These translations preserve the semantics of formulas and grammars.

Hyperedge-replacement grammars [1] are a natural extension of context-free string grammars to the world of hypergraphs. An HR grammar defines language of structures that can be constructed from an initial graph.

Separation logic [2] is a recently-developed logic for specifying the properties of heaps that extends normal first-order logic with a so-called *separating* conjunction. This allows a formula to specify the *spatial relationships* between assertions in the heap. Recent work based on separation logic has made considerable progress in verifying programs with pointers [3].

Our translations demonstrate that formulas in our fragment of separation logic are of *corresponding expressive power* to HR grammars under our restrictions. We have used this correspondence to prove some interesting results about our fragment of separation logic using the theoretical results for hyperedge-replacement grammars.

2 The Intuitive Correspondence

A correspondence exists between separation logic formulas and hyperedge replacement grammars because (1) the recursive definitions commonly used in separation logic closely resemble hyperedge replacement productions, and (2) the separating property enforced by separating conjunction corresponds to the context-free property of hyperedge-replacement grammars.

The following example illustrates this correspondence.

Example 1 (Binary tree). The predicate $bt(x)$ is defined as the least predicate satisfying the following equality.[1]

[*] Work completed during study towards a PhD at the University of York.

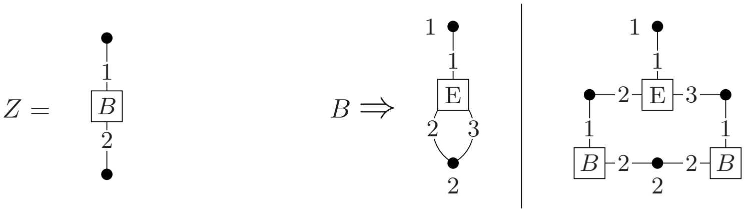
[1] To express this formula in our fragment of separation logic we make use of a recursive 'let' operator. For simplicity this is omitted from the example.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 484–486, 2008.

$$bt(x_1) = (x_1 \mapsto \mathsf{nil}, \mathsf{nil}) \vee (\exists x_2, x_3 : (x_1 \mapsto x_2, x_3) * bt(x_2) * bt(x_3))$$

$bt(x)$ is satisfied if either x points to a location holding a pair of nil-values, or if x points to a pair of locations, both of which also satisfy bt. The separating conjunction $*$ between the branch and the two subtrees differs from conventional conjunction in that it prohibits sharing between its conjuncts. This enforces the tree property by preventing sharing between the subtrees.

This predicate definition corresponds to the hyperedge replacement grammar $BT = \langle T, N, Z, P\rangle$. This defines the language of binary tree graphs with a shared leaf. The sets of terminal and non-terminal edge labels are respectively $T = \{E\}$ and $N = \{B\}$. The initial graph Z and set of productions P are:



The root (top node) of the initial graph Z corresponds to the variable x with which the predicate bt is called, while the leaf (bottom node) of Z corresponds to the nil constant. The individual cases of the production defined for label B in the grammar correspond to the two disjuncts defining the predicate bt. The first disjunct corresponds to a terminal branch, and the second a branch and a pair of child trees.

3 Defining the Translations

We define a mapping $g[\![\circ]\!]$ from separation logic formulas to hyperedge replacement grammars, and a reverse mapping $s[\![\circ]\!]$ from hyperedge-replacement grammars to separation logic formulas.

The mappings $g[\![\circ]\!]$ and $s[\![\circ]\!]$ are defined syntactically as functions over the structure of a grammar and a formula respectively. The elements of hyperedge replacement grammars and separation logic formulas are related as follows:

- The productions over a single nonterminal symbol in the grammar corresponds to the definition of a single recursive predicate in separation logic.
- Terminal edges in the grammar's initial graph and in the right-hand sides of productions correspond to separation logic's points-to assertion ($\mapsto$).
- Non-terminal edges in the grammar correspond to instances of recursively-defined predicates in separation logic. The attachment nodes of the edges correspond to the arguments passed to the predicate.

Hyperedge replacement grammars define languages of graphs, while separation logic formulas define classes of graph-like states. To resolve this mismatch between the domains we define the notion of a *heap-graph* for graphs which correspond to states, and a bijective mapping α from states to heap-graphs. Each node in a heap graph can be the first attachment point to at most one terminal edge. Our mappings are defined only over grammars which construct sets of heap-graphs.

Our mappings are defined over a fragment of full separation logic as given in (for example) [2]. This fragment includes separating conjunction ($*$), disjunction ($\vee$), 'points-to' ($\mapsto$) and existential quantification ($\exists$). To allow recursive definitions, we introduce a recursive let construct ($\mathsf{let}\ \Gamma\ \mathsf{in}\ P$). Omitted operators include conjunction, negation, and separating implication (the adjoint of separating conjunction).

4 Results

Our major results are as follows:

1. We have proved that our definitions of both $g[\![\circ]\!]$ and $s[\![\circ]\!]$ are *semantics preserving* modulo the mapping α. That is, $\alpha \circ g = g \circ \alpha$, and $\alpha^{-1} \circ s = s \circ \alpha^{-1}$.
2. As a consequence of (1), our fragment of separation logic is of *equivalent expressive power* to hyperedge-replacement grammars for heap graphs.
3. We have proved that the operators omitted from our fragment of separation logic *cannot* be simulated by a corresponding hyperedge replacement grammar. Notably conjunction corresponds to language intersection, and negation to language complement, both of which are known to be HR-inexpressible.
4. As a consequence of (2), results for hyperedge replacement languages, such as the inexpressibility results, can be imported into the fragment of separation logic. For example, the languages of red-black trees, balanced binary trees, grid graphs are all known to be HR-inexpressible. Therefore no formula exists in our fragment which is satisfied by the class of states containing these structures.

References

1. Drewes, F., Kreowski, H.-J., Habel, A.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1, pp. 95–162. World Scientific, Singapore (1997)
2. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science (2002)
3. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2006)

Security, Cryptography and Directed Bigraphs

Davide Grohmann

Department of Mathematics and Computer Science, University of Udine, Italy

Bigraphical reactive systems are an emerging graphical framework proposed by Milner and others [4,5,2,3] as a unifying theory of process models for distributed, concurrent and ubiquitous computing. A bigraphical reactive system consists of a category of *bigraphs* (usually generated over a given *signature of controls*) and a set of *reaction rules*. Bigraphs can be seen as representations of the possible configurations of the system, and the reaction rules specify how these configuration can evolve. The advantage of using bigraphical reactive systems is that they provide general results for deriving a labelled transition system *automatically* from the reaction rules, via the so-called *IPO construction*. Notably, the bisimulation on this transition system is always a congruence.

A bigraph is a set of nodes (the *controls*), endowed with two independent structures, the *place graph* and the *link graph*. The place graph is a tree over the nodes, representing the spatial arrangement (i.e., nesting) of the various components of the system. The link graph represents the communication connections, possibly traversing the place structure. Intuitively, edges represent *(delocalized) resources*, or *knowledge tokens*, which can be *accessed* by controls. Arcs are arrows from ports of controls to edges, possibly through names on the interfaces of bigraphs. A bigraph may be "not ground", in the sense that it may have one or more "holes", or *sites* to be instantiated; these holes are specific leaves of the place graph, where other bigraphs can be grafted, respecting the connection links. This notion of composition between bigraphs yields a categorical structure.

In this paper, we will study how to model key-passing and authentication protocols on insecure networks inside bigraphical framework. In literature, many calculi have been introduced to deal with security aspects, as the spi-calculus [1], and they are used as frameworks to prove process equivalences. Indeed, many protocols can be analyzed checking if its specification and implementation are observational equivalent, that is the implementation does all the requested actions defined by the specification and nothing more. For these reasons, we will study the IPO bisimilarity, and which security properties are ensured by it.

For describing how to model security protocols inside bigraphs, we consider as a running example the *wide mouth frog* protocol. This protocol can be described as follows (using the well-known Alice-Bob notation):

$$\begin{aligned} A \to S:&\quad id_A, \{K_{AB}, id_B\}_{K_{AS}} \\ S \to B:&\quad \{K_{AB}, id_A\}_{K_{BS}} \end{aligned} \tag{1}$$

The protocol suppose the existence of a server (S) trusted by both parties, i.e. A and B share a key each with S (K_{AS} and K_{BS}, respectively).

Informally, the agent A aims to communicate with B opening a secure channel using the key K_{AB}. To do so, A sends a message to S saying that it wants to talk

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 487–489, 2008.

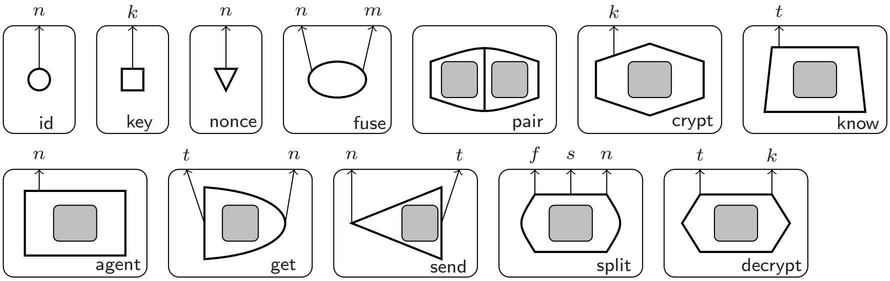


Fig. 1. Examples of nodes in the signature for protocols

with B using the key K_{AB}. Then, S sends a message to B, saying that A desires to talk with him using K_{AB} for encrypting messages. At this point, both A and B know the key K_{AB} and they are able to establish a secure communication.

The bigraphical signature used to encode agents, keys, messages and terms is graphically and informally described in Figure 1.

As shown in the example protocol (1), the exchanged messages are tuples of (possibly encrypted) basic data: keys, identifiers or nonces (i.e., "number used once", which are useful in avoiding reply attacks). Terms can be naturally encoded using the place structure of the bigraph. For this purpose, there is a control for every basic term: id, key and nonce. Next, there are constructor controls with holes: pair for constructing pairs: its holes contain the encoding of the two subterms; and encrypt for constructing encrypted data: its hole contains the term to encrypt, and its port is linked to the key used in the encryption.

The control known is used to assign aliases to complex terms, this is useful for two reasons: in the communication we can send/receive only aliases, and an agent knowledge is encoded (and inferred) by arcs from agent controls to edges linked to "terms", i.e., know, id, key and/or nonce controls.

The agents are represented as locations containing actions and "pointers" to known terms or keys. The actions are encoded using the controls get, send for the communication and the destructors split and decrypt, that resemble the case-operator of the spi-calculus. As usual, the prefix is encoded using the place graph structure: an "action node" is disabled if it is inside another one.

The reactive rules are described in Figure 2. The first three rules describe the communication mechanism, similarly to the Fusion calculus [6,3]. The last two rules describe how to access the subterms using the destructors. Due to lack of space, these rules are described only in the case where pair and crypt contain complex terms (know nodes), but there are rules for the other cases too.

Finally, in Figure 3, it is shown an encoding of the wide mouth frog protocol. The agent A is sending the encoding of the first message to the server S along the channel c_{AS}. The two holes are fitted with some bigraphs P and Q containing the next steps necessary to complete the protocol: P contains actions to communicate with B, and Q contains the controls necessary to deconstruct the message from A and to construct a message for B using the new information.

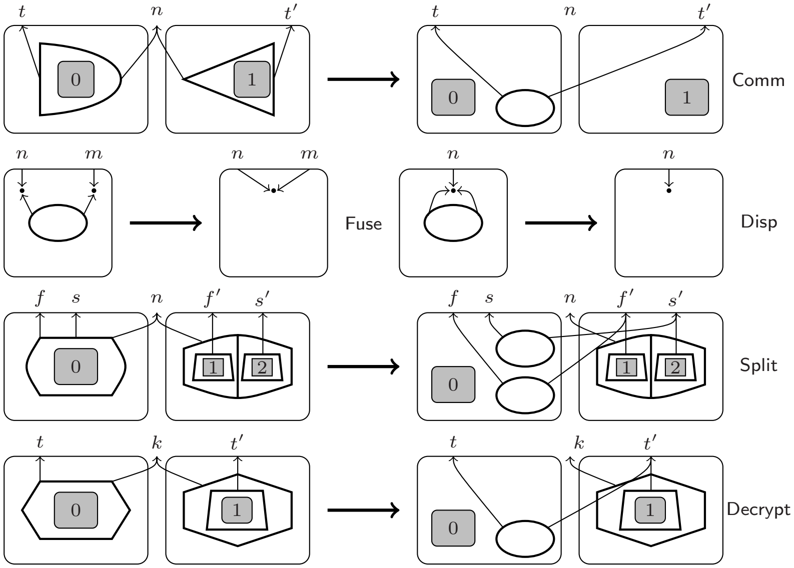


Fig. 2. Reaction rules for protocols

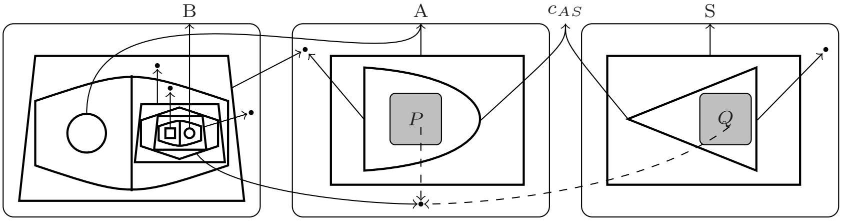


Fig. 3. The first step of the wide mouth frog protocol

References

1. Abadi, M., Gordon, A.D.: A Calculus for Cryptographic Protocols: The Spi Calculus. In: ACM Conference on Computer and Communications Security (1997)
2. Grohmann, D., Miculan, M.: Directed bigraphs. In *Proc. XXIII MFPS*. Electronic Notes in Theoretical Computer Science, vol. 173. Elsevier, Amsterdam (2007)
3. Grohmann, D., Miculan, M.: Reactive systems over directed bigraphs. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703. Springer, Heidelberg (2007)
4. Milner, R.: Bigraphical reactive systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154. Springer, Heidelberg (2001)
5. Milner, R.: Pure bigraphs: Structure and dynamics. Information and Computation 204(1), 60–122 (2006)
6. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proceedings of LICS 1998. IEEE Computer Society Press, Los Alamitos (1998)

Modelling Clustering of Sensor Networks with Synchronised Hyperedge Replacement

Mohammad Hammoudeh

School of Computing and IT, University of Wolverhampton,UK
m.h.h@wlv.ac.uk

1 Introduction

This paper proposes *Synchronised Hyperedge Replacement* (SHR) as a suitable modelling framework for Wireless Sensor Networks (WSN). SHR facilitates explicit modelling of WSN applications environmental conditions (that greatly affect the applications performance) while providing a sufficiently high level of abstraction for the specification of the adopted clustering mechanism. We model with SHR few communication and coordination aspects of a new algorithm, *Balanced Minimum Radius Clustering* (BMRC), for solving the balanced clustering problem.

The key contribution of this work is to demonstrate that SHR is sufficiently expressive to describe algorithms, such as balanced clustering algorithms, and can describe their behaviour at a suitable level of abstraction to allow onward analysis. We also propose a new cluster balancing algorithm to be used as a working example. We have chosen this particular algorithm because it is an intractable problem to solve in a distributed manner, and distribution is important, by reason of both avoiding specialised node vulnerability and minimising message overhead.

Developing, testing, and evaluating network protocols, architectures and services for WSNs are typically undertaken through test-beds or simulation. The substantial cost of deploying and maintaining large-scale WSNs and the time needed for setting up the network for experimental goals makes simulation invaluable in developing reliable and portable applications for WSNs. However, simulation in WSNs has its own set of problems. For instance, test-bed implementation is not always viable because it does not easily scale to a large number of nodes; also, some simulators are acknowledged to be difficult to use, platform dependent, containing inherent bugs, distributed under commercial licenses, etc.

We argue that abstract models may provide cost effective methods for analysing WSNs before deployment. They can, for example, help in assessing the scalability of algorithms independently from target hardware platforms and with a certain degree of accuracy; or else, they may simplify the software development process for WSN applications. One of the main difficulties in modelling WSNs lies in their intrinsic interdependency with the environment in which they operate. WSN applications attribute integration of communication, computation, and interaction with the physical environment. The environment is the physical phenomena and layout which has effect on the functionality of a sensor

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 490–492, 2008.

network application. The layout includes different types of surfaces, each affecting radio propagation characteristics in a different way. Therefore, interactions, coordination and performances of WSNs are greatly affected by environmental conditions. As a result, the lack of the environmental conditions in any modelling framework may diminish the realism of the modelling results.

To the best of our knowledge, a suitable abstract modelling framework encompassing these peculiarities of WSNs are still missing. Available models represent and study various network aspects of WSNs but neglect environmental issues. We contend that a modelling framework for WSNs to cover these concerns is required. In [2], a first attempt to specify such a model has been proposed; it is shown how SHR can suitably represent several facets of WSNs mechanisms in a unique formal framework.

We apply here the ideas in [2] to represent the communication and coordination aspects of a new clustering algorithm. As discussed later, SHR allows us to uniformly represent different abstraction levels. Specifically, we show how we can explicitly model very low level aspects (e.g., the wireless communications required in the preliminary phase of BRMC) along with abstract ones (e.g., spatial distribution or the sensors interaction mechanisms more suitable in modelling communications between sensors and cluster-heads). Another advantage offered by SHR is that it separates the coordination aspects of the algorithm from the data-related aspects. In fact, the metrics used to balance the clusters is parametric with respect to the SHR productions describing sensors coordination. This makes our approach suitable for studying classes of algorithms obtained by varying the metric criteria.

2 Balanced Minimum Radius Clustering

Clustering is another very important optimization problem in WSNs. For brevity, we refer readers to the recent survey on clustering algorithms for WSNs by Abbasi et. al [1].

We propose the BMRC algorithm to generate optimally balanced clusters based on unbalanced clusters. The distributed balanced clustering consists of four different steps: (1) Local clustering; (2) Determination of a local model; (3) Determination of a global model which is based on all local models; (4) Finally, updating of all local models.

In BMRC, the nodes are clustered locally and suitable representatives out of these clusters are extracted by the respective cluster-head. These representatives are sent to the sink node where the complete clustering based on the local representatives is balanced. This approach is very efficient, because the local clustering can be carried out quickly and independently. Furthermore, it achieves lower transmission cost, as the number of transmitted representatives is much smaller than the cardinality of the complete data set. Based on this small number of representatives, the global cluster balancing can be done very efficiently. All parts of the protocol have been modelled using SHR, from cluster formation, to cluster balancing.

3 Final Remarks

This paper has posited a novel approach to modelling one of the major practical problems facing designers of large WSNs. As has been argued, the traditional approach of simulation can be of limited use, both due to the difficulty of building a simulation using

Current tools, and due to the computational resources needed to analyse a network of substantial size. Also, a simulation is not an intellectually satisfying proof of good behaviour. Analytical examination of the network offers the possibility of proving at least some of its behaviours, and may therefore be able to provide answers where simulation cannot. It is argued here that SHR is a suitable and appropriate modelling framework for this type of problem and this has been demonstrated by the modelling of the BMRC cluster balancing protocol using it. Cluster balancing was chosen as a working example because it is an intractable problem to solve in a distributed manner.

This paper provides a model around which an analytical proof enterprise may be situated. In the process, it has been demonstrated that SHR is sufficiently expressive to describe such a protocol, and can describe its behaviour at a suitable level of abstraction to allow onward analysis. All parts of the protocol have been modelled, from cluster formation, to cluster balancing. These models lay the groundwork for future analysis of BMRC and other protocols, and the prospect of implementation of systems which are genuinely scalable, efficient and reliable, by dint of the proof of those properties as part of the design process. This type of analysis will not replace simulation, but it does provide a means for obtaining the kind of open-ended guarantees that simulation cannot give.

References

1. Abbasi, A.A., Younis, M.: A survey on clustering algorithms for wireless sensor networks. Comput. Commun. 30(14-15), 2826–2841 (2007)
2. Tuosto, E.: Tarzan: Communicating and Moving in Wireless Jungles. In: Cerone, A., Di Pierro, A. (eds.) 2nd Workshop on Quantitative Aspects of Programming Languages. ENTCS, vol. 112, pp. 77–94. Elsevier, Amsterdam (2005)

Grammar Morphisms and Weakly Adhesive Categories

Tobias Heindel

Institut für Informatik und angewandte Kognitionswissenschaften,
Universität Duisburg-Essen, Germany

In the area of specification and modelling of concurrent systems, Petri nets have become a standard tool, and they still work behind the scenes in tools for graph transformation systems (cf. [1]). Moreover there is still potential for cross-fertilization between the graph transformation and Petri net community. Even a better understanding of adhesive categories [2] and the related concepts of [3] seems possible in light of the proposed notion of *weakly adhesive categories* [4], which has emerged during work on the generalization of the co-reflective semantics of Petri nets to the realm of graph transformation and adhesive rewriting systems.

As an example of the interplay between graph transformation and Petri nets, take the notion of net *morphism*; it was introduced in [5] as a means to compose Petri Net models of concurrent computational systems via synchronization, which corresponds to taking the product in the *category* of Petri nets. While "it took the quarter of a century from the inception of Petri nets [. . .] to the definition of their morphisms"[1] it took another ten years to find *grammar morphisms* that are suitable to generalize the unfolding of Petri nets in the style of [7] to graph grammars (cf. [8,9]).

In this abstract, it will be outlined how the first attempts to generalize the unfolding semantics of graph grammars to adhesive rewriting systems have led to the definition of *weakly adhesive categories* (see Definition 1). The first approach was based on the restriction that grammar morphisms between T- and T'-typed grammars, for given type objects T and T' of a suitable category $\mathbb{C}$ (of graphs), have to be based on $\mathbb{C}$-spans $T \overset{u}{\leftarrowtail} U \xrightarrow{f} T'$ with monic u. Later it turned out that these grammar morphism might better be thought of as "rule-structure"-preserving functors of the form $f_! \circ u^*\colon \mathbb{C}{\downarrow}T \to \mathbb{C}{\downarrow}T'$, where the functor $u^*\colon \mathbb{C}{\downarrow}T \to \mathbb{C}{\downarrow}U$ acts by pulling back along u, and $f_!\colon \mathbb{C}{\downarrow}U \to \mathbb{C}{\downarrow}T'$ (post-)composes with f. Note that the action of the functor $f_! \circ u^*$ is closely related to the idea of retyping of [8], and recall that a $\mathbb{C}$-span $T \overset{u}{\leftarrowtail} U \xrightarrow{f} T'$ with monic u is essentially a *partial map* (see [10] for details).

The crucial observation about the unfolding construction of [8] is that it cannot only be characterized as a functor that forms part of a co-reflection, but moreover it can be described as an (ω-chain) co-limit in the category of grammars and grammar morphisms. More precisely, the unfolding U_G of a T-typed grammar G is the co-limit object of a chain of (occurrence) grammars

[1] This quote is from [6].

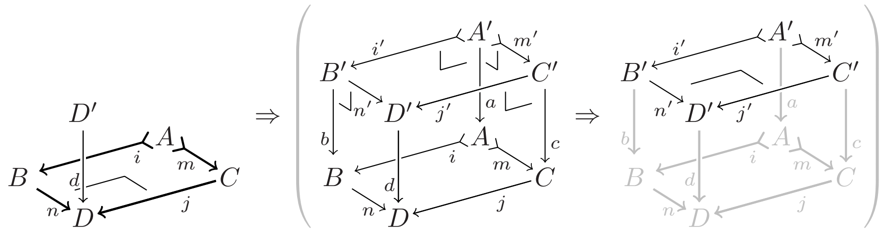
H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 493–495, 2008.

G_0 "$\subseteq$" G_1 "$\subseteq$" $\cdots$ "$\subseteq$" G_n "$\subseteq$" $\cdots$. Moreover the type object T_U of the unfolding U_G is the $\mathbb{C}$-co-limit object of the chain of type objects $T_0 \rightarrowtail T_1 \rightarrowtail \cdots T_n \rightarrowtail \cdots$ where each T_i is the type object of the grammar G_i.

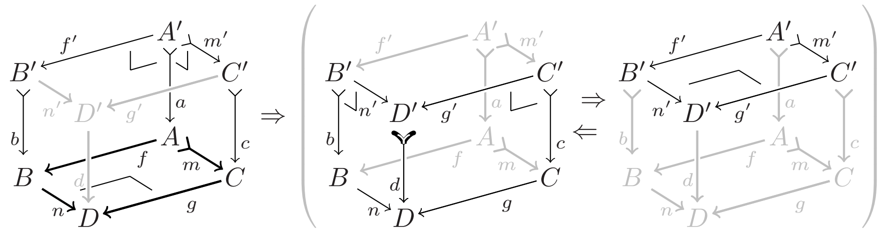
The main ingredient of this co-limit characterization of the unfolding relies on the fact that the latter $\mathbb{C}$-co-limit is preseved by the inclusion of the category $\mathbb{C}$ into the associated category of partial maps $\mathrm{Par}(\mathbb{C})$ (see [10]). This is ensured by the theorem that (any topos) $\mathbb{C}$ is a co-reflective subcategory of the category of partial maps $\mathrm{Par}(\mathbb{C})$, which implies that *all* co-limits are preserved by the inclusion $\mathbb{C} \subseteq \mathrm{Par}(\mathbb{C})$. Further, in (any topos) $\mathbb{C}$, all co-limits are universal. These two phenomena, namely universality of co-limits, and preservation of co-limits by the inclusion $\mathbb{C} \subseteq \mathrm{Par}(\mathbb{C})$, were the main inspiration for the following definition.

Definition 1 (Weakly adhesive category). *A category is* weakly adhesive *if*

1. *pullbacks and pushouts along monomorphisms exist, i.e. for each co-span $A \xrightarrow{f} D \xleftarrow{m} M$ with monic m, a pullback $A \xleftarrow{n} N \xrightarrow{g} M$ exists, yielding a* pullback square $\substack{A\\D}\!\downarrow\!\overset{\leftarrow}{\underset{\leftarrow}{\lrcorner}}\!\downarrow\!\substack{N\\M}$, *and for each span $B \xleftarrow{f} A \xrightarrow{m} C$ with monic m, a pushout $B \xrightarrow{n} D \xleftarrow{g} C$ exists, yielding a* pushout square $\substack{B\\D}\!\downarrow\!\overset{\leftarrow}{\underset{\leftarrow}{\lrcorner}}\!\downarrow\!\substack{A\\C}$*;*
2. *pushouts of pairs of monomorphisms are universal (or stable under pullback), i.e. in each commutative cube over a pushout square $\substack{B \leftarrow A \searrow \\ \searrow D \leftarrow C}$ having pullback squares as lateral faces as shown in the middle diagram in the display below, the top face is a pushout square;*



3. *pushouts along monomorphisms are* mono-universal *and* converse mono-universal: *in each commutative cube on top of a pushout square $\substack{B \leftarrow A \searrow \\ \searrow D \leftarrow C}$ as in the left diagram in the display below, with pullback squares as back faces and the "corner"-arrows b and c monic, its* top face is a pushout square *if and only if* the front faces are pullback squares and the morphism d is monic.



Note that adhesive categories [2], apart from having all pullbacks, satisfy the simpler (and stronger) version of Condition 3 that does not contain any conditions on the vertical morphisms a, b, c and d. Condition 3 is equivalent to the requirement that pushouts along monomorphisms are *hereditary* in the sense of [11]. Finally, the subtle difference to the weak adhesive HLR-categories of [3] is that in the definition of the latter, the vertical morphism d into the "tip" of the bottom pushout is required to be monic already in the antecedent, which implies the "top face-front faces"-equivalence. Note that this definition can be generalized straightforwardly to $\mathcal{M}$-weakly adhesive categories by using an *admissible* class $\mathcal{M}$ (see [10]) of monomorphisms instead of the class of all monomorphisms.

I conclude with the fact that every adhesive category is weakly adhesive, and every $\mathcal{M}$-weakly adhesive category is a weak HLR category w.r.t. $\mathcal{M}$, while it is open whether the converse of the latter holds in general, although I expect that virtually all weak adhesive HLR categories w.r.t. $\mathcal{M}$ that will occur in practical applications are actually $\mathcal{M}$-weakly adhesive.

References

1. König, B., Kozioura, V.: Augur – a tool for the analysis of graph transformation systems. Bulletin of the EATCS 87, 126–137 (2005)
2. Lack, S., Sobocinski, P.: Adhesive categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
3. Ehrig, H., Prange, U.: Weak adhesive high-level replacement categories and systems: A unifying framework for graph and petri net transformations. In: Futatsugi, K., Jouannaud, J.P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 235–251. Springer, Heidelberg (2006)
4. Baldan, P., Corradini, A., König, B., Heindel, T., Sobociński, P.: Unfolding weakly adhesive grammars. CONCUR 2008 (submitted, 2008)
5. Winskel, G.: Petri nets, algebras, morphisms, and compositionality. Information and Computation 72(3), 197–238 (1987)
6. Bednarczyk, M.A., Borzyszkowski, A.M.: General morphisms of petri nets (extended abstract). In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 190–199. Springer, Heidelberg (1999)
7. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)
8. Baldan, P., Corradini, A., Montanari, U.: Unfolding of double-pushout graph grammars is a coreflection. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 145–163. Springer, Heidelberg (2000)
9. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: The category of typed graph grammars and its adjunctions with categories. In: Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G. (eds.) Graph Grammars 1994. LNCS, vol. 1073, pp. 56–74. Springer, Heidelberg (1996)
10. Robinson, E., Rosolini, G.: Categories of partial maps. Information and Computation 79(2), 95–130 (1988)
11. Kennaway, R.: Graph rewriting in some categories of partial morphisms. In: Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 490–504. Springer, Heidelberg (1991)

Process Construction and Analysis for Workflows Modelled by Adhesive HLR Systems with Application Conditions

Frank Hermann

Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Germany

Abstract. Graph transformation systems (GTS) are suitable for modelling concurrent and distributed behaviour of systems and in particular of workflows. Analysis of the behaviour of these models is in general highly complex, but it is of main interest, especially for optimizing the system execution. Main focus of the PhD project is a formal approach for constructing the process of a workflow scenario to support possibilities of efficient analysis and execution. Based on the abstract framework of adhesive high level replacement systems the developed techniques will be applied on two levels. First, the framework is instantiated to different kinds of graph as well as Petri net transformation systems, which are key ingredient for modelling mobile networks in [1]. In the second level, the modelling techniques are used to specify the production of industrial products, which can involve several thousands of production steps. A case study will show how a chain of production steps taken from a real production facility can be modelled as GTS derivation. Formal techniques for process construction and analysis known for basic cases only have to be extended in various dimensions in order to be applied to the model and in general to the domain of workflows. A practical evaluation will compare the results with those derived by standard techniques for process analysis.

State of the Art. Graph transformation offers a rule based modelling of systems. One execution of the system corresponds to a specific derivation of the GTS. A process of this execution describes all possible equivalent executions. Correspondingly, a process of a derivation defines an equivalence class of derivations. Processes of graph transformation systems based on the double pushout (DPO) approach [2] were defined as occurrence grammars in [3], where causal relation together with asymmetric conflict relation allow analysis of dependencies and conflicts. Occurrence grammars were lifted to the abstract setting of adhesive rewriting systems [4] in order to generalize the process construction. This opened possibilities for analyzing processes of transformation systems based on arbitrary adhesive categories [5], such as typed graphs, graphs with scopes and graphs with second order edges.

Several extensions for GTSs, such as attribution, type graphs with inheritance as well as positive and negative application conditions [6], were introduced to allow specification of practical relevant models. These features imply additional

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 496–498, 2008.

constraints for dependencies in a derived process such that equivalences change as well. Main challenge is an extension of the process construction, such that it can be applied to the mentioned highly featured models.

First contributions. Subobject transformation systems (STSs) [7] are generalized occurrence grammars, which coincide with them if constructed as a process of a derivation, but they are less restrictive in general. New basic relations for STSs were shown to compose to the existing compound ones and allow a more detailed view on dependencies. A first description of the adaption of the process construction to attribution was presented in [8] including also an extension to transformation systems with arbitrary matching while monomorphic matches were required in the cases before.

Aims of the PhD project. Main challenge of the forthcoming PhD project is a complete foundation of the process construction and analysis techniques in the abstract setting of adhesive high level replacement (HLR) systems and the evaluation for practical workflow applications. Furthermore, a model transformation of the constructed STSs into Petri nets will allow the application of efficient analysis techniques available for different kinds of Petri nets. Selected aspects of the PhD project are explained in more detail in the following paragraphs and will be presented in the Doctorial Symposium.

Processes for Adhesive HLR Systems. The abstract framework of adhesive rewriting systems was extended to adhesive HLR systems [6] to integrate some additional systems of practical relevance, where Petri net and attributed graph transformation systems are important examples. Since adhesive HLR systems fulfil the conditions of HLR systems [9,10], all main results can be transferred including e.g. the local Church-Rosser, embedding, extension, parallelism and concurrency theorems. Transformation systems with attribution necessarily need possibly non-injective matches, because attributes in rules are terms, which may be evaluated to the same value. Therefore, an appropriate process construction needs to allow arbitrary matching. The solution in [11,8] is based on $\mathcal{E}-\mathcal{M}$-Factorization, which can be performed in many categories for graph structures. The process construction is extended by first factorizing the matches of the derivation. All rule applications are instantiated according to the factorization and lead to an instantiated derivation, which contains $\mathcal{M}$-morphisms only. Thereafter, the former process construction can be performed, but it is adapted to the class of $\mathcal{M}$-morphisms instead of general monomorphisms.

Extension for Application Conditions. Case studies of graph transformations often use the concept of application conditions and in particular negative application conditions (NACs), which restrict rule applications by prohibiting the presence of a negative pattern. In order to integrate application conditions in the process construction they need to be instantiated for each possible occurrence, because they define properties outside the image of rule matches. The instantiation can cause several copies of one condition, but some conditions may also disappear, if their patterns never occur at the corresponding matches. Furthermore, rule occurrences may depend on each other in several ways causing equivalent derivations that are not shift equivalent. While a derivation step may be independent from a sequence of steps it can depend on some steps in between. Therefore, a process will define a class of equivalent permutations of the steps of a derivation.

Conclusion. The proposed extension of the process construction for adhesive rewriting systems with monomorphic matches to adhesive HLR systems with application conditions and arbitrary matches will build a general and abstract foundation for both, formal semantics of true concurrency for various instantiations and a reliable framework for practical applications. The extraction of the minimal process model will additionally help to increase efficiency while the case study shall show possibilities for calculating equivalent executions that show maximal parallelism or improved properties with respect to the application domain.

References

1. Bottoni, P., De Rosa, F., Hoffmann, K., Mecella, M.: Applying Algebraic Approaches for Modeling Workflows and their Transformations in Mobile Networks. Journal of Mobile Information Systems 2(1), 51–76 (2006)
2. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Foundations. World Scientific, Singapore (1997)
3. Baldan, P.: Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars. PhD thesis, Computer Science Department - University of Pisa (2000)
4. Baldan, P., Corradini, A., Heindel, T., König, B., Sobocinski, P.: Processes for adhesive rewriting systems. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 202–216. Springer, Heidelberg (2006)
5. Lack, S., Sobociński, P.: Adhesive Categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 273–288. Springer, Heidelberg (2004)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
7. Corradini, A., Hermann, F., Sobociński, P.: Subobject Transformation Systems. Applied Categorical Structures 16(3) (June 2008), http://springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s10485-008-9127-6
8. Hermann, F., Ehrig, H.: Process Definition using Subobject Transformation Systems. EATCS Bulletin (to appear, 2008)
9. Ehrig, H., Habel, A., Kreowski, H.J., Parisi-Presicce, F.: Parallelism and concurrency in high-level replacement systems. Mathematical Structures in Computer Science 1, 361–404 (1991)
10. Ehrig, H., Habel, A., Kreowski, H.J., Parisi-Presicce, F.: From graph grammars to high level replacement systems. In: Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) Graph Grammars 1990. LNCS, vol. 532, pp. 269–291. Springer, Heidelberg (1991)
11. Hermann, F.: Process Definition of Adhesive HLR Systems. Technical Report 2008/09, TU Berlin, Fak. IV (to appear, 2008)

Towards a Two Layered Verification Approach for Compiled Graph Transformation

Ákos Horváth

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar Tudósok krt. 2
ahorvath@mit.bme.hu

1 Introduction

As model driven software development (MDSD) is being applied more and more in the safety critical (SC) and dependable system development processes there is an increasing need for verified model transformations to guarantee certain semantic properties to hold after their execution. For instance, when transforming UML models into Petri nets, the results of a formal analysis can be invalidated by erroneous model transformations when the system developer cannot easily distinguish whether an error is in the design or in the transformation.

In this paper we introduce our vision for verifying property preservation of graph transformation systems with a two layered approach.

2 Overview of the Approach

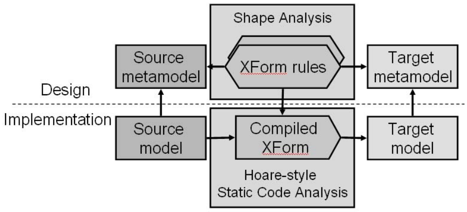


Fig. 1. Overview of the approach

Fig 1 gives an overview of our common model transformation process. The model transformation (*XForm rules*) is specified by a number of graph transformation rules. The GT rules are specified with respect to the metamodels of the source and the target metamodel. From these rule specifications, a compiled transformation (*compiled XForm*) is generated (see in [1]). The automatically derived compiled *XForm* transformation transforms a source model into a target model.

Our goal is to verify property preservation for the compiled transformation, meaning that, if a certain property holds in the source model after executing the transformation it will also hold in the target model. To achieve this we separated the verification process into two steps.

- First, we plan to apply shape analysis [2] on the *XForm* rules to summarize the behavior of a statement on an infinite set of possible rundown states of the GT rules. Shape analysis concerns the problem of determining *shape invariants* for

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 499–501, 2008.

programs that perform destructive updates on dynamically allocated storage. This way correctness of transformation rules applied to *any* model of the specified type can be verified (the concrete instances of the metamodels are irrelevant for the proof).

- Then, as the result of the shape analysis is based on the assumption that the GT rule specification are ”executed” semantically correct, in the second step we focus on the correctness check of the compiled GT rules. As the correctness of the generated compiled code depends on the correctness of the generator itself, that is usually a complex software components which cannot be verified easily. We use an alternative assurance approach, in which the generator is extended with formal program specification to enable Hoare-style [3] safety analysis for each individually generated GT rule. The crucial step in this approach is to extend the generator to produce all required annotations without compromising the assurance provided by the subsequent verification phase.

2.1 Analysis of Model Transformation Specification

Shape analysis: In our approach we plan to use the TVLA [4] (Three-Valued-Logic Analyzer), a system for automatically generating a static (shape) analysis implementation from the operational semantics of *XForm* rules. The small-step structural operational semantics is written in a meta-language based on first-order predicate logic with transitive closure. The main idea is that program states are represented as logical structures and the program transition system is defined using first order logical formulas. TVLA automatically generates the abstract semantics, and, for each program point, produces an abstract representation of the program states at that point. TVLA relies on a fundamental abstraction operation for converting a potentially unbounded structure into a bounded 3-valued structure (logic). 3-valued logic extends boolean logic by introducing a third value 1/2 denoting values that may be 0 or 1. A 3-valued logical structure can be used as an abstraction of a larger 2-valued logical structure. This is achieved by allowing an abstract state (i.e., a 3-valued logical structure) to include summary nodes, i.e., individuals that correspond to one or more individuals in a concrete state represented by that abstract state.

Our initial examples with the TVLA system shows that the mapping of the metamodel to the TVLA is a key problem for efficient shape analysis generation.

2.2 Analysis of Model Transformation Implementation

Hoare-style platform specific code analyzers: Hoare logic is a formal system to provide a set of logical rules in order to reason about the correctness of computer programs with the rigour of mathematical logic. The central feature of Hoare logic is the *Hoare triple*. A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form $\{P\}\ C\ \{Q\}$ where P, Q and C are *precondition*, *postcondition* and *command*, respectively. Based on the concept of pre-/postcondition introduced in the Hoare triple, *design by contract* [5] (DBC or programming by contract) prescribes that software designers should define precise verifiable interface specifications (pre/postconditions) for software components based upon the theory of abstract data types and the concept of a business contract. This means that *contracts* provides semantics to formally describe the behavior of a program module, removing potential ambiguity with regard to the module implementation.

Tools built upon the DBC methodology include the logic of predicate calculus and Dijkstra's weakest precondition calculations. We focused our studies on two of the most widely used frameworks: the (i) *Spec#* [6] programming system having developed at Microsoft Research to extend C# with formally verifiably method contracts in the form of pre-/postconditions as well as object invariants, and the (ii) *KeY* [7] formal software development system built upon a semi-automated prover over the Java Dynamic Logic (JavaDL) calculus (with support to Java Modeling Language (JML)) which covers the complete Java Card language, and additionally supports some Java SE features such as multi-dimensional arrays and dynamic object creation.

Both approaches look promising but our initial experiments show that none of them provide efficient support for: (i) dynamic casting of complex data structures (e.g., arrays), (ii) effective handling of nested loop invariants, (iii) contracts for library functions and finally (iv) user-friendly feedback from proof obligations.

3 Conclusion and Future Work

We have presented an ongoing work how graph transformations can be verified with a combination of shape analysis (with TVLA) and static code analyzer (e.g., Spec#, KeY). In the current state of our research, we have studied the boundaries of Hoare-style static code analyzers with respect to complex object navigation (as being the core of transformation implementation). It resulted in state space explosion in case of common implementations of GT rules and have to be further studied to achieve analyzable implementation.

As for the future, we plan to finish formalizing GT rules in 3-valued logic to achieve feasible shape analyze results and adapt the model logic described in [8] to capture properties of graph models.

References

1. Balogh, A., Varró, G., Varró, D., Pataricza, A.: Compiling model transformations to EJB3-specific transformer plugins, April 2006, pp. 1288–1295 (2006)
2. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3–valued logic. In: Symposium on Principles of Programming Languages, pp. 105–118. ACM Press, New York (1999)
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
4. Lev-Ami, T., Manevich, R., Sagiv, S.: Tvla: A system for generating abstract interpreters. In: Jacquart, R. (ed.) IFIP Congress Topical Sessions, pp. 367–376. Kluwer, Dordrecht (2004)
5. Meyer, B.: Applying "design by contract". Object-Oriented Systems and Applications 25(10), 40–51 (1994)
6. Spec#: The Spec# programming system, http://research.microsoft.com/specsharp/
7. The KeY Project: Integrated deductive software design, http://www.key-project.org/
8. Boneva, I.B., Rensink, A., Kurban, M.E., Bauer, J.: Graph abstraction and abstract graph transformation. Technical Report TR-CTIT-07-50, University of Twente (2007)

Model-Based Analysis of Network Reconfigurations Using Graph Transformation Systems

Ajab Khan

Department of Computer Science, University of Leicester
ak271@mcs.le.ac.uk

1 Introduction

P2P VoIP traffic potentially suffers from performance issues like packet loss, delay, jitter and echo [1], which greatly affect the quality of the service (QoS). Packet loss and delay mostly occur due to network reconfiguration caused by peer dynamics. In case of VoIP traffic, the network has to recover fast enough so that the quality of service is not affected [2].

Various solutions have been proposed to this problem, e.g., [3] proposes that an incentive has to be given to intermediate nodes and resource owners, whereas [4] proposes to keep many redundant links between peers, and [5] suggests that traffic has to be stopped and the routing may be changed. However, peer dynamics and complexity of P2P networks make it hard and expensive to validate these solutions [6] through classical means.

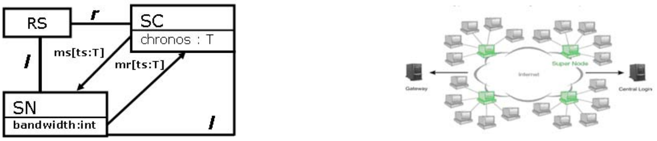
We propose to use model-based analysis and simulation to study the reconfiguration in P2P networks. The aim is to model different protocols in order to evaluate and improve their QoS properties with specific focus on VoIP applications. We consider the P2P network architecture as a graph, in which network nodes are represented by graph vertices and graph edges represent the network connections. Then reconfiguration in such a network can naturally be modeled by graph transformation [4] and stochastic analysis techniques can be used for validation. This extended abstract will motivate the idea by means of an example and discuss some of the challenges faced.

2 Case Study: Skype Network Architecture

The Skype P2P network offers three major services: VoIP, instant messaging and file transfer[7,8]. The architecture of the Skype network is described by the figures below. Skype nodes are distinguished into the Skype clients (SC) and super nodes (SN). Between them, SNs maintain an overlay network, while SCs have to connect to one of the SNs, which act as telephone switches or routers for the client. The Skype network is subject to architectural reconfiguration whenever a new SC joins the Network, an SN leaves the network, an intermediate SN fails to route the traffic or has problems of connectivity. Then, one of the SCs needs to be promoted to SN.

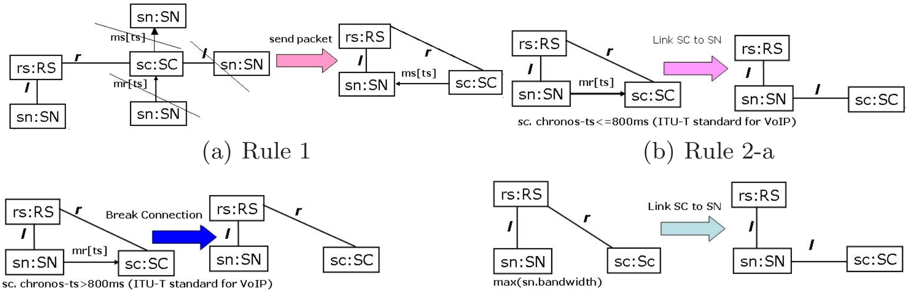
H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 502–504, 2008.

In our model we consider that after registration each Skype client keeps a permanent connection with a SN node: first the SC finds the latency of the SN node and if the latency is in the range of the standard i.e 800 ms then it establish a connection. Alternatively the SC can connect to SN which has the maximum spare bandwidth. The model can be best described with the help of the type graph below. i.e., registration servers (RS), super nodes (SN) and Skype clients (SC), while edge types model links (l), registrations (r), sending and receiving of time-stamped messages (ms, mr).



(a) Type Graph (b) Skype Network

Fig. 1. Network Model



(a) Rule 1 (b) Rule 2-a

(c) Rule 2-b (d) Rule 3

Fig. 2. Transformation Rules

3 Skype Reconfiguration as Graph Transformation

We refer to [4] for the basic notions of graph and graph transformation. We model the following scenario. When an SC tries to establish a network connection, it has to get registered with a central registration server. Then, the SC receives the addresses of a number of SNs, among which it has to make a selection based on their reachability, available bandwidth, etc. Before an SC links to an SN, it sends a packet to the node and waits for its return in order to measure the delay. This is the time it takes packets to travel from the SC to the SN and back. If the delay is in the range of the standards set by the International Telecommunication Union (ITU-T) [9] then it establishes a link with the SN. After the SC is linked, it can communicate directly with other peers, obtaining their address from the global index maintained by the central login server. The time stamping allows the peer to connect to a SN which offers the lowest latency as per ITU-T. If the latency is not acceptable, the connection attempt is abandoned and a new SN is selected.

The modelling of time follows the approach of a unique time stamp attribute chronos, associated with nodes or edges of the graph, as required.[10] The SC will be linked to the SN if the SCs current time when receiving the message mr[ts] has advanced no more than 800ms from the point when the time stamp ts has been set while sending ms[ts]. An alternative way of selecting an SN is to choose the one with the most available bandwidth. This is an example where, among the matches possible for a rule a selecting is performed which maximises a certain property(see Rule 3). Note that the example presented here is far from complete. Other aspects that need to be addressed include the promotion of peers to super peers, reaction to failures and disappearance of nodes, the connection to peers in local domains and behind firewalls, etc. which will be addressed in the extended version of the paper.

4 Challenges and Future Work

To carry out the research programme outlined in the introduction, of developing a methodology based on graph transformation for the modelling and validation of QOS properties of network protocols with dynamic topology, a number of problems have to be addressed in regard to the expressivity of the graph transformation approach, its formal semantics, and analysis techniques. In particular, these include the integration of the modeling of time with stochastic analysis techniques like model checking and simulation, and the formalization and implementation of optimising matches to support selecting the best match according to some specified metric.

References

1. Sounds, G.I.: Measuring Voice Quality (2006), http://www.gipscorp.com/files/english/white_papers/Measuring.doc
2. Arif, M.J., Karunasekera, S., Sovoip, S.K.: True convergence of data and voice network. In: 16th World wide web conference Banff, Alberta, Canada (May 2007)
3. Gupta, R., Somani, A.: Pricing strategy for incentivizing selfish nodes to share resources in peer-to-peer (p2p) networks. In: IEEE(ICON 2004), vol. 2 (2004)
4. Heckel., R.: Stochastic analysis of graph transformation systems: A case study in p2p networks. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722. Springer, Heidelberg (2005)
5. Lysne, O., Montanana, J.M., Pinkston, T.M.: Simple deadlock-free dynamic network reconfiguration. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296. Springer, Heidelberg (2004)
6. Merugu, S., Zegura, E.S.S.: P-sim: a simulator for peer-to-peer networks, publication. In: 11th IEEE, MASCOTS 2003 (October 2003)
7. Guha, S., Daswani, N., Jain., R.: n experimental study of the skype peer-to-peer voip system. In: 5th international workshop on peer-to-peer systems (iptps 2006) (2006)
8. Baset, S.A., Schulzrinne, H.G.: An analysis of the skype peer-to-peer internet telephony protocol. In: 25th IEEE, INFOCOM 2006 (April 2006)
9. Sounds, G.I.: VoIP-Better than PSTN? (2007), www.gipscorp.com/files/english/white_papers/VoIP_PSTN.WP.doc
10. Gyapay, S., Heckel, R., Varro, D.: Graph transformation with time: Causality and logical clocks. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505. Springer, Heidelberg (2002)

Service Extraction from Legacy Systems

Carlos Matos[1,2]

[1] Department of Computer Science, University of Leicester, United Kingdom
cmm22@mcs.le.ac.uk
[2] ATX Technologies Ltd, United Kingdom

Abstract. This paper presents a methodology for migration of legacy systems towards Service-Oriented Architectures. This approach is based on source code analysis and graph transformation with a central goal of allowing a high degree of automation.

The work presented here is developed in the context of a collaboration between academia and industry and is targetted to be applied systematically in software reengineering projects.

1 Introduction

The frequent changes of business requirements and evolution in technology occurring nowadays result in the need to develop new methods to support software evolution and, in particular, the transition of legacy systems to modern architectures. This need is observed frequently over the years and examples include the adoption of object-oriented programming languages, the advent of Web technologies and Service-Oriented Architectures (SOAs).

The adoption of SOAs has been growing and is becoming a prevailing software engineering practice [1]. However, experience indicates that SOA initiatives rarely start from scratch. The properties that derive from Service-Orientation include major challenges for reengineering of legacy systems:

1. The separation of business logic from presentation logic
2. The loosely coupled relationship between services
3. The coarse-grained nature of services

Legacy systems were not built with these concerns in mind thus much effort is necessary to accommodate them. This paper presents a methodology to address migration of legacy software to SOA while complying to the above properties.

2 General Methodology

Architecture migration involves different types of restructuring. Depending on the intended target, these are made along either technological and functional dimensions or both. The latter is the case of SOAs. Technological restructuring is used in the layering of software systems and may lead to a 3-tiered architecture,

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 505–507, 2008.

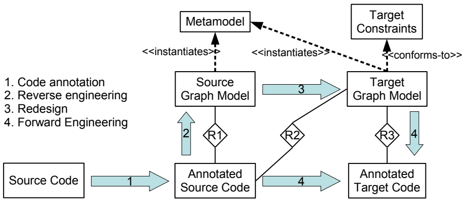


Fig. 1. General methodology

separating logic, data, and user interface (UI). Functional restructuring separates components which, after having replaced their UI tier with an appropriate interface and being grouped according to specific parameters, represent services.

The general methodology instanced for both technological and functional dimensions is presented in Figure 1. This consists of the steps: (1) code annotation-categorises blocks of source code according to the different elements of the target architecture they will be mapped to - in the technological dimension this is achieved by using code patterns to identify the parts of the code that belong to UI, Logic and Data; (2) reverse engineering - obtains a graph representation of the annotated code; (3) redesign - uses graph transformation rules to achieve the target architecture - in the technological dimension the rules aim to re-organise the model into a 3-tier architecture, thus complying to the first SOA property from the Introduction; and (4) forward engineering - uses the result of step 3 and traceability to the code (relations *R1* and *R2*) to obtain the target code.

One of the main goals of this methodology is to allow a high degree of automation. The manual intervention is in step 1 even though this is highly automated as well. A more detailed description of the methodology, and technological dimension, can be found in [2]. The next section details the functional dimension, except steps 2 and 4 as these are similar to the technological dimension.

3 Identification of Services

The functional code annotation phase is performed by two tasks: 1. operation identification and 2. grouping of operations into services. Identifying operation entry points is performed by using a combination of techniques that include:

- Code belonging to the Logic layer that is invoked by the UI (Figure 2);
- External API's (e.g. from IDL files);
- Code that falls into a typical pattern of control / data flow;
- Entry point for code that is mapped to more than one operation;
- Known feature location techniques such as LSI [3] and SBP [4].

The dependencies between each operation entry point and the remaining code can be determined using slicing techniques. The second task of service extraction is where the operations previously obtained are grouped into coherent services. Ranked groupings of operations are proposed by using metrics, including: overlapping between operations, actors involved, information about data accessed

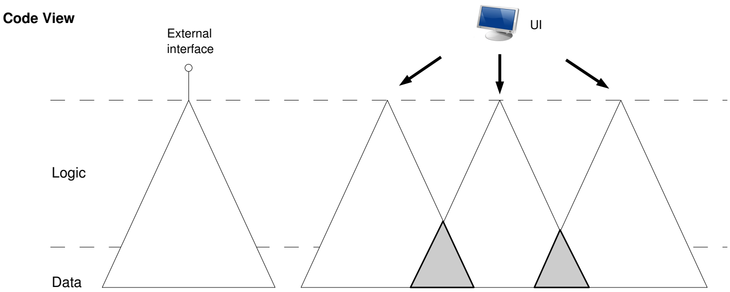


Fig. 2. Identification of operations

and similarity measure (e.g. using LSI). The graph transformation rules used in this dimension are designed so that operations are grouped in meaningful services (as per the categorisation) and that services have loosely coupled relations, thus complying to the two last SOA properties mentioned in the Introduction.

4 Conclusion

The main contribution of this work is the definition of a methodology that can be used for service extraction from legacy systems, while having a high level of automation. Code pattern matching and graph transformation are central to achieve this. By applying those in the technological and functional dimensions, the result is a concrete process of addressing SOA migration projects in a systematic way. This work is still ongoing but originated from problems found in projects and is developed in collaboration with the industry.

Acknowledgments. This work is partially supported by the IST-FET IP SENSORIA (IST-2005-16004). Thanks to Rui Correia (ATX Software), Prof. Reiko Heckel (University of Leicester) and Dr. Mohammad El-Ramly (Cairo University) for their participation.

References

1. McCoy, D., Natis, Y.: Service-oriented architecture: Mainstream straight ahead. Technical Report LE-19-7652, Gartner Research (April 2003)
2. Correia, R., Matos, C., Heckel, R., El-Ramly, M.: Architecture migration driven by code categorization. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758. Springer, Heidelberg (2007)
3. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: WCRE, pp. 214–223. IEEE Computer Society, Washington (2004)
4. Antoniol, G., Gueheneuc, Y.G.: Feature identification: A novel approach and a case study. In: ICSM, pp. 357–366. IEEE Computer Society Press, Washington (2005)

Development of Correct Graph Transformation Systems

Karl-Heinz Pennemann

University of Oldenburg, Germany*
pennemann@informatik.uni-oldenburg.de

Abstract. A major goal of this thesis is the ability to determine the correctness of graphical specifications consisting of a graph precondition, a graph program and graph postcondition. According to Dijkstra, the correctness of program specifications can be shown by constructing a weakest precondition of the program relative to the postcondition and checking whether the precondition implies the weakest precondition. With the intention of tool support, we investigate the construction of weakest graph preconditions, consider fragments of graph conditions, for which the implication problem is decidable, and investigate an approximative solution of said problem in the general case. All research is done within the framework of adhesive high-level replacement categories. Therefore, the results will be applicable to different kinds of transformation systems and petri nets.

Graph transformation has many application areas in computer science, such as software engineering or the design of concurrent and distributed systems. It is a visual modeling technique and expected to play a decisive role in the development of growingly larger and complex systems. However, the use of visual modeling techniques alone does not guarantee the correctness of a design. In context of rising standards for trustworthy systems, there is a growing need for the verification of graph transformation systems and programs. The research of appropriate methods for this purpose is the topic of this thesis. More precisely, a major goal of this thesis is the ability to determine the correctness of graph program specifications consisting of a graph precondition, a graph program and a graph postcondition. For an overview, see Figure 1.

Graph conditions. As language for the specification of state properties, graph conditions [5,6,7,8,9,10,11,12] are investigated. Graphs conditions are a graphical and intuitive, yet precise formalism, well-suited to describe structural properties of first-order.

Graph programs. Graph transformations rules with application conditions form the elementary steps of the considered computation model. Additional program constructs are nondeterministic choice, sequential composition, conditional execution, and various iterative constructs. Graph programs [13,1] are able to model

* This work is supported by the German Research Foundation (DFG), grants GRK 1076/1 (Graduate School on Trustworthy Software Systems) and HA 2936/2 (Development of Correct Graph Transformation Systems).

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 508–510, 2008.

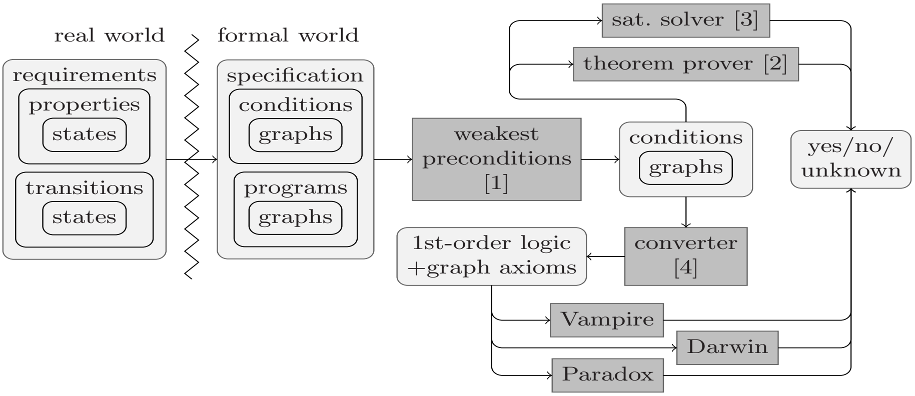


Fig. 1. The big picture

transactions that deal with an unbounded number of elements and are computationally complete.

Weakest preconditions. According to Dijkstra, the correctness of program specifications can be shown in a classical way by constructing a weakest precondition of the program relative to the postcondition and checking whether the precondition implies the weakest precondition. The construction of weakest preconditions and strongest postconditions of graph programs and graph postconditions is described in [1] and [4], respectively.

Implication problem. To determine, whether or not a graph precondition implies a weakest precondition, either a proof or a counterexample must be found. In [2], a correct calculus for proving graph conditions was proposed. In [3], a satisfiability algorithm was presented, proven to be correct and complete, and a decidable fragment of conditions was identified.

Implementation. Following the outlines of [14], the components sketched in Figure 1 are implemented. The weakest precondition transformer, the theorem prover and the satisfiability solver rely on a small number of structure-specific operations, provided by [15] for directed, labeled graphs.

Evaluation. A number of case studies are conducted, e.g. a railroad control system [9,11], an access control system [1,2], and a car platooning case study. The prover and solver components are evaluated against existing tools such as Vampire, Darwin and Paradox applied onto straightforward translations of graph conditions into first-order graph formulas [10,16,4], using the axiomatization of Courcelle, which is extended to labeled graphs.

Preliminary conclusion and further work. Graphs conditions are convenient to describe system requirements as well as suited to infer knowledge about system behavior. The developed methods and their implementations are able to automatically prove or refuse all test specifications of the access control case study. Surprisingly, existing tools such as Vampire, Darwin and Paradox fail to determine the correctness of some of these specifications. Reasons for this may include the axiomatization of direct, labeled graphs that tends to become a part of the problem to be solved. In contrast, condition-based algorithms are constructively restricted to the considered structure, e.g. directed labeled graphs.

References

1. Habel, A., Pennemann, K.-H., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer, Heidelberg (2006)
2. Pennemann, K.H.: Resolution-like theorem proving for high-level conditions. In: Ehrig, H., et al. (eds.) Proc.ICGT 2008. LNCS, vol. 5214. Springer, Heidelberg (2008)
3. Pennemann, K.H.: An algorithm for approximating the satisfiability problem of high-level conditions. In: Proc. GT-VC 2007. ENTCS, vol. 213, pp. 75–94. Elsevier, Amsterdam (2008)
4. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. MSCS (accepted for publication, 2008)
5. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae 26(3/4), 287–313 (1996)
6. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars. In: SEGRAGRA 1995. ENTCS, vol. 2, pp. 95–104 (1995)
7. Koch, M., Mancini, L., Parisi-Presicce, F.: Graph-based specification of access control policies. JCSS 71, 1–33 (2005)
8. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Constraints and Application Conditions: From Graphs to High-Level Structures. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 287–303. Springer, Heidelberg (2004)
9. Pennemann, K.H.: Generalized constraints and application conditions for graph transformation systems. Master's thesis, Department of Computing Science, University of Oldenburg, Oldenburg (2004)
10. Rensink, A.: Representing first-order logic by graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004)
11. 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)
12. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae 74, 135–166 (2006)
13. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 230–245. Springer, Heidelberg (2001)
14. Azab, K., Habel, A., Pennemann, K.H., Zuckschwerdt, C.: ENFORCe: A system for ensuring formal correctness of high-level programs. In: Proc. GraBaTs 2006. ECEASST, vol. 1, pp. 82–93 (2007)
15. Zuckschwerdt, C.: Ein System zur Transformation von Konsistenz in Anwendungsbedingungen. Berichte aus dem Department für Informatik 11/06, 114 pages, Universität Oldenburg (2006)
16. Habel, A., Pennemann, K.H.: Satisfiability of high-level conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 430–444. Springer, Heidelberg (2006)

Graph Transformation for the Semantic Web: Queries and Inference Rules

Hong Qing Yu* and Yi Hong

Department of Computer Science, University of Leicester.
University Road, Leicester, LE1 7RH, United Kingdom
{hqy1,yh37}@mcs.le.ac.uk

1 Introduction

Pervasive computing becomes an important characteristic of today's IT systems. In particular, context awareness is a core technology to enhance pervasive computing functionalities [1]. In recent years, Semantic Web (SW) is widely used in the development of mainstream software systems; it becomes increasingly important and feasible for improving conventional web data sharing process by providing machine-readable information and metadata expressed in various semantic web technologies such as Web Ontology Language (OWL) and Resource Description Framework (RDF). SW technology itself has already indicated the solutions to context modeling and sharing. However, the context querying and inferencing methodologies are still missing which leads to 2 major challenges.

- Specifying context inference rules at abstract level to provide validation and composition support for domain experts and users with ease-of-use front-end.
- Mapping abstract level rules into lower level executable rule language.

In this paper, we will address above issues by applying graph transformation theory.

2 General Approach

In our work, the context information is organized and stored as RDF/OWL documents. Thus, both context querying and reasoning are essential steps in terms of RDF/OWL processing. It is reasonable to use rules guiding their activities. The rule allows us to specify preconditions and postconditions of the query or inferencing procedure. The postconditions represent corresponding query or inference results. In additional, an RDF/OWL document can be seen as nodes and edges in as a directed and labelled graph. These characteristics are the prerequisite of graph transformation theory [2].

Moreover, RDF/OWL schema can be treated as the type graph (TG) while concrete instances of model can be shown as instance graph (IG) in terms of

* Thanks to Dr. Stephan Reiff-Marganiec and inContext (Interaction and Context Based Technologies for Collaborative Teams) partners for their discussions.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 511–513, 2008.

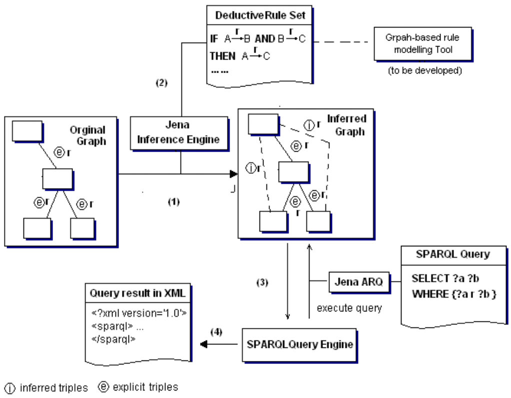


Fig. 1. General approach

graph transformation theory. Furthermore, pre- and post-conditions within reasoning rules are actually OWL/RDF graph pattern, which clearly indicates graph transformation would be a proper way for illustrating inference mechanism. The general approach that implements the graph transformation based context information inference and query includes four major steps (see Fig. 1): (1) Using graph transformation process to specify the reasoning rules both for query and inference. (2) Mapping the graph based rules into the Jena-based deductive rule syntax. (3) Executing SPARQL queries combining with the Jena reasoning rules. (4) Getting the query results as XML document.

3 RDF/OWL Graph and Graph Transformation

OWL ontology is a RDF graph which contains a set of triples. Both RDF subject and object are vertices V in the graph while predicate (property) are labelled edges E that represent a statement of a relationship between them [3]. RDF graph include two levels of abstraction: static structure of the model can be represented in OWL schema, which corresponds to type graph TG, while the snapshot at a specified time can be also described in OWL corresponding to instance graph. OWL provides mechanism to specify additional property restriction including symmetric property, transitive property and functional property etc, for example isFriendOf is a symmetric property that means if Person Alice isFriendOf Tom then Tom isFriendOf Alice. Similarly, isLocatedIn relationship can be declared as transitive property. Therefore, the approach of reasoning is to derive implicit triples from existing triples by means of graph transformation. The creation of implicit triples is modeled as graph transformation rules. The Fig. 2 shows how OWL transitive property is defined with graph transformation rules.

Similar graph transformation rules can be used to define other metamodel level properties such as symmetric and inverse property. Moreover, graph transformation rules also can specify inference at domain-specific level, which addresses

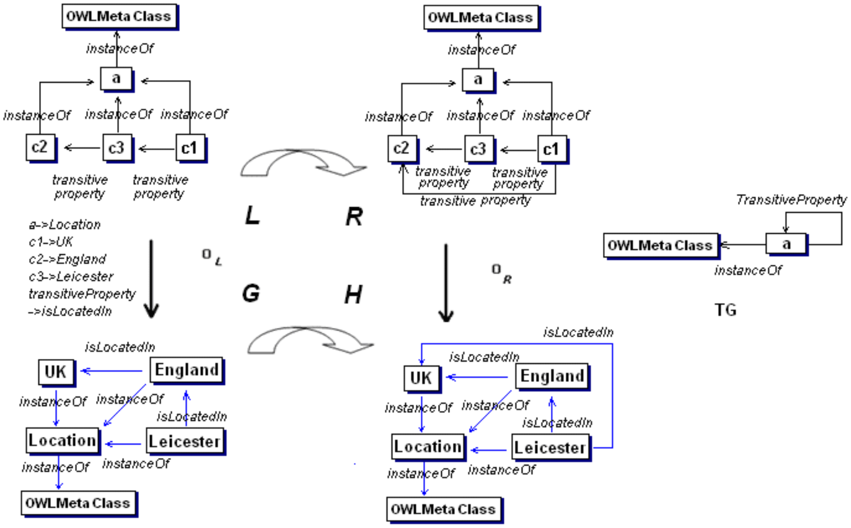


Fig. 2. Type graph (TG) and instance graph (IG) example

problems in a particular domain. In order to make the graph transformation rules executable, we mapped from our abstract rules to Jena-based inference syntax, which can be defined as follow:

```
[rule_name: (pre-condition 1),(pre-condition 2),...

-> (post-condition 1),(post-condition 2),...].
```

The left hand side of "→" consists of a set of triple patterns which correspond to graph L, right hand side stands for graph R.

In conclusion, we introduced a graph transformation based reasoning method in this paper. In particularly, we discussed the overview of the approach, the way to use RDF/OWL graph presenting the graph transformation rules and the mapping to lower level Jena syntax. To develop a debugable and visual graph rule design tool as shown is still an important future work.

References

1. El-Sayed, A.R., Black, J.P.: Semantic-Based Context-Aware Service Discovery in Pervasive-Computing Environments. In: Proc. of IEEE Workshop on Service Integration in Pervasive Environments (2006)
2. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools. World Scientific Publishing Co. Ptc. Ltd (1997) ISBN: 981-02-4020-1
3. Braatz, B., Brandt, C.: Graph Transformations for the Resource Description Framework. Journal of ECEASST (2008)

Using a Triple Graph Grammar for State Machine Implementations

Michael Striewe

Universität Duisburg-Essen
michael.striewe@s3.uni-due.de

1 Introduction and Concept

State machines can be comprehensively specified, simulated and validated at design time to get a formally founded skeleton for a software application. A direct implementation of state machines in Java source code can realize states as classes, transitions as methods and variables as well as pre-conditions and variable updates as auxilliary methods, invoking either arbitrary application methods to retrieve the current variable values or evaluating expressions [1]. Viewed from an abstract level, implementation and state machine are two models, sharing the same semantics. To maintain the modelled software systems, it is vitally important to perserve as much of this semantics as possible in the source code to be able to track back changes and errors [2,3]. In contrast to these considerations, current techniques of model-driven development use several unidirectional steps, starting from an abstract model and resulting in platfrom specific source code.

Following the ideas of [4], this problem leads to the idea of a Triple Graph Grammar, translating a state machine to an abstract syntax graph G_C for its implementation as well as to an abstract syntax graph G_V for a data format suitable for model verification. The necessary correspondence graph G_M can be understood as a meta-model of the actual state machine. This approach should not only allow simultaneous manipulation of two graphs, but open a way to transform between formal model and concrete implementation without loss of information and impact on source code that is not part of the meta-model realization. Note, that this bidirectional transformation is important, because in practice manipulations of source code can hardly be considered as strict application of transformation rules but as structured text editing. Therefore changes are not primarily performed by rewriting the meta-model, but triggered by either changing code or verification model. These changes have to be propagated through the graph triple, resulting in a virtually simultaneous change of both syntax graphs.

2 Mappings and Rules

In order to realize the desired transformations, two graph mappings have to be defined: one between the meta-model G_M and the syntax graph G_V and one between the meta-model G_M and the syntax graph G_C. Figure 1 shows a part of

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 514–516, 2008.

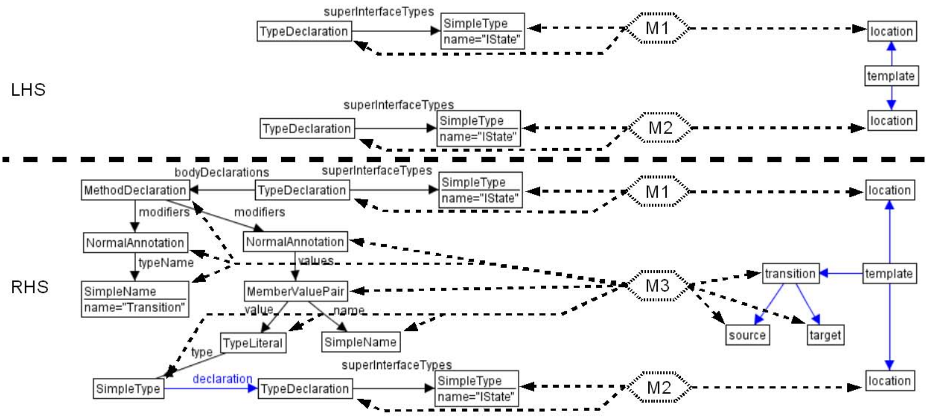


Fig. 1. TGG rule with mappings for inserting a new transition into an existing state machine. Mappings M1 and M2 map the states, while M3 maps the transition.

this mapping as an example, covering the correspondency in the rule for adding a new transition. In this example, G_C is placed on the left side and G_V on the right, while dashed boxes and arrows denote the mappings. The LHS consists of two states, mapped by M1 and M2 in the correspendence graph. The RHS adds the representation of a transition to both syntax graphs and conntects them through M3. From this rule we can derive three pairs of actual transformation rules. They are used to cover simultaneous transformations of all three graphs of the triple as well as to propagate changes of G_C to G_V and vice versa.

The first pair is the one for adding and removing a transition simultaneously in both syntax graphs. The rule can be derived directly from the TGG rule using the LHS and RHS for the manipulation of the syntax graphs. The second pair would be used to propagate changes from G_C to G_V. If a transition is added manually in this case, it is only present in the left syntax graph and M3 is missing. So the derived rule has to add the necessary nodes in the right syntax graph and extend the correspondence graph by inserting M3. If a transition is removed manually in this case, it is missing in the left syntax graph, but still present in the right one and the remainings of M3 still exist. So the derived rule has to remove the nodes from the right syntax graph and remove M3 from the correspondence graph. The same applies to the third pair of rule, but with switched sides to propagate changes from G_V to G_C.

Another set of rules is needed for adding and removing states. It is simpler than the example shown above as can be expected by looking at M1 or M2 in figure 1. In contrast to this, the rules for simultaneous manipulation and propagating changes in preconditions and variable updates are much more complex. Syntax trees for condition expressions may consist of virtually any number of syntactical elements. A series of rules has to be applied after adding a condition to handle the expression stepwise. Removing conditions is simpler, because they can be disconnected from the transitions by deleting one edge and then be removed by a generic ruleset for removing the dangling subtree. Further rules are needed as well, for example to mark the initial state of the state machine or to compose a list of model variables.

3 Implementation and Further Work

The presented approach has been preliminary implemented using the AGG tool environment for algebraic graph transformations [5]. Rules where derived manually from the described TGG rules. For retrieving and writing the abstract syntax graphs for Java, a plugin for the Eclipse IDE has been written. UPPAAL [6] was used as a state machine modeling and verification tool, storing its data in a XML file format which is easy to parse. This preliminary implementation confirmed the expected benefits of the concept, but did also reveal several drawbacks. Object structures representing the abstract syntax trees inside editing systems are often not prepared for the application of graph transformations. This make it necessary to export these structures into a graph format suitable for graph transformation and write them back afterwards. The result is multistage toolchain, which induces a significant slowdown and makes it difficult to realize simultaneous editing. Moreover, imports and exports may be error-prone, so the use for tracking back errors is limited.

Therefore one of the next goals is more direct tool support, using object structures directly by accessing editors through appropiate APIs. Then there would be no longer the need to use a toolchain and write the graph structure explicitly to disk, so a better performance can be expected. In an optimal solution an IDE would integrate source code editor, model editor and graph transformation engine, all sharing the same graph based data structure.

References

1. Goedicke, M., Balz, M., Striewe, M.: UPPAAL-Modelle als ausführbare Spezifikation in Java. In: Proceedings of Workshop Modellgetriebene Softwarearchitektur - Evolution, Integration und Migration (MSEIM 2008), Munich (2008)
2. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In: Briand, L., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)
3. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45(3), 451–461 (2006)
4. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903. Springer, Heidelberg (1995)
5. AGG-Team: The AGG website, http://tfs.cs.tu-berlin.de/agg/
6. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Int. Journal on Software Tools for Technology Transfer 1(1–2), 134–152 (1997)

Model-Driven Development of Model Transformations

Pieter Van Gorp

University of Antwerp
pieter.vangorp@ua.ac.be

Abstract. The "model-driven development of model transformations" requires both a technique to *model* model transformations as well as a means to *transform* transformation models. Therefore, the thesis underlying this paper evaluates and extends state-of-the-art model transformation approaches. For example, the thesis contributes a new language construct for modeling subgraph-copy operations. Perhaps surprisingly, this thesis intentionally does *not* propose a fundamentally new transformation language and toolset. Instead, the thesis is based on a small UML profile for controlled graph transformation. The profile only relies on class diagrams, activity diagrams, and the UML's extension mechanism. The proposed techniques have emerged from several case studies that involve model *evolution*, model *refinement*, as well as model *synchronization*.

1 Problem: Lack of Portability and Reuse

Controlled graph transformation gained industrial credibility in the nineties, thanks to the application of the Progres language (and tool) within industrial tool integration projects. After working within the Progres team, so-called "Story Diagrams" were proposed as a UML based syntax for modeling graph transformation systems. Since some implementation challenges were hard to overcome on top of the C based implementation of the Progres tool, the Java based Fujaba tool was implemented. With the advent of the MDA, several comparable tools were constructed once more. Unfortunately, several tools have proposed yet another syntax for existing graph transformation constructs. Moreover, none of the graph transformation tools relied on common libraries to reuse existing infrastructure for pattern matching, control flow, etc.

2 Solution: A Standard Transformation Modeling Profile

The first step to transformation tool integration is the agreement on a common metamodel for representing transformation models. Whenever possible, such a metamodel should be aligned with mainstream standards. As stated in the previous section, "Story Diagrams" was the first language for representing controlled graph transformation models in mainstream UML syntax. Unfortunately, the language was initially based on a proprietary and rather implicit metamodel.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 517–519, 2008.

Therefore, as a first contribution to the thesis, we aligned that controlled graph transformation language with the UML metamodel. Several off-the-shelf UML tools can now be used to edit the transformation models based on Story Diagrams. Moreover, the tool that supports this thesis relies on another mainstream MDA tool to transform transformation models into standard (i.e., MOF/JMI) compliant repository code. This lowers the cost and risk for adopting the proposed approach in an industrial context. Unlike the emerging QVT standard, there is no need to learn a completely new language and buy into a completely new toolset.

The core profile for modeling model transformations only supports the basic concepts of controlled graph transformation: it has the notion of a rewrite rule (matched elements, created elements, deleted elements and updated elements) and control flows (iterative loops, conditionals and called transformations). Interestingly, these constructs already enable one to model refactorings [3] as well as refinements [5] in a human friendly manner. Nevertheless, the proposed approach enables one to add more expressive language constructs instead of prematurely standardizing all transformation tools to the bare minimum of their "lowest common denominator".

3 Extensibility: Higher Order Transformations

The previous section only indicated how transformation model editors can be integrated at the syntactic level. The thesis proposes to define new language constructs as extensions to a small (the "core") transformation modeling profile. The role of higher order transformations is to transform transformation models that conform to an extension of the profile into transformation models that conform to the core profile (of which the semantics has been standardized).

A key to the proposed approach is that the higher order transformations themselves are modeled using the core profile for transformation modeling [4, Chapter 8]. Therefore, any transformation engine that supports the core profile can execute the higher order transformations. Consequently, any such engine can normalize transformation models that apply a new language construct into more primitive transformation models. Today, only a *Copy* operator has been realized using this approach. However, as new operators (such as *Merge*, *Diff*, ...) are introduced, a transformation tool may need to execute a series of publicly available higher order transformations before executing the result on its native graph transformation engine.

4 Related Work

First of all, Graph Transformation eXchange Language (GTXL [2]) has been proposed as a standard for exchanging transformation models. Unlike the proposed profile, GTXL has no relation to a mainstream modeling language such as the UML. Therefore, there are no off-the-shelf industrial tools for editing GTXL models. Secondly, the GTXL metamodel relies on a XML DTD instead of on the MOF. Therefore, it requires more integration effort in an MDA tool integration context. Finally, GTXL only supports uncontrolled rules whereas the proposed profile supports rules that are controlled by activity diagrams. Finally, due to the lack of a profile concept, GTXL cannot be extended without breaking metamodel compatibility with its implementations.

Secondly, the Queries/Views/Transformations (QVT) standard presents three languages for transformation modeling. Apart from the MOF basis, it has the same limitations as GTXL. The QVT standard does promote bridges between its sublanguages by means of higher order transformations. In fact, the mapping between the relations and core language is formalized in the QVT relations language. Unfortunately, the QVT relations language is not as generally applicable as the profile presented in this paper.

Within the VIATRA tool, the transformation process from human-oriented transformation models into machine-oriented transformation code is supported by higher order transformations too [1]. Unlike the proposed approach, the transformation models do not conform to any standards. Moreover, the higher order transformation is not written in a standard transformation language either.

5 Conclusions

This paper presented a new approach to the integration and extension of transformation languages and tools. A realization of the proposed approach enables transformation tool builders to focus on user-oriented added value (such as editor usability, run-time performance, ...) and new, *declarative* language constructs (such as a *Copy* operator), instead of spending time on the implementation of evaluation code that was already been realized in other tools before. A unique characteristic of the approach is that it only requires transformation tool builders to implement a small core, and provides support for more declarative language constructs without breaking interoperability.

References

1. Horváth, Á., Varró, D., Varró, G.: Automatic generation of platform-specific transformation. Info-Communications-Technology LXI(7), 40–45 (2006)
2. Lambers, L.: A new version of GTXL: An exchange format for graph transformation systems. Electronic Notes in Theoretical Computer Science 127, 51–63 (2005)
3. Schippers, H., Van Gorp, P., Janssens, D.: Leveraging UML profiles to generate plugins from visual model transformations. Electronic Notes in Theoretical Computer Science 127(3), 5–16 (2004); Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation
4. Van Gorp, P.: Model-driven Development of Model Transformations. PhD thesis, University of Antwerp (April 2008)
5. Van Gorp, P., Muliawan, O., Keller, A., Janssens, D.: Executing a platform independent model of the UML-to-CSP transformation on a commercial platform. In: Täntzer, G., Rensink, A. (eds.) AGTIVE 2007 Tool Contest (January 2008)

Transformation-Based Operationalization of Graph Languages

Erhard Weinell

RWTH Aachen University, Department of Computer Science 3,
Ahornstrasse 55, D-52074 Aachen, Germany
Weinell@cs.rwth-aachen.de

Abstract. Graph Languages[1] emerged during the seventies from the necessity to process data structures with complex interrelations. Nowadays, various variants of these languages can be found for querying [1][2], in-place transforming [3][4], and translating graph structures [5][6]. Still, new graph languages supporting different paradigms and usage scenarios are proposed regularly. In fact, languages tailored for a dedicated application domain can be restricted to a concise and clear syntax representation, thus reducing effort to learn and apply them. Effectively aiding the development of graph languages, even though considering the already existing ones, therefore remains an important working topic.

Constructing specialized graph languages, considering them as special case of domain-specific modeling languages, is supported by various frameworks and so-called Meta-CASE tools, e.g. [7]. Operational implementations of these languages is usually achieved by customizing template-based code generators. However, graph languages, in contrast to purely static modeling languages, are inherently complex to implement due to the required pattern matching facility, and the possibly required non-deterministic execution engine.

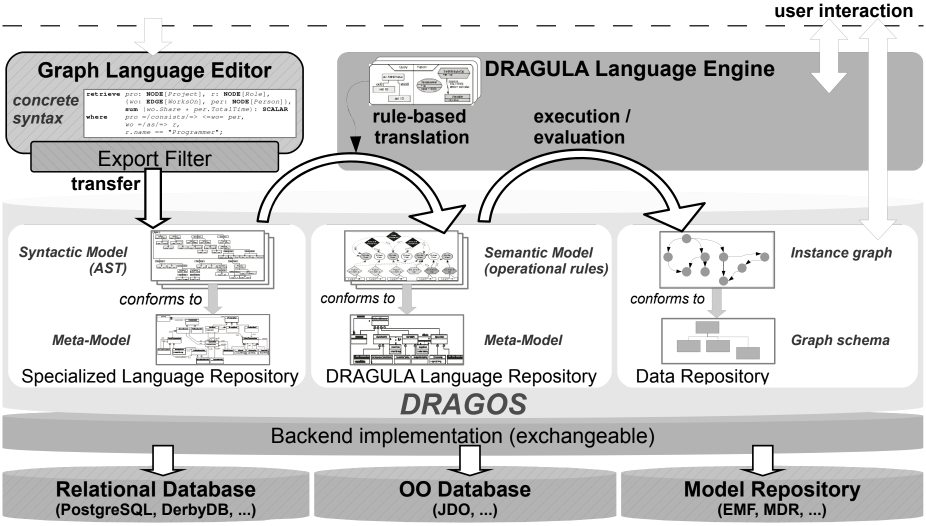
As alternative to the usual code generation approach, I propose a solution to implement graph languages by *transformation*. The approach is based on an extensible core graph language, to which rules modeled in a specialized graph language are transformed. Extensions can be added to the core language to approximate both languages's conceptual levels, and thus to narrow their "semantic gap". In contrast, a code generation module would have to span a significantly larger gap from a high-level specification language to an imperative or object-oriented programming language.

A coarse-grained overview on the presented approach is given in Figure 1. Technically, this platform is built on top of the graph-oriented database DRAGOS [8]. Thanks to DRAGOS' exchangeable backends, language implementations gain access to established storage solutions like relational databases or model repositories.

To construct a new graph language, developers usually build an editor based on the language's concrete syntax model, be that a textual or visual one. Based on this, a partly generic export facility transfers

[1] The term *Graph Language* subsumes languages for querying and transforming graphs, and especially from the overlapping of these areas.

H. Ehrig et al. (Eds.): ICGT 2008, LNCS 5214, pp. 520–522, 2008.



```
retrieve pro: NODE[Project], r: NODE[Role],
         {wo: EDGE[WorksOn], per: NODE[Person]},
         sum (wo.Share * per.TotalTime): SCALAR
where    pro =/consists/=> <=wo= per,
         wo =/as/=> r,
         r.name == "Programmer";
```

Fig. 1. Architectural overview

instances of this language, e.g. entered by the user at runtime, into the graph database as abstract syntax trees (ASTs). This intermediate storage facility decouples further processing from the actual concrete syntax and from the applied editing technology.

Afterwards, as the first curved arrow suggests, the ASTs are transformed into instances of the provided core graph language, DRAGULA. This transformation, which needs to precisely capture the specialized language's intended semantics, can be modeled in a rule-based way. For this purpose, a simple uni-directional model transformation language is provided. Technically, this language's rule instances are stored in an additional repository in the database, and are transformed to the core language, too.

Finally, the generated core language rules can be evaluated by the corresponding engine, thereby referring to the database's data repository. Both the rule engine and the data repository are subject to user interaction, e.g. to select rules for invocation or to directly inspect the stored graphs. This talk primarily discusses the first curved arrow, whereas the second one has been described in [9], and the core language's extensibility in [10].

Summary. The proposed solution eases operational implementations of graph languages, using a rule-based transformation approach. The DRAGULA language is well-suited to implement different kinds of languages, e.g. for queries and transformations. Extensions allow to capture additional functionality.

Related work. Modeling a domain-specific language's dynamic semantics is offered by some meta-case tools, e.g. presented in [11]. In contrast to the present work relying on a core graph language as target domain, the authors apply petri nets for this purpose. Though restricting their approach in expressing graph languages, petri nets naturally provide valuable analytical properties.

Existing core graph languages can also be found in the literature, such as the lately proposed GP [12]. GP offers core functionality for a basic graph model, whilst DRAGULA is tailored towards complex models including hierarchical graphs and hypergraphs. Furthermore, DRAGULA focusses on extensibility to allow concise mapping of specialized graph languages [10]. Like GP already does, DRAGULA will soon support non-deterministic rule application.

References

1. Consens, M., Mendelzon, A.: GraphLog: a visual formalism for real life recursion. In: Proc. of the ACM Symp. on Principles of Database Systems, pp. 404–416 (1990)
2. Kullbach, B., Winter, A.: Querying as an enabling technology in software reengineering. In: Proc. of the 3rd Europ. Conf. on Software Maintenance and Reengineering, pp. 42–50. IEEE Computer Society Press, Los Alamitos (1999)
3. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: Language and environment. In: [13], pp. 487–550
4. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: Language and environment. In: [13], pp. 551–603
5. Agrawal, A., et al.: The design of a language for model transformations. Software and Systems Modeling 5(3), 261–288 (2006)
6. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: SAC 2006, pp. 1280–1287. ACM Press, New York (2006)
7. Ebert, J., Süttenbach, R., Uhe, I.: Meta-CASE in practice: a case for KOGGE. In: Olivé, À., Pastor, J.A. (eds.) CAiSE 1997. LNCS, vol. 1250, pp. 203–216. Springer, Heidelberg (1997)
8. Böhlen, B.: Ein parametrisierbares Graph-Datenbanksystem für Entwicklungswerkzeuge. PhD thesis, RWTH Aachen (2006)
9. Weinell, E.: Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Proc. of the 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE). Lect. Notes in Comp. Sci, vol. 5088, pp. 369–411. Springer, Heidelberg (2008)
10. Weinell, E.: Extending graph query languages by reduction. In: [14]
11. de Lara, J., Vangheluwe, H.: Translating model simulators to analysis models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 77–92. Springer, Heidelberg (2008)
12. Manning, G., Plump, D.: The GP programming system. In: [14]
13. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools, vol. 2. World Scientific, Singapore (1999)
14. Ermel, C., Heckel, R., de Lara, J. (eds.): Graph Transformation and Visual Modeling Techniques, 7th Intl. Workshop. Elec. Comm. of the EASST, vol. 10 (2008)

Author Index

Arnould, Agnès 147
Azab, Karl 211

Baldan, Paolo 242, 257, 467
Bauderon, Michel 366
Bauer, Jörg 321
Bergmann, Gábor 396
Bisztray, Dénes 475
Bonchi, Filippo 226, 478
Boneva, Iovka 321
Bruggink, H.J. Sander 336

Cenciarelli, Pietro 131
Chalopin, Jérémie 54
Chen, Rui 366
Comet, Jean-Paul 147
Corradini, Andrea 257, 473

Dang, Duc-Hanh 481
de Lara, Juan 426
Derbel, Bilel 99
Dodds, Mike 484
Dörr, Heiko 38
Drewes, Frank 442

Echahed, Rachid 84
Ehrig, Hartmut 162, 178, 194, 242, 257
Ermel, Claudia 194

Gadducci, Fabio 115, 226
Gorla, Daniele 131
Grohmann, Davide 487
Gruner, Stefan 99
Guerra, Esther 426

Habel, Annegret 211, 460
Hammoudeh, Mohammad 490
Heckel, Reiko 458
Heindel, Tobias 226, 493
Hermann, Frank 496
Hoffmann, Berthold 442
Hong, Yi 511
Horváth, Ákos 396, 499

Jelínková, Eva 379

Khan, Ajab 502
Klar, Felix 411
König, Barbara 242, 257, 305, 336, 467
Kozioura, Vitali 305
Kratochvíl, Jan 379
Kreowski, Hans-Jörg 351
Kurbán, Marcos E. 321
Kuske, Sabine 351

Lambers, Leen 162, 242
Le Gall, Pascale 147
Ly, Olivier 366

Maier, Sonja 39
Matos, Carlos 505
Mazanek, Steffen 39
Mazurkiewicz, Antoni 54
Métivier, Yves 54
Minas, Mark 39, 442
Monreale, Giacoma Valentina 115
Mosbah, Mohamed 99, 460

Noll, Thomas 69

Orejas, Fernando 162, 274

Pennemann, Karl-Heinz 289, 508
Petre, Ion 470
Poudret, Mathieu 147
Prange, Ulrike 162, 178

Rangel, Guilherme 242
Ráth, István 396
Rensink, Arend 321, 463
Rieger, Stefan 69
Rozenberg, Grzegorz 470

Schürr, Andy 411
Stevens, Perdita 1
Striewe, Michael 514

Tuosto, Emilio 131, 473

van der Aalst, W.M.P. 18
Van Gorp, Pieter 463, 517
Varró, Dániel 396

Weinell, Erhard 520

Yu, Hong Qing 511